hep-aid: A Python Library for Sample Efficient Parameter Scans in Beyond the Standard Model Phenomenology

Mauricio A. Diaz[a], Srinandan Dasmahapatra[b], Stefano Moretti[a,c]

[a]School of Physics & Astronomy, University of Southampton, Southampton, SO17 1BJ, UK

[b]School of Electronics & Computer Science, University of Southampton, Southampton, SO17 1BJ, UK

[c]Department of Physics & Astronomy, Uppsala University, Box 516, 75120, Uppsala, Sweden

Abstract

This paper presents hep-aid, a modular Python library conceived for utilising, implementing, and developing parameter scan algorithms. Originally devised for sample-efficient, multi-objective active search approaches in computationally expensive Beyond Standard Model (BSM) phenomenology, the library currently integrates three Machine Learning (ML)-based approaches: a Constraint Active Search (CAS) algorithm, a multi-objective Active Search (AS) method (called b-CASTOR), and a self-exploration method named Machine Learning Scan (MLScan). These approaches address the challenge of multi-objective optimisation in high-dimensional BSM scenarios by employing surrogate models and strategically exploring parameter spaces to identify regions that satisfy complex objectives with fewer evaluations. Additionally, a Markov-Chain Monte Carlo method using the Metropolis-Hastings algorithm (MCMC-MH) is implemented for method comparison. The library also includes a High Energy Physics (HEP) module based on SPheno as the spectrum calculator. However, the library modules and functionalities are designed to be easily extended and used also with other external software for phenomenology. This manual provides an introduction on how to use the main functionalities of hep-aid and describes the design and structure of the library. Demonstrations based on the aforementioned parameter scan methods show that hep-aid methodologies enhance the efficiency of BSM studies, offering a versatile toolset for complex, multi-objective searches for new physics in HEP contexts exploiting advanced ML-based approaches.

Keywords: High Energy Physics, Beyond the Standard Model phenomenology, Python Library, Parameter Scan, Active Search, Machine Learning

Contents

1. Introduction

Within Beyond Standard Model (BSM) phenomenology, the Parameter Scan (PS) problem [1] involves a systematic exploration of the multi-dimensional parameter space of a new physics scenario. This process includes calculating numerical values for model predictions across various points in its parameter space, applying experimental and theoretical constraints, and identifying satisfactory regions that can explain multiple phenomena. The satisfactory regions in the parameter space are found by checking whether a theoretical prediction matches within some error margin measured features of anomalous data or respects exclusion limits if no BSM observations have been made.

PS methods must address several computational challenges, covering high-dimensional parameter spaces, the computational cost of numerical evaluations, and satisfying a large number of constraints in the physical observable space. Additionally, PS methods may be further limited by the computational resources available for the study. Therefore, selecting a PS method suitable for a specific phenomenological study is not trivial. Expert knowledge of the BSM model and its computational demands must be evaluated to perform a successful phenomenological study.

The PS problem is commonly framed as a sampling problem [2], tackled using Bayesian in-

ference techniques [3], such as Markov-Chain Monte Carlo (MCMC) methods (see, e.g., [4]), to estimate the probability density functions across the parameter space. Recent advances have explored the integration of Machine Learning (ML) [5, 6] methods into PS techniques offering a promising approach to addressing challenges related to efficiency and scalability. Neural Network (NN)-based methods [7, 8, 9, 10] have been investigated by framing the problem as a regression, classification, or generative task. This involves training a NN model to approximate the observables from the inputs or to classify and assess the viability of parameter configurations under experimental constraints. These models are subsequently incorporated into informed sampling strategies. Active Learning (AL) [11, 12] techniques have also been utilised to train NNs to approximate the decision boundaries for the allowed parameter space. Furthermore, the PS problem has been framed for black-box optimisation techniques, employing methods such as Bayesian optimisation and evolutionary strategies [13, 14, 15]. Recently, another formulation was investigated by applying Active Search (AS) approaches to enhance sample efficiency in computationally expensive BSM scenarios [16]. This kind of work led to the development of the library introduced here.

Several PS and sampling libraries have been developed for phenomenology, including `BSM Toolbox` [17], `xBit` [18], `EasyScan_HEP` [19], and `BSMart` [20]. Each of these libraries addresses the PS problem with unique software designs and specific usage goals. They share common features such as integration with a set of High Energy Physics (HEP) packages, implementation of various PS algorithms, and the use of configuration files to simplify setup. These tools provide the community with a range of resources tailored to different applications.

In this work, we introduce a new Python library, `hep-aid`, which provides a modular framework for developing PS algorithms for BSM phenomenology and also adopts the principles of related libraries. It manages the HEP software and provides components to ease the utilisation, implementation, and development of PS algorithms for phenomenological studies. The library comprises two main modules: the `hep` module and the `search` module.

The `hep` module facilitates the integration of the HEP software ecosystem into Python scripts. It allows users to perform a first-principles computation of observable quantities to compare with experimental data for each parameter space point using a stack of HEP software, collecting the output with a single function call. Currently, the SARAH [21, 22] family of programs is implemented. The `search` module manages PS algorithms, following an AS [23] paradigm in which a search policy and a surrogate model are employed to explore the parameter space of a multi-objective function to find parameter configurations where the objectives satisfy a set of constraints. This framework allows the integration of potentially any PS method, such as MCMC or ML based sampling methods. The connection between the PS algorithms in the `search` module and the HEP software in the `hep` module is established through the construction of an *objective function*. The `search` module includes an *objective function* constructor, which defines the search space, objectives, and constraints based on a predefined configuration. It also maintains an internal dataset of samples with functionalities for saving, loading, and exporting datasets in formats such as a Pandas DataFrame [24] or a PyTorch [25] tensor.

We demonstrate the use of the library through quick tests and real BSM scan examples. We

employ an AS parameter scan method to study the $(B-L)$ Supersymmetric Standard Model ($(B-L)$SSM) [26], accommodating results from an observed signal at approximately 95 GeV in neutral (pseudo)scalar searches in the $h \to \gamma\gamma$ channel, as observed by CMS [27].

The article is structured as follows. In Section 2, we provide a brief overview of the library, followed by code examples for different use cases. (This section also serves as a quick start guide.) In Section 3, we describe the integration of HEP software, i.e., the `hep` component. Section 4 explains the internal structure of the `search` component of the library and details the parameter scans implemented in `hep-aid`. We then conclude in Section 5.

2. The `hep-aid` Library

The `hep-aid` library provides a modular framework for performing parameter scans in BSM scenarios, currently using SPheno [28, 29], HiggsBounds (HB) [30], HiggsSignals (HS) [31] and MadGraph (MG) [32], which we call the *HEP-stack*. It is focused on the AS [23] paradigm, where a multi-dimensional multi-objective function needs to be defined and the PS algorithm searches for satisfactory configurations in the parameter space given a set of constraints on the objectives. The search is done using a surrogate model, which is fitted to the collected data, to approximate the objective function and to assess which regions of parameter space to explore by querying the HEP-stack. This yields parameter configurations where the objectives satisfy the constraints. We call this region the *satisfactory region*. Originally the library was created to give the user simple access to use the b-CASTOR [16] and Constraint AS (CAS) [33] algorithms for parameter scans but, given the modular structure of it, many parameter scan algorithms can be implemented: e.g., `hep-aid` already includes a MCMC method using the Metropolis-Hastings sampling algorithm (MCMC-MH) [34, 35], MLScan [7], and a NN based sampling algorithm for BSM phenomenology. The library is divided into two key modules, the `hep` module and the `search` module, illustrated in Figure 1.

The `hep` module facilitates the integration of the HEP software ecosystem into Python scripts. The `hep.stack` module integrates SPheno, HB, HS and MG in a sequential stack of software. Technically, this has four HEP-stacks, ranging from SPheno used independently to the full chain incorporating all four HEP programs, and the user can utilise any of these HEP-stacks depending on the phenomenological study. Each HEP-stack includes operational utilities, assisting in handling and managing data, mainly SUSY Les Houches Accord (SLHA) [36] files, and running the software externally. The HEP-stacks need to be initialised with a pre-defined configuration file. This configuration file contains information about the SPheno inputs that will undergo a parameter scan and the necessary directories for the HEP tools used in the scan.

The library offers a quick installation method integrated as a command-line feature, along with a function to create a template HEP-stack configuration file. Once the configuration file is defined and the HEP-stack is initiated, the `hep.stack` module enables the user to run a parameter space configuration defined as a simple array. This array corresponds to the input parameters for SPheno, as specified in the configuration file. The result contains all the input and output information of the HEP tools used in the HEP-stack, formatted as a Python dictionary.

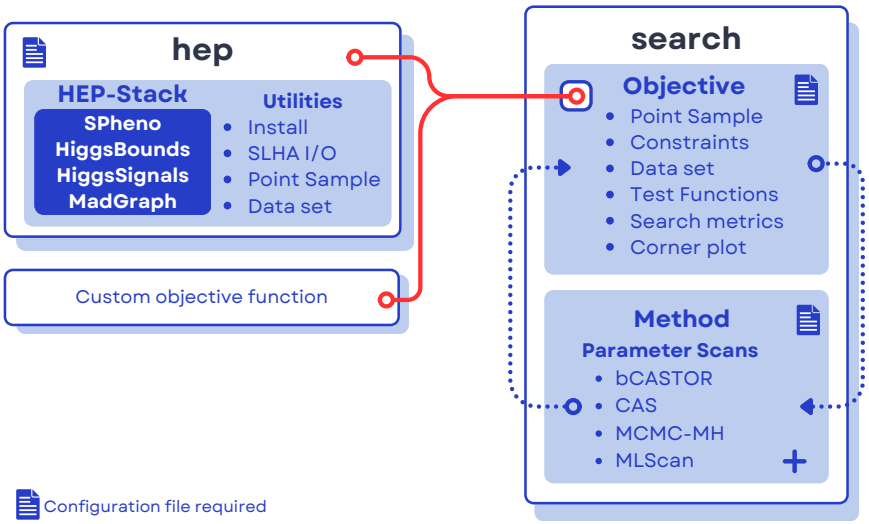

Figure 1: Diagram showing the general structure of the `hep-aid` library, which includes two main modules: `hep` and `search`. The search module contains the objective and methods submodules. The `objective` submodule allows users to plug in any custom objective function. The `hep` module serves as a specialised external objective function that integrates with HEP software and can be seamlessly used by the objective submodule for integration.

The `search` module provides all the necessary components for performing a search on a multi-objective function, either by using the `hep` module or by defining a custom function. It also includes PS algorithms and tools to support experimentation and the development of new PS methods. Since we adhere to the AS approach, `hep-aid` also includes *surrogate models* to approximate the objective function under study. These features are implemented in three main sub modules, `search.objective`, `search.methods` and `search.models`. The `objective` module contains the `Objective` class, which is the objective function constructor it needs to be initialised with a configuration file stating the search space dimensions with their ranges, the objectives variables and the information of the constraints. The `Objective` class internally manages the sampling of the objective function, stores the dataset, performs data processing to export to the *surrogate models*, and saves the dataset to disk. The `methods` module contains all the PS methods available currently, namely, MCMC-MH [34, 35], b-CASTOR [16], CAS [33] and MLScan [7]. Each method is constructed by inheriting the `Method` base class which loads a specific configuration file and saves or loads checkpoints to continue the search. A set of metrics is recorded in each PS method, such as the total number of parameters sampled, and those that satisfy the constraints. Further metrics may be customised for different use cases. Finally, if the parameter scan algorithms follow an AS approach, they will use the surrogate models implemented in the `search.models` module currently containing Gaussian Processes (GPs) for the AS algorithms and a simple Multi-Layer Perceptron (MLP) for the MLScan method.

The workflow idea that `hep-aid` proposes is that the users define an objective function using the `Objective` class and define its configuration file. They can then run the preferred PS method which uses the initiated `Objective` object. For BSM phenomenological analyses using the `hep` module, the workflow needs to be more elaborated since the objective function needs to be con-

```python
# Import the HEP-Stack module
from hepaid.hep.stack import HEPSTACK

# Initialise the HEP-Stack with a configuration file
hep_config = 'hep_stack_config.yml'
hepstack = HEPSTACK['SPhenoHBHSMG5'](hep_config=hep_config)

# Define the benchmark point for sampling
benchmark_point = [800, 2300, 20, 3800, 3900, 3900, 1e6, 1e6]

# Perform the sampling
result = hepstack.sample(benchmark_point)

# Extract key results from the output
mhp = result['SLHA']['MASS']['entries']['25']['value']    # Mass of particle 25
mh = result['SLHA']['MASS']['entries']['35']['value']     # Mass of particle 35
r_hb = result['HB']['obsratio']                           # Observed ratio in HB
csq_hs = result['HS']['csq(tot)']                         # Chi-squared total in HS
```

Figure 2: Example code demonstrating the initialisation of the HEP-stack with SPheno, HB, HS, and MG using a HEP-stack class specific to the $(B-L)$SSM. The corresponding `SPhenoHBHSMG5` object runs a parameter configuration defined by an array, returning results in a Python dictionary. Key outputs, such as the Higgs particle masses (`mhp`, `mh`), the HB ratio (`r_hb`), and the HS χ^2 (`csq_hs`), can be extracted for analysis.

structed in a appropriate manner compatible with the `Objective` class, by retrieving the necessary values of masses, cross sections, Branching Ratios (BRs), etc.: see the code example in Figure 2.

2.1. Library Overview

Here, we describe in detail how to make use of `hep-aid`.

2.1.1. Installation

The library is publicly available in GitHub[1]. To install `hep-aid` one needs to clone the repository first, then proceed as illustrated in Figure 3.

```
git clone https://github.com/mjadiaz/hep-aid.git
pip install.
```

Figure 3: Commands to clone the repository and install the `hep-aid` package.

2.1.2. Test Objective Functions

To provide a practical overview of how to use various functionalities of `hep-aid`, we utilise the test function introduced in [16]. This is a two-dimensional, double-objective function comprising both uni-modal and multi-modal objectives, with partially overlapping constraints for each objective that defines the satisfactory region. The ground truth satisfactory region is shown in Figure 4. The general pipeline for using a parameter scan with `hep-aid` is illustrated in the example code in Figure 5. When the search method is executed in the command line, a `progress` bar displays key metrics related to the search. In Figure 5, which demonstrates the b-CASTOR method, the

[1] https://github.com/mjadiaz/hep-aid.

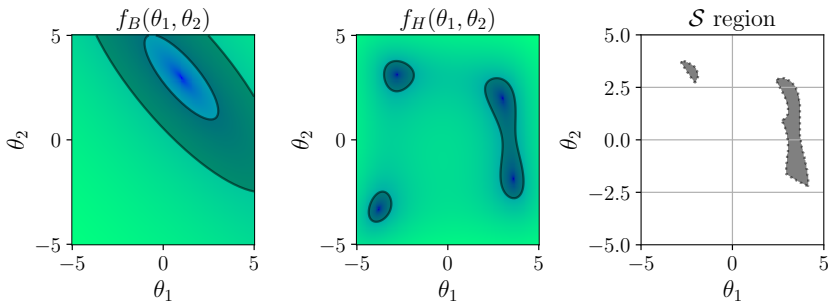

Figure 4: Two-dimensional double-objective test function, where the objectives must satisfy the shaded areas. The overlapping shaded areas define the satisfactory region, denoted by $\mathcal{S}$ on the right plot.

```python
from hepaid.search.objective.test import init_him_boo_fn
from hepaid.search.method import bCASTOR

# Initialise the test function
test_objective = init_him_boo_fn()

# Initialise the method algorithm
bcastor = bCASTOR(objective=test_objective)
# Run search
bcastor.run()
```

Figure 5: Quick start guide illustrating the following: initialising a two-dimensional double objective test function, selecting and starting the b-CASTOR method, then running the search.

progress bar will track the success rate, total points sampled, valid points count, and satisfactory points count. It also provides the current iteration and the parameter r, defining the radius of the neighbourhood around each parameter vector. Further details on this algorithm are provided in Section 4.2. The dataset generated from the search is stored in `test_objective.dataset`. This dataset can be visualised using a corner plot, a technique that displays pairwise correlations between multiple variables alongside their marginal distributions. In this context, the plot illustrates the various dimensions of the search space and potentially the objective dimensions. The corner plot can be generated using the utility module `hepaid.search.objective.plot`, as demonstrated in Figure 6.

Every ML-based active PS method which is used retains a copy of the fitted surrogate model in the `method.method` attribute. This model can be accessed after completing the search and will correspond to the model used in the latest iteration of the search loop. Figure 7 demonstrates how to use routines from the `.plot` module, e.g., `generate_meshgrid` and `reshape_model_output`, and to create filled contour plots that visualise the objective approximations generated by the surrogate model fitted to the current dataset.

The `hep-aid` framework currently includes four main PS methods: MCMC-MH [34, 35], b-CASTOR [16], CAS [33], and MLScan [7]. (The details of each algorithm are described in

```python
from hepaid.search.objective.plot import CornerPlot

# Define 'good' for satisfactory points, 'bad' for the rest
good = test_objective.satisfactory.prod(axis=1).astype(bool)
bad = ~good

# Export dataset as a DataFrame
df = test_objective.as_dataframe()

# Initialise CornerPlot for visualisation
cp = CornerPlot(
    figsize=(4, 4),
    parameters=test_objective.input_parameters,
    colormap='winter_r'
)

# Add traces for satisfactory and unsatisfactory points
cp.add_trace(df[good], r'$\in \mathcal{S}$', alpha=1)
cp.add_trace(df[bad], r'$\notin \mathcal{S}$', alpha=1)

# Update legend and clean up figure
cp.update_legend()
cp.clean()

# Save figure
cp.fig.savefig('plots/bcastor_run_result_himboo.png', dpi=300)
```

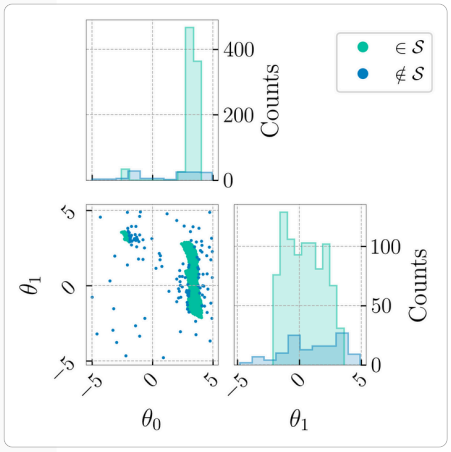

Figure 6: Code snippet that demonstrates the use of the `CornerPlot` utility to visualise data points classified as satisfactory or unsatisfactory. The dataset is filtered using the `Objective.satisfactory` attribute, and traces are added to distinguish between the two classifications.

```python
from hepaid.search.objective.plot import generate_meshgrid
from hepaid.search.objective.plot import reshape_model_output

# Generate the grid
dim_ranges = [(0, 1), (0, 1)]
steps_per_dim = [50, 50]
grid_points, meshgrids = generate_meshgrid(
    dim_ranges, steps_per_dim
    )

# Use the trained surrogate model
mean, sigma = bcastor.model.predict(grid_points)

# Reshape mean with the grid dimensions
reshaped_preds= reshape_model_output(
        model_output=mean[:,i],
        steps_per_dim=steps_per_dim
        ).detach().numpy()
    xx, yy = meshgrids

# Create plot with routines like contourf or imshow
```

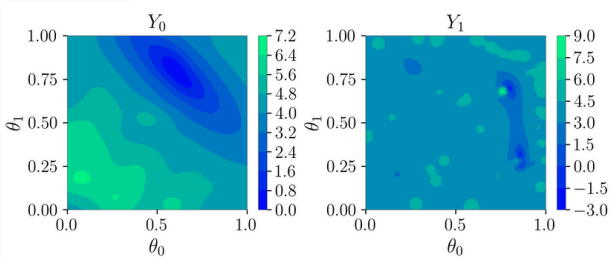

Figure 7: Example code for the application of the `generate_meshgrid` and `reshape_model_output` routines from the `hepaid.search.objective.plot` module, as well as the evaluation of the GP surrogate model `b-CASTOR.model` on the generated grid, illustrating the fitted model predictions based on the current dataset.

Section 4.2.) In short, CAS is an AS approach that uses a GP as a surrogate model. Its search policy aims to discover the entire satisfactory region in the parameter space where the objectives meet specific constraints, while also ensuring sample diversity within this set. Here, b-CASTOR

is a batched variant of the CAS algorithm, offering faster discovery of the satisfactory region and a denser filling of the set. MCMC-MH is an algorithm used to sample a complex target distribution. In the context of parameter scanning, the latter represents the satisfactory region and is sampled by defining a likelihood over the objectives. Finally, the MLScan method fits within the active search paradigm, employing a fully connected NN as the surrogate model. The input and output dimensions of this network are determined by the search space and the objectives, respectively. The MLScan policy involves performing rejection sampling over a likelihood, which is computed using the surrogate NN model.

The flexible structure of the `hep-aid` library allows users to switch between methods seamlessly by following the procedure displayed in Figure 5. This flexibility facilitates straightforward performance comparisons between different algorithms, as demonstrated in Figure 8. The figure presents the efficiency, measured by the ratio of satisfactory to total points $\mathcal{S}_r$, as a function of dataset size $\mathcal{D}_{size}$ for b-CASTOR, MLScan, and MCMC-MH on a two-dimensional, double-objective test function. The search for each algorithm is conducted over five independent runs for b-CASTOR and MLScan, and ten runs for MCMC-MH. Figure 8 highlights the superior sample efficiency of b-CASTOR compared to the other two methods; however, MLScan exhibits greater exploration within the parameter space, showing robustness for mode discovery. MCMC-MH demonstrates an $\mathcal{S}_r$ of zero in some runs due to suboptimal random starting points, highlighting a key weakness of this algorithm[2]. This comparison illustrates the importance of selecting a PS method based on user needs, as each algorithm has distinct strengths that can be used depending on the requirements of the specific use case.

2.2. Searching for BSM Physics

To perform parameter searches in BSM phenomenology studies, the HEP-stack software must be installed separately. Therefore `hep-aid` provides a command-line utility for installing the specific HEP software used in [16]. This setup aims to facilitate reproducible phenomenological scans by sharing BSM spectrum files. By running the command `hepaid install-HEP-stack-cli` in the directory containing the SPheno and Universal FeynRules Output (UFO) [37] files, `hep-aid` will install the required software with the appropriate versions[3]. In this manual, we demonstrate how to conduct a parameter scan within the $(B-L)$SSM to identify Higgs mass values that can explain an experimental signal around 95 GeV in terms of a BSM Higgs boson, alongside the 125 GeV SM-like Higgs state. This signal is supported by multiple experimental analyses searching for new Higgs bosons, including a di-photon ($\gamma\gamma$) excess observed by CMS [27], a di-tau ($\tau^+\tau^-$) excess also reported by CMS [38], and a $b\bar{b}$ excess detected by LEP [39]. The search space in this case is defined by

$$\mathcal{X} = \left\{ x \in \mathbb{R}^8 : x = \left(M_0, M_{1/2}, \tan\beta, A_0, \mu, \mu', B_\mu, B_{\mu'} \right) \right\}. \tag{1}$$

[2]This analysis can be replicated with the code examples available in github.com/mjadiaz/hepaid_tutorials.

[3]The companion repository provides the UFO and SPheno files for the $(B-L)$SSM model, enabling the replication of the example study.

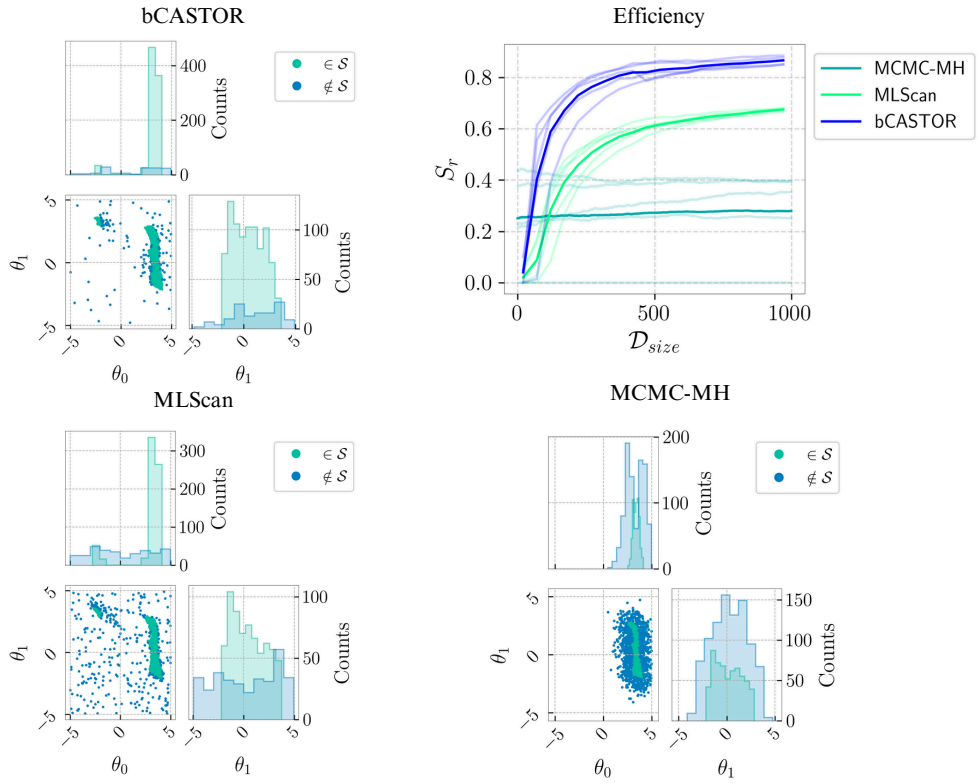

Figure 8: Performance comparison of b-CASTOR, MLScan, and MCMC-MH for a two-dimensional, double-objective test function. The mean is displayed for five runs for b-CASTOR as well as MLScan, and ten runs for MCMC-MH. The top-right plot displays the efficiency measured as the ratio of satisfactory to total points, $\mathcal{S}_r$, as a function of the dataset size, $\mathcal{D}_{\in\notin}$. The top-left plot shows the corner plot for b-CASTOR, while the bottom-left and bottom-right plots present the corner plots for MLScan and MCMC-MH, respectively.

For demonstration purposes, we simplify the objective space to include only the masses of the two lightest Higgs particles in the $(B-L)$SSM, i.e., the output space is defined by

$$\mathcal{Y} = \left\{ y \in \mathbb{R}^2 : y = (m_{h'}, m_{h^{\rm SM}}) \right\}. \tag{2}$$

Then the task for the search method is to find parameter space points that satisfy the constraints

$$\boldsymbol{\tau} = \begin{cases} m_{h^{\rm SM}} & = 125 \pm \delta m \text{ GeV}, \\ m_{h'} & = 95 \pm \delta m \text{ GeV}, \end{cases} \tag{3}$$

where δm is a user defined mass window. In this case we will consider $\delta m = 5$ GeV. Figure 9 illustrates the use of the b-CASTOR algorithm to conduct a search within the $(B-L)$SSM. The objective function must be defined: in this example, we use the `hep_stack_fn` utility function, which enables the use of a HEP-stack object for evaluating a single parameter space point. The result is returned as a nested Python dictionary containing, in this instance, the input and output files for SPheno. To generate the required output from the objective function, we use the `create_simple_dict` utility function, which queries the result dictionary as shown in Figure 2 but in an automated manner. This function takes the list of keys from the `masses_spheno_config.yml` configuration file and returns only the specified keys and their values for the input and output parameters defined in the configuration file. The configuration format is designed to be compatible with the `Objective` object constructor by including the `key_chain` and `output_parameters` elements.

The results can be visualised using the `CornerPlot` class of `hep-aid`, as shown in Figure 10, which displays the distribution of points within and outside the $\mathcal{S}$ region. In this case, the two classes of points are not clearly separable, as the distributions of satisfactory and non-satisfactory points overlap, as shown in the marginal histograms in the corner plots. However, the number of non-satisfactory points is relatively small, given that b-CASTOR demonstrates nearly 95% efficiency in the top-right plot. The search was conducted with an initial dataset of 400 points. The configuration file for b-CASTOR used in this example is shown in Figure 3.

3. HEP Module

The HEP module in `hep-aid`, located in `hepaid.hep`, provides infrastructure for managing and executing HEP software. As mentioned, for phenomenological analysis, a collection of HEP tools, referred to as a HEP-stack, is typically required to run sequentially for a single parameter space point of the BSM scenario under study. In fact, `hep-aid` aims to simplify the setup and execution of a HEP-stack by abstracting away the details of each tool initialisation and execution. It thus allows the user to perform HEP phenomenology analyses in a simple manner from a Python script or a Jupyter notebook.

Currently, part of the SARAH [21, 22] family of programs is implemented in `hep-aid`. SARAH is a Mathematica package for both Supersymmetric and non-Supersymmetric model building. SARAH allows the generation of SPheno and UFO model files for a specific BSM model. SPheno

```python
from hepaid.search.objective import Objective
from hepaid.hep.stack import hep_stack_fn
from hepaid.hep.utils import create_simple_dict
from hepaid.search.method import bCASTOR

# Configuration file paths
stack_config_path = 'configs/hep_stack_config.yml'
objective_config_path = 'configs/masses_spheno_config.yml'
method_config_path = 'configs/bcastor.yml'

# Define HEP objective function
def calculate_higgs_masses(x):
    results = hep_stack_fn(x,stack_config_path)
    return create_simple_dict(objective_config_path, results)

# Create the Objective
hep_objective = Objective(
    function=calculate_higgs_masses,
    function_config=objective_config_path
)

# Initialise method and run search
method = bCASTOR(
    objective=hep_objective,
    hyper_parameters=method_config_path
)
method.run()
```

```yaml
# masses_spheno_config.yml
input_space:
  m0:
    lower: 100.
    upper: 1000.
    distribution: "uniform"
    key_chain: ["LHS", "MINPAR", "entries", "1", "value"]
  m12:
    lower: 1000.
    upper: 4500.
    distribution: "uniform"
    key_chain: ["LHS", "MINPAR", "entries", "2", "value"]
  TanBeta:
    lower: 1.
    upper: 60.
    distribution: "uniform"
    key_chain: ["LHS", "MINPAR", "entries", "3", "value"]
  Azero:
    lower: 100.
    upper: 4000.
    distribution: "uniform"
    key_chain: ["LHS", "MINPAR", "entries", "5", "value"]
  MuInput:
    lower: 1000.
    upper: 8000.
    distribution: "uniform"
    key_chain: ["LHS", "EXTPAR", "entries", "11", "value"]
  MuPInput:
    lower: 1000.
    upper: 8000.
    distribution: "uniform"
    key_chain: ["LHS", "EXTPAR", "entries", "12", "value"]
  BMuInput:
    lower: 1e3
    upper: 1e8
    distribution: "uniform"
    key_chain: ["LHS", "EXTPAR", "entries", "13", "value"]
  BMuPInput:
    lower: 1e3
    upper: 1e8
    distribution: "uniform"
    key_chain: ["LHS", "EXTPAR", "entries", "14", "value"]
output_parameters:
  Mh(1): ["SLHA", "MASS", "entries", "25", "value"]
  Mh(2): ["SLHA", "MASS", "entries", "35", "value"]
objectives:
  double_constraint:
    Mh(1): [["gt", 90.4], ["lt", 100.4]]
    Mh(2): [["gt", 120.0], ["lt", 130.4]]
```

Figure 9: Code example demonstrating the definition of the objective function, `calculate_higgs_masses`, and the initialisation of the Objective constructor. Here, `create_simple_dict` resolves the key chains in the nested `result` dictionary, retaining only the objective keys and their corresponding values. The search configuration file for HEPstack is also provided, showing the search space configuration, the objectives and their constraints.

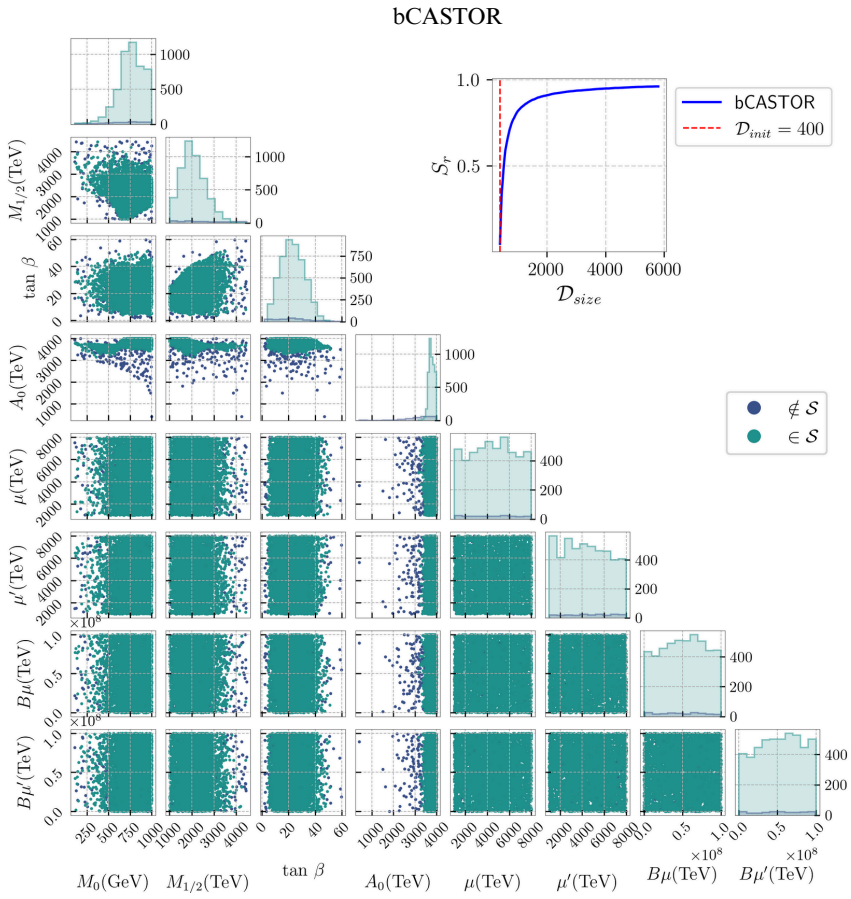

Figure 10: Corner plot visualisation of point distributions within and outside the $\mathcal{S}$ region. The top-right plot shows the b-CASTOR efficiency, with nearly 95% of points meeting satisfactory constraints. The red dashed line shows the initial number of points for the search.

is a model spectrum generator that, given an input SLHA file with a particular parameter space point, calculates loop-corrected masses, couplings, BRs, decay widths, and even flavour observables [28, 29], and write the output in SLHA format. Using the UFO files generated by SARAH, MG [32] can utilize the SLHA output file from SPheno, known internally as the `param_card.dat`, to calculate process cross-sections and generate events. Furthermore, access to the HB [30] and HS [31] programs are also implemented. HB and HS are used for experimental testing of model parameter space points. HB compares existing exclusion limits from Higgs searches with the model predictions of the Higgs sector, generating an upper limit to a corresponding signal cross-section prediction. Therefore, with HB, we can check whether a given model, whose spectrum is evaluated at a particular configuration, is excluded at the 95% Confidence Level (C.L.) by existing Higgs boson searches. In contrast, HS tests the model prediction of a Higgs sector with an arbitrary number of Higgs bosons against the properties of the observed state as measured by the LHC experiments ATLAS [40] and CMS [41] in 2012 (and thereafter).

3.1. HEP-stack

Every HEP-stack is implemented as a Python class and stored in the `HEPSTACK` dictionary. Each HEP-stack needs to be initialised with a pre-defined configuration file. This configuration file contains information about the input parameters that will undergo a parameter scan and the necessary directories for the HEP tools used in the scan. For instance, the HEP-stack composed by the SPheno-HB-HS-MG sequence can be accessed as shown in Figure 2. For this HEP-stack, a configuration file is shown in Figure 11. With the HEP-stack initialised, we can define a Benchmark Point (BP) within the search space specified in Eq. (1). We then execute the HEP-stack using the `.sample(x)` method, as shown in Figure 2. The result includes all the input and output information from the four HEP tools used in the HEP-stack, presented as a Python dictionary. To obtain the objective parameters within $\mathcal{Y}$, defined in Eq. (2), we can simply query the result dictionary using their corresponding chain of keys, as illustrated in Figure 2.

3.2. HEP Tools

The `hep-aid` package implements the `hepaid.hep.tools` module, which provides classes and methods for managing high-energy physics tools such as SPheno, MG, HB, and HS. This module offers a simplified interface for running computations with minimal boilerplate code. It also handles the automatic management of input and output files, including the creation of necessary directories, and parses tool-specific output formats into structured Python objects for further analysis.

The classes defined within this module corresponds to a specific tool. The structure of each tool class is defined by the `BaseTool` class which includes `run()` and `results()` methods. As an example, Figure 12 demonstrates how SPheno can be initialised and run in a straightforward manner. HB and HS are also implemented in `hep-aid`, as mentioned[4]. The HB and HS tools can

[4]The two programs have recently been incorporated into the HiggsTools distribution [42], which would also be easily embedded in `hep-aid`.

```yaml
# hep_stack_config.yml
hep_stack:
    name: 'SPhenoHBHSMG5'
    scan_dir: 'path/to/output'
    final_dataset: 'datasets'
    delete_on_exit: True
spheno:
    model: 'BLSSM'
    reference_slha: 'path/to/reference_slha'
    directory: 'path/to/SPheno'
higgsbounds:
    neutral_higgs: 6
    charged_higgs: 1
    directory: 'path/to/hb/build'
higgssignals:
    neutral_higgs: 6
    charged_higgs: 1
    directory: 'path/to/hs/build'
madgraph:
    directory: 'path/to/MG5_aMC'
    scripts:
        gghaa: 'path/to/mg5_script.txt'
```

```yaml
model:
  name: 'BLSSM'
  input:
    m0:
        block_index: 1
        block_name: 'MINPAR'
    m12:
        block_index: 2
        block_name: 'MINPAR'
    TanBeta:
        block_index: 3
        block_name: 'MINPAR'
    Azero:
        block_index: 5
        block_name: 'MINPAR'
    MuInput:
        block_index: 11
        block_name: 'EXTPAR'
    MuPInput:
        block_index: 12
        block_name: 'EXTPAR'
    BMuInput:
        block_index: 13
        block_name: 'EXTPAR'
    BMuPInput:
        block_index: 14
        block_name: 'EXTPAR'
```

Figure 11: Configuration file for the HEP-stack named `SPhenoHBHSMG5`. The file includes all relevant information for the HEP tools, and on the right, details about the parameters needed to conduct a search using SLHA block information.

```python
# Import the SPheno tool from the hepaid.hep.tools module
from hepaid.hep.tools import SPheno

# Define paths and model name
heptool_dir = "path/to/SPheno-4.0.4"
output_dir = "path/to/output/files"
model_name = "MODEL"

# Initialise the SPheno tool
spheno = SPheno(heptool_dir, output_dir, model_name)

# Run SPheno with a specified input SLHA file
spheno.run(input_slha_file)

# Retrieve the output SLHA file
output_slha_file = spheno.results
```

```python
# Import the HiggsSignals tool from hepaid.hep.tools
from hepaid.hep.tools import HiggsSignals

# Initialise the HiggsSignals
hs = HiggsSignals(
    heptool_dir='path/to/HS',
    output_dir='path/to/output/files',
    neutral_higgs=6,
    charged_higgs=1
)

# Run HiggsSignals and retrieve results
hs.run()
hs_result = hs.results
```

Figure 12: Using SPheno and `HiggsSignals`: The parameter `heptool_dir` corresponds to the path to SPheno's directory, `output_dir` corresponds to the directory where the output will be stored, and `model_name` is the name of the previously compiled model. The process is analogous for `HiggsSignals`, here `heptool_dir` refers to the directory containing the build.

be used as shown in Figure 12, following the same structure as SPheno. Lastly, the MG HEP tool is implemented to run in script mode. In this case, the `run(mg5_script_path)` method takes the path to the script as input. The `results("path/to/process/output")` method finally takes the path of the MG output generated by the command `output example_process`.

3.3. Reading and Writing SLHA Files

The basis for input/ouput in the HEP module in `hep-aid` is the SLHA format [43, 36]. Libraries for manipulating SLHA files exist in literature already. The Pyslha [44] library is a well-established and widely used tool for SLHA file manipulation. Pyslha includes functions for calculating spectrum properties, making it a comprehensive solution for handling SLHA files in various contexts. Additionally, it offers capabilities for generating mass spectrum plots in various formats through the `slhaplot` script. Another notable library is [45], which efficiently manages both individual SLHA files and directories containing multiple files, with a particular emphasis on reading speed optimisation. This library allows for efficient data filtering through Targeted Data Extraction when specific blocks and entries are known. Furthermore, xSLHA (a Python parser for files written in the SLHA format) can process large files where spectra are separated by specific keywords.

However, in `hep-aid` implements its own SLHA, `module.lass`, which treats SLHA files like Python dictionaries, allowing users to access blocks and entries with familiar syntax. This simplifies data extraction and manipulation without the need for a specialised Application Programming Interface (API). The `SLHA` class retains file comments, preserving valuable metadata, explanations, and references essential for reproducibility and context. It supports the `DECAY1L` and `HiggsTools` blocks, with a focus on SPheno input files, which include a unique `"on/off"` configurations. An example on how the `SLHA` module is used is displayed in Figure 13. The main objective of creating such a module is to support the construction of large spectrum datasets, by providing functionalities for exporting SLHA files as nested Python dictionaries, facilitating integration with JavaScript Object Notation (JSON) and enabling the storage of multiple SLHA files within a single zipped JSON file.

Under the hood, every line representing data in a block is structured with a `BlockLineSLHA` class which stores the numerical entries, values, comments, and line category. Note that entries are defined as everything apart from the value of each line. The blocks are represented with a `BlockSLHA` class, which stores each line, the block header and additional information in this header. Lastly, the `SLHA` extracts automatically the information of an SLHA file and organises it with `BlockSLHA` and `BlockLineSLHA`. This modular design lets users customise the library behaviour or add support for new SLHA blocks, as needed. The `hepaid.hep.read` module also includes utilities for reading results from HB, HS, and single MG process generation, to extract the cross-section and the number of events. These routines are used in the `hepaid.hep.tools` module, where each HEP tool includes its results by calling the corresponding `hepaid.hep.read` routine. As an example, Figure 12 shows how SPheno returns the resulting SLHA file by internally calling the `SLHA` class in `hepaid.hep.read`.

```python
# Importing the SLHA reader module from the hep-aid package
from hepaid.hep.read import SLHA

# Load the SLHA file containing particle physics parameters
slha_file = 'LesHouches.in.BLSSM'
slha_data = SLHA(slha_file)

# Print a summary of the SLHA data
print(slha_data)
# Output: SLHA object containing 8 parameter blocks

# Display the available parameter blocks in the SLHA file
parameter_blocks = slha_data.keys()
print(parameter_blocks)
# Output: ['MODSEL', 'SMINPUTS', 'MINPAR', 'EXTPAR',
#          'SPHENOINPUT', 'DECAYOPTIONS', 'YXIN', 'YVIN']

# Access and display the 'MINPAR' block, which contains input
parameters
minpar_block = slha_data['MINPAR']
print(minpar_block)
# Output:
# Block MINPAR   # INPUT PARAMETERS
# 1   6.28483500e+02   # M0 (Universal scalar mass)
# 2   2.45986100e+03   # M12 (Universal gaugino mass)
# 3   4.40308500e+01   # TANBETA (Ratio of Higgs vacuum
expectation values)

# Other blocks can be accessed similarly by their names
```

Figure 13: Using the SLHA class for the default input SLHA file of the $(B-L)$SSM. Blocks and entries exhibit dictionary-like behaviour and can be exported as a Python dictionary for efficient storage and communication with other modules.

4. Search Module

In `hep-aid`, a Parameter Scan Algorithm is implemented as an AS process. AS is an iterative search methodology that utilises existing evaluations of an *objective function* $\mathcal{H}(\mathbf{x})$ to identify suitable points to sample within a required category. In this context, the required category corresponds to parameter values $\mathbf{x}$ for which the corresponding observables $\mathbf{y} = \mathcal{H}(\mathbf{x})$ satisfy a specified set of constraints $\boldsymbol{\tau}$. Therefore, an AS method consists of three main components: the *objective function*, the *search policy*, and the *surrogate model*. At each iteration t, a dataset $\mathcal{D}_t$ is updated with new evaluations from the *objective function*. The *surrogate model* is then made to fit this dataset to refine its approximations and guide the *search policy* in proposing new candidate points within the satisfactory region. Therefore, technically `hep-aid` comprises three core components. In this section, we describe the main components, modules and additional utility functions that `hep-aid` implements to utilise, implement and develop parameter scan methods.

4.1. *Objective*

The interface between an objective function and a parameter scan method is managed by the `Objective` class. This class handles an external multi-objective function, where the objective function is treated as a black-box. The objective function is defined as a mapping of an n-dimensional input vector $\mathbf{x}$ to an m-dimensional output vector $\mathbf{y}$, with m representing the number of objectives. The objective function is expressed as $\mathbf{y} = \mathcal{H}(\mathbf{x})$. Constraints are defined by the vector $\boldsymbol{\tau}$, and the satisfactory region in the parameter space, denoted as $\mathcal{S}$, is the set of configurations that meet these constraints:

$$\mathcal{S} = \{\mathbf{x} \mid \mathbf{y} = \mathcal{H}(\mathbf{x}) \wedge y_i \succeq \tau_i, i = 1, \ldots, m\}\,. \tag{4}$$

By design, the multi-objective function must be defined as a Python function. This function takes the parameter configuration input $\mathbf{x}$ as an array and outputs a dictionary. The dictionary includes keys and values that correspond to the names of the input dimensions and the objectives, along with their respective values. The necessary parameters are defined in the configuration file to initialise the `Objective` class. This is illustrated in Figure 14 for a two-dimensional, single-objective Egg Box model function.

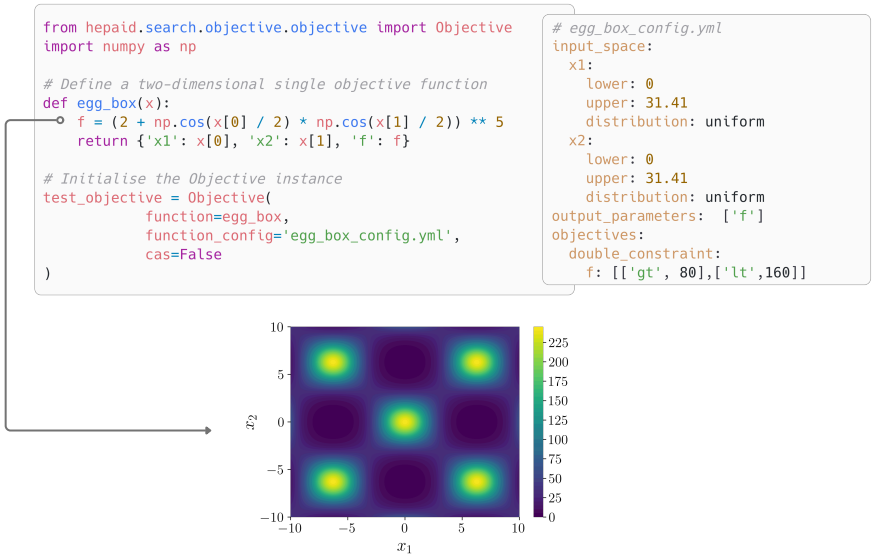

Figure 14: Definition of the Egg Box model as a function, which serves as the objective function to initialise the Objective class. The function accepts an array of input parameters and outputs a Python dictionary containing all the parameters, both input and output. These parameters and their ranges are defined in a configuration file, along with the objective dimensions and their constraints. Additionally, the contour plot of the `egg_box` function is displayed at the bottom.

The `Objective` class provides a key feature in `hep-aid`: the ability to integrate any black-box function. This allows users to input any function, including external software, as long as the function accepts input $\mathbf{x}$ as an array and outputs a vector $\mathbf{y}$. This capability also serves as the mechanism by which the HEP module is integrated into the `Objective` class.

The `Objective` object stores the function in its `Objective.function` attribute and reads the configuration file to create a search space utility using `scikit-optimize` [46]. This search space facilitates the normalisation of the internal dataset, stored in `Objective.dataset`, which is essential for fitting *surrogate models*. The dataset is dynamically updated whenever the function is called through the `Objective.sample(x)` method, adding the new set $\mathbf{x}, \mathbf{y}$ to the internal dataset. The historical `Objective.dataset` is formatted as a Python dictionary. However, the `Objective.x`, `Objective.X`, and `Objective.Y` attributes hold the input data in the original space, the input data in the normalised space, and the output data in the original space, respectively. The method `Objective.as_dataframe(satisfactory=True)` return the dataset as a Pandas DataFrame [24]. The `Objective.satisfactory` attribute generates a boolean matrix, S_{ij}, with the same dimensions as `Objective.Y`. This matrix identifies which elements in `Objective.Y` satisfy the constraints defined in the configuration file. Consequently, taking the vertical product, $S_j = \prod_i S_{ij}$, of `Objective.satisfactory` produces a boolean array. This array labels as `True` the configurations in `Objective.dataset` that meet the specified constraints.

Lastly, the method `Objective.save(path)` saves the internal dataset as a compressed JSON file. Then, the method `Objective.load(path, process=True)` loads a previously saved JSON file, updating the internal dataset and all relevant information if the `process` argument is set to `True`. This allows the continuation of the search process with a parameter scan algorithm or for plotting purposes.

4.2. Parameter Space Sampling

The `hepaid.search.methods` module includes all the implemented PS algorithms. Currently, these methods are MCMC-MH [34, 35], b-CASTOR [16], CAS [33], and MLScan [7]. Due to the varying requirements of each parameter scan algorithm, `hep-aid` implements a base class called `Method`. This class is designed to be inherited by PS algorithms and encapsulates the essential functionalities needed for managing hyperparameters, tracking standard and custom metrics, handling checkpoints, and managing file directories. Every method that inherits from this base class will receive an `Objective` instance and a hyperparameter file as arguments. The file can be provided by the user as a string path or a `DictConfig` [47]. If neither is provided, the default configuration file will be used. The `Method` class also offers utility methods to save and load the state of the search process, including the state of the objective function, current metrics being tracked, and the hyperparameter configuration file. Each instance of the `Method` class initialises a `Metrics` object responsible for tracking standard metrics related to the search process. These metrics include the total number of points evaluated, the number of valid points, the number of satisfactory points, the success rate, and the current iteration counter. Additionally, users can extend the metrics functionality by defining custom metrics that can be updated during the search process.

There is a defined general structure for the hyperparameter configuration file, as illustrated in Figure 15. Each instance of the `Method` class will read the following hyperparameters. The `run_name` refers to the path to the directory where checkpoints and configuration files will be saved. The `parallel` parameter is defined as a boolean parameter to enable the parallel eval-

uation of specific processes during the search. This works in conjunction with the `n_workers` parameter, which dictates the number of parallel processes based on available resources. For example, this setup is used for evaluating the objective function of the batch $\boldsymbol{X}^*$ suggested by the policy, by using the function `batch_evaluation` located in `hepaid.search.objective.utils`. Except for MCMC-MH, search methods require an initial dataset, which is configured via the `initial_dataset` hyperparameter. The `initial_dataset.n_points` specifies the number of points to be generated, while `initial_dataset.generate` is a boolean indicator for the dataset generation. This is useful, for instance, when the user wishes to continue from previous checkpoints where the complete initial dataset is not needed. The initial dataset is generated by the function `generate_initial_dataset`. The `total_iteration` parameter sets the total number of iterations for the search loop. Additional parameters will depend on the specific algorithm and are explained below.

Hyperparameter	Description
`run_name`	Directory path for saving checkpoints and configuration files.
`parallel`	Boolean parameter to enable parallel evaluation of processes.
`n_workers`	Number of parallel processes based on available resources.
`initial_dataset`	Configures initial dataset required by most search methods.
`initial_dataset.n_points`	Number of points to generate in the initial dataset.
`initial_dataset.generate`	Specifies whether the initial dataset should be generated.
`total_iteration`	Sets the total number of iterations for the search loop.

Table 1: Hyperparameters for a PS method implemented by inheriting the `Method` class.

4.2.1. AS Methods

As mentioned, `hep-aid` implements two AS methods, CAS [33] and the b-CASTOR [16] algorithm, which utilises GPs as surrogate models to approximate each objective function. Herein, GPs provide a probabilistic model where each point in the search space has an estimated mean, the predicted objective value, and a standard deviation, uncertainty of the prediction. The Expected Coverage Improvement (ECI) function is then applied to guide the selection of points to evaluate. ECI operates by defining a hypersphere around each point in the current dataset and measuring the volume covered by these hyperspheres across the search space by using the predictions from the GP surrogates. The main goal is to find a location $\mathbf{x}^*$ in the search space for a new hypersphere that will maximally increase the coverage of the $\mathcal{S}$ region. At each iteration, a global optimisation technique is then applied to the ECI function to determine the best new point, $\mathbf{x}^*$. The resulting dataset from this search process will consist of a diverse set of points that effectively cover the $\mathcal{S}$ region, with diversity determined by the radius of each hypersphere, denoted by r. This radius r is a fixed hyperparameter within the method. However, the `hep-aid` implementation of CAS includes a linear decay on the radius, referred to in the library as the resolution parameter.

Therefore, the additional hyperparameters relevant for CAS are located in `eci.num_samples`, which specifies the number of sample points for which an individual hypersphere is approximated, and the `resolution` block. In this context, `resolution.constant_resolution` allows

for switching between a fixed radius or a linear decay in the radius. The `resolution.value` sets the value, typically ranging between 0 and 1, as the search space is normalised. The parameters `resolution.initial` and `resolution.final` are used when the linear decay mode is enabled. Additionally, `resolution.r_decay_steps` specifies the number of iteration steps during which the decay takes place.

Hyperparameter	Description
`eci.num_samples`	Number of sample points for approximating each hypersphere.
`resolution.constant_resolution`	Switches between fixed radius and linear radius decay.
`resolution.value`	Sets radius value, typically between 0 and 1 (normalised space).
`resolution.initial`	Initial radius value for linear decay.
`resolution.final`	Final radius value for linear decay.
`resolution.r_decay_steps`	Number of steps for radius decay.

Table 2: Hyperparameters used for CAS in `hep-aid`.

CAS operates sequentially, selecting only a single point per iteration to add to the dataset. This approach leads to relatively slow convergence, as the search progresses incrementally, expanding the $\mathcal{S}$ region step-by-step to nearby areas. A batched version of this method, called b-CASTOR, was developed in [16]. It uses the same foundational components as CAS, employing GPs as surrogates and the ECI function to guide the search. However, different optimisation and sampling methods are applied to the ECI function to generate a batch of samples for evaluation in the objective function during each iteration. Specifically, b-CASTOR leverages the Tree-structured Parzen Estimator (TPE) to optimise the ECI. Furthermore, a Stochastic Prioritisation (SP) sampling technique is employed to select the batch $\boldsymbol{X}^*$ from the TPE optimisation history. This technique alternates between an exploitation phase – sampling from high-potential regions as predicted by the *surrogate model* – and an exploration phase, which uniformly samples the parameter space. This batched approach accelerates the search process by enabling more diverse and densely populated sample collections in each iteration, thereby improving convergence speed and coverage of high ECI regions.

The additional hyperparameters required by b-CASTOR include those from CAS related to the resolution parameter, as well as the following ones. For the batch sampling settings, we have `batch_sampling.tpe_trials`, which indicates the number of trials for optimising the ECI function using the TPE algorithm. Next, `batch_sampling.rank_samples` specifies the number of samples obtained from the historical data of the TPE optimisation using the stochastic prioritisation technique. This process determines the size of the batch $\boldsymbol{X}^*$. The `batch_sampling.alpha` parameter controls the degree of prioritisation in rank-based sampling; a higher value prioritises sample efficiency. The interaction between the `alpha` and `rank_samples` hyper-parameters determines the balance between exploration and exploitation in the algorithm. Empirically, setting `alpha=2` provides a good balance of efficiency and diversity within the satisfactory region. A higher number of `rank_samples`, i.e., a larger batch size, allows for more chances for exploration.

Figure 15 shows the complete configuration file for the b-CASTOR algorithm, which can also be used for the CAS algorithm. The file is displayed in `YAML` format.

```yaml
# configs/bcastor.yml
run_name: "blssm/run"
parallel: True
initial_dataset:
  generate: True,
  n_workers: 30
  n_points: 400
checkpoint:
  name: "checkpoint"
  n_step_save: 5
total_iterations: 250
```

```yaml
resolution:
  value: 1e-6
  constant_resolution: False
  r_decay_steps: 100
  initial: 1e-2
  final: 1e-6
batch_sampling:
  tpe_trials: 2500
  rank_samples: 30
  n_evaluation_workers: 30
  alpha: 2
eci:
  num_samples: 500
```

Figure 15: Configuration file for the b-CASTOR algorithm. The left side displays the general structure of a configuration file for a parameter scan implemented in `hep-aid`. The right side displays the parameters relevant to the CAS algorithm in the resolution block, where additional batch sampling blocks are required for the b-CASTOR algorithm.

Hyperparameter	Description
`batch_sampling.tpe_trials`	Number of TPE trials for optimising the ECI function.
`batch_sampling.rank_samples`	Number of samples obtained through rank based sampling (batch $\boldsymbol{X}^*$ size).
`batch_sampling.alpha`	Prioritisation degree in rank-based sampling.

Table 3: Additional hyperparameters (with respect to the CAS ones) relevant to the b-CASTOR algorithm.

The technical implementation for both the CAS and b-CASTOR algorithms is as follows. We used the ECI acquisition function implementation provided by the BoTorch library [48]. Additionally, we integrated GP models, available in `hepaid.search.models`, from the GPytorch library [49], a repository for scalable GP inference built on PyTorch [25]. We further utilise the TPE implementation available in Optuna [50], an open-source hyperparameter optimisation framework.

4.2.2. MCMC-MH

The MCMC-MH algorithm is a method for sampling from a complex probability distributions [2]. The use of this sampling method and its advanced variants are well-established in HEP research [3]. The algorithm generates samples that approximate the target distribution by constructing a Markov chain, which stationary distribution is the desired target distribution. The MCMC-MH algorithm starts by selecting an initial state x_0 from the state space. During each iteration t, a candidate state x' is generated from a proposal distribution $q(x' \mid x_t)$. The acceptance probability is then computed as

$$\alpha(x', x_t) = \min\left(1, \frac{\pi(x')q(x_t \mid x')}{\pi(x_t)q(x' \mid x_t)}\right), \tag{5}$$

where $\pi(x)$ is the target distribution and $q(x' \mid x_t)$ is the proposal distribution. A common choice for the proposal distribution in the MCMC-MH algorithm is a Gaussian distribution centered on the current sample which simplifies the acceptance probability. This proposal is implemented in

hep-aid. The proposal variance is crucial, as a large one can lead to proposals being frequently rejected, which slows down the algorithm and decreases sample efficiency. Conversely, a low variance can cause the chain to explore the target distribution inefficiently, potentially preventing the chain from converging to a stationary distribution. (The variance is also known as the step size or scale parameter.) A uniform random number u is generated from the interval $[0, 1]$. If $u \leq \alpha(x', x_t)$, the candidate is accepted by setting $x_{t+1} = x'$, otherwise, the candidate is rejected, and one sets $x_{t+1} = x_t$. This iteration process is repeated until a sufficient number of samples have been generated.

The hyperparameters defined in the configuration file include `burn_in`, which defines the number of initial iterations to discard, allowing the chain to converge to the target distribution and helping to eliminate dependence on the starting value. The `initial_scale` parameter sets the initial step size for generating proposal states from the current state. The `adapt_frequency` parameter defines how often to adjust the scale of the proposal distribution during the sampling process, based on the observed acceptance rate. Lastly, the `target_acceptance_rate` specifies the desired acceptance rate for proposal moves, guiding the adaptation of the proposal distribution scale and influencing the amount of exploration of the parameter space while ensuring efficient convergence to the target distribution.

Hyperparameter	Description
`burn_in`	Number of initial iterations for burn-in.
`initial_scale`	Initial step size for generating proposal states.
`adapt_frequency`	Frequency of adjusting the scale on burn-in.
`target_acceptance_rate`	Desired acceptance rate for proposals.

Table 4: Hyperparameters relevant for the MCMC-MH algorithm.

For a PS, the target distribution is the likelihood constructed from the constraints in the objectives. In hep-aid, this likelihood can be provided as a function. If not, hep-aid will use a default likelihood constructed with sigmoid windows for each objective and its constraints [16],

$$\mathcal{L}(y_i) = \begin{cases} \sigma(y_i, a) & y_i > a \\ 1 - \sigma(y_i, a) & y_i < a \\ \sigma(y_i, a) - \sigma(y_i, b) & a < y_i < b \end{cases} \tag{6}$$

where y_i is an objective and a, b constraints. Here, σ is the sigmoid function and is defined as

$$\sigma(y, a) = \frac{1}{1 + e^{-(y-a)/\epsilon}}, \tag{7}$$

where ϵ is a parameter that controls the smoothness of the sigmoid function and a shifts the centre of the sigmoid. Then, the total likelihood for a point $(\mathbf{x}, \mathbf{y})$ used for MCMC-MH sampling is defined as

$$\mathcal{L}(\mathbf{y}) = \prod_i \mathcal{L}(y_i). \tag{8}$$

4.2.3. MLScan

The Machine Learning Scan method [7], termed MLScan in `hep-aid` was designed for efficient sampling and exploration of parameter spaces using a Machine Learning surrogate model for the observables, fitting in the description of AS. It uses an MLP Neural Network as a *surrogate model*. The search policy in this case is performing rejection sampling over the likelihood, but this time the likelihood is evaluated using the predictions from the MLP model.

The implementation in `hep-aid` begins with an initial dataset, similar to previous methods. In each iteration, the MLP is trained using the currently available dataset, retaining the model parameters from previous iterations to perform incremental learning. The rejection sampling technique is used to generate the batch $\boldsymbol{X}^*$, with the size of this batch controlled by the hyper-parameter `num_samples`. The scaling of the acceptance probability for the rejection sampling method is controlled by the parameter `m_factor`. By adjusting this factor, users can manage the balance between sample quality and the diversity of generated samples, thus affecting both the efficiency and effectiveness of the sampling process. The MLScan method also adds extra random samples to encourage exploration, with the number of samples controlled by the hyperparameter `extra_random_samples`.

```python
import torch
from hepaid.search.objective.test import init_egg_box_fn
from hepaid.search.method.mlscan import MLScan
from hepaid.search.method.eci import smooth_box_mask

# Define likelihood
def likelihood(x):
    if not isinstance(x, torch.Tensor):
        x = torch.tensor(x)
    f = x[:,1]
    lh = smooth_box_mask(f, 80., 160., 1e-3 )
    return  lh

# Initialise Eggs model
test_objective = init_egg_box_fn()

# Define the configuration file
hp = 'datasets/mlscan/run/eggs/hprms.yaml'

# Initialise method and run
mlscan= MLScan(
    objective=test_objective,
    likelihood=likelihood,
    hyper_parameters=hp
    )
mlscan.run()
```

```yaml
# hprms.yaml
num_samples: 100
checkpoint:
  n_step_save: 500
  name: chckpnt
run_name: mlscan/run
total_iterations: 100
parallel: false
n_workers: 4
initial_dataset:
  generate: True,
  n_points: 200
model:
  layer_sizes: [2, 60,60,60,1]
  dropout_prob: 0.0
  learning_rate: 0.01
  num_epochs: 500
  step_size: 5
  gamma: 0.999
  threshold: 1e-8
```

Figure 16: A code example demonstrating the utilisation of the MLScan method, a neural network-based sampling method introduced in [7], is provided. Users must define a likelihood function. Since the Egg Box model is already implemented in `hep-aid`, it can be readily used. Initialising the MLScan method requires a configuration file, as shown in the code snippet on the right.

For MLScan an additional hyper-parameter block, named `model_hyperparameters`, is required for configuring the MLP's architecture and training. The `layer_sizes` parameter receives a list specifying an input layer, an arbitrary number of hidden layers with their respective number of neurons, and the output layer neurons. The `dropout_prob` indicates the level of dropout regulari-

sation in each layer. The `learning_rate` controls the magnitude of weight updates during training, while `num_epochs` defines the number of complete passes through the training dataset. Additionally, the `step_size` determines how frequently the learning rate is adjusted, and `gamma` specifies the decay rate for the learning rate, allowing it to decrease gradually. Finally, the `threshold` acts as a stopping criterion for training when the loss function fall below this value.

Finally, the `MLScan` class takes an argument for the likelihood function, which the user must define, as shown in Figure 16. In this example, `hep-aid` can be used to replicate the results from [7] using the Egg Box model test function, which is currently implemented in `hep-aid`. The results of running the MLScan search on the Egg Box model are displayed in Figure 17.

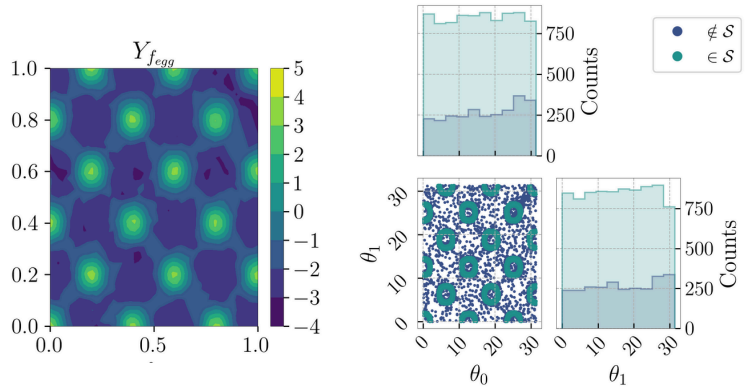

Figure 17: On the left, the predictions across the entire parameter space of the surrogate neural network model in the final iteration of the search from Figure 16 are shown. On the right, the points identified by the MLScan method are displayed using the corner plot functionalities of `hep-aid`.

Hyperparameter	Description
`num_samples`	Batch size for rejection sampling.
`m_factor`	Scales acceptance probability in rejection sampling.
`extra_random_samples`	Number of additional random samples for exploration.
`model_hyperparameters`	Configures MLP architecture and training.
`layer_sizes`	Defines neuron counts for MLP layers.
`dropout_prob`	Dropout rate for regularisation.
`learning_rate`	Step size for weight updates.
`num_epochs`	Total training dataset passes.
`step_size`	Interval for adjusting learning rate.
`gamma`	Learning rate decay factor.
`threshold`	Loss threshold for stopping criterion.

Table 5: Hyperparameters relevant for the MLScan algorithm.

5. Conclusions

This paper introduced `hep-aid`, a new Python library designed to facilitate PS algorithms for BSM phenomenology. The library provides a modular framework that integrates HEP software,

simplifying the implementation and development of PS algorithms while offering essential functionalities for phenomenological analysis. Originally the library was created to give researchers simple access to use the b-CASTOR and CAS algorithms for parameter scans. However, its development lead to a modular structure allowing the implementation of further PS algorithms existing in the literature such as MLScan.

We demonstrated the utility of the library by performing multi-objective searches on test functions and comparing the performance of different PS algorithms. In this connection, b-CASTOR exhibited superior sample efficiency while achieving a comprehensive characterisation of the satisfactory parameter region. The experiments with the MLScan algorithm instead highlighted its robust exploration capabilities, due to the stochastic nature of its policy. However, for BSM phenomenological studies, a NN classifier needs to be implemented to enhance the sample efficiency of this algorithm. (All ML-based such approaches were also demonstrated to be superior to the more standard MCMC-HS one.) Additionally, we illustrated the application of the library in a real BSM case study, fitting the masses of the lightest Higgs particles in the $(B-L)$SSM to explain new physics signals in $\gamma\gamma$, $\tau^+\tau^-$, and/or $b\bar{b}$ final states [16].

Looking ahead, future work will focus on improving the `hep-aid` design and modularity to encourage the development of new PS algorithms, possibly starting from combinations of existing components: e.g., using the rejection sampling component from MLScan to sample from the ECI policy function, which is used by CAS and b-CASTOR, could enhance exploration capabilities. This is, in fact, a direction currently under investigation. Another promising development is integrating `hep-aid` with other existing PS libraries, particularly those that interface with a broad range of HEP software. Ultimately, the primary goal of `hep-aid` is to serve as a versatile testing ground for developing and benchmarking PS algorithms, with a focus on ML-driven approaches.

Acknowledgments

The work of SM is supported in part through the NExT Institute and the STFC Consolidated Grant No. ST/X000583/1. MAD and his work were supported by ANID BECAS DE DOCTORADO EN EL EXTRANJERO, BECAS CHILE 2020, 72210042.

References

[1] O. Brein, Adaptive scanning—a proposal how to scan theoretical predictions over a multi-dimensional parameter space efficiently, Computer Physics Communications 170 (1) (2005) 42–48. doi:10.1016/j.cpc.2005.03.104.
URL http://dx.doi.org/10.1016/j.cpc.2005.03.104

[2] D. W. Hogg, D. Foreman-Mackey, Data analysis recipes: Using markov chain monte carlo*, The Astrophysical Journal Supplement Series 236 (1) (2018) 11. doi:10.3847/1538-4365/aab76e.
URL http://dx.doi.org/10.3847/1538-4365/aab76e

[3] J. Albert, C. Balazs, A. Fowlie, W. Handley, N. Hunt-Smith, R. R. de Austri, M. White, A comparison of bayesian sampling algorithms for high-dimensional particle physics and cosmology applications (2024). arXiv:2409.18464.

[4] J. S. Speagle, A Conceptual Introduction to Markov Chain Monte Carlo Methods (9 2019). arXiv:1909.12313.

[5] M. Feickert, B. Nachman, A living review of machine learning for particle physics (2021). arXiv:2102.02770.
[6] R. Baruah, S. Mondal, S. K. Patra, S. Roy, Probing intractable beyond-standard-model parameter spaces armed with machine learning, The European Physical Journal Special Topics (Jul. 2024). doi:10.1140/epjs/s11734-024-01236-w.
URL http://dx.doi.org/10.1140/epjs/s11734-024-01236-w
[7] J. Ren, L. Wu, J. M. Yang, J. Zhao, Exploring supersymmetry with machine learning, Nuclear Physics B 943 (2019) 114613. doi:https://doi.org/10.1016/j.nuclphysb.2019.114613.
URL https://www.sciencedirect.com/science/article/pii/S0550321319300938
[8] A. Hammad, M. Park, R. Ramos, P. Saha, Exploration of parameter spaces assisted by machine learning, Computer Physics Communications 293 (2023) 108902. doi:10.1016/j.cpc.2023.108902.
URL http://dx.doi.org/10.1016/j.cpc.2023.108902
[9] M. Binjonaid, Multi-label classification of parameter constraints in bsm extensions using deep learning (2024). arXiv:2409.05453.
[10] J. Hollingsworth, M. Ratz, P. Tanedo, D. Whiteson, Efficient sampling of constrained high-dimensional theoretical spaces with machine learning, The European Physical Journal C 81 (12) (Dec. 2021). doi:10.1140/epjc/s10052-021-09941-9.
URL http://dx.doi.org/10.1140/epjc/s10052-021-09941-9
[11] M. D. Goodsell, A. Joury, Active learning bsm parameter spaces, The European Physical Journal C 83 (4) (Apr. 2023). doi:10.1140/epjc/s10052-023-11368-3.
URL http://dx.doi.org/10.1140/epjc/s10052-023-11368-3
[12] M. D. Goodsell, A. Joury, Bsmart: Simple and fast parameter space scans, Vol. 297, 2024, p. 109057. doi:https://doi.org/10.1016/j.cpc.2023.109057.
URL https://www.sciencedirect.com/science/article/pii/S0010465523004022
[13] F. Abreu de Souza, M. Crispim Romão, N. F. Castro, M. Nikjoo, W. Porod, Exploring parameter spaces with artificial intelligence and machine learning black-box optimization algorithms, Phys. Rev. D 107 (2023) 035004. doi:10.1103/PhysRevD.107.035004.
URL https://link.aps.org/doi/10.1103/PhysRevD.107.035004
[14] J. C. Romão, M. C. Romão, Combining evolutionary strategies and novelty detection to go beyond the alignment limit of the z_3 3hdm (2024). arXiv:2402.07661.
[15] C. Balázs, M. van Beekveld, S. Caron, B. M. Dillon, B. Farmer, A. Fowlie, E. C. Garrido-Merchán, W. Handley, L. Hendriks, G. Jóhannesson, A. Leinweber, J. Mamužić, G. D. Martinez, S. Otten, R. R. de Austri, P. Scott, Z. Searle, B. Stienen, J. Vanschoren, M. White, The DarkMachines High Dimensional Sampling Group, A comparison of optimisation algorithms for high-dimensional particle and astrophysics applications, Journal of High Energy Physics 2021 (5) (2021) 108. doi:10.1007/JHEP05(2021)108.
URL https://doi.org/10.1007/JHEP05(2021)108
[16] M. A. Diaz, G. Cerro, S. Dasmahapatra, S. Moretti, Bayesian active search on parameter space: a 95 gev spin-0 resonance in the $(b-l)$ssm (2024). arXiv:2404.18653.
URL https://arxiv.org/abs/2404.18653
[17] F. Staub, T. Ohl, W. Porod, C. Speckner, A Tool Box for Implementing Supersymmetric Models, Comput. Phys. Commun. 183 (2012) 2165–2206. arXiv:1109.5147, doi:10.1016/j.cpc.2012.04.013.
[18] F. Staub, xBIT: an easy to use scanning tool with machine learning abilities (6 2019). arXiv:1906.03277.
[19] L. Shang, Y. Zhang, EasyScan_HEP: A tool for connecting programs to scan the parameter space of physics models, Comput. Phys. Commun. 296 (2024) 109027. arXiv:2304.03636, doi:10.1016/j.cpc.2023.109027.
[20] M. D. Goodsell, A. Joury, BSMArt: Simple and fast parameter space scans, Comput. Phys. Commun. 297 (2024) 109057. arXiv:2301.01154, doi:10.1016/j.cpc.2023.109057.
[21] F. Staub, SARAH (6 2008). arXiv:0806.0538.
[22] F. Staub, Sarah 4: A tool for (not only susy) model builders, Computer Physics Communications 185 (6) (2014) 1773–1790. doi:10.1016/j.cpc.2014.02.018.
URL http://dx.doi.org/10.1016/j.cpc.2014.02.018
[23] R. Garnett, Y. Krishnamurthy, X. Xiong, J. Schneider, R. Mann, Bayesian optimal active search and survey-

ing, in: Proceedings of the 29th International Conference on International Conference on Machine Learning, ICML'12, Omnipress, Madison, WI, USA, 2012, p. 843–850.

[24] T. pandas development team, pandas-dev/pandas: Pandas (Feb. 2020). doi:10.5281/zenodo.3509134.
URL https://doi.org/10.5281/zenodo.3509134

[25] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, S. Chintala, Pytorch: An imperative style, high-performance deep learning library (2019). arXiv:1912.01703.

[26] A. A. Abdelalim, B. Das, S. Khalil, S. Moretti, Di-photon decay of a light Higgs state in the BLSSM, Nucl. Phys. B 985 (2022) 116013. arXiv:2012.04952, doi:10.1016/j.nuclphysb.2022.116013.

[27] Search for a standard model-like Higgs boson in the mass range between 70 and 110 GeV in the diphoton final state in proton-proton collisions at $\sqrt{s}$ = 13 TeV (2023).

[28] W. Porod, Spheno, a program for calculating supersymmetric spectra, susy particle decays and susy particle production at e+e- colliders, Computer Physics Communications 153 (2003) 275–315.
URL https://api.semanticscholar.org/CorpusID:7905927

[29] W. Porod, F. Staub, Spheno 3.1: extensions including flavour, cp-phases and models beyond the mssm, Computer Physics Communications 183 (11) (2012) 2458–2469. doi:10.1016/j.cpc.2012.05.021.
URL http://dx.doi.org/10.1016/j.cpc.2012.05.021

[30] P. Bechtle, O. Brein, S. Heinemeyer, G. Weiglein, K. Williams, Higgsbounds: Confronting arbitrary higgs sectors with exclusion bounds from lep and the tevatron, Computer Physics Communications 181 (1) (2010) 138–167. doi:10.1016/j.cpc.2009.09.003.
URL http://dx.doi.org/10.1016/j.cpc.2009.09.003

[31] P. Bechtle, S. Heinemeyer, O. Stål, T. Stefaniak, G. Weiglein, Higgssignals: Confronting arbitrary higgs sectors with measurements at the tevatron and the lhc, The European Physical Journal C 74 (2) (Feb. 2014). doi:10.1140/epjc/s10052-013-2711-4.
URL http://dx.doi.org/10.1140/epjc/s10052-013-2711-4

[32] J. Alwall, R. Frederix, S. Frixione, V. Hirschi, F. Maltoni, O. Mattelaer, H. S. Shao, T. Stelzer, P. Torrielli, M. Zaro, The automated computation of tree-level and next-to-leading order differential cross sections, and their matching to parton shower simulations, JHEP 07 (2014) 079. arXiv:1405.0301, doi:10.1007/JHEP07(2014)079.

[33] G. Malkomes, B. Cheng, E. H. Lee, M. Mccourt, Beyond the pareto efficient frontier: Constraint active search for multiobjective experimental design, in: M. Meila, T. Zhang (Eds.), Proceedings of the 38th International Conference on Machine Learning, Vol. 139 of Proceedings of Machine Learning Research, PMLR, 2021, pp. 7423–7434.
URL https://proceedings.mlr.press/v139/malkomes21a.html

[34] 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 (1953) 1087–1092. doi:10.1063/1.1699114.

[35] W. K. Hastings, Monte Carlo Sampling Methods Using Markov Chains and Their Applications, Biometrika 57 (1970) 97–109. doi:10.1093/biomet/57.1.97.

[36] B. C. Allanach, C. Balazs, G. Belanger, M. Bernhardt, F. Boudjema, D. Choudhury, K. Desch, U. Ellwanger, P. Gambino, R. Godbole, T. Goto, J. Guasch, M. Guchait, T. Hahn, S. Heinemeyer, C. Hugonie, T. Hurth, S. Kraml, S. Kreiss, J. Lykken, F. Moortgat, S. Moretti, S. Penaranda, T. Plehn, W. Porod, A. Pukhov, P. Richardson, M. Schumacher, L. Silvestrini, P. Skands, P. Slavich, M. Spira, G. Weiglein, P. Wienemann, SUSY Les Houches Accord 2, Comput. Phys. Commun. 180 (2009) 8–25. arXiv:0801.0045, doi:10.1016/j.cpc.2008.08.004.

[37] L. Darmé, C. Degrande, C. Duhr, B. Fuks, M. Goodsell, G. Heinrich, V. Hirschi, S. Höche, M. Höfer, J. Isaacson, O. Mattelaer, T. Ohl, D. Pagani, J. Reuter, P. Richardson, S. Schumann, H.-S. Shao, F. Siegert, M. Zaro, UFO 2.0 – The Universal Feynman Output format, Eur. Phys. J. C 83 (7) (2023) 631. arXiv:2304.09883, doi:10.1140/epjc/s10052-023-11780-9.

[38] Searches for additional Higgs bosons and vector leptoquarks in $\tau\tau$ final states in proton-proton collisions at $\sqrt{s}$ = 13 TeV, Tech. rep., CERN, Geneva (2022).
URL https://cds.cern.ch/record/2803739

[39] R. Barate, et al., Search for the standard model Higgs boson at LEP, Phys. Lett. B 565 (2003) 61–75. arXiv:hep-ex/0306033, doi:10.1016/S0370-2693(03)00614-2.

[40] G. Aad, et al., Observation of a new particle in the search for the standard model higgs boson with the atlas detector at the lhc, Physics Letters B 716 (1) (2012) 1–29. doi:https://doi.org/10.1016/j.physletb.2012.08.020.
URL https://www.sciencedirect.com/science/article/pii/S037026931200857X

[41] S. Chatrchyan, et al., Observation of a new boson at a mass of 125 gev with the cms experiment at the lhc, Physics Letters B 716 (1) (2012) 30–61. doi:https://doi.org/10.1016/j.physletb.2012.08.021.
URL https://www.sciencedirect.com/science/article/pii/S0370269312008581

[42] H. Bahl, T. Biekötter, S. Heinemeyer, C. Li, S. Paasch, G. Weiglein, J. Wittbrodt, HiggsTools: BSM scalar phenomenology with new versions of HiggsBounds and HiggsSignals, Comput. Phys. Commun. 291 (2023) 108803. arXiv:2210.09332, doi:10.1016/j.cpc.2023.108803.

[43] P. Z. Skands, et al., SUSY Les Houches accord: Interfacing SUSY spectrum calculators, decay packages, and event generators, JHEP 07 (2004) 036. arXiv:hep-ph/0311123, doi:10.1088/1126-6708/2004/07/036.

[44] A. Buckley, Pyslha: a pythonic interface to susy les houches accord data (2015). arXiv:1305.4194.
URL https://arxiv.org/abs/1305.4194

[45] F. Staub, xslha: An les houches accord reader for python and mathematica, Computer Physics Communications 241 (2019) 132–138. doi:10.1016/j.cpc.2019.03.013.
URL http://dx.doi.org/10.1016/j.cpc.2019.03.013

[46] T. Head, M. Kumar, H. Nahrstaedt, G. Louppe, I. Shcherbatyi, scikit-optimize/scikit-optimize (Oct. 2021). doi:10.5281/zenodo.5565057.
URL https://doi.org/10.5281/zenodo.5565057

[47] O. Yadan, J. Sommer-Simpson, O. Delalleau, omegaconf (2019).
URL https://github.com/omry/omegaconf

[48] M. Balandat, B. Karrer, D. R. Jiang, S. Daulton, B. Letham, A. G. Wilson, E. Bakshy, BoTorch: A Framework for Efficient Monte-Carlo Bayesian Optimization, in: Advances in Neural Information Processing Systems 33, 2020.
URL http://arxiv.org/abs/1910.06403

[49] J. Gardner, G. Pleiss, K. Q. Weinberger, D. Bindel, A. G. Wilson, Gpytorch: Blackbox matrix-matrix gaussian process inference with gpu acceleration, in: S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, R. Garnett (Eds.), Advances in Neural Information Processing Systems, Vol. 31, Curran Associates, Inc., 2018.

[50] T. Akiba, S. Sano, T. Yanase, T. Ohta, M. Koyama, Optuna: A next-generation hyperparameter optimization framework, in: Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 2019.