FlexibleSUSY extended to automatically compute physical quantities in any Beyond the Standard Model theory: Charged Lepton Flavor Violation processes, Higgs decays, and user-defined observables

Uladzimir Khasianevich [a,*], Wojciech Kotlarski [b], Dominik Stöckinger [a], Alexander Voigt [c]

[a] *Institut für Kern- und Teilchenphysik, TU Dresden, Zellescher Weg 19, 01069 Dresden, Germany*
[b] *National Centre for Nuclear Research Pasteura 7, 02-093 Warsaw, Poland*
[c] *Institute for Theoretical Solid State Physics, RWTH Aachen University, Sommerfeldstraße 16, 52074 Aachen, Germany*

arXiv:2402.14630v1 [hep-ph] 22 Feb 2024

Abstract

`FlexibleSUSY` is a framework for the automated computation of physical quantities (observables) in models beyond the Standard Model (BSM). This paper describes an extension of `FlexibleSUSY` which allows to define and add new observables that can be enabled and computed in applicable user-defined BSM models. The extension has already been used to include Charged Lepton Flavor Violation (CLFV) observables, but further observables can now be added straightforwardly. The paper is split into two parts. The first part is non-technical and describes from the user's perspective how to enable the calculation of predefined observables, in particular CLFV observables. The second part of the paper explains how to define new observables such that their automatic computation in any applicable BSM model becomes possible. A key ingredient is the new `NPointFunctions` extension which allows to use tree-level and loop calculations in the model-independent setup of observables. Three examples of increasing complexity are fully worked out. This illustrates the features and provides code snippets that may be used as a starting point for implementation of further observables.

Keywords: Beyond the Standard Model, New Physics, Supersymmetry, Charged Lepton Flavor Violation

Program summary

Program Title: `NPointFunctions`
CPC Library link to program files: (to be added by Technical Editor)
Developer's repository link: https://github.com/FlexibleSUSY/FlexibleSUSY
Code Ocean capsule: (to be added by Technical Editor)
Licensing provisions: GPLv3
Programming language: `C++`, `Wolfram Language`, `Fortran`, `Bourne shell`
Journal reference of previous version: Comput. Phys. Commun. 230 (2018) 145–217; PoS CompTools2021 (2022) 036
Does the new version supersede the previous version?: Yes
Reasons for the new version: Program extension including new observables and file structures

*Corresponding author.
E-mail address: uladzimir.khasianevich@tu-dresden.de

Nature of problem: Determining observables for an arbitrary extension of the Standard Model supported by FlexibleSUSY, input by the user.
Solution method: Generation of the code from automated algebraic manipulations. Automatic filling and compiling of predefined template files.
Additional comments including restrictions and unusual features: Vertices with a direct product of Lorentz and color structures are supported. Settings of the advanced NPointFunctions mode rely on explicit specification of topologies.

Contents

1 Introduction **3**

2 Available observables and how to calculate them with FlexibleSUSY **4**
2.1 Installation . . . 4
2.2 Output defined observable with FlexibleSUSY . . . 4
2.3 Output Wilson coefficients with FlexibleSUSY . . . 6

3 File structure of new observables O_i **7**
3.1 Content of the file O_i/Observables.m . . . 9
3.1.1 Example 1: output a given number . . . 10
3.1.2 Example 2: show fermion masses . . . 11
3.1.3 Example 3: lepton self-energy with NPointFunctions . . . 12
3.2 Content of the file O_i/WriteOut.m . . . 12
3.2.1 Example 1: output a given number . . . 14
3.2.2 Example 2: show fermion masses . . . 14
3.2.3 Example 3: lepton self-energy with NPointFunctions . . . 15
3.3 Content of the file O_i/FlexibleSUSY.m, simple settings for NPointFunctions . . . 15
3.3.1 Example 1: output a given number . . . 19
3.3.2 Example 2: show fermion masses . . . 20
3.3.3 Example 3: lepton self-energy with NPointFunctions (Observable -> None) 21
3.4 Content of the C++ template files . . . 23
3.4.1 Example 1: output a given number . . . 25
3.4.2 Example 2: show fermion masses . . . 25
3.4.3 Example 3: lepton self-energy with NPointFunctions . . . 26
3.5 Content of the optional file O_i/NPointFunctions.m, advanced settings . . . 27
3.5.1 Enabling advanced settings, overview . . . 27
3.5.2 topologies . . . 28
3.5.3 diagrams (generic level modifications) . . . 30
3.5.4 amplitudes (class level modifications) . . . 31
3.5.5 order, chains (Dirac algebra modifications) . . . 32
3.5.6 Other modifications . . . 33
3.5.7 Example 3: lepton self-energy with NPointFunctions (Observable -> O_i[]) 34
3.6 Content of the optional file O_i/FSMathLink.m . . . 36
3.7 New observables and how to calculate them with FlexibleSUSY . . . 36
3.7.1 Example 1: output a given number . . . 36

3.7.2 Example 2: show fermion masses . . . 38
3.7.3 Example 3: lepton self-energy with `NPointFunctions` . . . 39

4 Conclusions **40**

Appendix A Available observables: physical details **41**
Appendix A.1 Two-body CLFV decays ($\ell_i \to \ell_j \gamma$) . . . 41
Appendix A.2 Three-body CLFV decays ($\ell_i \to \ell_j \ell_k \ell_k^c$) . . . 42
Appendix A.3 Coherent conversion in nuclei (μ–e conversion) . . . 44

Appendix B Additional features and examples **46**
Appendix B.1 Example 4: post-processing in $h \to gg$ . . . 46
Appendix B.2 Example 5: `flavio` output in $b \to s\mu\mu$ . . . 47
Appendix B.3 Example 6: additional LH input blocks in μ–e conversion . . . 48

1. Introduction

Exploring the parameter space of Beyond the Standard Model (BSM) theories, researchers frequently employ software packages to automate both intricate calculations and parameter scans. Nevertheless, these software tools are often restricted to a limited range of models, closely related to the Standard Model (SM), the Minimal Supersymmetric Standard Model (MSSM), the Two-Higgs-Doublet Model (2HDM), and their extensions involving higher-dimensional operators. This is only a slice of all intriguing possibilities that include a broad set of models.

To address this limitation, `FlexibleSUSY` [1, 2] was created to be used for a broad class of supersymmetric (SUSY) or non-SUSY models, providing a tool for the comprehensive investigation of diverse theoretical scenarios.[1] It is a software application primarily implemented in the `Wolfram Language` [7] and `C++` based on `SARAH` [3, 8–10] and components from `SoftSUSY` [11, 12], designed to produce an efficient and accurate `C++` spectrum generator (a program searching for a consistent set of model parameters and calculating the pole mass spectrum and a set of observables) for physical models specified by the user.

The produced `C++` program applies user-defined boundary conditions at up to three distinct energy scales within the model, incorporating Renormalization Group Equation (RGE) evolution between these scales. It further generates a collection of mixing matrices, pole masses, and auxiliary quantities. Recent versions of `FlexibleSUSY` have introduced the computation of several important observables that are suitable for phenomenological investigations and comparisons with experimental data. In particular, we highlight the extensions `FlexibleAMU` and `FlexibleCPV` introduced in Ref. [2] (they are responsible for the calculations of the anomalous magnetic moment and Electric Dipole Moment (EDM) of leptons), an update for `FlexibleMW` on precise calculation of the W-boson pole mass from Ref. [13], and `FlexibleDecay` [14] (a tool to calculate decays of scalars in a broad class of BSM models).

A key point of these observables is that they are integrated in `FlexibleSUSY` such that they are ready to be computed for any desired BSM scenario `FlexibleSUSY` is applied to. In order to achieve this, the mentioned extensions store information about the observables in suitable `Wolfram`

[1] A software with similar capabilities is `SARAH`/`SPheno`/`FlavorKit` [3–6].

Language and C++ meta code which is then automatically converted into actual C++ code specifically for each BSM scenario.

So far, new observables were added by individually modifying the internals of FlexibleSUSY, hence users could not add new observables in such a model-independent way. In the present paper, we explain a new FlexibleSUSY 2.8 design structure which solves this problem. It allows to integrate new observables on the meta code level, and it provides powerful options to define and finetune the computation of new observables without having to touch internals of FlexibleSUSY.

A number of new observables has already been integrated by using this new structure (they correspond to various Charged Lepton Flavor Violation (CLFV) processes), and in the future, further additional observables may be integrated either by FlexibleSUSY developers or by users of FlexibleSUSY.

To streamline and modularize the integration of new observables into FlexibleSUSY, an extension named NPointFunctions [15] was developed. This extension serves to automate the calculation of amplitudes and other quantities that rely on them for any high-energy model supported by FlexibleSUSY. In essence, NPointFunctions incorporates a well-defined approach of widely used packages FeynArts [16], FormCalc [17], and ColorMath [18] into FlexibleSUSY (up to technical implementation details to be mentioned later in appropriate sections).

The article is separated into two parts depending on the readers' interests:

1. Section 2 describes how to install and use FlexibleSUSY to calculate any of the available observables. This section is of interest for all users of FlexibleSUSY who may want to switch on the computation of observables. Reading it does not require knowledge of the internal structure of FlexibleSUSY. To get physical insights about the new CLFV observables, one is invited to read Appendix A.

2. Section 3 presents details on how to implement new observables. It provides a general outline and background information relevant for all observables, and it covers three specific examples of increasing complexity. It thus illustrates the range of possibilities and equips users with code snippets which can be used as a basis for further developments. Interesting features improving the functionality of FlexibleSUSY are mentioned separately in Appendix B.

2. Available observables and how to calculate them with FlexibleSUSY

Observables that are currently available in FlexibleSUSY are shown in Table 1. This section shows the reader how to calculate them with FlexibleSUSY.

2.1. Installation

There are no changes to the previous FlexibleSUSY version of Refs. [2, 14] with respect to mandatory installation steps. All missing dependencies will be highlighted by FlexibleSUSY during the execution of the configure script (their list and hints about the installation can be found at developer's repository, see the program summary). To use observables that rely on NPointFunctions module (in particular, CLFV ones), FeynArts and FormCalc must be installed.

2.2. Output defined observable with FlexibleSUSY

To switch on the calculation of a desired, predefined observable O_i (for example, bsgamma or LToLConversion from Table 1) for a physical model M_a (like SM or MSSM), one needs to modify

Observable	FlexibleSUSY name	Loop level	Hints and comments
Δa_ℓ	AMM, AMMUncertainty		Anomalous magnetic moment of a lepton [2]
d_ℓ	EDM		Electric dipole moment of a lepton [2]
$\ell_i \to \ell_j \gamma$	BrLToLGamma	1	See Appendix A.1
$\ell_i \to \ell_j \ell_k \ell_k^c$	BrLTo3L	0–1	See Table 2 and Appendix A.2
μ–e conversion	LToLConversion	0–1	See Table 3 and Appendix A.3
$b \to s\gamma$	bsgamma	1	—
—	FlexibleDecay	known SM (0–4), LO (0–1) for BSM	Decays of scalars [14]

Table 1: All observables currently supported by FlexibleSUSY.

Usage		
``FlexibleSUSYObservable`BrLTo3L[lep_, iI_ -> {iJ_, iK_, iK_}, contr_, loopN_]``		
Abbreviation	**Values**	**Hints**
lep	symbol	Leptons ℓ in $\ell_i \to \ell_j \ell_k \ell_k^c$ in SARAH model (Fe in SM or MRSSM)
iI, iJ, iK	integer	Generations i, j, k in $\ell_i \to \ell_j \ell_k \ell_k^c$, starting from 1
contr	Vectors, Scalars, Boxes	Contributions (a synonym or a subset is allowed), see FlexibleSUSY.m, NPointFunctions.m in the directory meta/Observables/BrLTo3L/
loopN	0–1	Loop level, FormCalc limitation

Table 2: Options for BrLTo3L, see meta/Observables/BrLTo3L/Observables.m.

either model_files/M_a/FlexibleSUSY.m or models/M_a/FlexibleSUSY.m. These files define the C++ spectrum generator output in the SUSY Les Houches Accord (SLHA) [19, 20] or the Flavour Les Houches Accord (FLHA) [21] formats.

The first file mentioned above is used by the script createmodel (see Sections 3.7.1–3.7.3) to create both the directory models/M_a/ and the second file, while the latter is executed by commands configure and make that create C++ spectrum generator itself. This means, that to always include the desired observables after the directory models/M_a/ is purged or cleaned, one modifies the first file.

The required modification of the file FlexibleSUSY.m is in the list ExtraSLHAOutputBlocks. Each observable that should be computed and appear in the spectrum generator output needs a corresponding entry which specifies the details of the observable and how it should appear in the output. For example, μ–e conversion is switched on in the Minimal R-Symmetric Supersymmetric Standard Model [22] (MRSSM) (for its particular FlexibleSUSY configuration called MRSSM2) as follows:

Usage		
FlexibleSUSYObservable`LToLConversion[lep_, iI_ -> iO_, nucl_, contr_, loopN_]		
Abbreviation	**Values**	**Hints**
lep	symbol	Leptons in SARAH model (Fe in SM or MRSSM)
iI, iO	integer	Muon and electron generations, starting from 1
nucl	Al or Au	Nucleus, see l_to_l_conversion.cpp.in in the directory templates/observables/
contr	Vectors, Scalars, Boxes	Contributions (a synonym or list of a subset is allowed), see FlexibleSUSY.m, NPointFunctions.m in the directory meta/Observables/LToLConversion/
loopN	0–1	Loop level, FormCalc limitation

Table 3: Options for LToLConversion, see meta/Observables/LToLConversion/Observables.m.

Listing 1: Adding μ–e conversion observable (example for models/MRSSM2/FlexibleSUSY.m).

```
ExtraSLHAOutputBlocks = {
   {
      FlexibleSUSYLowEnergy,
      {
         {41, FlexibleSUSYObservable`LToLConversion[Fe, 2 -> 1, Al, All, 1]},
         ...
      }
   },
   ...
};
```

The numerical value of μ–e conversion after the execution of the spectrum generator will be stored in SLHA format under the user-chosen number 41 in the block FlexibleSUSYLowEnergy (see meta/WriteOut.m). More details for arguments of LToLConversion are provided in Table 3.

To numerically calculate all added observables (apart from FlexibleDecay), the observable calculation must be enabled in the SLHA input file for C++ spectrum generator:[2]

Listing 2: In models/Mₐ/LesHouches.in.Mₐ. All added observables are enabled.

```
Block FlexibleSUSY
   ...
   15   1                    # calculate all observables
```

2.3. *Output Wilson coefficients with FlexibleSUSY*

One can also output Wilson coefficients used in derivation of LToLConversion or BrLTo3L. To do that, one places the corresponding observable into the FWCOEF (IMFWCOEF) block to output their real (imaginary) part, for example:

[2]Here, the number 15 corresponds to the FlexibleSUSY conventions for the SLHA block Block FlexibleSUSY, introduced in Appendix B of Ref. [2]. Calculation of observables using FlexibleDecay is controlled by a separate flag as explained in Ref. [14].

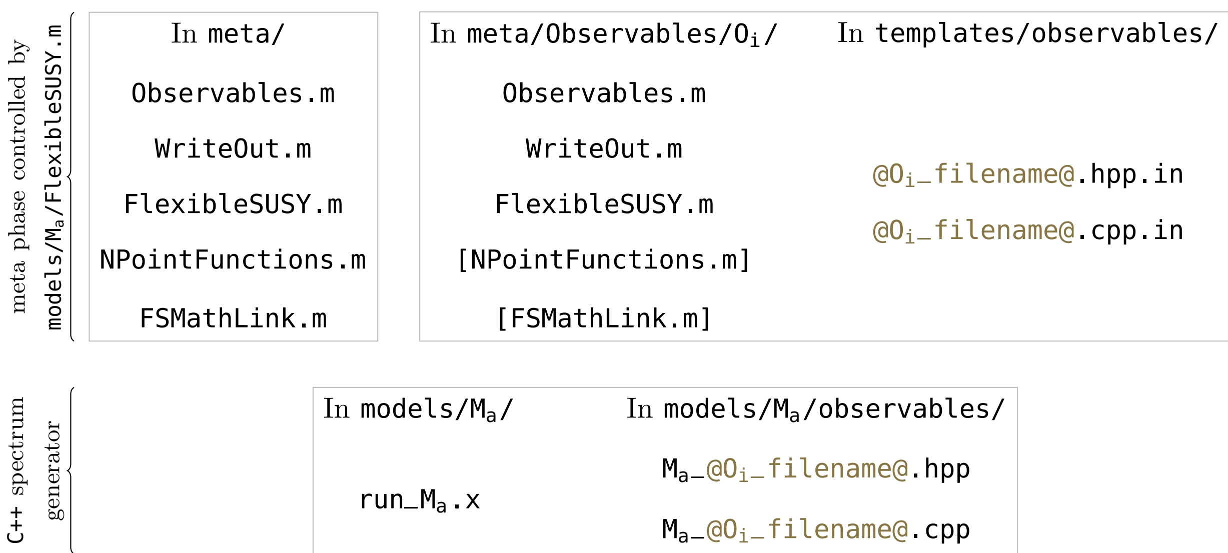


Figure 1: The place and role of new files, required by new observables. The files in square brackets are optional as described in the text. The name of C++ files contains @O$_i$_filename@, replaced by the rule GetObservableFileName, see Listing 5.

Listing 3: Showing Wilson coefficients (example for models/MRSSM2/FlexibleSUSY.m).

```
ExtraSLHAOutputBlocks = {
   {
      FWCOEF,
      {
         {1, FlexibleSUSYObservable`LToLConversion[Fe, 2 -> 1, Al, All, 1]},
         ...
      }
   },
   ...
};
```

The coefficients are defined in files meta/Observables/O$_i$/WriteOut.m.

3. File structure of new observables O$_i$

Two new FlexibleSUSY features are described in this paper: a way to add new observables to FlexibleSUSY (implying new auxiliary files), and a way to generate C++ code to numerically calculate amplitudes that are required by these observables (NPointFunctions extension). This section addresses both extensions and explains how to implement new observables with the help of toy examples and code snippets of already implemented observables.

To create a spectrum generator, FlexibleSUSY usually first runs Wolfram Language code (located in meta/) to obtain model-specific expressions for mass matrices, self-energies, amplitudes, etc. The obtained expressions are converted to C++ form and are filled into C++ template files (located in templates/) to generate model-specific C++ code (located in model/M$_a$/). Figure 1 shows the files relevant for the calculation of observables. The top left block shows Wolfram

Language files in meta/ that contain general routines and observables implemented in previous versions of FlexibleSUSY. The top right block shows the corresponding observable-specific Wolfram Language files (located in meta/Observables/O_i/) and the necessary C++ template files (located in templates/observables/). The bottom block shows the generated relevant C++ files that are eventually compiled and combined with other C++ files to the final spectrum generator executable (models/M_a/run_M_a.x).

In the following, by creating a new observable we mean the generation of a specific C++ function that calculates the observable within the final spectrum generator. That is, one needs to create several C++ code blocks in appropriate places of the generated C++ spectrum generator, as the following template-listing with observable- and model-specific blocks shows:[3]

Listing 4: C++ template code to calculate an observable for a specific model M_a.

```
// 1. Function definition in Ma_@Oi_filename@.cpp:
namespace Ma_@Oi_namespace@ {
@Oi_output_type@ @Oi_calculate(...)@ {
   // Complicated function body generated during the meta phase and specific for Oi and Ma
}
}

// 2. Internal calculation of the observable:
observables.@Oi_name@ = Ma_@Oi_namespace@::@Oi_calculate(...)@;

// 3. Writing the observable in Les Houches format to an output stream:
block << FORMAT_*(observables.@Oi_name@, ...);
```

In the C++ source code snippet above, @O_i_name@ is replaced by the name of the observable and @O_i_output_type@ is its C++ type. The @O_i_calculate(...)@ pattern will be replaced by the name of function that performs the calculation, which is defined in the M_a_@O_i_namespace@ namespace. The body of the function will depend on the observable and the model under consideration.

In the following subsections it is described how the expressions are determined that replace the different patterns in the above C++ template code snippet. The files where these expressions are determined are the Wolfram Language files located in the directory meta/Observables/O_i/, which will be described in the following subsections. In general, to define a new observable one creates five new files filled with the content described in the following steps:

1. In Section 3.1 we show how to define the C++ name of the observable, the observable's C++ type, etc. (done in the file meta/Observables/O_i/Observables.m).

2. In Section 3.2 we show how to connect the calculation of the observable to the spectrum generator output (done in the file meta/Observables/O_i/WriteOut.m).

3. In Sections 3.3–3.4 we describe how to fill the body of the function that calculates the numeric value of the observable for a given parameter point (done in the meta-phase file

[3]These C++ expressions might seem to be trivial at this point and the natural question arises whether it is really necessary to make use of the file structure described in Sections 3.1–3.4. One needs to keep in mind, that the mentioned expressions depend both on the physical model M_a and the observable O_i settings, which change the parts generated during the meta phase. These modifications can, in general, be very complex, and FlexibleSUSY meta-phase routines help to apply them.

meta/Observables/O_i/FlexibleSUSY.m in Section 3.3 and two C++ template files in Section 3.4).

4. As a last step, we show how to modify the model-specific FlexibleSUSY model file named models/M_a/FlexibleSUSY.m to register the observable to the model under consideration.

Each subsection will begin with general explanations and definitions. Then three recurring examples will be used for illustration. The first example illustrates the definition of an observable with minimal complexity — the output of a single numerical constant, where the numerical constant is not hard-coded but can be specified separately for each model FlexibleSUSY is applied to. The second example outputs the value of fermion masses, where the decision of which fermions to select can be specified separately for each model. The third example outputs the value of one-loop self-energies and thus illustrates how to use loop calculations in the definition of observables.

3.1. Content of the file O_i/Observables.m

General case

The creation of a new observable O_i starts with the file meta/Observables/O_i/Observables.m. This file is supposed to define the C++ tokens such as the observable name (@O_i_name@) or the observable's C++ type (@O_i_output_type@), etc. The file contains only one call to the function DefineObservables, which defines all C++ tokens. The general structure of this function call is shown in the following Wolfram Language source code snippet:

Listing 5: General content of meta/Observables/O_i/Observables.m.

```
Observables`DefineObservable[
   FlexibleSUSYObservable`Oi[parA_, parB_, ...], (* Wolfram Language symbol for observable *)
   GetObservableType        -> {1},                (* @Oi_output_type@ *)
   GetObservablePrototype   -> "calc_parA(int parB, ...)", (* @Oi_calculate(...)@ prototype *)

   (* Optional: to override defaults, use commands below *)
   GetObservableName        -> "name_with_parAparB...",     (* @Oi_name@ *)
   CalculateObservable      -> "calc_parA(parB+1, ...)",    (* @Oi_calculate(...)@ call *)
   GetObservableNamespace   -> "observable_namespace",      (* @Oi_namespace@ *)
   GetObservableFileName    -> "observable_file",           (* @Oi_filename@ *)
   GetObservableDescription -> "observable: parA, parB"     (* description *)
];
```

The first argument of the function DefineObservables defines the Wolfram Language symbol of the observable O_i, which is used in the file models/M_a/FlexibleSUSY.m to register the observable to the model-specific FlexibleSUSY model file, see Tables 2–3 for examples. The optional parameters (like parA_) can be freely chosen and may be used to specify anything that might be changed from one model M_a to another (like the SARAH name of a particle multiplet, relevant contributions, etc.), or may be specific for the observable itself (like a particle generation index, the loop level, etc.). The parameters may be wrapped by other Wolfram Language functions (like Rule) to improve readability. Note that by convention, the names of all FlexibleSUSY observables belong to FlexibleSUSYObservable` context.

All further arguments of the function DefineObservables define how C++ tokens (for example, @O_i_output_type@) for the observable O_i should be replaced in code from Listing 4. Note that the names of the parameters defined for the observable (like parA, parB, etc.) are replaced

in the strings by their values given in the model-specific `FlexibleSUSY` model file (the only exception: arguments in the prototype). For example, if one writes in the `FlexibleSUSY` model file `FlexibleSUSYObservable`O_i[Fe, 3]`, then the strings `parA` and `parB` are replaced by `Fe` and `3`, respectively, in all strings on the r.h.s. in the function `DefineObservable`, so that for example the execution of the function `GetObservableName` produces `"name_with_Fe3"`.[4]

The meaning of two other mandatory arguments of the function `DefineObservables` is the following:

`GetObservableType` defines the `C++` type of the observable and replaces `@O_i_output_type@`. In general, it can be any (real or complex) scalar, array or matrix `C++` type. The syntax `{1}` in Listing 5 defines a complex-valued array of length 1 (corresponds to the `C++` type `Eigen::Array<std::complex<double>,1,1>`). See also `BrLToLGamma/Observables.m` for another example. Note that array, vector or matrix types are convenient for storing Wilson coefficients, debugging information, etc. As described later, the connection between the observable type and the `FlexibleSUSY` Les Houches output is specified in the file named `meta/Observables/O_i/WriteOut.m`.

`GetObservablePrototype` defines the `C++` prototype for the function that calculates the observable (`@O_i_calculate(...)@`). Note that the function name is modified with the values of the parameters of the first argument of the function `DefineObservable` (`parA` is replaced with value of `parA_` in Listing 5), while the function arguments are kept intact (`parB` is kept).[5]

Let us now illustrate how to create new observables in `FlexibleSUSY` starting from the usage of the function `DefineObservable` in `Observables.m` file. The examples are of increasing complexity and they will be continued in the next subsections.

3.1.1. Example 1: output a given number

Let us create an observable, called `ExampleConstantObservable`, that outputs a numerical constant specified by the user in the `FlexibleSUSY.m` file, so that the value of the numerical constant is not hard-coded in the body of the observable `C++` code but rather configured with the `Wolfram Language` files. This example will demonstrate the basic usage of the function that calculates an observable and allows us to familiarize ourselves with the workflow, as no physical calculation is required. First, one creates the `meta/Observables/ExampleConstantObservable/` directory. Afterwards one creates the file `Observables.m` inside this directory with the following content:

---

[4]More advanced ways to name the `C++` code parts are supported: 1. One may use expression like `"$(parA+1)"`, which evaluates the content inside the parenthesis after substitution of observable pattern values (`parA_`). 2. The description of observable in the SLHA output can be generated from its name automatically by replacing underscores with spaces. This can be redefined by `GetObservableDescription`. 3. Both names for `C++` templates and `C++` namespace can be automatically generated from the `Wolfram Language` name of the observable by inserting an underscore before capital letters (or before numbers in front of them) and lowering the letter case. If the generated replacement for `@O_i_namespace@` (`@O_i_filename@`) leads to undesired side effects, then one may use `GetObservableNamespace` (`GetObservableFileName`) to override the default behavior. 4. `GetObservableName` uniquely defines the `C++` name of the observable, which replaces `@O_i_name@`. This name is used to store the numerical value of the observable internally and is generated automatically but can be overridden.

[5]The following two additional function arguments may be added: `auto model` and `auto qedqcd`. The first one allows to access all numeric quantities of the model (model parameters, masses, etc.). The second one provides access to known low-energy input observables.

Listing 6: Content of ExampleConstantObservable/Observables.m.

```
Observables`DefineObservable[
   FlexibleSUSYObservable`ExampleConstantObservable[num_],
   GetObservableType      -> {1},
   GetObservablePrototype -> "calculate_example_constant_observable(double num)"
];
```

As in the general case discussed above, the function call of DefineObservables defines the Wolfram Language symbol of the observable to be ExampleConstantObservable, and it specifies that the observable depends on one parameter num_; the meaning of the parameter will be the value of the numerical constant to be output. Accordingly, as C++ return type for the function that calculates the observable we use an array of size one in GetObservableType. GetObservablePrototype defines the prototype of the function that performs the calculation and we use double num as function argument.

3.1.2. Example 2: show fermion masses

Let us turn to a little more involved example to illustrate the usage of model-specific quantities in the calculation of an observable and the possibility of filling the C++ template files with model-specific content. As an example, we define an "observable" which can output several masses whose values are determined in FlexibleSUSY. Specifically we choose to allow the output the Modified Minimal Subtraction Scheme ($\overline{\text{MS}}$)/Dimensional Reduction Scheme ($\overline{\text{DR}}$) mass of a user-selected fermion at a given renormalization scale and/or the pole mass of a SM lepton. Again, similar to the example above, the fermion field and its generation number will be specified by the user in the FlexibleSUSY.m file, so that these values are not hard-coded in the body of the observable C++ code but are configured with the Wolfram Language files.

To set up the observable, we create the directory meta/Observables/ExampleFermionMass/ and the file Observables.m within it, with the following content:

Listing 7: Content of ExampleFermionMass/Observables.m.

```
Observables`DefineObservable[
   FlexibleSUSYObservable`ExampleFermionMass[fermion_[gen_]],
   GetObservableType         -> {2},
   GetObservablePrototype    -> "ex_fermion_mass(int gen, auto model, auto qedqcd)",
   GetObservableDescription -> "fermion[gen] (lepton[gen] if in Block ExampleLeptonMass) mass"
];
```

Again, the function call of DefineObservables first defines the Wolfram Language symbol of the observable to be ExampleFermionMass, and it specifies that the observable depends on two arguments, merged into one Wolfram Language function. We assume that the concrete name of the fermion to output will be defined in the FlexibleSUSY model file models/M_a/FlexibleSUSY.m, and we assume that it has the form fermion[gen], where fermion is the name of the fermion multiplet and gen is its index in the multiplet. Therefore, the observable ExampleFermionMass has the form of a function with the fermion_[gen_] pattern as argument, which matches the multiplet name with fermion_ and the index in the multiplet with gen_.[6] We would like to return

[6]Note that in the generated C++ code the multiplet index is decreased by one in order to match the C++ index convention. I.e. if one writes Fe[1] in the Wolfram Language model file FlexibleSUSY.m and refers in the generated C++ code to the argument of Fe, then its value will be 0, not 1.

two numerical values, corresponding to the $\overline{\text{MS}}/\overline{\text{DR}}$ mass of the fermion `fermion[gen]` at a given renormalization scale and the pole mass of a SM lepton of generation `gen`. Hence we define the observable type to be an array of length 2 in `GetObservableType`. The prototype of the function that calculates the observable is defined in `GetObservablePrototype`. The function takes the index of the fermion in the multiplet as first argument. The remaining two arguments `model` and `qedqcd` allow the access to the model parameters, masses, mixing matrices, etc. In the name of the function prototype `ex_fermion_mass`, the string `fermion` is replaced by the value of the variable `fermion`, which may be `Fe`, `Fd`, etc. depending on the user-selected model $\mathtt{M_a}$. As we plan to calculate different masses depending on the SLHA output block (see Section 3.2.2), let us implement the description of the observable as specified with `GetObservableDescription` above to have an explicit indication of the generated numerical values. There, the strings `fermion` and `gen` are replaced by the user-specified values of `fermion` and `gen`, respectively.

3.1.3. Example 3: lepton self-energy with `NPointFunctions`

As a third example, let us implement an observable to calculate the self-energy of a user-specified lepton at the one-loop level. This example will illustrate how to use the `NPointFunctions` extension in both simple and advanced ways, as well as how to output quantities in the FLHA format. As in the previous examples, we start by defining the observable and a suitable `C++` function prototype that allow us to select the lepton (from a multiplet) whose self-energy shall be calculated and the contributions that should be taken into account:

Listing 8: Content of `ExampleLeptonSE/Observables.m`.

```
Observables`DefineObservable[
   FlexibleSUSYObservable`ExampleLeptonSE[field_[gen_], contr_],
   GetObservableType       -> {2},
   GetObservablePrototype -> "ex_lepton_se_contr(int gen, auto model)"
];
```

We call the observable `ExampleLeptonSE`, which takes the multiplet name (`field_`) and the index of the particle in the multiplet (`gen_`) as first argument, as in the previous example. Furthermore, we'd like to be able to select a subset of the full one-loop contribution. To achieve this, we provide a second parameter (`contr_`) which we will use later to only calculate a certain part of the full one-loop self-energy. In general, a fermion self-energy can be split into different covariants involving left-handed or right-handed projection operators $P_{L,R}$ and possibly the covariant $\not{p}$, where p^μ is the external fermion momentum. Let us in this example for simplicity output only the coefficients of the two covariants $\not{p}P_L$ and $\not{p}P_R$, so we choose an array of length 2. The `C++` function needs the index of the particle in the multiplet (`gen`) and the model object (`model`) as arguments.

3.2. Content of the file $\mathtt{O_i}$`/WriteOut.m`

General case

After the definition of the observable name, type, etc. in the file $\mathtt{O_i}$`/Observables.m`, one can now connect the numerical value(s) of the observable to the Les Houches output of the `C++` spectrum generator in the file `meta/Observables/`$\mathtt{O_i}$`/WriteOut.m`. Currently, there are two automated ways to output the results of calculations in `FlexibleSUSY`: via the SLHA (exists since `FlexibleSUSY 1.0`) and the FLHA (the usage is explained in this article) formats. Both are defined in the `meta/WriteOut.m` file, which provides functions to write numbers, arrays or matrices to specified output blocks.

In the simplest case the value of an observable `Oᵢ[parA, parB, ...]` is a single complex number (see Examples 1–2 below or the $\ell_i \to \ell_j \gamma$ decay; their Les Houches output is shown in Sections 3.7.1–3.7.2). If we want to write its real part to the `FlexibleSUSYLowEnergy` SLHA output block, we define a function called `WriteObservable` in `meta/Observables/Oᵢ/WriteOut.m` as follows:

Listing 9: Content of `meta/Observables/Oᵢ/WriteOut.m`.

```
WriteOut`WriteObservable[
   "FlexibleSUSYLowEnergy",
   obs:FlexibleSUSYObservable`Oᵢ[parA_, parB_, ...]
] := "Re(observables." <> Observables`GetObservableName[obs] <> "(0))";
```

The first argument to the `WriteObservable` function is a string with the SLHA output block name (here: `"FlexibleSUSYLowEnergy"`), which reflects the SLHA block name in the `FlexibleSUSY` model file `models/Mₐ/FlexibleSUSY.m`. Since the observable type is a complex-valued array of length 1 in this example, we have to output the 0-th element of the array (in `C++` index convention). We obtain the real part of the array element by applying the `Re` function.

As another example we consider the output of the real parts of several Wilson coefficients (see also Example 3 below and its Les Houches output in Section 3.7.3). This requires a more involved definition of the function `WriteObservable`:

Listing 10: Content of `meta/Observables/Oᵢ/WriteOut.m` to output the real parts of several Wilson coefficients.

```
WriteOut`WriteObservable[
   "FWCOEF", (* Or "IMFWCOEF" with Re below replaced with Im *)
   obs:FlexibleSUSYObservable`Oᵢ[parA_, parB_, ...]
] :=
StringReplace[
   {
      "fermions₁, operator₁, Oalpha₁, Oalphas₁, contributions₁, num_value, \"comment₁\"",
      "fermions₂, operator₂, Oalpha₂, Oalphas₂, contributions₂, num_value, \"comment₂\"",
      ...
   },
   {
      "num_value" -> "Re(observables." <> Observables`GetObservableName[obs] <> ")",
      ...
   }
];
```

We define the function `WriteObservable` to write the real parts of the Wilson coefficients to the `FWCOEF` Les Houches output block.[7] The return value of `WriteObservable` is a list of strings, with each string consisting of a comma-separated tuple of a fermion name (`fermions₁`), an operator name (`operator₁`), etc., which should be replaced by appropriate values. See Ref. [21] for a description of the FLHA format. The numbering convention for the FLHA format in `FlexibleSUSY` is the following: Wilson coefficients must occupy the positions from the end of the `C++` output array. In the definition of the function `WriteObservable` the substring `"num_value"` is replaced

[7] If the imaginary parts of the Wilson coefficients shall be written to the output, one should use `"IMFWCOEF"` as block name and replace `Re` with `Im`.

by the real parts of the Wilson coefficients (accessed via GetObservableName) via the function StringReplace. Note that there is no need to explicitly specify the numbering of the Wilson coefficients in the replacement rule for "num_value", as it is done automatically.

3.2.1. Example 1: output a given number

We continue our minimal Example 1 from Section 3.1.1, where the observable is defined to just be a user-defined numeric constant specified in the FlexibleSUSY model file. The second step is to connect the numeric value of the observable to the SLHA (or FLHA) format, which is the output of the spectrum generator. This is done by the following definition placed in the file meta/Observables/ExampleConstantObservable/WriteOut.m:

Listing 11: Content of ExampleConstantObservable/WriteOut.m.

```
WriteOut`WriteObservable[
   "FlexibleSUSYLowEnergy",
   obs:FlexibleSUSYObservable`ExampleConstantObservable[_]
] := "Re(observables." <> Observables`GetObservableName[obs] <> "(0))";
```

As the first argument of the function WriteObservable shows, the numeric value of the observable is written to the SLHA output block FlexibleSUSYLowEnergy, which matches a corresponding definition in the model file models/M_a/FlexibleSUSY.m. The function returns a string (which must be valid C++ syntax), where we take the real part of the first entry from the array of length 1, where the observable is stored (with the zero-based index convention in C++). Since for the output no information about the observable is needed, one uses the _ pattern in the specification of ExampleConstantObservable. The code listing above leads to the usage of the C++ parser for the SLHA format named FORMAT_ELEMENT, see Listing 4.

3.2.2. Example 2: show fermion masses

Now we turn back to Example 2 from above and connect our observable (two given fermion masses) to the output of the spectrum generator. Similarly to Example 1, one creates the file meta/Observables/ExampleFermionMass/WriteOut.m and places the definitions of the function WriteObservable there. In this example we would like to output two different masses at the same time. To store them internally, we have defined the observable type to be an array of length 2, see the specification of GetObservableType in line 3 of Listing 7. To write the two fermion masses to the output, one could do the following:

Listing 12: Content of ExampleFermionMass/WriteOut.m.

```
WriteOut`WriteObservable[
   "FlexibleSUSYLowEnergy",
   obs:FlexibleSUSYObservable`ExampleFermionMass[_]
] := "Re(observables." <> Observables`GetObservableName[obs] <> "(0))";

WriteOut`WriteObservable[
   "ExampleLeptonMass",
   obs:FlexibleSUSYObservable`ExampleFermionMass[_]
] := "Re(observables." <> Observables`GetObservableName[obs] <> "(1))";
```

Here, we define two distinct behaviours of our observable with two different Les Houches output blocks (which must be reflected by appropriately named lists in models/M_a/FlexibleSUSY.m):

"FlexibleSUSYLowEnergy" and "ExampleLeptonMass". In both cases, definitions lead to the C++ parser named FORMAT_ELEMENT, see Listing 4. The fermion mass in the first entry of our two-component observable array is written to the block FlexibleSUSYLowEnergy, while the second entry is written to the block ExampleLeptonMass. Note that the name of this second block is not a part of the official SLHA standard, but is a non-standard addition we use for this example.

3.2.3. Example 3: lepton self-energy with NPointFunctions

In Example 3, we aim to output the two components of a lepton self-energy. We can use the automatic FLHA output format to achieve this:

Listing 13: Content of ExampleLeptonSE/WriteOut.m.

```
WriteOut`WriteObservable[
   "FWCOEF",
   obs:FlexibleSUSYObservable`ExampleLeptonSE[_[gen_], _]
] :=
StringReplace[
   {
      "leptons, 31, 0, 0, 2, num_value, \"left\"", (* P_L *)
      "leptons, 32, 0, 0, 2, num_value, \"right\"" (* P_R *)
   },
   {
      "num_value" -> "Re(observables." <> Observables`GetObservableName[obs] <> ")",
      "leptons"   -> Switch[gen, 0, "1111", 1, "1313", 2, "1515"]
   }
];
```

In the definition of WriteObservable we provided a pattern for the index of the lepton in the lepton multiplet (gen_) explicitly in the first argument of the observable name ExampleLeptonSE, because the concrete value of this index must be known to select the appropriate self-energy for the output.

3.3. Content of the file O_i/FlexibleSUSY.m*, simple settings for* NPointFunctions

General case

In the previous sections it was described how to define a new observable (done in the file Observables.m) and how to write the numeric value of the observable to the Les Houches output of the spectrum generator (defined in the file WriteOut.m). In this section we describe how to generate the content of the C++ function, that calculates the numeric value of the observable. This C++ function is defined in the C++ template files located in templates/observables/, which contains placeholders that are replaced by expressions to calculate the observable. The rules that specify how to replace the placeholders by appropriate expressions are defined in the observable-specific file meta/Observables/O_i/FlexibleSUSY.m. This file contains at least the function WriteClass, which performs the following tasks:

1. Fill the C++ templates from templates/observables/ with appropriate C++ code.
2. Move the filled C++ templates into models/M_a/observables/.

The following source code listing shows an example for the function WriteClass that does some basic replacements that we will explain in the following:

Listing 14: General content of meta/Observables/O_i/FlexibleSUSY.m.

```
FlexibleSUSY`WriteClass[obs:FlexibleSUSYObservable`Oi, allObs_, files_] :=
Module[
   {
      observables = DeleteDuplicates[Cases[Observables`GetRequestedObservables[allObs], _obs]],
      prototypes = {}, definitions = {},
      npfHeaders = "", npfDefinitions = {}, cxxVertices = {}
   },

   If[observables =!= {},
      Utils`PrintHeadline["Creating " <> SymbolName[obs] <> " class ..."];

      prototypes = TextFormatting`ReplaceCXXTokens[
         "@type@ @prototype@;",
         {
            "@type@"      -> CConversion`CreateCType[Observables`GetObservableType[#]],
            "@prototype@" -> Observables`GetObservablePrototype[#]
         }
      ] &/@ observables;

      (* Task 1: filling definitions *)
   ];

   (* Task 2: filling templates and moving them into models/Ma/observables/ *)
   WriteOut`ReplaceInFiles[
      files,
      {
         "@npf_headers@"           -> npfHeaders,
         "@npf_definitions@"       -> StringRiffle[DeleteDuplicates[npfDefinitions], "\n\n"],
         "@calculate_prototypes@"  -> StringRiffle[DeleteDuplicates[prototypes],     "\n\n"],
         "@calculate_definitions@" -> StringRiffle[DeleteDuplicates[definitions],    "\n\n"],
         "@include_guard@"         -> SymbolName[obs],
         "@namespace@"             -> Observables`GetObservableNamespace[obs],
         "@filename@"              -> Observables`GetObservableFileName[obs],
         Sequence @@ FlexibleSUSY`Private`GeneralReplacementRules[]
      }
   ];

   (* Task 3: returning something to the outside world *)
   {
      "for_outside_usage1" -> ...,
      "for_outside_usage2" -> ...,
      ...
   }
];
```

The function WriteClass has three parameters: the explicit name of the observable (obs), the list of all observables that are requested — by the variable ExtraSLHAOutputBlocks from the file models/M_a/FlexibleSUSY.m — to be calculated (allObs_), and a list of the C++ template file names, where the replacements should be performed, and the corresponding output file names (files_).

The body of the function WriteClass consists of three parts:

1. The first part is the If statement, where all C++ expressions are created (Task 1 as indicated

in the listing above). In general, we are interested in defining one observable `Oi` unified by similar calculations, e.g. we define one observable `Oi` for the set of processes $\ell_i \to \ell_j \ell_k \ell_k^c$ instead of multiple observables $\mu \to 3e$, $\tau \to 3\mu$, etc. Then, we enable specific realizations of `Oi` in `ExtraSLHAOutputBlocks`, see also explicit examples in Listing 31 and 35. So, we start with selecting unique realizations of chosen `Oi` from all `allObs` and storing them into `observables`. Then, inside `If` statement we make changes to the `C++` code generation based on the possible realization-specific features (e.g. the process $\tau \to 3\mu$ might require additional contributions, compared to $\mu \to 3e$). Changes in `C++` prototypes stored in the variable `prototypes` are handled automatically due to the function `WriteObservable`, see Listing 5, while the `C++` definitions in the variable `definitions` usually require manual coding, see Listings 16–18. In the above example source code listing `prototypes` strings are created by replacing the `"@type@"` and `"@prototype@"` tokens by the observable's `C++` type and the prototype of the function that calculates the numeric value of the observable, respectively.

2. In the second part (`Task 2` indicated in the listing) the `C++` tokens (`"@npf_headers@"`, `"@npf_definitions@"`, etc.) are replaced in the `C++` template files (`files`) that are passed to the function with the help of the `ReplaceInFiles` function.

3. The third part (`Task 3`) is to specify the function's returned expression. In the example above the function returns a list of replacement rules, whose use is described below.

   Note that in `Task 2` the tokens in the `C++` template files can be replaced by strings that can in principle be arbitrarily large. However, we recommend to put as much generic information as possible into the `C++` template files and replace the tokens in the template files only by model-specific information. For the latter we recommend to use the full power of `FlexibleSUSY`'s helper routines and functions located in `meta/TextFormatting.m`, `meta/CConversion.m` and `meta/Utils.m`. We refer the reader also to Section 3.5, where we describe the `NPointFunctions` extension and how one can use it to generate `C++` code for amplitudes.

The expression returned by the `WriteClass` function (`Task 3`) should be a list of replacement rules, which gets stored internally in the variable `ObservablesExtraOutput["Oi"]`. The replacement rules stored in the returned list can be accessed and re-used later, if needed. For example, one can access expression stored in the rule defined in line 40 from Listing 14 as follows:

Listing 15: Accessing an expression returned by the `WriteClass` function.

```
expr = Cases[ObservablesExtraOutput["Oi"], ("for_outside_usage1" -> res_) :> res];
```

Besides the possible manual re-use of the returned expressions, `FlexibleSUSY` automatically performs the following two tasks with the returned list of replacement rules:

1. If there exists a replacement rule of the form `"C++ vertices" -> list1`, then the expression `list1` must be a list of vertices that are required to calculate the observable. Each vertex is represented by a list of `SARAH` fields, e.g. `{bar[Chi], Chi, VP}` in the MRSSM (represents the $\bar{\chi}_a^0 \chi_b^0 \gamma$ vertex with two neutralinos and a photon). For each required vertex `FlexibleSUSY` creates a corresponding `C++` function for its numerical evaluation when calculating the numeric value of the observable.

Usage		
`NPointFunction[..., Option_i -> Value_j]`		

Option	Values	Hints
`Regularize`	`MSbar`, `DRbar`	Corresponds to `` FormCalc`Dimension ``: `D`, `4`; can be overriden by `regularization` setting
`ZeroExternalMomenta`	`True`, `False`, `OperatorsOnly`, `ExceptLoops`	External momenta and mass treatment
`OnShellFlag`	`True`, `False`	Usage of on-shell external particles
`UseCache`	`True`, `False`	Cache usage
`Observable`	`None` or $\texttt{O}_\texttt{i}$`[]`	Mode of `NPointFunctions`: simple or advanced, that enables $\texttt{O}_\texttt{i}$`/NPointFunctions.m`
`KeepProcesses`		For `Observable -> None`, see allowed values in the function `GetExcludeTopologies` in the file `meta/NPointFunctions/Topologies.m`
`LoopLevel`	`0` or `1`	Loop level, `FormCalc` limitation

Table 4: Mandatory options for the function `NPointFunction`, see the function `CheckOptionValues` in `meta/NPointFunctions.m` file for allowed values, and `meta/Observables/`$\texttt{O}_\texttt{i}$`/FlexibleSUSY.m` files for examples.

2. If there exists a replacement rule of the form `"C++ replacements" ->` $\texttt{list}_2$, then the expression $\texttt{list}_2$ must be a list of replacement rules that should be applied to all `C++` template files, see Appendix B.3 for an example.

Enabling optional `NPointFunctions` extension, simple settings

If the observable relies on tree-level or one-loop level Feynman diagrams then there is an automated way to generate them in `FlexibleSUSY` with the help of the `NPointFunctions` extension already announced in Ref. [15]. `NPointFunctions` is typically used from within the `WriteClass` function. Internally, the extension calls `FeynArts` [16], `FormCalc` [17], and `ColorMath` [18] to generate analytic expressions and converts them into `C++` form for their numeric evaluation in `FlexibleSUSY`. It thus allows access to Feynman diagrammatic computations in the definition of observables.

`NPointFunctions` can be used in two modes which we will refer to as simple and advanced modes. The modes differ by the accessible settings. Both types of settings serve to modify the calls of the `FeynArts` and `FormCalc` routines: In the simple mode only the settings listed in Table 4 are accessible, which allow topology-indepedent modifications. The advanced mode allows many more settings, enabling in particular to select specific option values for selected topologies. In the present section, we focus on the simple settings from Table 4. These simple settings are also illustrated with the help of Example 3 in Section 3.3.3. Later, the advanced settings are explained in detail in Section 3.5 and exemplified in an extra Section 3.5.7.

`ZeroExternalMomenta` can currently be `True`, `False`, `OperatorsOnly`, and `ExceptLoops`. This option also specifies the way external masses should be treated for specific topologies, fermions

bispinors, and in loop integrals. If set to `True`, all external momenta are set to zero everywhere, while if set to `ExceptLoops`, the scalar products in loop integrals are kept. This allows one to correctly implement the expressions for self-energy-like diagrams relevant for CLFV processes in particular.

`UseCache` stores the `FeynArts` and `NPointFunctions` output for future usage, if enabled. This speeds up the code generation if `model/M_a/` is purged or parts of the meta code are modified.

`KeepProcesses` specifies the mode of `NPointFunctions`. There are two modes to calculate an amplitude: using simple settings only (`Observable -> None`) and with the help of advanced ones defined in `meta/Observables/O_i/NPointFunctions.m` files (`Observable -> O_i[]`), see Section 3.5.[8]

3.3.1. Example 1: output a given number

Each observable undergoes various calculational steps before being evaluated into an actual numerical result. In general, the definitions of the calculational steps are distributed between the `Wolfram Language` file `meta/Observables/O_i/FlexibleSUSY.m` and the `C++` template files discussed. Specifically, the `WriteClass` function in the `FlexibleSUSY.m` file is supposed to generate model-specific expressions to be placed into `C++` template files from Section 3.4.1. As stated before, in general it is recommended to put as much information as possible into the `C++` template files and keep the `WriteClass` function minimal. In case of Example 1, however, the example is so simple that we would like `WriteClass` to generate the complete `C++` code for all calculations (just output a number in this case), thus the `C++` template files will not contain much code. This code generation is done with the definition of `Task 1` and `Task 3` in the `WriteClass` function (by replacing line 20 and line 38 in Listing 14) as follows:

Listing 16: Content of `ExampleConstantObservable/FlexibleSUSY.m`.

```
(* Task 1: generate function definition by replacing tokens by concrete C++ code *)
definitions = TextFormatting`ReplaceCXXTokens["
   @type@ @prototype@ {
      @type@ res {num};
      return res;
   }",
   {
      "@type@"      -> CConversion`CreateCType[Observables`GetObservableType[#]],
      "@prototype@" -> Observables`GetObservablePrototype[#]
   }
] &/@ observables;

...

(* Task 3: return an empty list *)
{}
```

[8]The simple mode with the simple settings discussed here was developed mainly to allow users a simple starting point. It is also used in unit testing of `FlexibleSUSY`. Currently, `FlexibleSUSY` performs unit tests of self-energy expressions in the SM/MSSM and Z-boson penguins in the MRSSM. It is also used in Example 3 to demonstrate the basic usage of the `NPointFunctions` package and can serve to write the first iterations of the code that relies on `NPointFunctions`. Typically, the code for actual physics observables such as the ones discussed in Section 2 will use the advanced settings of `NPointFunctions`.

In the above source code listing the variable definitions is defined to contain the entire definition of the function that calculates the observable (including function body). It is generated by replacing the @type@ token by the concrete C++ type Eigen::Array<std::complex<double>,1,1> and the function name @prototype@ (including its argument double num) as specified by the definitions of Section 3.1.1 in a generic string. In the body of the function, we initialize a local variable res of type @type@ and set its value to the value of num. The value of res is then returned to further fill the SLHA block entries. The final function definition in the definitions variable will be used later in Section 3.4.1 together with the C++ template files to construct the complete C++ code to output the numerical value of the observable.

The WriteClass function finally returns an empty list (Task 3), because we do not need any of the defined expressions anywhere else in this example.

Note that the function definition generated in Task 1 is later put into the C++ template files in templates/observables/ an is thus closely connected to their content, as shown in Section 3.4.1.

3.3.2. Example 2: show fermion masses

Now we return to Example 2 where we output fermion masses. In this example we follow the general recommendation to put as much C++ code as possible into the C++ template files and as little as possible into FlexibleSUSY.m. Thus, in this example we fill the following code into the WriteClass function in the FlexibleSUSY.m file of Listing 14:

Listing 17: Content of ExampleFermionMass/FlexibleSUSY.m.

```
(* Task 1: generate function definition by replacing tokens by concrete C++ code *)
definitions = TextFormatting`ReplaceCXXTokens["
   @type@ @prototype@ {
      return forge<@type@, fields::@fermion@>(gen, model, qedqcd);
   }",
   {
      "@fermion@"   -> SymbolName[Head[First[#]]],
      "@type@"      -> CConversion`CreateCType[Observables`GetObservableType[#]],
      "@prototype@" -> Observables`GetObservablePrototype[#]
   }
] &/@ observables;

...

(* Task 3: return an empty list *)
{}
```

Again, the variable definitions contains the definition for each C++ function for the observable calculation of the function that calculates the numerical value of the observable. This function has the name and type that were specified via GetObservablePrototype and GetObservableType, respectively. In this example the function body contains only a single line, which contains a call of the template function forge. By this call we delegate the computation of the observable to the forge function, which is defined entirely at the C++ level in Section 3.4.2. The same strategy is applied in several of the predefined observables discussed in Section 2 such as $\ell_i \to \ell_j \ell_k \ell_k^c$ or μ–e conversion. The template parameter fields::@fermion@ of the forge function above is an internal C++ type that represents the fermion whose mass shall be output (in the MRSSM, for example, the token @fermion@ is replaced by Fe).

Again, we do not want to use any expressions defined in the `WriteClass` function anywhere else, so we return an empty list from the function (`Task 3`).

3.3.3. Example 3: lepton self-energy with `NPointFunctions` (`Observable -> None`)

This example illustrates the use of the `NPointFunctions` extension of `FlexibleSUSY`. We will use the `NPointFunctions` extension in the simplified mode, where we do not make use of the advanced settings in `NPointFunctions.m` and just specify the option `Observable -> None`. The following code snippet shows the content of the `WriteClass` function:

Listing 18: Content of `ExampleLeptonSE/FlexibleSUSY.m` with the option `Observable -> None`.

```
(* Task 1: generate function definition by replacing tokens by concrete C++ code *)

(* Task 1a: delete duplicates and ignore fermion generation index *)
observables = DeleteDuplicates[observables /. f_[_Integer] :> f[_]];

(* Task 1b: run NPointFunction *)
Module[{field, contr, npf, basis, name},
   field = Head[First[#]];
   contr = Last[#];
   npf = NPointFunctions`NPointFunction[
      {field}, (* Incoming particles *)
      {field}, (* Outgoing particles *)
      NPointFunctions`UseCache            -> False,
      NPointFunctions`OnShellFlag         -> True,
      NPointFunctions`ZeroExternalMomenta -> True,
      NPointFunctions`LoopLevel           -> 1,
      NPointFunctions`Regularize          -> FlexibleSUSY`FSRenormalizationScheme,
      NPointFunctions`Observable          -> None,
      NPointFunctions`KeepProcesses       -> {Irreducible}
   ];

   basis =
   {
      "left_wilson"  -> NPointFunctions`DiracChain[SARAH`DiracSpinor[field[{SARAH`gt2}], 0,
↪ 0], 7, SARAH`DiracSpinor[field[{SARAH`gt1}], 0, 0]],
      "right_wilson" -> NPointFunctions`DiracChain[SARAH`DiracSpinor[field[{SARAH`gt2}], 0,
↪ 0], 6, SARAH`DiracSpinor[field[{SARAH`gt1}], 0, 0]]
   };

   npf = WilsonCoeffs`InterfaceToMatching[npf, basis];
   name = "se_irr";

   AppendTo[cxxVertices, NPointFunctions`VerticesForNPointFunction[npf]];
   AppendTo[npfDefinitions, NPointFunctions`CreateCXXFunctions[npf, name, Identity, basis
↪ ][[2]]];
   AppendTo[definitions,
      TextFormatting`ReplaceCXXTokens["
         @type@ @prototype@ {
            const auto npf = npointfunctions::@name@(model, {gen, gen}, {});
            return {npf[0], npf[1]};
         }",
         {
            "@type@"      -> CConversion`CreateCType[Observables`GetObservableType[#]],
            "@prototype@" -> Observables`GetObservablePrototype[#],
```

```
42              "@name@"       -> name
43          }
44       ]
45    ];
46 ] &/@ observables;
47
48 (* Task 1c: obtain list of C++ header files *)
49 npfHeaders = NPointFunctions`CreateCXXHeaders[];
50
51 ...
52
53 (* Task 3: returning something to the outside world *)
54 {"C++ vertices" -> Flatten[cxxVertices, 1]}
```

First of all, as `NPointFunctions` returns the same code for different generations of external particles, we remove potential duplicates by replacing integer generation with the _ pattern for simplicity (it could have been any symbol or number), see `Task 1a`.

Afterwards, we run `NPointFunctions` (`Task 1b`) to generate the relevant `C++` code, and all required `C++` vertices. For this we create a local function (defined as a `Module`), which we map to all observables in the `observables` list. Note, however, that in more involved calculations it may be better to define a non-local, separate function to process all `observables`, instead of a local one via a `Module`, as done here. The local function to obtain the `C++` code to numerically evaluate the observable(s) stores its results in the following local variables:

`field` represents the lepton particle that we specify in the definition of the observable (like `Fe` for example in the SM or MRSSM).

`contr` contains an expression that allows one to select certain contributions from the self-energy that should be output. In this example we do not make use of this selection feature. See Section 3.5.7 for an advanced example.

`npf` contains the main result of `NPointFunctions`: an object with generic amplitudes and, sometimes, replacement rules. See Table 4 for a list of all possible options. In this example, we calculate the `Fe -> Fe` amplitude, from which we will extract the self-energy (note that the outgoing particle is typed as `Fe` and not `bar[Fe]`). We do not store the results in the cache. `OnShellFlag` is used internally by the option `FormCalc`OnShell` for the function `FormCalc`CalcFeynAmp`. `ZeroExternalMomenta` is passed to the function `FormCalc`OffShell`. The loop level is specified explicitly to be `1`. The renormalization scheme is decided to be set by `FlexibleSUSY`.

We chose a simple operation mode of the `NPointFunctions` extension by setting `Observable -> None`. This implies, that no optional file O_i`/NPointFunctions.m` with advanced settings is used, see Section 3.5 for more details and the usage example in Section 3.5.7. The option `KeepProcesses` allows selecting certain topologies, provided by the list of default values within the function `GetExcludeTopologies` in `meta/NPointFunctions/Topologies.m`. To extend this list, we refer to the usage of `FeynArts`$ExcludeTopologies`.

`name` contains the `C++` name of `NPointFunctions`-generated function that calculates the numerical value of the observable.

basis is a list of replacement rules to extract certain sub-expressions of the amplitude and store them in local C++ variables. The rhs. of each replacement rule defines the expression, whose prefactor should be extracted. The lhs. defines the name of the C++ variable in which this prefactor shall be stored. Note that patterns are currently not supported to select the sub-expression. The replacement rules stored in the basis variable should be passed to the InterfaceToMatching function, which performs the extraction of the prefactors.[9]

cxxVertices contains the list of all vertices that are required to numerically calculate the observable. They have to be returned from the WriteClass function (see Task 3), because FlexibleSUSY stores the C++ code for the vertices in other files.

npfDefinitions contains strings with all generated amplitude-related C++ functions required for the numerical evaluation of the amplitude. The function CreateCXXFunctions converts the NPointFunctions object into C++ code. The user provides the name of the C++ function and specifies which color structures are expected (Identity or SARAH`Delta).

definitions contains the C++ function that calculates the numerical value of the observable with the help of C++ code created by the NPointFunctions extension. The function body consists of a call to the generated irr_se function (the function name is stored in the name variable), to which the model parameters (model object) and the indices of the incoming and outgoing particles (both are gen here) are passed. The last argument contains the set of momenta of the external particles. In the current implementation external momenta are assumed to be zero and so the last function argument should be an empty initializer list {}.

Since basis contains two replacement rules that define prefactors of certain Lorentz structures, irr_se will return a two-valued array. The generated function eventually returns the two components of this array.

In Section 3.5.7 the example described here is extended to use advanced NPointFunctions options.

3.4. Content of the C++ template files

General case

The function WriteClass defined in the file meta/FlexibleSUSY.m in the previous section fills C++ templates with model-specific information, and places the resulting files into the directory models/M_a/. The content of the C++ template files is discussed in this section.

For each observable one has to create two C++ template files in the templates/observables directory: a .hpp.in and .cpp.in file. These files may contain the tokens that are replaced by the WriteClass function. The following two source code listings show generic examples of these template files:

[9]Sometimes the question arises how to know the exact form of the expressions in basis? As a practical recommendation, one can place the Print[npf]; statement right before the definition of basis and then iteratively run FlexibleSUSY several times to figure out the exact form of the basis elements. In general, they have to explicitly represent the structures, no patterns are supported here at this moment.

Listing 19: General content of C++ header template `templates/observables/@O_i_filename@.hpp.in`.

```
#ifndef @ModelName@_@include_guard@_H
#define @ModelName@_@include_guard@_H
#include "lowe.h"
// Auxiliary #include directives could come here

namespace flexiblesusy::@namespace@ {

// Auxiliary observable-specific expressions could come here

// declaration of the function that calculates the observable
@calculate_prototypes@

} // namespace flexiblesusy::@namespace@

#endif
```

Listing 20: General content of C++ definitions template `templates/observables/@O_i_filename@.cpp.in`.

```
#include "@ModelName@_mass_eigenstates.hpp"
#include "cxx_qft/@ModelName@_qft.hpp"
#include "@ModelName@_@filename@.hpp"
@npf_headers@
// Auxiliary #include directives could come here

namespace flexiblesusy {
namespace @ModelName@_cxx_diagrams::npointfunctions {

@npf_definitions@

} // namespace @ModelName@_cxx_diagrams::npointfunctions

using namespace @ModelName@_cxx_diagrams;
namespace @namespace@ {

// Auxiliary observable-specific expressions could come here

// Definition of the function that calculates the observable
@calculate_definitions@

} // namespace @namespace@
} // namespace flexiblesusy
```

Both template files contain C++ tokens (e.g. `@calculate_definitions@`) to be replaced by the `WriteClass` function, defined in `FlexibleSUSY.m` during `FlexibleSUSY`'s meta phase. For example, `@npf_headers@` will be replaced with the content of `npfHeaders` and similar for other tokens, see lines 27–34 of Listing 14. If no replacement rule for a token is found, the token is replaced by an empty strings.

The numerical calculation of the observable might require more C++ auxiliary functions or classes. To use extra functions/classes one could add appropriate preprocessor `#include` directives.

Note again, that there is some freedom to move code between the `O_i/FlexibleSUSY.m` and `@O_i_filename@.cpp.in` files. However, as stated before, we recommend to move as much C++

code as possible into the dedicated directory templates/observables/, as C++ source code is often more robust compared to Wolfram Language scripts due to static type checking.

3.4.1. Example 1: output a given number

To output a single number we create two C++ template files in templates/observables/, as stated above. They will be filled by the WriteClass function defined in FlexibleSUSY.m created in Section 3.3.1. The content of the header template file example_constant_observable.hpp.in is the same as in general Listing 19. In principle, we can fill the C++ definitions template file with the default content provided in Listing 20. Nevertheless, as we do not use the NPointFunctions module for the observable calculation (realized with lines 3–6 in Listing 16), we can simplify the file as follows:

Listing 21: Content of C++ definitions template example_constant_observable.cpp.in.

```
#include "@ModelName@_@filename@.hpp"

namespace flexiblesusy::@namespace@ {

@calculate_definitions@

} // namespace flexiblesusy::@namespace@
```

In this file, the template C++ token @calculate_definitions@ will be replaced with the content of the variable definitions defined in the file O_i/FlexibleSUSY.m described in Section 3.3.1. In this way, during the meta phase of FlexibleSUSY the complete C++ code will be generated for the function calculate_example_constant_observable (defined in Listing 6) which outputs the number num. We do not need to provide any additional observable-specific code, as everything is already provided by the WriteClass function in O_i/FlexibleSUSY.m during the Wolfram Language meta phase.

3.4.2. Example 2: show fermion masses

As stated above, we need to provide two C++ template files in templates/observables/. They will be filled during the FlexibleSUSY meta phase and embedded into the rest of the C++ spectrum generator. In this example the content of the template header file example_fermion_mass.hpp.in is the same as in the general Listing 19. The template file example_fermion_mass.cpp.in, however, contains two changes compared to the generic example from Listing 20. First of all, all commands related to NPointFunctions can be omitted, since we do not use Feynman diagrammatic calculations in this example (like in Example 1). Apart from this, our goal with this example is to demonstrate how one can move as many calculations as possible to the C++ template file, while keeping the flexibility to insert model-specific information. As discussed in Section 3.3.2, we prepared for this by having a minimal function body defined on the Wolfram Language level in the file ExampleFermionMass/FlexibleSUSY.m, where the function body only calls another C++ function named forge. This function shall now be defined. The following source code listing shows the content of the C++ template file that contains this definition:[10]

[10]Instead of the switch statement one can also use qedqcd.displayLeptonPoleMass(idx);.

Listing 22: Content of C++ definitions template example_fermion_mass.cpp.in.

```
#include "@ModelName@_mass_eigenstates.hpp"
#include "cxx_qft/@ModelName@_qft.hpp"
#include "@ModelName@_@filename@.hpp"
#include "error.hpp"

namespace flexiblesusy {
using namespace @ModelName@_cxx_diagrams;
namespace @namespace@ {

template <typename RTYPE, typename FIELD>
auto forge(int idx, const @ModelName@_mass_eigenstates& model, const softsusy::QedQcd& qedqcd)
{
   context_base context {model};
   auto context_mass = context.mass<FIELD>({idx});

   std::complex<double> lepton_mass;
   switch (idx) {
      case 0:  lepton_mass = qedqcd.displayPoleMel();
               break;
      case 1:  lepton_mass = qedqcd.displayPoleMmuon();
               break;
      case 2:  lepton_mass = qedqcd.displayPoleMtau();
               break;
      default: throw OutOfBoundsError("fermion index out of bounds");
   }
   RTYPE res {context_mass, lepton_mass};
   return res;
}

@calculate_definitions@

} // namespace @namespace@
} // namespace flexiblesusy
```

In the code listing above, the template C++ token @calculate_definitions@ will be replaced by the content of the variable definitions, defined in the ExampleFermionMass/FlexibleSUSY.m file in Section 3.3.2, which calls the forge function. The forge function defined above illustrates how to access two different types of particle masses. First, the running $\overline{\text{MS}}/\overline{\text{DR}}$ mass of the fermion specified by FIELD and idx (which can correspond to, e.g. Fe[2] or Fd[3] at the Wolfram Language level in the selected BSM model) is obtained by calling the mass function template. Afterwards, the pole mass of the SM lepton of generation idx is obtained from the qedqcd object. The forge function finally returns an array of length 2 containing these two masses.

3.4.3. Example 3: lepton self-energy with NPointFunctions

As in the examples above, we have to create two C++ template files that are responsible for the numerical calculation of the observable. This example uses NPointFunctions to generate analytical expressions for the self energies, but it is otherwise a structurally simple example. Hence, like in Example 1 (but unlike Example 2) we do not delegate computations to a forge function; rather, the main definition of the observable is done by the function WriteClass in FlexibleSUSY.m created in Section 3.3.3. For this reason we use the C++ template files given in Listing 19–20 without

Option	Usage	Purpose and hints
`topologies`	See Table 6	Select topologies based on adjacency matrices
`diagrams`	See Table 7	Exclude contribution based on generic fields
`amplitudes`	See Table 7	Remove contribution based on class insertions
`chains`	See Table 8	Specify operators to neglect
`order`	See Section 3.5.5	Select fermion order
`sum`	See Section 3.5.6	Define particle generations to skip on the `C++` level
`regularization`	See Section 3.5.6	Change scheme for a specified topologies
`momenta`	See Section 3.5.6	Eliminate given external momenta for a given topology
`mass`	See Section 3.5.6	Treat masses in an selected way

Table 5: Purpose of all available advanced settings and their usage.

any changes in this example. Note that these general template files are generally appropriate for observables that use the `NPointFunctions` extension, hence we can keep the present subsection very short.

We recall that in this example we use `NPointFunctions` in the simple mode specified by the option `Observable -> None`; the example will be continued in Section 3.7.3, which shows how to enable and use the computation of the self-energy in a concrete spectrum generator. An alternative version of the example is provided in Section 3.5.7, where the example is modified to illustrate the use of the advanced settings of `NPointFunctions`.

3.5. Content of the optional file `Oi/NPointFunctions.m`, advanced settings

The `NPointFunctions` extension allows to generate Feynman diagrammatic calculations to obtain analytical expressions for observables in any specific model `Ma` when the `FlexibleSUSY` meta phase is executed. The simple usage of `NPointFunctions` was demonstrated in Section 3.3.3, but frequently it is necessary to fine-tune calculations done with `NPointFunctions`. Such a fine-tuning is possible via advanced settings of `NPointFunctions` explained in this section. It may involve the selection of subsets of Feynman diagrams, remove contributions, selecting the regularization scheme, or specifying the fermion order in e.g. four-fermion amplitudes. Parts of the fine-tuning may be accessible to users of the observable (such as the arguments `Vector`, `Scalars`, etc. of observables in Tables 2–3), other parts may be found only in the definition of the advanced settings.

In the following we begin by explaining how to enable the advanced settings and giving an overview, then we will explain all advanced settings and finally provide an example.

3.5.1. Enabling advanced settings, overview

We begin by explaining how to enable and access the advanced settings when setting up the definition of an observable. The `NPointFunctions` extension is called from the observable-specific file `Oi/FlexibleSUSY.m` discussed in Section 3.3. Listing 18 combined with Listing 14 provides an example for using `NPointFunctions` in its simple mode without advanced settings. To use the advanced mode, this file must be modified as follows:

Listing 23: Modifications in `O_i/FlexibleSUSY.m` for the option `Observable -> O_i[]`.

```
(* Instead of lines 18 and 19 in Listing 18 *)
NPointFunctions`Observable    -> obs[],
NPointFunctions`KeepProcesses -> If[Head[contr] === List, contr, {contr}]
```

Here the line `Observable -> O_i[]` enables the advanced settings (note that the variable `obs` was defined in the first line in Listing 14). This option requires the existence of the corresponding file `meta/Observables/O_i/NPointFunctions.m` which should contain the detailed advanced settings described further below.

The second line defines the option `KeepProcesses` via the variable `contr` (instead of the hard-coded value `Irreducible` in Listing 18) and opens up an important way to access advanced settings. To explain the meaning of this line we briefly recall the overall structure of setting up an observable. From the user's perspective, an observable ultimately is called as described in Section 2, and observables may have options `contr`, see e.g. Tables 2–3, with possible values being a keyword or a list of keywords such as `Vectors`, `Scalars`, `Boxes`, etc. Using these options influences how the observable is evaluated. The definition of such keywords and how they influence the observable is part of the advanced settings of `NPointFunctions`.

Using such a `contr` option has to be prepared in the file `O_i/Observables.m` as exemplified in Section 3.1.3 by specifying the appropriate argument in the definition of the observable. Via line 9 of Listing 18 and line 3 of Listing 23 the variable `contr` propagates into the option of `KeepProcesses`. The construction in the code makes sure that the option is always a list.

The advanced settings are listed in Table 5 and explained in detail in the following Sections 3.5.2–3.5.6. As an overview, `NPointFunctions` relies on `FeynArts` and `FormCalc` to produce `Wolfram Language` expressions for model-specific class-level amplitudes, and the advanced settings fine-tune the calls to `FeynArts` and `FormCalc` routines, so familiarity with these tools is required to some extent.[11] Among the advanced settings, `topologies`, `diagrams` and `amplitudes` can be used to define the keywords which can be set by users as values of `contr` as explained above; these keywords then influence the execution of the functions `FeynArts`CreateTopologies`, `FeynArts`InsertFields`, and `FormCalc`CalcFeynAmp`. The further advanced settings influence the calls to `FeynArts` and `FormCalc` and manipulate their output in ways which are fixed for the observable and can be influenced only by directly modifying the file `O_i/NPointFunctions.m`.

3.5.2. `topologies`

The observable calculation by the `NPointFunctions` extension starts from the generation of topologies. It is possible to select particular topologies in the calculation by using the option `KeepProcesses` in line 3 of Listing 23 as explained above. The possible values of `contr` are (lists of) keywords: `Vectors`, `Scalars`, `TreeLevelSChannel`, etc.

In this section we focus on how to define such keywords, generally now called *`Contribution_i`*, and to associate them with topologies of Feynman diagrams. Table 6 shows the required command in the file `O_i/NPointFunctions.m`. It connects each keyword *`Contribution_i`* with certain

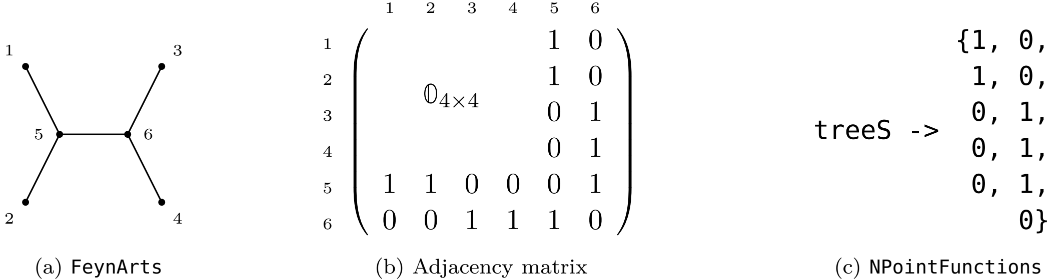
[11] Nevertheless, due to `C++ template` capabilities, we use these packages in a non-standard way. For example, the particle content of the model, vertices, and masses are stored as `C++` structures. This allows the `C++` compiler to substitute all particle classes and sum over generation indices. Thus, the amplitudes are modified before the usage of `FormCalc` to avoid particle-level computations. In addition, all color factors for amplitudes are computed with `ColorMath`.

Usage

```
topologies[LoopNumber] = {
   Contribution_i -> TopologyName_j,
   ...
};
```

Abbreviation	Values	Hints
`LoopNumber`	`0` or `1`	`FormCalc` limitation
		Symbols for CLFV observables: `contr` in Tables 2–3
`Contribution` $_i$	∀ `Symbol`	Synonyms: in `O`$_i$`/FlexibleSUSY.m`
		Connection to topologies: in `O`$_i$`/NPointFunctions.m`
`TopologyName` $_j$	∀ `Symbol`	Definitions in `meta/NPointFunctions/Topologies.m`

Table 6: Syntax and options values for `topologies`.



(a) `FeynArts` (b) Adjacency matrix (c) `NPointFunctions`

Figure 2: The connection between `FeynArts` topology, adjacency matrix, and `NPointFunctions` name `treeS`.

topology names. The latter uniquely define topologies by connections to adjacency matrices, as shown in Figure 2. This explicit naming of topologies is useful, as they can also be used to modify diagrams and amplitudes, as described in the following subsections.

The name of a topology can be any `Wolfram Language` symbol that has to be connected with the actual adjacency matrix. All relevant topology names can be defined in a separate file `meta/NPointFunctions/Topologies.m`, as follows:

Listing 24: Example for definitions of topologies (in `meta/NPointFunctions/Topologies.m`).

```
AllTopologies[{2, 2}] = {
   treeS   -> {1,0,1,0,0,1,0,1,0,1,0},
   treeT   -> {1,0,0,1,1,0,0,1,0,1,0},
   treeAll -> {treeS, treeT}
};
```

Above, the `{2, 2}` argument represents the number of incoming and outgoing particles. The definitions are stored in a list of key–value pairs of two kinds:

1. A connection between the user-given name of topology, e.g. `treeS`, and `NPointFunctions` internal representation of this specific topology. It is obtained in a few steps, as follows

Usage

```
diagrams[LoopNumber, WhenToApply] = { (* Or amplitudes[...] *)
   Contributioni -> {
      TopologyNamej -> {Descriptiona, Commandb},
      ...
   },
   ...
};
```

Abbreviation	Values	Hints
LoopNumber	`0` or `1`	As in Table 6
WhenToApply	`Present` or `Absent`	Apply $\mathit{Command}_b$ for $\mathit{TopologyName}_j$ if $\mathit{Contribution}_i$ is present (absent) in `KeepProcesses`, see Table 4
$\mathit{Contribution}_i$	∀ `Symbol`	As in Table 6
$\mathit{TopologyName}_j$	∀ `Symbol`	As in Table 6
$\mathit{Description}_a$	∀ `String`	Printed in the terminal during meta phase
$\mathit{Command}_b$	Special syntax	See the text description

Table 7: Syntax and options for `diagrams` and `amplitudes`.

from Figure 2. One draws the `FeynArts` topology with explicit vertex numbers, creates an adjacency matrix, and eliminates all entries with redundant information: the zero top-left submatrix representing propagators between external particles, and entries below the diagonal as the adjacency matrix is symmetric. The rest is combined line by line into a one-dimensional list that is stored in `meta/NPointFunctions/Topologies.m`. These steps are done by the function `AdjaceTopology` defined there.

2. A connection between a set of simple topologies and their group name, like `treeAll`. This is useful to shorten the application of other settings.

If the topology names are defined like this, an example content of the file $\mathtt{O}_\mathtt{i}$`/NPointFunctions.m` might define the keyword `TreeLevelSChannel` and associate it to the topology `treeS` as follows:

Listing 25: Example content of $\mathtt{O}_\mathtt{i}$`/NPointFunctions.m`.

```
topologies[0] = {
   TreeLevelSChannel -> treeS
};
```

3.5.3. `diagrams` *(generic level modifications)*

The keywords $\mathit{Contribution}_i$ can also be associated with generic-level field patterns (in `FeynArts` nomenclature) to remove certain generic-level fields in the Feynman diagrams. Table 7 shows the required command in the file $\mathtt{O}_\mathtt{i}$`/NPointFunctions.m` to define the associations with field patterns. The logic is the same as for `topologies`, but the main new ingredient is the $\mathit{Command}_b$ option. It defines the kind of generic fields that should be removed, and it should be a pure function which returns `True` or `False` and which takes several arguments while traversing the

tree of FeynArts insertions: FeynArts`TopologyList, the current topology, and the current class level insertion. Apart from standard Wolfram Language expressions, the following ingredients may be used to specify *Command*$_b$:[12]

- Fields appearing in tree-parts of diagrams are specified by TreeFields.
- Fields appearing in loops are specified by LoopFields.
- Any generic field of scalar type FeynArts`S, fermion type FeynArts`F or vector boson type FeynArts`V.
- A specific field, derived from external particles. For example, one can remove from the diagram fields that correspond to the external particles numbered 1 or 3 with the usage of the argument FieldPattern[#, 1|3] in the function FreeQ below.

Altogether, an example is given by the setting:

```
diagrams[1, Present] = {
   Vectors -> {penguinT -> {"no 1,3 particles", FreeQ[LoopFields[##], FieldPattern[#, 1|3]]&}}
};
```

Then, the function call NPointFunction[..., KeepProcesses -> {Vectors}] deletes the diagrams with the topology penguinT when the external particles 1, 3 do appear inside the loop. A second example, demonstrating the Absent mode of *WhenToApply*, is given by:

```
1 diagrams[1, Absent] = {
2    Vectors -> {penguinT -> {"no tree vectors", FreeQ[TreeFields[##], FeynArts`V]&}},
3    Scalars -> {penguinT -> {"no tree scalars", FreeQ[TreeFields[##], FeynArts`S]&}}
4 };
```

In this way, if KeepProcesses -> {Vectors} is specified (thus Scalars is *not* specified), the third line of the listing applies and eliminates diagrams of penguinT topology which contain scalar particles in their tree-level parts.

3.5.4. amplitudes (class level modifications)

Often, modifications on the level of topologies and generic-level amplitudes allowed by the topologies and diagrams settings, together with other options of Table 4, provide sufficient fine-tuning. Sometimes, however, removing amplitudes on the classes-level is required. This is possible via specifying the amplitudes[*LoopNumber*, *WhenToApply*] setting. The syntax is identical to the one of diagrams and given in Table 7. As an example, amplitudes with massless particles can require special treatment, and the following setting allows to remove them from the generated expressions (assuming they will be handled properly elsewhere):

```
amplitudes[1, Present] = {
   Vectors ->
      {penguinT -> {"no tree photons", FreeQ[{##}, InternalMass[FeynArts`V, 5] -> 0]&}}
};
```

[12]The syntax of *Command*$_b$ follows from the internal structure of the diagram used by the NPointFunctions extension, and for more details we refer to the file meta/NPointFunctions/Settings.m shipped with FlexibleSUSY.

Usage		

`chains[LoopNumber] = {`
`   Momenta_i -> {Rule_j, ...},`
`   ...`
`};`		

Abbreviation	Values	Hints
`LoopNumber`	`0` or `1`	As in Table 6
`Momenta`$_i$	∀ `Symbol` from `ZeroExternalMomenta`	See Table 4
`Rule`$_j$	Special syntax	See `DiracChains.m`

Table 8: Syntax and options for `chains`. Files in column "Hints" are located in `meta/NPointFunctions/`.

3.5.5. `order`, `chains` *(Dirac algebra modifications)*

`NPointFunctions` can be used to extract Wilson coefficients. For observables with external fermions, there might exist a need to change or fine-tune fermion chains to achieve the desired operator structure. This is done by the settings `order` and `chains`, where both the desired fermion order of external fermions as well as chains to be dropped are specified.

The syntax for `order` can be understood as follows. Imagine that we would like to obtain Wilson coefficients corresponding to the process $\mu^- \to e^- e^- e^+$, or more conveniently for the crossed $2 \to 2$ process $\mu^- e^- \to e^- e^-$ instead, where the outgoing positron is replaced by an incoming electron. That means we need to obtain coefficients of expressions $\bar{u}(e^-)\Gamma_i u(\mu^-)\, \bar{u}(e^-)\Gamma_j u(e^-)$, where Γ represent structures involving Dirac matrices and momenta. For example, in the source code for the observable `BrLTo3L`, the `NPointFunctions` extension is accordingly called as

```
NPointFunction[{lep, lep}, {lep, lep}, ...]
```

where the variable `lep` will correspond to leptons during the meta phase execution for concrete models. Interpreting the incoming particles as muon and electron, we specify the required fermionic structure by the following line in the file `NPointFunctions.m` (the field numbers are as in `FeynArts`InsertFields`):

```
order[] = {3, 1, 4, 2};
```

Once the fermionic order is fixed by `order`, and amplitudes are calculated, one obtains multiple chains consisting of Dirac matrix products. Often certain Dirac chains may be simplified or neglected, thanks to the desired level of precision or the choice of basis of Wilson coefficients. Dropping specific types of Dirac chains can be implemented by using the `chains` setting described in Table 8. An example code which serves to neglect expressions proportional to the mass of the electron from the example above is given by:

```
chains[1] = {
   ExceptLoops -> {1[k[4|2], ___] -> 0, 2[k[3|1], ___] -> 0}
};
```

In general, the `Rule`$_j$ appearing on the right-hand side of this example (or the general case in Table 8) must be of the form

```
ChainNumber[Entry1, Entry2, ...] -> 0
```

The syntax can be described as follows (the source code which interprets these expressions is in the file `meta/NPointFunctions/DiracChains.m`, and further details can be found there):

`ChainNumber` is an integer which defines a chain number, e.g. for the fermion order `{3,1,4,2}` the chain numbered `1` consists of particles `{3,1}` and chain numbered `2` is `{4,2}`.

`Entry1` corresponds to a pattern for a Dirac chain which may be a non-commuting product of projection operators $P_{L,R}$, γ^μ matrices with open indices or γ^μ matrices contracted with external momenta. We mimic `FormCalc` notation inside the Dirac chains so that the integer `6` represents P_R, `7` is P_L, a negative number leads to the antisymmetrized chain, `k[`i`]` means γ_μ`k[`i`]`$^\mu$ with `k[`i`]`$^\mu$ being external momenta of ith particle. Further, the short form `l[`i`]` is converted to `Lor[`i`]` and represents a Lorentz index connecting two Dirac chains. Accordingly, *`Entry1`* can take the following values: `6`, `-6`, `7`, `-7`, `k[`i_1`]`, `l[`i_1`]` (`k` and `l` may have several integer arguments simultaneously such as `k[`i_1`|`i_2`|...]`, `l[`i_1`|`i_2`|...]`). In the case of `k[`i_1`|`i_2`|...]` and `l[`i_1`|`i_2`|...]`, all projection operators are prepended automatically so that `2[k[3|4]]` becomes `2[6|-6|7|-7, k[3|4]]`; finally i_j correspond to the integer numbers of external particles.

`Entry2` and further entries are similar but may take only the following values: `k[...]`, `l[...]`, `_`, `__`, `___`.

3.5.6. Other modifications

Here we briefly describe and exemplify a set of settings which are typically of minor importance.

In some cases, one should not sum over all generations for some generic field in the amplitude. For example, in CLFV observables, there often appear penguin contributions with external self-energy-like corrections. In such diagrams, the fermion in the tree-level propagator should differ from the external fermion. This behavior can be defined via the setting `sum`:

```
sum[1] = {
   inSelfT -> {"skip initial lepton", {6, Field[#, 1]&}}
};
```

This changes the expressions at the loop level *`LoopNumber`* `= 1` in the following way. For the topology identified as `inSelfT` one modifies the summation over generic fields in the propagator under `FeynArts` number `6`, so that the sum over the field being equal the first external particle is omitted on `C++` level.

The setting `momenta` can be used to eliminate certain external momenta by using momentum conservation for a given topology, for instance

```
momenta[1] = {
   penguinT -> 2
};
```

This modifies the options for the function `FormCalc`CalcFeynAmp` so that for *`LoopNumber`* `= 1` and topology `penguinT` the momenta of the second external particle is replaced by the momenta conservation expression.

The setting `mass` allows to neglect masses, and improve the readability of the code, if desired:

```
mass[1] = {
   inSelfT -> {"explicit final lepton mass", InternalMass[FeynArts`F, 6] :> ExternalMass[3]},
   inSelfT -> {"keep initial lepton mass untouched", Hold :> ExternalMass[1]}
};
```

The first example appends an additional replacement rule that allows the use of the explicit expression for the mass of the particle in the generic propagator. The second one prevents some simplification which would otherwise lead to incorrect amplitude expressions, see the explicit implementation for more details.

Finally, `regularization` overrides the option `FormCalc`Dimension` for given topologies:

```
regularization[1] = {
   boxS -> D,
   boxU -> D
};
```

This option might be useful to obtain the desired Wilson coefficients faster or more optimally: e.g. sometimes the option `chains` might be skipped, as the default `FormCalc` setting has already produced the required expressions.

3.5.7. Example 3: lepton self-energy with `NPointFunctions` (`Observable -> O_i[]`)

Let us exemplify how one enables the usage of the advanced settings for `NPointFunctions`, by modifying Example 3 which calculates fermion self energies. We continue from Section 3.3.3 and Section 3.4.3, where the self-energy calculation was defined by using only the simple mode of `NPointFunctions`. In order to use the advanced settings, at first we need to apply the following modifications in the file `FlexibleSUSY.m`:

Listing 26: Modifications in `ExampleLeptonSE/FlexibleSUSY.m` for the option `Observable -> O_i[]`.

```
NPointFunctions`Observable    -> obs[],
NPointFunctions`KeepProcesses -> If[Head[contr] === List, contr, {contr}]
...
name = "se_" <> CConversion`ToValidCSymbolString[contr];
```

Here the first lines are as explained in Section 3.5.1 and enable the advanced settings, including the variable `contr` which allows users later to select different Feynman diagrams for the evaluation of the self energies via keywords corresponding to different $\texttt{Contribution}_i$. In the last line of the listing, the `name` of the function is changed to reflect different possible values of the variable `contr`.

The goal in this example is to allow users to compute "self energies" by including either only one-particle irreducible diagrams, or only diagrams with a tadpole part, or both. Hence we intend to use the advanced settings to define three keywords and associate them with the appropriate topologies via the `topologies` setting described in Table 6.

In the following we describe a typical workflow how one might interactively obtain the relevant information to create the advanced settings file `meta/Observables/O_i/NPointFunctions.m` which achieves this. We begin with the definition of topologies we want to enable/disable for the calculation.

As the self-energy process is $1 \to 1$, we start by looking for already defined topologies inside `meta/NPointFunctions/Topologies.m` in the form `AllTopologies[{1, 1}]`. Currently, this

		Matrix	NPointFunctions
T2:	1, 2, 3, 4	$\begin{pmatrix} \mathbb{0}_{2\times 2} & \begin{matrix} 1 & 0 \\ 1 & 0 \end{matrix} \\ \begin{matrix} 1 & 1 \\ 0 & 0 \end{matrix} & \begin{matrix} 0 & 1 \\ 1 & 1 \end{matrix} \end{pmatrix}$	`tadpole -> {1, 0, 1, 0, 0, 1, 1}`
T3:	1, 3, 4, 2	$\begin{pmatrix} \mathbb{0}_{2\times 2} & \begin{matrix} 1 & 0 \\ 0 & 1 \end{matrix} \\ \begin{matrix} 1 & 0 \\ 0 & 1 \end{matrix} & \begin{matrix} 0 & 2 \\ 2 & 0 \end{matrix} \end{pmatrix}$	`sunset -> {1, 0, 0, 1, 0, 2, 0}`

Figure 3: Topologies, relevant for one-loop $1 \to 1$ processes, their `FeynArts`, mathematical and `NPointFunctions` representations, as in Figure 2.

definition is missing, so its specification becomes our first task. We can open a `Mathematica` notebook and evaluate the following code to figure out, which one-loop $1 \to 1$ topologies exist in general and which ones are of interest to us:

Listing 27: Execute in a `Mathematica` notebook.

```
<<FeynArts`
(topologies = CreateTopologies[1, 1 -> 1]) // Paint
```

The printed output is similar to the content of the first column in Figure 3. In this example, we are not interested in the first topology, while others represent what would like to include. We need to convert the `FeynArts` representation of the topology to the `NPointFunctions` one, this can be done as shown in Figure 3 or by execution of the code:[13]

Listing 28: Execute in a `Mathematica` notebook.

```
AdjaceTopology[topologies[[2]]]
AdjaceTopology[topologies[[3]]]
```

Let us name the obtained topologies as in Figure 3 and add their definition to the observable-independent file `meta/NPointFunctions/Topologies.m` as:

Listing 29: In `meta/NPointFunctions/Topologies.m`.

```
AllTopologies[{1, 1}] = {
   tadpole -> {1,0,1,0,0,1,1},
   sunset  -> {1,0,0,1,0,2,0},
   fermi   -> {tadpole, sunset}
};
```

where the last line creates the synonym for both topologies combined.

Now we can finally define the content of the advanced settings file for this observable:

[13]The general definition of the function `AdjaceTopology` exists in `meta/NPointFunctions/Topologies.m`; one has to replace the specification of the pattern in the definition by `topology_` and explicitly define `external = 2;` as this reflects the number of external vertices.

Listing 30: Content of the file `ExampleLeptonSE/NPointFunctions.m`.

```
topologies[1] = {
   Tadpoles -> tadpole,
   Sunsets  -> sunset,
   Fermi    -> fermi
};
```

This achieves the goal. Now the observable `ExampleLeptonSE` has an option `contr`, like all the examples in Section 2. This option may be set to the values `Tadpoles`, `Sunsets`, or `Fermi`. If some model-specific configuration file `models/M_a/FlexibleSUSY.m` contains the option `Tadpoles`, then only the `tadpole` topology will be included, and similarly for `Sunsets`. Once we specify `Fermi` (or `{Tadpoles, Sunsets}`), then both topologies will be used to compute the (generalized) self-energy. In this setup, `Sunsets` has the same effect as the `Irreducible` setting from the simplified example.

3.6. Content of the optional file `O_i/FSMathLink.m`

The spectrum generators created by `FlexibleSUSY` can be called from within a `Wolfram Language` notebook or kernel. The necessary definitions to output the numerical values of the generated observables at the `Wolfram Language` level are done in the general file `meta/FSMathLink.m`. For a specific observable `O_i` one can create an optional file `meta/Observables/O_i/FSMathLink.m` to specify the desired interface via the function `PutObservable`. We refer the reader to existing example files shipped with `FlexibleSUSY` for more details.

3.7. New observables and how to calculate them with `FlexibleSUSY`

General case

To activate the `C++` code generation for a desired observable with `FlexibleSUSY`, one carries out the same steps as for predefined observables in Section 2 (though now it is clear where all definitions come from). For a user-selected model `M_a` one modifies `models/M_a/FlexibleSUSY.m` by changing `ExtraSLHAOutputBlocks`. The pattern of observable `O_i` is defined inside the file called `meta/Observables/O_i/Observables.m` while the Les Houches blocks where this observable can be placed — inside `meta/Observables/O_i/WriteOut.m`. Finally, by execution of `make` one obtains the desired `C++` spectrum generator.

3.7.1. Example 1: output a given number

In summary, to add the new observable corresponding to Example 1 to `FlexibleSUSY`, one needs to create several files:

1. `O_i/Observables.m` with the `Wolfram Language` interface, the `C++` return type, and the `C++` prototype of the function that will be used to calculate the observable, see Section 3.1.1.
2. `O_i/WriteOut.m` with the connection of observable's return values and the `C++` spectrum generator output in Les Houches format, see Section 3.2.1.
3. `O_i/FlexibleSUSY.m` which will fill the `C++` template files with the model-specific information, see Section 3.3.1.
4. Two `C++` template files that will contain the `C++` code calculating the observable, see Section 3.4.1.

After all this is done, the observable can be used in the same way as the predefined observables discussed in Section 2, and we need to proceed as described in that section.

We first need to choose a desired physical model to perform the calculations (to continue this example we choose the SM), create it via `./createmodel --name=SM`, then configure `FlexibleSUSY` to make the `C++` spectrum generator for this model via `./configure --with-models=SM`.

Then, two model-specific settings files have to be modified to enable the computation of the observable `ExampleConstantObservable`. To be specific, we modify the file with the meta-level model settings `models/SM/FlexibleSUSY.m` to contain

Listing 31: In `models/SM/FlexibleSUSY.m`.

```
ExtraSLHAOutputBlocks = {
   {
      FlexibleSUSYLowEnergy,
      {
         {1, FlexibleSUSYObservable`ExampleConstantObservable[3]},
         {2, FlexibleSUSYObservable`ExampleConstantObservable[4]}
      }
   }
};
```

In this way we specify that the observable is called twice, with two different arguments (the arguments simply correspond to the numeric constants which should be printed in the output), and that the output will be part of the `Block FlexibleSUSYLowEnergy` with numbers 1 and 2, respectively. Finally, the calculation of all observables is enabled in the runtime model-specific SLHA input file:

Listing 32: In `models/SM/LesHouches.in.SM`.

```
Block FlexibleSUSY
   ...
   15   1                    # calculate all observables
```

The execution of `make` command runs the meta phase and compiles the final `C++` spectrum generator. `FlexibleSUSY` provides shell scripts which merge these steps, as follows:

Listing 33: Execute in terminal from the `FlexibleSUSY` directory.

```
./createmodel --name=SM
./configure --with-models=SM
./examples/new-observable/make-observable SM-example-1
make
./models/SM/run_SM.x --slha-input-file=models/SM/LesHouches.in.SM
```

to produce the following lines among the SLHA output:

```
Block FlexibleSUSYLowEnergy Q= 1.73340000E+02
   1     3.00000000E+00   # exampleconstantobservable 3
   2     4.00000000E+00   # exampleconstantobservable 4
```

Here we see the appearance of the block numbers and the values of the numeric constants in agreement with the definitions given above.

3.7.2. Example 2: show fermion masses

In order to add the observable of Example 2 to FlexibleSUSY and obtain a SM spectrum generator including the output of this observable one needs to go through analogous steps. Again, FlexibleSUSY is shipped with a script which merges all steps:

Listing 34: Execute in terminal from the FlexibleSUSY directory.

```
./createmodel --name=SM
./configure --with-models=SM
./examples/new-observable/make-observable SM-example-2
make
./models/SM/run_SM.x --slha-input-file=models/SM/LesHouches.in.SM
```

After executing these steps successfully, the SLHA output will contain the results corresponding to four instances of the observable ExampleLeptonMass (where the fermion index 1 here means 2nd generation due to the C++ numbering convention):

```
Block FlexibleSUSYLowEnergy Q= 1.73340000E+02
   1     1.04187667E-01   # Fe[1] (lepton[1] if in Block ExampleLeptonMass) mass
   2     5.71258815E-02   # Fd[1] (lepton[1] if in Block ExampleLeptonMass) mass
Block ExampleLeptonMass Q= 1.73340000E+02
   1     1.05658357E-01   # Fe[1] (lepton[1] if in Block ExampleLeptonMass) mass
   2     1.05658357E-01   # Fd[1] (lepton[1] if in Block ExampleLeptonMass) mass
```

The script generates all the files discussed in the previous Sections 3.1.2, 3.2.2, 3.3.2, and 3.4.2. Then it modifies the model-specific file models/SM/FlexibleSUSY.m as:

Listing 35: In models/SM/FlexibleSUSY.m.

```
ExtraSLHAOutputBlocks = {
   {
      FlexibleSUSYLowEnergy,
      {
         {1, FlexibleSUSYObservable`ExampleFermionMass[Fe[2]]},
         {2, FlexibleSUSYObservable`ExampleFermionMass[Fd[2]]}
      }
   },
   {
      ExampleLeptonMass,
      {
         {1, FlexibleSUSYObservable`ExampleFermionMass[Fe[2]]},
         {2, FlexibleSUSYObservable`ExampleFermionMass[Fd[2]]}
      }
   }
};
```

and assumes that the calculation of observables in the SLHA input file should be already enabled:

Listing 36: In models/SM/LesHouches.in.SM.

```
Block FlexibleSUSY
   ...
   15   1                    # calculate all observables
```

Here the file models/SM/FlexibleSUSY.m defines the four instances of the observable: they are distinguished by their argument (either Fe[2] or Fd[2]) and by the Les Houches block (either FlexibleSUSYLowEnergy or ExampleLeptonMass). We recall that in the definition of the observable the running $\overline{\text{MS}}/\overline{\text{DR}}$ mass of the fermion in the BSM model described by the argument and the SM lepton pole mass of the specified generation (in this case generation 2) are returned, see Section 3.4.2. We also recall that the output depends on the Les Houches block, see Section 3.2.2. This explains the output given above, where the FlexibleSUSYLowEnergy block shows the $\overline{\text{MS}}/\overline{\text{DR}}$ masses of the muon and the strange quark in the BSM model, while the ExampleLeptonMass block shows twice the muon pole mass from the qedqcd object.

3.7.3. *Example 3: lepton self-energy with NPointFunctions*

Technically, there are two versions of this example which differ by $\texttt{O}_\texttt{i}$/FlexibleSUSY.m: using the simple mode of NPointFunctions in Section 3.3.3 or using the advanced mode in Section 3.5.7. To add the advanced version of the observable of Example 3 to FlexibleSUSY and obtain a SM spectrum generator, one needs to go through steps analogous to previous examples. Again, FlexibleSUSY provides a script which combines all steps:

Listing 37: Execute in terminal from the FlexibleSUSY directory.

```
./createmodel --name=SM
./configure --with-models=SM
./examples/new-observable/make-observable SM-example-3
make
./models/SM/run_SM.x --slha-input-file=models/SM/LesHouches.in.SM
```

to successfully see among the SLHA output:

```
Block FWCOEF Q= 1.73340000E+02
   1111   31   00   2    -3.18762513E-05   # left
   1111   32   00   2    -3.18762513E-05   # right
```

Again, the script generates all files related to the definition of the observable. Then it modifies the model-specific file models/SM/FlexibleSUSY.m as

Listing 38: In models/SM/FlexibleSUSY.m.

```
ExtraSLHAOutputBlocks = {
   {
      FWCOEF,
      {
         {1, FlexibleSUSYObservable`ExampleLeptonSE[Fe[1], {Sunsets}]}
      }
   }
};
```

Note that the option Sunsets is used. It was defined via the advanced settings of NPointFunctions in Section 3.5.7. It is further assumed that the calculation of observables is enabled in the SLHA input file:

Listing 39: In models/SM/LesHouches.in.SM.

```
Block FlexibleSUSY
   ...
   15   1                    # calculate all observables
```

4. Conclusions

In this paper, we describe two novel essential features of FlexibleSUSY: a streamlined approach for incorporating new observables to FlexibleSUSY, and a simplified way to generate C++ code for Feynman diagrams essential for the computation of these observables and other physical quantities. To enhance the accessibility of this article we have divided the content into two distinct parts.

The first part is concise, requires no detailed knowledge of the code and is directed to readers interested in the computation of predefined FlexibleSUSY observables. The currently predefined observables include CLFV observables such as $\ell_i \to \ell_j \gamma$, μ–e conversion and $\ell_i \to \ell_j \ell_k \ell_k^c$, whose physics definitions are summarized in the appendix. Any user of FlexibleSUSY can now enable the computation of these observables for any desired model by just adding the appropriate flag in the model files.

The second part deals with the procedure of implementing new observables. It is more extensive and technical and is of interest to those seeking insights into the internal structure of the NPointFunctions code used for the creation of new observables. In essence, the implementation of a new observable requires five files (three Wolfram Language and two C++ template files) which essentially define in a model-independent way how the observable is computed, how it is called and how its output is organized. The second part also illustrates the utilization of the NPointFunctions extension to streamline the generation of C++ code for Feynman diagrams. This is achieved with the help of FlexibleSUSY-specialized wrappers designed for FeynArts, FormCalc, and ColorMath packages.

Three fully worked out examples are provided. They correspond to a minimal "observable" which simply outputs a single number, an observable which outputs a set of particle masses, and an observable which outputs the values of one-loop self energy diagrams. The code snippets can be used as efficient starting points for future implementations of further observables. In the appendix additional features and details are discussed which may be useful to accommodate special properties of new observables. They correspond to actual use-cases motivated by implementing e.g. the $b \to s\mu\mu$ or $h \to gg$ decays.

The FlexibleSUSY extensions presented here have been thoroughly cross-checked and used for concrete phenomenological applications in non-SUSY and in SUSY models. Ref. [23] uses and validates CLFV observables in a leptoquark model, where some observables arise at the one-loop level and some observables arise at tree-level. Refs. [24, 25] provide validations of loop-induced CLFV observables in a model of neutrino masses where several neutrino masses are themselves loop-induced. Finally, extensive validations have been carried out in the context of a non-trivial SUSY realization by comparing with results of Ref. [26] on CLFV phenomenology in the MRSSM. These applications demonstrate the reliability and versatility of the code for observables defined via Feynman diagrammatic calculations across a broad spectrum of models for wide variety of observables. In the future, it is planned to add further observables to the default distribution of FlexibleSUSY. In addition, users may add observables individually. Finally, we introduce the possibility to request the implementation of desired observables via github issues, see the developer's repository.

Acknowledgements

We thank the other FlexibleSUSY authors for helpful discussions. In particular, U.Kh. thanks Jobst Ziebell [27] and Kien Dang Tran [28] for the creation of an initial version of NPointFunctions.

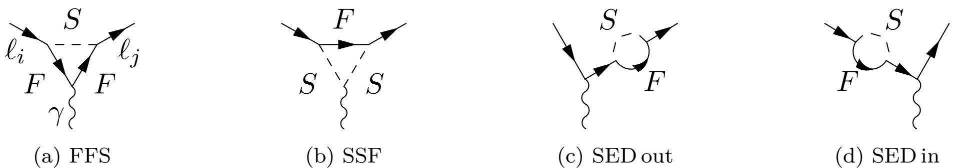


Figure A.4: Generic one-loop scalar-fermion Feynman diagrams contributing to $\ell_i \to \ell_j\gamma$. Arrows show particle propagation. Diagrams of Fermion-Fermion-Scalar (FFS) and Scalar-Scalar-Fermion (SSF) types contribute to A_2^X, while all four are responsible for A_1^X.

We acknowledge support by the German Research Foundation (DFG) under grants STO 876/4 and STO 876/7. W.K. was supported by the National Science Centre (Poland) grant 2022/47/D/ST2/03087.

Appendix A. Available observables: physical details

In this section we discuss physics details of calculations for CLFV observables implemented in `FlexibleSUSY`. Phenomenological applications connected to this section were presented in Refs. [23–26]. In all expressions of this section, the masses are treated in the following way: in the amplitudes we use $\overline{\text{MS}}/\overline{\text{DR}}$ values, everywhere else pole masses.

Appendix A.1. Two-body CLFV decays ($\ell_i \to \ell_j\gamma$)

The partial decay width is given by:

$$\Gamma_{\ell_i\to\ell_j\gamma} = \frac{1}{4\pi} m_i^5 \sum_{X=L,R} \left|C_X^{\mathcal{D}}\right|^2, \quad C_X^{\mathcal{D}} = -\frac{1}{2}A_2^X, \tag{A.1}$$

where the minus sign corresponds to the outgoing photon momentum. The form-factors A_2^X are defined as (the contributions for both scalar and fermion insertions are shown in Figure A.4):

$$i\Gamma_{\bar{\ell}_j\ell_i\gamma} = i\bar{u}_j\Big[\Big(q^2\gamma^\mu - q^\mu \not{q}\Big)\Big(A_1^L P_L + A_1^R P_R\Big) + im_i\sigma^{\mu\nu}q_\nu\Big(A_2^L P_L + A_2^R P_R\Big)\Big]u_i. \tag{A.2}$$

It is convenient to uniquely separate contributions proportional to f_γ and s_γ ($f_\gamma = g^L_{[\bar{F}F\gamma]}/(32\pi^2 m_S^2)$ is proportional to fermion electric charge; $s_\gamma = g^L_{[S^*S\gamma]}/(32\pi^2 m_S^2)$ contains scalar electric charge):

$$\begin{aligned}
A_1^L &= -f_\gamma g^R_{[\bar{F}_jFS]}\;\; g^L_{[\bar{F}F_iS^*]}\frac{F_D(x)}{18}, \\
A_2^L &= -f_\gamma g^L_{[\bar{F}_jFS]}\left[g^R_{[\bar{F}F_iS^*]}\frac{F_E(x)}{12} + g^L_{[\bar{F}F_iS^*]}\frac{m_F}{m_i}\frac{2F_F(x)}{3}\right],
\end{aligned} \tag{A.3}$$

$$\begin{aligned}
A_1^L &= -s_\gamma g^R_{[\bar{F}_jFS]}\;\; g^L_{[\bar{F}F_iS^*]}\frac{F_A(x)}{18}, \\
A_2^L &= -s_\gamma g^L_{[\bar{F}_jFS]}\left[g^R_{[\bar{F}F_iS^*]}\frac{F_B(x)}{18} + g^L_{[\bar{F}F_iS^*]}\frac{m_F}{m_i}\frac{F_C(x)}{3}\right].
\end{aligned} \tag{A.4}$$

In both Eqs. (A.3)–(A.4) the argument of the loop functions is $x = m_F^2/m_S^2$ and the summation over repeated fields of model $\mathtt{M_a}$ is assumed. Also, the expressions for the right-handed amplitudes $A^R_{1,2}$ can be obtained from a substitution $[L \leftrightarrow R]$ in the generic couplings. All $g^X_{[abc]}$ terms correspond to interaction vertices with removed imaginary prefactor. Finally, $F_y(x)$ are loop functions, see the aforementioned file and Refs. [29, 30].

Appendix A.2. Three-body CLFV decays ($\ell_i \to \ell_j \ell_k \ell_k^c$)

The partial decay width expressed in terms of Wilson coefficients has the form:

$$\Gamma_{\ell_i \to 3\ell_j} = \frac{m_i^5}{192\pi^3} \sum_{X=L,R} \left\{ e^2 \left| C_X^{\mathcal{D}} \right|^2 \left(8 \ln \frac{m_i}{m_j} - 11 \right) + e\,\mathrm{Re} \left[\left(2C_{XX}^{\mathcal{V},j} + C_{X\bar{X}}^{\mathcal{V},j} \right) C_{\bar{X}}^{\mathcal{D}*} \right] \right.$$
$$\left. + \frac{1}{64} \left(\left| C_{XX}^{\mathcal{S},j} \right|^2 + 16 \left| C_{XX}^{\mathcal{V},j} \right|^2 + 8 \left| C_{X\bar{X}}^{\mathcal{V},j} \right|^2 \right) \right\}, \tag{A.5}$$

where $\bar{X}$ means complementary chirality, i.e. $\bar{L}$ stands for R (see the `width_same` `C++` template in `templates/observables/br_l_to_3l.cpp.in`). In case of different final particles the expression becomes:

$$\Gamma_{\ell_i \to \ell_j \ell_k \ell_k^c} = \frac{m_i^5}{192\pi^3} \sum_{X=L,R} \left\{ e^2 \left| C_X^{\mathcal{D}} \right|^2 \left(8 \ln \frac{m_i}{m_k} - 12 \right) + e\,\mathrm{Re} \left[\left(C_{XX}^{\mathcal{V},k} + C_{X\bar{X}}^{\mathcal{V},k} \right) C_{\bar{X}}^{\mathcal{D}*} \right] \right.$$
$$\left. + \frac{1}{32} \left(\left| C_{XX}^{\mathcal{S},k} \right|^2 + \left| C_{X\bar{X}}^{\mathcal{S},k} \right|^2 \right) + \frac{1}{8} \left(\left| C_{XX}^{\mathcal{V},k} \right|^2 + \left| C_{X\bar{X}}^{\mathcal{V},k} \right|^2 \right) + \frac{3}{2} \left| C_{XX}^{\mathcal{T},k} \right|^2 \right\}, \tag{A.6}$$

see the `width_diff` `C++` template in `templates/observables/br_l_to_3l.cpp.in`.

The Wilson coefficients are defined via the Effective Field Theory (EFT) Lagrangian:

$$\mathcal{L}_{\mathrm{EFT}} = \mathcal{L}_{\mathrm{QED}} + \sum_{X,Y} \left(C_X^{\mathcal{D}} O_X^{\mathcal{D}} + \sum_{I=\mathcal{S},\mathcal{V},\mathcal{T}} \sum_f C_{XY}^{I,f} O_{XY}^{I,f} + \mathrm{h.c.} \right), \quad \mathcal{D}_\mu = \partial_\mu + ieQ_f A_\mu. \tag{A.7}$$

where the following set of operators is relevant for the considered CLFV processes, see Refs. [31–33]:

$$\begin{aligned} O_X^{\mathcal{D}} &= m_\mu [\bar{\ell}_j \sigma^{\mu\nu} P_X \ell_i] F_{\mu\nu}, & O_{XY}^{\mathcal{S},k} &= [\bar{\ell}_j P_X \ell_i][\bar{\ell}_k P_Y \ell_k], \\ O_{XY}^{\mathcal{V},k} &= [\bar{\ell}_j \gamma^\mu P_X \ell_i][\bar{\ell}_k \gamma_\mu P_Y \ell_k], & O_{XX}^{\mathcal{T},k} &= [\bar{\ell}_j \sigma^{\mu\nu} P_X \ell_i][\bar{\ell}_k \sigma_{\mu\nu} P_X \ell_k], \end{aligned} \tag{A.8}$$

with the definitions of left-right chiral projectors $P_{L/R} = (1 \mp \gamma_5)/2$; the square brackets emphasize different fermion chains with corresponding particles; $O_{LR}^{\mathcal{T},k}$, $O_{RL}^{\mathcal{T},k}$ are zero. The basis in Eq. (A.8) is a redundant one: scalar $O_{LR}^{\mathcal{S},k}$, $O_{RL}^{\mathcal{S},k}$ and tensor $O_{LL}^{\mathcal{T},k}$, $O_{RR}^{\mathcal{T},k}$ operators should be ignored because corresponding Wilson coefficients can be chosen to be zero.

Let us explain the amplitude matching procedure between EFT and full `Ma`-model Lagrangians specifically for the $\mu \to 3e$ process to simplify the notation and highlight the most essential details. One needs to keep the same external fermion order for both theories. It is enough to use the approximation of zero external momenta, and a convenient way to extract amplitudes is to consider the process $\mu_1^- e_2^- \to e_3^- e_4^-$. Using the two Lagrangians $\mathcal{L}_{\mathrm{EFT}}$ and $\mathcal{L}_{\mathtt{Ma}}$ we get the following amplitudes (up to δ-function and usual normalization factors; summation over repeated capital indices is assumed):

$$\langle e_4^- e_3^- | T \exp\left(i \int \mathcal{L}\, \mathrm{d}^4 x \right) | e_2^- \mu_1^- \rangle = \begin{cases} -iC_{XY}^{I,e} ([\bar{u}_3 \Gamma_X^I u_1][\bar{u}_4 \Gamma_Y^I u_2] - [3 \leftrightarrow 4]) & \text{for } \mathcal{L} = \mathcal{L}_{\mathrm{EFT}}, \\ iF_{XY}^{I,e} [\bar{u}_3 \Gamma_X^I u_1][\bar{u}_4 \Gamma_Y^I u_2] & \text{for } \mathcal{L} = \mathcal{L}_{\mathtt{Ma}}. \end{cases} \tag{A.9}$$

The first line contains the definition of the Wilson coefficients. Note that there is no $[3 \leftrightarrow 4]$ for $F_{XY}^{I,e}$ as we associate them with the results of `FormCalc` and the application of the setting `order` from Section 3.5.5; the derivation for coefficients is explained further, right below the matching.

$iF =$ inSelfT + outSelfT + triangleT + u-channel + boxS + other boxes

Figure A.5: Calculation of four-lepton coefficients F in the full model $\mathsf{M_a}$. For self-energy and triangle topologies only t-channel is drawn. Decay $\mu \to 3e$ is considered so that u-channel for self-energy and penguin topologies should be added. All box channels are calculated directly. For convenience, `NPointFunctions` topology names are shown.

We can match $F^{I,e}_{XY}$ onto low-energy coefficients $C^{I,e}_{XY}$ by applying Fierz identities onto crossed $[3 \leftrightarrow 4]$ terms of Eq. (A.9):

$$\begin{aligned} -F^{\mathcal{S},e}_{XX} &= \frac{1}{2}C^{\mathcal{S},e}_{XX} - 6C^{\mathcal{T},e}_{XX}, & -F^{\mathcal{S},e}_{X\bar{X}} &= C^{\mathcal{S},e}_{X\bar{X}} - 2C^{\mathcal{V},e}_{X\bar{X}}, \\ -F^{\mathcal{V},e}_{XX} &= 2C^{\mathcal{V},e}_{XX}, & -F^{\mathcal{V},e}_{X\bar{X}} &= C^{\mathcal{V},e}_{X\bar{X}} - \frac{1}{2}C^{\mathcal{S},e}_{X\bar{X}}, \\ -F^{\mathcal{T},e}_{XX} &= \frac{3}{2}C^{\mathcal{T},e}_{XX} - \frac{1}{8}C^{\mathcal{S},e}_{XX}. \end{aligned} \tag{A.10}$$

To solve these equations one may set:

$$C^{\mathcal{S},e}_{X\bar{X}} = 0, \qquad C^{\mathcal{T},e}_{XX} = 0, \tag{A.11}$$

then $F^{\mathcal{S}}_{XX}/F^{\mathcal{T}}_{XX} = -4$ and $F^{\mathcal{S}}_{X\bar{X}}/F^{\mathcal{V}}_{X\bar{X}} = -2$ should be fulfilled, which was checked numerically.

Now let us explain, how the $F^{I,e}_{XY}$ are calculated in the full model $\mathsf{M_a}$ via `NPointFunctions`. As it is shown in Figure A.5, the $F^{I,e}_{XY}$ coefficients take into account self-energy, penguin, and box diagrams. If we consider non-box contributions (e.g. for the MRSSM [26] for convenience), then they can be categorized by the particle which is connected to the bottom chain of electrons to be the photon, Z- and Higgs-boson. The photon contributes via dipole coefficients, and becomes a part of four-lepton vector operators as well, while the Z-boson contributes to the latter only, as follows from comparison with Ref. [26]:[14]

$$iF^{\mathcal{V},e}_{XY}\big|_{t\text{-channel for }\gamma} = -ieQ_e A^X_1, \quad iF^{\mathcal{V},e}_{XY}\big|_{t\text{-channel for }Z} = -i\frac{g_Z^2}{m_Z^2}A^X_Z Z^Y_e, \tag{A.12}$$

where the rhs. incorporates an additional minus sign from embedding of A^X_1 coefficients into four-fermion amplitude.[15] Scalar and pseudoscalar Higgs bosons contribute purely to scalar coefficients, and they are implemented and computed in all models. Specifically for the MRSSM, they are much smaller than other diagrams,

$$F^{\mathcal{S},e}_{XY}\big|_{t\text{-channel for }h_i,A_i} \approx 0. \tag{A.13}$$

Now, as one sees from Figure A.5, the expressions for crossed diagrams are required in addition to the mentioned t-channel self-energies and penguins. They can be calculated with the help of

[14] This reference provides the calculation of the CLFV processes in the context of the MRSSM, which represents a powerful way to validate the code in a BSM model for which no other public codes exist, in contrast to the MSSM.

[15] We use the left equation explicitly in the calculation of $\ell_i \to \ell_j \ell_k \ell_k^c$ with `FlexibleSUSY`. Instead of the right equation we use `NPointFunctions` directly and cross-checked it with unit test implemented in `FlexibleSUSY`.

the same Fierz identities which were used for matching in Eq. (A.10) because this channel differs only by the bispinor order. One can take any equality from Eq. (A.10) and remove the matching minus from the lhs. and replace the $C^{I,e}_{XY}$ in the rhs. by t-channel contributions of $F^{I,e}_{XY}$, e.g. the first equality shown in Eq. (A.10) transforms to the following one (see `fierz C++` template in `templates/observables/br_l_to_3l.cpp.in`):

$$F^{\mathcal{S},e}_{XX}|_{t\text{- and }u\text{-channel}} = \frac{1}{2}F^{\mathcal{S},e}_{XX}|_{t\text{-channel}} - 6F^{\mathcal{T},e}_{XX}|_{t\text{-channel}}. \tag{A.14}$$

Box diagrams contribute to all types of four-lepton coefficients. Also, there is no sense to distinguish between different channels, because we neglect external momenta so that all boxes directly contribute to the complete $F^{I,e}_{XY}$ coefficients. The final contribution for `Ma` being the MRSSM can be written as

$$F^{I,e}_{XY} = F^{I,e}_{XY}|_{t\text{- and }u\text{-channel for }\gamma,Z} + F^{I,e}_{XY}|_{\text{boxes}}. \tag{A.15}$$

The numerical output of the code generated in this way for the MRSSM has been successfully validated against Ref. [26].

Appendix A.3. Coherent conversion in nuclei (μ–e conversion)

The conversion rate of the process is given by Refs. [26, 34–37]:

$$\omega_{\mu-e} = 4m_\mu^5 \sum_{X=L,R} \left| \frac{1}{4}DC^{\mathcal{D}}_X - \sum_{N=n,p} \left(S^{(N)} g^{\mathcal{S},N}_X + V^{(N)} g^{\mathcal{V},N}_{\bar{X}} \right) \right|^2 \tag{A.16}$$

with dimensionless integrals $D, S^{(N)}, V^{(N)}$ defined in Ref. [34]; the minus sign reflects the different definitions of photon field, coming from a comparison of covariant derivatives (this observable is implemented in `templates/observables/l_to_l_conversion.cpp.in`).

We use EFT Lagrangian (with covariant derivative $\mathcal{D}_\mu = \partial_\mu + ieQ_f A_\mu$):

$$\mathcal{L}_{\text{EFT}} = \mathcal{L}_{\text{QED}} + \mathcal{L}_{\text{QCD}} + \sum_{X,Y} \left(\sum_{I=\mathcal{D},\mathcal{G}} C^I_X O^I_X + \sum_{I=\mathcal{S},\mathcal{V},\mathcal{T}} \sum_f C^{I,f}_{XY} O^{I,f}_{XY} + \text{h.c.} \right) \tag{A.17}$$

where the following set of operators is relevant, see Ref. [32]:[16]

$$\begin{aligned} O^{\mathcal{D}}_X &= m_\mu [\bar{e}\sigma^{\mu\nu} P_X \mu] F_{\mu\nu}, & O^{\mathcal{G}}_X &= \alpha_s m_\mu G_F [\bar{e} P_X \mu] G^a_{\mu\nu} G^{a\mu\nu}, \\ O^{\mathcal{S},f}_{XY} &= [\bar{e} P_X \mu][\bar{f} P_Y f], & O^{\mathcal{V},f}_{XY} &= [\bar{e}\gamma^\mu P_X \mu][\bar{f}\gamma_\mu P_Y f], \\ O^{\mathcal{T},f}_{XX} &= [\bar{e}\sigma^{\mu\nu} P_X \mu][\bar{f}\sigma_{\mu\nu} P_X f] \end{aligned} \tag{A.18}$$

with minor changes:

$$\begin{aligned} C^{\mathcal{S},q}_X &= \frac{1}{2}(C^{\mathcal{S},q}_{XL} + C^{\mathcal{S},q}_{XR}), & O^{\mathcal{S},q}_X &= [\bar{e} P_X \mu][\bar{q} q], \\ C^{\mathcal{V},q}_X &= \frac{1}{2}(C^{\mathcal{V},q}_{XL} + C^{\mathcal{V},q}_{XR}), & O^{\mathcal{V},q}_X &= [\bar{e}\gamma^\mu P_X \mu][\bar{q}\gamma_\mu q], \\ C^{\mathcal{T},q}_X &= C^{\mathcal{T},q}_{XX}, & O^{\mathcal{T},q}_X &= [\bar{e}\sigma^{\mu\nu} P_X \mu][\bar{q}\sigma_{\mu\nu} q]. \end{aligned} \tag{A.19}$$

$-iC =$ inSelfT + outSelfT + triangleT + boxS + other boxes

Figure A.6: Amplitude matching $\mathcal{L}_{\text{EFT}} \leftrightarrow \mathcal{L}_{\texttt{M}_\texttt{a}}$ for quark coefficients. μ–e conversion is considered so that all existing self-energy and penguin topologies are in t-channel only and are shown explicitly. All channels for boxes are calculated directly. The sum over all represented diagrams can be matched to the corresponding C_X^q coefficients without intermediate steps. For convenience, `NPointFunctions` topology names are shown.

In contrast to $\mu \to 3e$, there are no u-channel penguin and self-energy diagrams and no coefficient vanishes due to Fierz identities. So that $\mathcal{L}_{\text{EFT}} \leftrightarrow \mathcal{L}_{\texttt{M}_\texttt{a}}$ matching can be done directly by the consultation with Figure A.6. Let us again compare with the MRSSM results of Ref. [26] for concreteness. For four-fermion coefficients one, in general, adds self-energy, penguin, and box diagrams:

$$-iC_X^{\mathcal{V},q}\big|_\gamma = -ieQ_q A_1^X, \quad -iC_X^{\mathcal{V},q}\big|_Z = -i\frac{g_Z^2}{2m_Z^2} A_Z^X (Z_q^L + Z_q^R), \quad C_X^{\mathcal{S},q}\big|_{h_i,A_i} \approx 0, \tag{A.20}$$

where we justify the lhs. by stressing that diagrams of Figure A.6 represent amplitudes. Note the additional minus sign from photon embedding, which corresponds to the chosen order of external fermions in $\mu_1^- q_2^- \to e_3^- q_4^-$. Box diagrams contribute to all types of Wilson coefficients, so that:

$$C_X^{I,q} = C_X^{I,q}\big|_\gamma + C_X^{I,q}\big|_Z + C_X^{I,q}\big|_{\text{boxes}}. \tag{A.21}$$

Gluonic coefficients might be relevant in general, and are changed when heavy quarks c, b are integrated out, as stated in Ref. [38]:

$$m_q \bar{q} q \to -\frac{\alpha_s}{12\pi} G^a_{\mu\nu} G^{a\mu\nu}, \quad C_X^{\mathcal{G}} \to \tilde{C}_X^{\mathcal{G}} = C_X^{\mathcal{G}} - \frac{1}{12\pi} \sum_{q=c,b} \frac{1}{m_q m_\mu G_F} C_X^{\mathcal{S},q}. \tag{A.22}$$

To match $\mathcal{L}_{\text{EFT}}$ onto $\mathcal{L}_{\text{N}}^{\text{coh}}$, on-mass-shell condition is used for the same initial and final nucleon state $|N\rangle$, see Refs. [32–37, 39–42]. This is realized for a nucleon N, quark q, and coefficient type I with a help of $G_N^{I,q}$ form factors:

$$\langle N|\mathcal{L}_{\text{EFT}}|N\rangle \approx \langle N|\mathcal{L}_{\text{N}}^{\text{coh}}|N\rangle \to \begin{array}{r} [\bar{q}\Gamma_I q] = \sum\limits_{N=p,n} G_N^{I,q} [\bar{N}\Gamma_I N], \\ \alpha_s G^a_{\mu\nu} G^{a\mu\nu} = \sum\limits_{N=p,n} m_N G_N^{\mathcal{G}} [\bar{N} N], \end{array} \tag{A.23}$$

where gluonic form factors are obtained using a trace of the energy-momentum tensor, while others can be found in Table A.9:

$$G_N^{\mathcal{G}} = -\frac{8\pi}{9}\left(1 - \sum_{q=u,d,s} \frac{m_q}{m_N} G_N^{\mathcal{S},q}\right). \tag{A.24}$$

[16] Gluon dimension-7 operators $O_X^{\mathcal{G}}$ are important for μ–e conversion in general, see Ref. [38]. It was not necessary for our purposes to use $m_\mu G_F$ prefactor in the gluonic operator, as for considered models it was neglected, but we keep it to preserve the form of the operator in Ref. [32].

$G_p^{\mathcal{S},u} = 0.021(2)\frac{m_p}{m_u}$	$G_p^{\mathcal{S},d} = 0.041(3)\frac{m_p}{m_d}$	$G_p^{\mathcal{S},s} = 0.043(11)\frac{m_p}{m_s}$
$G_n^{\mathcal{S},u} = 0.019(2)\frac{m_n}{m_u}$	$G_n^{\mathcal{S},d} = 0.045(3)\frac{m_n}{m_d}$	$G_n^{\mathcal{S},s} = 0.043(11)\frac{m_n}{m_s}$
$G_p^{\mathcal{V},u} = G_n^{\mathcal{V},d} = 2$	$G_p^{\mathcal{V},d} = G_n^{\mathcal{V},u} = 1$	$G_p^{\mathcal{V},s} = G_n^{\mathcal{V},s} = 0$
$G_p^{\mathcal{T},u} = G_n^{\mathcal{T},d} = 0.77(7)$	$G_p^{\mathcal{T},d} = G_n^{\mathcal{T},u} = -0.23(3)$	$G_p^{\mathcal{T},s} = G_n^{\mathcal{T},s} = 0.008(9)$

Table A.9: Matching ($\mathcal{L}_{\mathrm{N}}^{\mathrm{coh}} \leftrightarrow \mathcal{L}_{\mathrm{EFT}}$) form factors, see Appendix B.3. Scalar form factors measure the contribution of the quark condensate to the mass of nucleon. They are determined from pion-nucleon $\sigma_{\pi N}$ term for u, d-quarks [43, 44] and from the lattice calculation for s-quark. Vector form factors do not suffer from theoretical uncertainty during the matching and are derived from the conservation of vector current (counting of valence quarks). For tensor form factors the values calculated in lattice Quantum Chromodynamics (QCD) at 2 GeV [41, 45] are taken. See both `src/observables/l_to_l_conversion/settings.cpp` used for the storage of form factors and `templates/observables/l_to_l_conversion.cpp.in` for observable implementation.

At finite recoil, tensor operator contributes to scalar ones [41, 42], which is expressed via the replacement:

$$[\bar{e}\sigma^{\mu\nu}P_X\mu][\bar{N}\sigma_{\mu\nu}N] \to \frac{m_\mu}{m_N}[\bar{e}P_X\mu][\bar{N}N]. \tag{A.25}$$

The relevant for $\mu - e$ conversion Lagrangian takes the following form:

$$\mathcal{L}_{\mathrm{N}}^{\mathrm{coh}} = \sum_{X=L,R}\left[C_X^{\mathcal{D}}O_X^{\mathcal{D}} + \sum_{N=n,p}\left(g_X^{\mathcal{S},N}O_X^{\mathcal{S},N} + g_X^{\mathcal{V},N}O_X^{\mathcal{V},N}\right) + \text{h.c.}\right], \tag{A.26}$$

where new coefficients and operators are introduced ($G_{N,s}^{\mathcal{V}}$ are zero and may be omitted):

$$\begin{gathered} g_X^{\mathcal{S},N} = m_N m_\mu G_F G_N^{\mathcal{G}}\tilde{C}_X^{\mathcal{G}} + \sum_{q=u,d,s}\left(G_N^{\mathcal{S},q}C_X^{\mathcal{S},q} + \frac{m_\mu}{m_N}G_N^{\mathcal{T},q}C_X^{\mathcal{T},q}\right), \quad O_X^{\mathcal{S},N} = [\bar{e}P_X\mu][\bar{N}N], \\ g_X^{\mathcal{V},N} = \sum_{q=u,d,s}G_N^{\mathcal{V},q}C_X^{\mathcal{V},q}, \quad O_X^{\mathcal{S},N} = [\bar{e}\gamma^\mu P_X\mu][\bar{N}\gamma_\mu N]. \end{gathered} \tag{A.27}$$

Appendix B. Additional features and examples

In this section we show several advanced ways to use FlexibleSUSY in order to obtain more freedom in both input and output formats, as well as to perform non-conventional intermediate calculations. Though it is currently possible to use these features in the way described below, they might be automatized in future.

Appendix B.1. Example 4: post-processing in $h \to gg$

Let us demonstrate a possible way to do post-processing when additional user input is required after NPointFunctions calculations. When all external particles are fermions, one can consult the implementation of $\ell_i \to \ell_j\ell_k\ell_k^c$ and μ–e conversion. In this example, we consider external bosons which implies several changes to aforementioned observables.

The code template of this section can be integrated into FlexibleSUSY by the execution of the following command:

```
./examples/new-observable/make-observable example-4
```

The observable defined in this example calculates amplitudes required for the $h \to gg$ process (but not the branching ratio itself) with the help of `NPointFunctions`. After `NPointFunctions` calculations are done, one is left with amplitudes containing many abbreviations. From physics we know that the computed amplitudes must contain several structures of covariants. For $h \to gg$, the possible covariants are

$$\epsilon_2^\mu \epsilon_{3\mu}, \quad \epsilon_2^\mu p_{3\mu}, \quad \epsilon_3^\mu p_{2\mu}, \quad \varepsilon^{\alpha\beta\mu\nu} \epsilon_{2\alpha} \epsilon_{3\beta} p_{2\mu} p_{3\nu}, \tag{B.1}$$

where ϵ_i is the polarization vector of the corresponding gluon and ε is the Levi-Civita tensor.

It is the coefficients of these structures which are relevant for the calculation of the branching ratio. Hence we need to extract the prefactors of these structures. This can be performed as shown in Listing 40. There, we apply all remaining sub-expressions in line 1, then abbreviate all basis structures in lines 2–8, and extract the coefficients that come as prefactors of the second argument of `InterfaceToMatching` in line 9 (similarly to line 28 in Listing 18). Finally, the `C++` code definitions are generated by the function `NPFDefinitions` in line 14 (similarly to line 32 in Listing 18; note, that `NPFDefinitions` may accept strings that are different to the last argument of `InterfaceToMatching`):

Listing 40: Content of `HiggsTo2Gluons/FlexibleSUSY.m`.

```
npf = NPointFunctions`ApplySubexpressions[npf];
npf = npf /. {
   SARAH`sum[__, FormCalc`ec[2, l_] FormCalc`ec[3, l_] SARAH`g[__]] :>         "e2e3",
   SARAH`sum[__, FormCalc`ec[2, l_] SARAH`Mom[3, l_] SARAH`g[__]]   :>         "e2m3",
   SARAH`sum[__, FormCalc`ec[3, l_] SARAH`Mom[2, l_] SARAH`g[__]]   :>         "e3m2",
   FormCalc`Eps[FormCalc`ec[2], FormCalc`ec[3], SARAH`Mom[2], SARAH`Mom[3]] :> "eps",
   SARAH`sum[__, SARAH`Mom[2, l_] SARAH`Mom[3, l_] SARAH`g[__]] :> SARAH`Mass[higgs]^2/2
};
npf = WilsonCoeffs`InterfaceToMatching[npf, {"eps", "e2e3", "e2m3" "e3m2"}];

...

AppendTo[npfDefinitions,
   NPointFunctions`NPFDefinitions[npf, "cpp_name", SARAH`Delta, {"eps", "e2e3", "e2m3_e3m2"}]
];
```

In principle, one can apply the routines from the code snippet above for the processes with external fermions as well. This might be relevant, in particular, if one prefers to ignore the setting `chains` from Section 3.5.5 and deal with Dirac chains in some other ways.

Appendix B.2. Example 5: `flavio` *output in* $b \to s\mu\mu$

One might want to generate the output of Wilson coefficients to be used later via other programs, like `flavio` [46] that requires a `.json` file with specific content. The code template of this section can be integrated into `FlexibleSUSY` by the execution of the following command:

```
./examples/new-observable/make-observable example-5
```

Currently, the way to fill a `.json` file can be demonstrated by the operators required for $b \to s\mu\mu$ using `JavaScript` Object Notation (JSON) library [47] distributed with `FlexibleSUSY`:

Listing 41: Content of `templates/observables/br_d_l_to_d_l.cpp.in`.

```
#include <fstream>
#include <iomanip>
#include "json.hpp"
// Definition of C9_bsmumu and other Wilson coefficients
nlohmann::json j;
j["eft"]    = "WET";
j["basis"]  = "flavio";
j["scale"]  = dynamic_cast<@ModelName@_mass_eigenstates const&>(context.model).get_scale();
j["values"] = {
   {"C9_bsmumu",   {{"Re", Re(C9_bsmumu)},   {"Im", Im(C9_bsmumu)}}},
   // Other Wilson coefficients
};
std::ofstream wc_json("WET_bsmumu.json");
wc_json << std::setw(4) << j << std::endl;
wc_json.close();
```

The code from the listing above will generate the file `WET_bsmumu.json` filled with the numerical values obtained by `FlexibleSUSY` and, in principle, ready to be used for the program `flavio`. It is clear, that one is generally required to calculate the complete set of operators closed under the RGE running and only then use it for phenomenological studies. In particular for this example, one is expected currently to generate several `.json` files and appropriately merge them before the execution of `flavio`, which is planned to be improved.

Appendix B.3. Example 6: additional LH input blocks in μ–e conversion

The process of μ–e conversion depends on several parameters, see Table A.9, and contains multiple factors that might not be relevant for a given model, see Appendix A.3. There is a way to modify these settings during the runtime of `C++` spectrum generator via an additional Les Houches input block that is automatically added if `FlexibleSUSY` is configured to calculate μ–e conversion.

Let us provide more information about how this functionality works and can be used for other observables as well if they require additional configuration options.

First of all, multiple changes within `C++` template files should be made in `FlexibleSUSY` globally. This is realized in the function `WriteClass` via its additional output value, called `"C++ replacements"`, see Section 3.3 and the `LToLConversion/FlexibleSUSY.m` file.

On the `C++` side, one defines files to store new values in the `src/obsevables/@O_i_filename@/` directory (see the definition for `descriptions` and their default values in the function `reset`) and lets `FlexibleSUSY` know about the new Les Houches block name in `src/slha_io.*` files (in this case, the name is `"LToLConversion"`).

For example, if we want to enable the tensor contributions from Eq. (A.25), then we need to include the following line into the input Les Houches file:

Listing 42: In `models/M_a/LesHouches.in.M_a`, if `FlexibleSUSY` is configured to calculate μ–e conversion.

```
Block LToLConversion
    0   1   # include tensor contribution
```

Similarly, one can change numerical values of all coefficients from Table A.9.

References

[1] P. Athron, J.-h. Park, D. Stöckinger, A. Voigt, FlexibleSUSY — A spectrum generator generator for supersymmetric models, Comput. Phys. Commun. 190 (2015) 139–172. arXiv:1406.2319, doi:10.1016/j.cpc.2014.12.020.

[2] P. Athron, M. Bach, D. Harries, T. Kwasnitza, J.-h. Park, D. Stöckinger, A. Voigt, J. Ziebell, FlexibleSUSY 2.0: Extensions to investigate the phenomenology of SUSY and non-SUSY models, Comput. Phys. Commun. 230 (2018) 145–217. arXiv:1710.03760, doi:10.1016/j.cpc.2018.04.016.

[3] F. Staub, SARAH 4: A tool for (not only SUSY) model builders, Comput. Phys. Commun. 185 (2014) 1773–1790. arXiv:1309.7223, doi:10.1016/j.cpc.2014.02.018.

[4] W. Porod, SPheno, a program for calculating supersymmetric spectra, SUSY particle decays and SUSY particle production at e^+e^- colliders, Comput. Phys. Commun. 153 (2003) 275–315. arXiv:hep-ph/0301101, doi:10.1016/S0010-4655(03)00222-4.

[5] W. Porod, F. Staub, SPheno 3.1: Extensions including flavour, CP-phases and models beyond the MSSM, Comput. Phys. Commun. 183 (2012) 2458–2469. arXiv:1104.1573, doi:10.1016/j.cpc.2012.05.021.

[6] W. Porod, F. Staub, A. Vicente, A Flavor Kit for BSM models, Eur. Phys. J. C 74 (8) (2014) 2992. arXiv:1405.1434, doi:10.1140/epjc/s10052-014-2992-2.

[7] Wolfram Research, Inc., Wolfram Language, Version 14.0, Champaign, Illinois, 2024.
URL https://www.wolfram.com/language

[8] F. Staub, From Superpotential to Model Files for FeynArts and CalcHep/CompHep, Comput. Phys. Commun. 181 (2010) 1077–1086. arXiv:0909.2863, doi:10.1016/j.cpc.2010.01.011.

[9] F. Staub, Automatic Calculation of supersymmetric Renormalization Group Equations and Self Energies, Comput. Phys. Commun. 182 (2011) 808–833. arXiv:1002.0840, doi:10.1016/j.cpc.2010.11.030.

[10] F. Staub, SARAH 3.2: Dirac Gauginos, UFO output, and more, Comput. Phys. Commun. 184 (2013) 1792–1809. arXiv:1207.0906, doi:10.1016/j.cpc.2013.02.019.

[11] B. C. Allanach, SOFTSUSY: a program for calculating supersymmetric spectra, Comput. Phys. Commun. 143 (2002) 305–331. arXiv:hep-ph/0104145, doi:10.1016/S0010-4655(01)00460-X.

[12] B. C. Allanach, P. Athron, L. C. Tunstall, A. Voigt, A. G. Williams, Next-to-Minimal SOFTSUSY, Comput. Phys. Commun. 185 (2014) 2322–2339, [Erratum: Comput.Phys.Commun. 250, 107044 (2020)]. arXiv:1311.7659, doi:10.1016/j.cpc.2014.04.015.

[13] P. Athron, M. Bach, D. H. J. Jacob, W. Kotlarski, D. Stöckinger, A. Voigt, Precise calculation of the W boson pole mass beyond the standard model with FlexibleSUSY, Phys. Rev. D 106 (9) (2022) 095023. arXiv:2204.05285, doi:10.1103/PhysRevD.106.095023.

[14] P. Athron, A. Büchner, D. Harries, W. Kotlarski, D. Stöckinger, A. Voigt, FlexibleDecay: An automated calculator of scalar decay widths, Comput. Phys. Commun. 283 (2023) 108584. arXiv:2106.05038, doi:10.1016/j.cpc.2022.108584.

[15] U. Khasianevich, W. Kotlarski, D. Stöckinger, NPointFunctions: a calculator of amplitudes and observables in FlexibleSUSY, PoS CompTools2021 (2022) 036. arXiv:2206.00745, doi:10.22323/1.409.0036.

[16] T. Hahn, Generating Feynman diagrams and amplitudes with FeynArts 3, Comput. Phys. Commun. 140 (2001) 418–431. arXiv:hep-ph/0012260, doi:10.1016/S0010-4655(01)00290-9.

[17] T. Hahn, M. Perez-Victoria, Automatized one loop calculations in four-dimensions and D-dimensions, Comput. Phys. Commun. 118 (1999) 153–165. arXiv:hep-ph/9807565, doi:10.1016/S0010-4655(98)00173-8.

[18] M. Sjödahl, ColorMath — A package for color summed calculations in $SU(N_c)$, Eur. Phys. J. C 73 (2) (2013) 2310. arXiv:1211.2099, doi:10.1140/epjc/s10052-013-2310-4.

[19] 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.

[20] B. C. Allanach, et al., SUSY Les Houches Accord 2, Comput. Phys. Commun. 180 (2009) 8–25. arXiv:0801.0045, doi:10.1016/j.cpc.2008.08.004.

[21] F. Mahmoudi, et al., Flavour Les Houches Accord: Interfacing Flavour related Codes, Comput. Phys. Commun. 183 (2012) 285–298. arXiv:1008.0762, doi:10.1016/j.cpc.2011.10.006.

[22] G. D. Kribs, E. Poppitz, N. Weiner, Flavor in supersymmetry with an extended R-symmetry, Phys. Rev. D 78 (2008) 055010. arXiv:0712.2039, doi:10.1103/PhysRevD.78.055010.

[23] U. Khasianevich, D. Stöckinger, H. Stöckinger-Kim, J. Wünsche, Constraint on scalar leptoquark from low-energy leptonic observables, Phys. Rev. D 108 (9) (2023) 095027. arXiv:2305.05016, doi:10.1103/PhysRevD.108.095027.

[24] V. Dūdėnas, T. Gajdosik, U. Khasianevich, W. Kotlarski, D. Stöckinger, Charged lepton flavor violating processes in the Grimus-Neufeld model, JHEP 09 (2022) 174. arXiv:2206.00661, doi:10.1007/JHEP09(2022)174.

[25] V. Dūdėnas, T. Gajdosik, U. Khasianevich, W. Kotlarski, D. Stöckinger, Box-enhanced charged lepton flavor violation in the Grimus-Neufeld model, Phys. Rev. D 107 (5) (2023) 055027. arXiv:2211.14384, doi:10.1103/PhysRevD.107.055027.

[26] W. Kotlarski, D. Stöckinger, H. Stöckinger-Kim, Low-energy lepton physics in the MRSSM: $(g-2)_\mu$, $\mu \to e\gamma$ and $\mu \to e$ conversion, JHEP 08 (2019) 082. arXiv:1902.06650, doi:10.1007/JHEP08(2019)082.

[27] J. Ziebell, Precise Higgs boson mass calculation in the MSSM with one-loop pole mass matching to the THDM, Master's thesis, TU Dresden (2018).

[28] K. D. Tran, B physics in the Minimal R-symmetric Supersymmetric Standard Model, Master's thesis, TU Dresden (2019).

[29] J. Hisano, T. Moroi, K. Tobe, M. Yamaguchi, Lepton flavor violation via right-handed neutrino Yukawa couplings in supersymmetric standard model, Phys. Rev. D 53 (1996) 2442–2459. arXiv:hep-ph/9510309, doi:10.1103/PhysRevD.53.2442.

[30] J. R. Ellis, J. S. Lee, A. Pilaftsis, Electric Dipole Moments in the MSSM Reloaded, JHEP 10 (2008) 049. arXiv:0808.1819, doi:10.1088/1126-6708/2008/10/049.

[31] Y. Okada, K.-i. Okumura, Y. Shimizu, $\mu \to e\gamma$ and $\mu \to 3e$ processes with polarized muons and supersymmetric grand unified theories, Phys. Rev. D 61 (2000) 094001. arXiv:hep-ph/9906446, doi:10.1103/PhysRevD.61.094001.

[32] A. Crivellin, S. Davidson, G. M. Pruna, A. Signer, Renormalisation-group improved analysis of $\mu \to e$ processes in a systematic effective-field-theory approach, JHEP 05 (2017) 117. arXiv:1702.03020, doi:10.1007/JHEP05(2017)117.

[33] Y. Kuno, Y. Okada, Muon decay and physics beyond the standard model, Rev. Mod. Phys. 73 (2001) 151–202. arXiv:hep-ph/9909265, doi:10.1103/RevModPhys.73.151.

[34] R. Kitano, M. Koike, Y. Okada, Detailed calculation of lepton flavor violating muon electron conversion rate for various nuclei, Phys. Rev. D 66 (2002) 096002, [Erratum: Phys.Rev.D 76, 059902 (2007)]. arXiv:hep-ph/0203110, doi:10.1103/PhysRevD.76.059902.

[35] V. Cirigliano, R. Kitano, Y. Okada, P. Tuzon, On the model discriminating power of $\mu \to e$ conversion in nuclei, Phys. Rev. D 80 (2009) 013002. arXiv:0904.0957, doi:10.1103/PhysRevD.80.013002.

[36] S. Davidson, Y. Kuno, A. Saporta, "Spin-dependent" $\mu \to e$ conversion on light nuclei, Eur. Phys. J. C 78 (2) (2018) 109. arXiv:1710.06787, doi:10.1140/epjc/s10052-018-5584-8.

[37] S. Davidson, Y. Kuno, M. Yamanaka, Selecting $\mu \to e$ conversion targets to distinguish lepton flavour-changing operators, Phys. Lett. B 790 (2019) 380–388. arXiv:1810.01884, doi:10.1016/j.physletb.2019.01.042.

[38] M. A. Shifman, A. I. Vainshtein, V. I. Zakharov, Remarks on Higgs Boson Interactions with Nucleons, Phys. Lett. B 78 (1978) 443–446. doi:10.1016/0370-2693(78)90481-1.

[39] T. S. Kosmas, S. Kovalenko, I. Schmidt, Nuclear muon- e- conversion in strange quark sea, Phys. Lett. B 511 (2001) 203. arXiv:hep-ph/0102101, doi:10.1016/S0370-2693(01)00657-8.

[40] T. S. Kosmas, Exotic $\mu \to e$ conversion in nuclei: Energy moments of the transition strength and average energy of the outgoing e^-, Nucl. Phys. A 683 (2001) 443–462. doi:10.1016/S0375-9474(00)00471-1.

[41] V. Cirigliano, S. Davidson, Y. Kuno, Spin-dependent $\mu \to e$ conversion, Phys. Lett. B 771 (2017) 242–246. arXiv:1703.02057, doi:10.1016/j.physletb.2017.05.053.

[42] M. Cirelli, E. Del Nobile, P. Panci, Tools for model-independent bounds in direct dark matter searches, JCAP 10 (2013) 019. arXiv:1307.5955, doi:10.1088/1475-7516/2013/10/019.

[43] A. Crivellin, M. Hoferichter, M. Procura, Accurate evaluation of hadronic uncertainties in spin-independent WIMP-nucleon scattering: Disentangling two- and three-flavor effects, Phys. Rev. D 89 (2014) 054021. arXiv:1312.4951, doi:10.1103/PhysRevD.89.054021.

[44] M. Hoferichter, J. Ruiz de Elvira, B. Kubis, U.-G. Meißner, High-Precision Determination of the Pion-Nucleon σ Term from Roy-Steiner Equations, Phys. Rev. Lett. 115 (2015) 092301. arXiv:1506.04142, doi:10.1103/PhysRevLett.115.092301.

[45] T. Bhattacharya, V. Cirigliano, R. Gupta, H.-W. Lin, B. Yoon, Neutron Electric Dipole Moment and Tensor Charges from Lattice QCD, Phys. Rev. Lett. 115 (21) (2015) 212002. arXiv:1506.04196, doi:10.1103/PhysRevLett.115.212002.

[46] D. M. Straub, flavio: a python package for flavour and precision phenomenology in the standard model and beyond (2018). arXiv:1810.08132.

[47] N. Lohmann, JSON for modern C++ (11 2023).
URL https://json.nlohmann.me