ScannerS manual: version-1.1.0

Raul Costa,[a] **Renato Guedes,**[a] **Marco O. P. Sampaio,**[b] **Rui Santos**[a,c]

[a] *Centro de Física Teórica e Computacional, Universidade de Lisboa*
1649-003 Lisboa, Portugal

[b] *Departamento de Física da Universidade de Aveiro and I3N*
Campus de Santiago, 3810-183 Aveiro, Portugal

[c] *Instituto Superior de Engenharia de Lisboa - ISEL*
1959-007 Lisboa, Portugal

E-mail: rauljcosta@ua.pt, renato@cii.fc.ul.pt, msampaio@ua.pt, rsantos@cii.fc.ul.pt

Abstract: ScannerS is a tool for automatizing scans of the parameter space of arbitrary scalar potentials beyond the Standard Model, from an expression for the scalar potential and a user defined analysis. The code provides automatic routines to determine the scalar spectrum, analyse stability and numerous interfaces to external libraries for analysis. In addition to the the tools which can be used for a general potential, ScannerS provides two model specific classes for the analysis of the xSM2 (a complex singlet extension) and the 2HDM (Two Higgs Double Model) already implemented. Please refer also to our first publication with the code for details of the numerical strategy [1] and a study with the 2HDM [2].

Keywords: Computational Tools, Higgs boson, Beyond Standard Model, 2HDM, xSM2

Contents

1 Getting started

1.1 Minimal requirements

To run the standalone examples of the code (with no interfaces to other HEP libraries), ScannerS needs the following packages installed:

- The Gnu Scientific Library (GSL – see `http://www.gnu.org/software/gsl/`): This is used by the source code for several mathematical operations.
- A `C++` compiler (such as `g++`) and gnu make (or equivalent).
- *Mathematica7* or later: This is to able to run the file `ScannerSInput.nb` where the scalar potential for the model is entered (as well as the options for the scan), which generates an input file, `model.in`, for the `C++` program. This is NOT required for potential models that already have a specific class (2HDM and xSM2). For such models, there are implementations ready to run and/or adapt.

For details on how to use the external interfaces please see Section 3.

1.2 Quick Start (xSM2 or 2HDM)

Let us start with an example for the two Higgs doublet model (2HDM) which performs a scan of the scalar potential of the $\mathbb{Z}_2$ symmetric class of models with the following potential[1]

$$V = m_{11}^2 \Phi_1^\dagger \Phi_1 + m_{22}^2 \Phi_2^\dagger \Phi_2 - m_{12}^2 (\Phi_2^\dagger \Phi_1 + c.c.) + \frac{1}{2}\lambda_1 |\Phi_1|^4 + \frac{1}{2}\lambda_2 |\Phi_2|^4 + \\ + \lambda_3 |\Phi_1|^2 |\Phi_2|^2 + \lambda_4 |\Phi_1^\dagger \Phi_2|^2 + \frac{1}{2}\lambda_5 \left((\Phi_1^\dagger \Phi_2)^2 + c.c. \right) \quad (1.1)$$

The steps to follow are:

a) **Download and unpack the latest source**:
https://www.hepforge.org/downloads/scanners

```
$ tar -xf scanners-x.x.x.tar.gz
$ cd scanners-x.x.x/
```

(replace x.x.x by the version numbers)

In this directory you find:

```
$ ls
 AUTHORS  ChangeLog  COPYING  doc/  examples/  makefile  model.in
 README  ScannerScore/  ScannerSInput.nb  ScannerSUser.cpp
```

- `ScannerScore/`: The source code of the program (not to be edited by the user).
- `doc/`: The documentation of the program with the manual and `.cpp` files describing some useful pieces of source code.
- `examples/`: Various example directories, each containing the analysis files necessary to run them, as well as README files for the xSM2 and 2HDM implementation.
- License information and `README` file.
- The user editable files: `makefile`, `ScannerSInput.nb`, `ScannerSUser.cpp`, `ScannerSUserTwoHDM.cpp`, `ScannerSUserxSM2.cpp`.

b) **Check the makefile**: Open the `makefile` and check that the compilers are set correctly.

```
#...
############################
# 1) Choose your compilers
############################
compiler=g++
compilerf77=gfortran
#...
```

If the `gsl` library is not installed in the default search paths specify additional paths

[1] The procedure is similar for the xSM2 model, using the xSM2 files instead.

```
#...
############################
# 2) Standard search Paths
############################
#Optionally add search paths here to include files and libraries
#Use format  = -I/Incpath1 -I/Incpath2 etc...
SystemIncPath= -I/usr/local/include
#Use format  = -L/libpath1 -L/libpath2 etc...
SystemLibPath= -L/usr/lib64
#...
```

c) Copy the `TwoHDM.in` file from the examples folder or use the `--model` command:

```
$ ./ScannerS --model 2HDM
```

This will generate an input file (`TwoHDM.in`) in which you can edit the parameters for the scan.

d) **Check/edit the User Analysis file** – `ScannerSUserTwoHDM.cpp`:

The final file to be edited is `ScannerSUserTwoHDM.cpp`. Here you have access to several template functions for: i) initial calculations which are executed before starting the scan (`UserInitCalcs`), ii) calculations to analyse each generated point (`UserAnalysis`) and iii) deciding what to print to the output file (`what2print`) iv) final calculations after the scan is over (`UserFinalCalcs`). There are also template functions for testing the stability of the potential (boundedness from below) and if the minimum is global. These functions are already implemented for the 2HDMs.

e) `UserInitCalcs`**:** This function only runs once and is used to set up whatever user defined extra variables, interfaces, etc... that your scan needs. What you are most likely to want to edit here is the following line

```
setprint(1,26);
```

This controls how many points you want to store in memory before printing to a file (first argument) and how many variables you are going to save for each point generated in the scan (second argument). For example, if you want to keep in memory 100 points before dumping to a file (say for example if you are running the code in a cluster) the first argument should be changed to 100.

Typically, you will want to save information on the parameter space point such as the masses, vacuum expectation values, mixing matrices, potential couplings or other variables which are already available in the code. For examples, for the 2HDM implementation there are also cross section ratios, branching ratios etc... which are already included internally in the class and are printed in the `what2print` function. But you are also likely to want to calculate new quantities (for each point) and then add them in the `what2print` function. For that effect there is a vector of double precision variables available that you can use to store extra data (in the `UserAnalysis` function), which you can access in other functions of the class (such

as the `what2print` function). You can resize this vector here (`UserInitCalcs`) by editing:

```
extra_data.resize(0);
```

f) **UserAnalysis:** The bulk of your analysis and calculations will be done inside `UserAnalysis`. This function is called for all points that pass the all internal tests on the potential (such as stability, etc... as mentioned above). Its purpose is to define any other tests you want to impose on the parameter space point. As your tests/calculations are performed, for each point, you also have the option to store the quantities in the `extra_data` vector.

```
bool TwoHDM::UserAnalysis(PhiRef & Phi,LambdaRef & L,MassRef & Mass,
     MmixingRef & Mixing){
  ////////////////////////////////////////////////////
  ////  ENTER CODE FOR YOUR TESTS DURING THE SCAN   ////
  ////////////////////////////////////////////////////
  ...
  if(SomeTest==false)
    return false;
  ...
  if(SomeOtherTest==false)
    return false;
  ...

  return true;
}
```

Note that this function returns `true` or `false`, so if you wish to reject the point before output you can introduce a test (as above) and the point will be rejected without saving (a new point will be attempted to generate). Such a condition could be for example a cross-section limit to be compared with a predicted value computed from the current parameter space point.

g) **what2print:** This function is used to set up what data to print for each successful point of the scan. You can simply use the the `points2print[store_vector_index][]` to select the variables to be printed. For example:

```
void TwoHDM::what2print(){

  // input parameters
  points2print[store_vector_index][0]=mHheavy;
  points2print[store_vector_index][1]=mHlight;
  points2print[store_vector_index][2]=mA;
  points2print[store_vector_index][3]=mHcharged;
  points2print[store_vector_index][4]=alpha;
  points2print[store_vector_index][5]=tanbeta;

  //couplings
  for(int i=0;i<Cp.L.size();++i)
    points2print[store_vector_index][6+i]=Cp.L[i];
  points2print[store_vector_index][14]=bsgamma;
```

Note: Do not try to change the `store_vector_index` yourself. The print internal routines do so automatically.

h) **Compile the code and generate 10 points**:

Now we are ready to compile the code. In the terminal do

```
$ make
```

to compile the code. Then use the command

```
$ ./ScannerS -i TwoHDM.in --nscan 10
```

to specify the name of the input file (the default is assumed to be `model.in`). An output file is generated with the default name `model.out`[2]. If you wish to specify a different file name run with the option `-o`, i.e.

```
$ ./ScannerS -i TwoHDM.in -o output_file_name --nscan 10
```

It is also possible to redirect `std::clog` and `std::cerr` so that log and error messages go to specific text files. For a list of available options run

```
$ ./ScannerS --help
```

Finally open the output file `model.out` to see the results for the 10 points. Note that the format of the output file is completely user defined, so you can change it in `ScannerSUserTwoHDM.cpp` according to your needs. Furthermore, if you are printing point by point (`setprint(1,x)`), you can use `std::cout` anywhere in the code to print in the output file.

i) **Run 1 point in `VERBOSE` mode**: A particularly useful feature of the code is the `VERBOSE` mode. This option allows for the user to see an automatic output of the internal analysis done by the code, for each point it attempts to generate before the user applies any rejection in the `UserAnalysis` function. To run in verbose mode you can compile with the following flag (first line cleans the compilation)

[2]Note that `std::cout` is redirected by default to output to `model.out`.

```
$ make clean
$ make MODE=-DVERBOSE
```
Then run
```
$ ./ScannerS -i TwoHDM.in
```
Since the program attempts various points before accepting a valid one, the output contain several summaries for each attempted point as well as the reason for the point rejection (for the already implemented functions). Focusing on the last one
```
**********************************************************
**********************************************************
******** SUMMARY INFO FOR THIS ATTEMPTED POINT *********
**********************************************************
**********************************************************
...
```
we find first information on the basis of states that was used to decompose the scalar states into physical states[3]
```
------------------
--- New basis ----
------------------

--- Blocks which will mix ---
*** Block 0
v[0]= 0.000000e+00 dphi0 + 0.000000e+00 dphi1 + 9.893147e-01 dphi2
+ 0.000000e+00 dphi3 + 0.000000e+00 dphi4 + 0.000000e+00 dphi5 +
 -1.457960e-01 dphi6 + 0.000000e+00 dphi7 +
v[1]= ...

--- Non-degenerate Curved diagonal directions ---
v[2]= ...

--- degenerate Curved diagonal directions ---
*** Group 0
v[3]= ...
v[4]= ...
```
```
--- Goldstone directions ---
v[5]= ...
v[6]= ...
v[7]= ...
```
where the first block contains the two CP even Higgses (h, H), then we have the CP odd A, two real degenerate degrees of freedom which correspond to the charged Higgs $H^{\pm}$, and finally the three Goldstones. The program always orders the states in this

[3]We show here the full numerical decomposition only for the first vector for brevity.

way, i.e. first mixing blocks, then non-degenerate eigen-states, degenerate eigenstates and at last massless states.

Next, there is some information on which parameters were left as independent

```
-----------------------------------------------------------------
--- Independent parameters generated after VEVs and Mixings ---
-----------------------------------------------------------------

Coupling 7 is independent.
Mass of state 0 is independent.
Mass of state 1 is independent.
Mass of state 2 is independent.
Mass of state 4 is independent.

----------------------
--- Mixing matrix ----
----------------------
...
```

and as expected there are 4 masses (for $h, H, A, H^{\pm}$) and a free coupling (m_{12}^2 in this example). The convention is that: i) the program always tries to leave as many physical masses as possible as independent (to be scanned over) ii) the leftover couplings that are left independent are the last ones in the numbering scheme entered in the expression provided in the *Mathematica* file `ScannerSInput.nb`. Thus the user can change which coupling is left as independent by changing the numbering.

Finally, note that VEVs and mixings (or corresponding parameterizations) are always scanned over, so they are independent parameters are well.

For this point we also have

```
----------------------
------ Masses --------
----------------------
M[0]=1.250000e+02
M[1]=2.425054e+02
M[2]=1.453113e+02
M[3]=2.601380e+02
M[4]=2.601380e+02
M[5]=0.000000e+00
M[6]=0.000000e+00
M[7]=0.000000e+00
----------------------
------ Couplings -----
----------------------
L[0]=-1.430257e+04
L[1]=2.420284e+04
```

```
L[2]=-2.252956e-01
L[3]=6.395826e-01
L[4]=6.437778e+00
L[5]=8.029378e-01
L[6]=-1.763944e+00
L[7]=2.035475e+03
----------------------
-------- VEVs --------
----------------------
Phi[0]=0.000000e+00
Phi[1]=0.000000e+00
Phi[2]=2.358903e+02
Phi[3]=0.000000e+00
Phi[4]=0.000000e+00
Phi[5]=0.000000e+00
Phi[6]=6.979807e+01
Phi[7]=0.000000e+00
```

2 General Potential (*Mathematica* generated)

ScannerS was primarily developed to be a general tool, capable of analysing any scalar potential. So, if you wish to study a potential which is not implemented in one of the example class models (2HDM and xSM2), you can do so without any change to the core program.

2.1 Inserting the model in *Mathematica* – `ScannerSInput.nb`

Open the `ScannerSInput.nb` notebook. It contains the expression for the scalar potential (`Vscalar`)

`Vscalar = ComplexExpand[L[0]Φ1Dag.Φ1+L[1]Φ1Dag.Φ1 +...`

as well as: i) definitions of the ranges for the various parameters in the scan, ii) switches to turn on and off options and iii) variables to indicate the number of couplings and fields. Table 1 lists all the variables available in the notebook. You can use the notebooks inside the examples folder as reference, those were used to generate the input (.in) files used by the 2HDM and xSM2 classes to define their potential.

Define the Potential: The first thing you need to do is set `Nreal` to the number of real fields and `Ncoup` to the number of couplings for the model. Then you need to define the expression for the potential. Following the notation of the code, an arbitrary potential `Vscalar` is a real function which can always be written in terms of a number `Nreal` of canonically normalised fields ϕ`[i]`, as a linear form over a set of `Ncoup` real couplings `L[a]`, i.e.

$$\texttt{Vscalar} = \sum_{a=0}^{\texttt{Ncoup-1}} V(\phi)_a L[a] \,. \tag{2.1}$$

Variable	Description	Remarks
`Nreal`	Number of (real) scalar degrees of freedom/fields (ϕ`[i]`).	A decomposition into canonically normalised real scalar fields is always possible.
`Ncoup`	Number of (real) couplings (`L[a]`).	A (linear) decomposition into real couplings is always possible.
`Vscalar`	Expression for the scalar potential in terms of ϕ`[i]` and `L[a]`.	
ϕ`Min[i]`, ϕ`Max[i]` `i` $= 0, ...,$ `Nreal-1`	Ranges for the scan over VEV of field ϕ`[i]`.	These are overridden if `NparamsVEVs`> 0.
`LMin[a]`, `LMax[a]` `a` $= 0, ...,$ `Ncoup-1`	Allowed ranges for the scan over couplings `L[a]`.	
`massMin[k]`, `massMax[k]` `k` $= 0, ...,$ `Nreal-1`	Allowed ranges for the scan over masses of the physical states.	
`NparamsVEVs`	Number of parameters used in the parametrisation of the VEVs.	OPTIONAL – set to 0, to turn off.
ϕ`ParMin[j]`, ϕ`ParMax[j]` `j` $= 0, ...,$ `NparamsVEVs-1`	Ranges for the scan over the parameters ϕ`Par[j]` used in the reparametrisation of the VEVs.	OPTIONAL – used if `NparamsVEVs` > 0.
`NparamsMix`	Number of parameters used in the parametrisation of the mixing matrix.	OPTIONAL – set to 0, to turn off.
`MixParMin[j]`, ϕ`MixParMax[j]` `j` $= 0, ...,$ `NparamsMix-1`	Ranges for the scan over the parameters `MixPar[j]` used in the parametrisation of the mixing matrix.	OPTIONAL – used if `NparamsMix` > 0.
`NparamsML`	Number of parameters used in the parametrisation to impose extra conditions at the last stages of the scan.	OPTIONAL – set to 0, to turn off.
`MLParMin[j]`, ϕ`MLParMax[j]` `j` $= 0, ...,$ `NparamsML-1`	Ranges for the scan over the parameters `MLPar[j]` used in the parametrisation of the extra conditions.	OPTIONAL – used if `NparamsML` > 0.
`InputFileName`	Name of the input file generated by the notebook.	Default is `model.in`.

Table 1. List of variables defining the run parameters in `ScannerSInput.nb`.

The current version of the code assumes that $V(\phi)_a$ are polynomials in the fields of order up to 4, i.e. a tree level re-normalisable potential. Thus the user must enter an expression which evaluates (up to constant numerical factors) to an expression in these two sets of quantities, (`L[a]`,ϕ`[i]`), only.

Scan boxes for fields, couplings and physical masses: There are several variables available to set the ranges for the parameters being scanned over. The recommended strategy in doing this is as follows.

a) For field vacuum expectation values (VEVs) the maximum and minimum range for fields that do not get a VEV must be set to zero. In particular it is always safer to start by setting all ranges to zero and then set the ranges that are non zero. For example:

```
(* Initialising all ranges to zero *)
For[i = 0, i < Nreal, ++i,
ϕMin[i] = 0;
ϕMax[i] = 0;
]
```

Then, the non-trivial ranges are set as follows (for example)

```
ϕMin[0] = 246;
ϕMax[0] = 246;
ϕMin[1] = 0;
ϕMax[1] = 500;
```

where we have set the first VEV to be fixed (but non-zero) and the second to be scanned in the interval ϕ`[1]` $\in [0, 500]$. For other cases where a parametrisation is necessary (such as the 2HDM) see the next paragraph. Note that it is up to you to choose units which are consistent with the other tests defined in the `UserAnalysis`. From the point of view of the potential there is only one mass scale unit set by the VEVs.

b) For the couplings of the potential, `L[a]`, a similar strategy applies to their ranges `LMin[a]`, `LMax[a]`.

c) For the mass ranges (`massMin[a]`, `massMax[a]`) please note that it is ScannerS's internal convention to have the Goldstone masses (or any other massless scalar state) at the end of the array. In the 2HDM, for example, you could set them as

```
(* Set all mass ranges to be zero -- last ones are the Goldstones *)
For[i = 0, i < Nreal, ++i,
massMin[i] = 0;
massMax[i] = 0;
]
(* Specific mass for a certain particle *)
massMin[0] = 125; (*Observed Higgs mass set to 125*)
massMax[0] = 125;
massMin[1] = 50; (*Other Higgs*)
massMax[1] = 125;
massMin[2] = 50; (*A*)
massMax[2] = 500;
massMin[3] = 50; (*Re(H+)*)
massMax[3] = 500;
massMin[4] = 50; (*Im(H+)*)
massMax[4] = 500;
```

Scan boxes for parametrisations: In addition, there are 3 sets of optional parameters which are associated with three types of re-parametrisations. Their role is to add flexibility so that the user is able to scan the parameters not necessarily in hypercubic boxes, but in more generic slices/volumes of the parameter space. These are discussed in detail in Sec. 2.2.3 (see also Table 1 for a short description).

Generate `model.in`: After defining your potential and ranges select *Evaluate Notebook*. An input file, `model.in` (you can change the file name in `InputFileName` at the end of the notebook) is generated in the working directory. This serves as input for the `C++` program.

2.2 Defining the analysis (`ScannerSUser.cpp`)

After the *Mathematica* notebook `ScannerSInput.nb` has been executed to generate the input file, you must define the analysis you wish to run in the file `ScannerSUser.cpp`. This file is simply a set of template functions (i.e. that are defined by the user) which are called before, during and after the scan. All other operations that are not defined in this file are performed automatically by the code to generate a valid local minimum[4].

The various template functions are discussed in the following sections.

2.2.1 User analysis functions

The first three functions are responsible for various user analysis tasks. They are:

a) `UserInitCalcs`: The function looks like

[4]For each point in the scan the physical spectrum is automatically detected and tree level unitarity constraints are applied for arbitrary models (see Sect. 2.2.4 for further details).

```
void Potential::UserInitCalcs(void){

////////////////////////////////////////////////////////////////////
// ENTER CODE FOR YOUR INITIAL CALCULATIONS BEFORE THE SCAN STARTS //
////////////////////////////////////////////////////////////////////

}
```

The user can enter here initial calculations/operations that are done (only once) before the scan starts. The most important of which will be to set up the print and extra data configurations (see `UserInitCalcs` in 1.2). You may also want to create a table of values to be used in the rejection/acceptance step during the scan, or write some headers in the output file using `std::cout<<...`.

b) `UserAnalysis`: This is where the user writes the analysis tests done for each point that is generated in the scan. The general structure looks like

```
bool Potential::UserAnalysis(PhiRef & Phi,LambdaRef & L,
MassRef & Mass, MmixingRef & Mixing){
  ////////////////////////////////////////////
  // ENTER CODE FOR YOUR TESTS DURING THE SCAN //
  ////////////////////////////////////////////

  if(_condition_)
    return false;

  ...

  return true;
}
```

Here you can write conditions to be tested. If such conditions imply that the point must be rejected then they must `return false;` otherwise the analysis of the point continues until the function returns `true`.

c) `what2print`: As explained in 1.2, use this function to add the variables you want to print to the `points2print[store_vector_index][]` vector. ScannerS will store the data from a few points (the 1st argument of `setprint(int,int)` that you set in `UserInitCalcs`), and then print all at once to a file once the buffer is filled (for basic runs we advise you to use a buffer with only one point, i.e. `setprint(1,x)`).

d) `UserFinalCalcs`: This is analogous to `UserInitCalcs`, except that it runs once after the scan is done. This function looks like

```
void Potential::UserFinalCalcs(void){

////////////////////////////////////////////////////////////////////
//ENTER HERE CODE FOR YOUR FINAL CALCULATIONS AFTER THE SCAN IS DONE
////////////////////////////////////////////////////////////////////

}
```

2.2.2 Global stability and boundedness from below

Two theoretical constraints that are not yet implemented for general models in the code are the global minimum condition (i.e. that the local minimum that was generated is actually the global one) and the boundedness from below condition (i.e. that the potential does not have runaway directions). For this purpose there are two (user defined) template functions, where the user can add any expression/procedure to test these conditions.

a) `CheckStability`: This function is supposed to contain the conditions that test whether the potential is bounded from below. The basic code structure is

```
bool Potential::CheckStability(LambdaRef & L){
  //This example is for the 2HDM
  if(L[3]>0 && L[4]>0 && L[5]+sqrt(L[3]*L[4])>0 && L[5]+L[6]
    -abs(L[2])+sqrt(L[3]*L[4])>0)
    return true;
  else
    return false;
}
```

so the user can define any function that depends on the parameters of the potential.

b) `CheckGlobal`: Similarly this function contains conditions that test whether the minimum is global. The basic code structure is similar:

```
bool Potential::CheckGlobal(PhiRef & Phi,LambdaRef & L,Potential & V)
{
  //// This example is for the 2HDM
  //// Compute discriminant D
  double kd = pow((L[3]/L[4]),0.25);
  double Disc=L[7]*(L[0]-kd*kd*L[1])*(Phi[6]/Phi[2]-kd);
  if(Disc <= 0)
    return false;//If condition not met, reject point

  return true;
}
```

2.2.3 Other user defined options

Finally, the last three template functions in `ScannerSUser.cpp` add flexibility to allow for more generic parametrisations of the scan. The first function is particularly important.

VEV scan re-parametrisation: An important feature for more advanced models, is to be able to impose more generic symmetry breaking patterns where, for example, there are relations among VEVs. Such an example is the 2HDM model, Eq. (1.1), where instead of having the two VEVs v_1, v_2 generated uniformly inside a square, one wants to generate them on a circle with radius $v = 246$ (see Eq. (2.2)).

$$\begin{aligned} v_1 &= v \cos\beta \\ v_2 &= v \sin\beta \ . \end{aligned} \tag{2.2}$$

Thus, the program allows for the user to define a generic re-parametrisation of the VEVs in the form

$$\begin{aligned} \texttt{Phi[0]} &= \texttt{f}_0\texttt{(PhiPar[0],...,PhiPar[NparamsVEVs-1])} \\ \ldots &= \ldots \\ \texttt{Phi[Nreal-1]} &= \texttt{f}_{\texttt{Nreal-1}}\texttt{(PhiPar[0],...,PhiPar[NparamsVEVs-1])} \end{aligned} \tag{2.3}$$

where the right hand side functions are defined in the function `MyPhiParametrization` of the `ScannerSAnalysis.cpp` file. The ranges for the parameters `PhiPar[0]` are defined in the notebook `ScannerSInput.nb` similarly to those for the couplings `L[a]`, etc... (see Table 1). For the 2HDM example, the code is simply (compare with Eq. (2.2))

```
void Potential::MyPhiParametrization(const PhiParamVec & PhiPar,
     PhiVec & Phi){
  ////////////////////////
  // Variables:
  //   PhiPar[] : Vector of VEV parameters
  //   Phi[] : Vector of VEVs
  ////////////////////////
  // Description:
  //...

  Phi[2]=246*cos(PhiPar[0]);
  Phi[6]=246*sin(PhiPar[0]);
}
```

Note the great flexibility of this function since the user could have called any other expression/function on the right hand side.

Mixing matrix parametrisation: Regarding the mixing matrix, the code generates it automatically regardless of any parametrisation. This is done by using a method which generates rotation matrices uniformly with respect to the Haar measure. However, in many models the user may want to use a specific parametrisation (say a set of angles). The code allows this through a template function where an internal mixing matrix can be specified generically in the form

$$\texttt{MixInternal[i][j]} = F_{ij}\texttt{(MixPar[k],Phi[k],PhiPar[k])} \tag{2.4}$$

where one notes that this depends on a set of parameters `MixPar[k]`, but it can also depend on the VEVs or respective parametrisation. The structure of the function is

```
void Potential::MyInternalMixing(const PhiParamVec & PhiPar,
     const PhiVec & Phi, MixingparamVec & MixPar,
     vector< vector<double> > & MixInternal, RandGen & r){
  // Here the parameter MixPar[] was chosen to actually depend on the
  // VEV parameters PhiPar[0]. This is actually a decoupling limit
  // relation for the 2HDM if one sets MixPar[0]=α and
  // PhiPar[0]=β in the 2HDM
  MixPar[0]=PhiPar[0]-acos(-1)/2e0;
  //Mixing matrix parametrised by the α angle
  MixInternal[0][0]=cos(MixPar[0]);
  MixInternal[0][1]=-sin(MixPar[0]);
  MixInternal[1][0]=sin(MixPar[0]);
  MixInternal[1][1]=cos(MixPar[0]);
}
```

Imposing extra conditions: Finally there is a template function which allows for extra conditions to be imposed among all remaining free parameters in the last stage of the generation of a point (to find out the independent couplings at the end of each point you should run a point in `VERBOSE` mode). This function is `MyCoupMassRelations`.

2.2.4 Automatic modules

A first description of the numerical strategy used for the automatic tasks of the code (to generate a local minimum obeying tree level unitarity constraints), was provided in [1]. The details of the method are not essential for a first use of the program so a full description with examples will be presented in a future version of the manual.

3 Using the external interfaces

We provide several interfaces to external programs (or their library versions), which allow for the user to access capabilities of each program library within the analysis in `ScannerSUser*.cpp`. Nevertheless, most of these external codes are written in different programming languages ranging from `Fortran77/90` to `C/C++`. Thus the instructions below should be followed carefully for each specific interface. For almost all the interfaces, there is a corresponding path variable in the editable header of the `makefile` which must be defined to activate the interface, or left empty[5] to de-activate it.

3.1 SuperIso (tested with v3.3)

The SuperIso library [3] is linked directly to the ScannerS code, so all SuperIso functions can be called directly in the code. The interface is done through a SLHA file which

[5] With no white space – hit enter to ensure this, i.e. by creating a newline after the = symbol.

is created to pass as an argument to the SuperIso functions in the analysis. Currently there is a function to make this automatic for the 2HDM model (instructions below).

The steps to use the interface are:

a) Specify the path to the SuperIso source files in the `makefile`, i.e. in the following line (to de-activate the interface leave this empty with NO white space):

```
SuperisoPath=<Path to directory here>
```

b) Edit your `ScannerSUser.cpp` analysis file at the line where you want to call SuperIso and write the following lines:

- **Create the `tempsuperiso.lha` file** – Write a line to call a function which creates the file `tempsuperiso.lha` in your ScannerS working directory. For the 2HDM there is a special function already defined to make this easier

```
void CreateInputFileSuperiso2HDM(double mHlight,double mHheavy,
     double mA,double mHcharged,double alpha,double tanbeta,
     int ModelType);
```

  Alternatively you can define your own function to create the `tempsuperiso.lha` file. All variable names follow the usual conventions for the 2HDM including the `ModelType` variable ($= 1, 2, 3$ or 4), which defines the Yukawa type as in SuperIso.

- **Call a SuperIso function** – Write a line to compute a specific observable that you wish to use. Call the corresponding SuperIso function as usual by passing the filename (see superiso manual [3]). ScannerS already contains an internal static variable to pass the file name, `superisofile`, which holds the name `tempsuperiso.lha`. For example to compute the $B \to X_s\gamma$ branching ratio one calls

```
bsgamma_calculator(superisofile);
```

3.2 HiggsBounds/Signals

Both HiggsBounds and HiggsSignals can be linked by indicating the correct path to the library in the makefile (if not in the standard search paths). All functions are available to be called (check declarations in `ScannerScore/HBWrap.h` and `ScannerScore/HSWrap.h`). In addition, we have provided a further convenient interface for HiggsBounds, through a special object, to ease the task of initialising, passing the input, and running HiggsBounds. The typical steps are described below.

Initialising and running HiggsBounds (tested with version 4.1.2)

The basic steps to use the interface are:

a) Specify the path to the HiggsBounds source files in the `makefile`, i.e. in the following line (to de-activate the interface leave this empty with NO white space):

```
HBPath=<Path to directory here>
```

Note: The next two steps are only necessary for *Mathematica* generated potentials, the TwoHDM and xSM2 classes already handle this in the constructor.

b) At the top of the `ScannerSUser.cpp` file declare a global `HBobject`.

```
...
//////////////////////////////
// USER ANALYSIS TEMPLATES //
//////////////////////////////

HBobject HiggsBoundsAnalysis;
```

c) Edit your `UserInitCalcs` function to define HiggsBounds's initial parameters.

```
void Potential::UserInitCalcs(void){

  int nHzero=3; //Number of neutral scalars
  int nHplus=1; //Number of singly charged scalars
  int whichanalyses=3; //which Higgs bounds analysis
  int whichinput=1; //type of input you will provide (1=hadronic)

  HiggsBoundsAnalysis.rewrite_values(nHzero,nHplus,whichanalyses,
  whichinput);
```

This object (`HiggsBoundsAnalysis`) contains all the arrays and functions that are necessary to run HiggsBounds.

d) In the `UserAnalysis` function, set the values for your HiggsBounds input as usual and run in the following way:

```
bool Potential::UserAnalysis(PhiRef & Phi,LambdaRef & L,
     MassRef & Mass,MmixingRef & Mixing){
  ////////////////////////////////////////////////////
  // ENTER CODE FOR YOUR TESTS/OUTPUT DURING THE SCAN //
  ////////////////////////////////////////////////////

  ...
  HiggsBoundsAnalysis.Mh[0]=126;
  HiggsBoundsAnalysis.Mh[1]=90;
  HiggsBoundsAnalysis.Mh[2]=300;

  HiggsBoundsAnalysis.CP[0]=1;
  HiggsBoundsAnalysis.CP[1]=-1;
  HiggsBoundsAnalysis.CP[2]=1;
```

```
  HiggsBoundsAnalysis.CP[0]=1;
  HiggsBoundsAnalysis.CP[1]=-1;
  HiggsBoundsAnalysis.CP[2]=1;

  HiggsBoundsAnalysis.BR_hjhihi[1][2]=0.3;
  HiggsBoundsAnalysis.BR_hjhihi[2][1]=0.5;
...

  int HBresult,chan, ncombined;
  double obsratio;
  //Call of HiggsBounds
  HiggsBoundsAnalysis.run(HBresult,chan,obsratio,ncombined);
...
```

Note that all vectors and matrices have exactly the same names as found in the appendix tables of the HiggsBounds manual [4]. We provide a list in Appendix A.

e) The output of the run is in the usual variables `HBresult,chan,...` as defined in HiggsBounds.

Initialising and running HiggsSignals (tested with version 1.2.0)

HiggsSignals must be called always together with HiggsBounds, i.e., in the initialisation step HiggsSignals must be initialised after HiggsBounds, and the same when running. The basic steps to use the interface are even simpler, so no special object is necessary in this case:

a) Specify the path to the HiggsSignals source files in the `makefile`, i.e. in the following line (to de-activate the interface leave this empty with NO white space):

```
HSPath=<Path to directory here>
```

b) In the `UserInitCalcs` (in the `ScannerSUser*.cpp` file) initialise HiggsSignals as follows:

```
...

void UserInitCalcs(void){
  ...
  initialize_higgssignals_latestresults_(nHzero,nHplus);
  int pdf=2;
  // choose probability density function of Higgs mass around
  // the measured central value (HiggsSignals)
  setup_pdf_(pdf);
  ...
}
...
```

c) In the `UserAnalysis` function run HiggsSignals right after running HiggsBounds, i.e.

```
  //Call of HiggsBounds
  HiggsBoundsAnalysis.run(HBresult,chan,obsratio,ncombined);

  //Higgs Signals example run
  int mode=1;
  double csqmu;
  double csqmh;
  double csqtot;
  int  nobs;
  double Pvalue;
  run_higgssignals_(mode,csqmu,csqmh,csqtot,nobs,Pvalue);
...
```

d) The output of the run is in the usual variables `csqmu,csqmh,...` as usual in HiggsSignals. Note also that the `mode` of the statistical analysis must be decided by the user (see the HiggsSignals manual for more information [5]).

3.3 SusHi (tested with v1.1.0)

Currently there is an interface to the `SusHi` cross section calculator [6], to extract the Higgs production cross section at NNLO in the gluon fusion and $b\bar{b}$ channels. In the analysis, the user has to call a function to create a sushi input file `tempsushi.in`. There is a special function already defined for the 2HDM model (instructions below), however users can define their own function to create the file.

The steps to use the interface are:

a) Specify the path to the `SusHi` source files in the `makefile`, i.e. in the following line (to de-activate the interface leave this empty with NO white space):

```
SusHiPath=<Path to directory here>
```

b) **Create the `tempsushi.in` file** – In your `ScannerSUser*.cpp` analysis file, write a line to call a function which creates the file `tempsushi.in` in your ScannerS working directory. For the SM and the 2HDM there are special functions already defined to make this easier

```
void CreateInputFileSusHi2HDM(int particle,int pp_ppbar,
     int order, double CM_energy,double mHlight,double mHheavy,
     double mA, double mHcharged,double alpha,double tanbeta,
     int ModelType);

void CreateInputFileSusHiSM(int pp_ppbar,int order,
     double CM_energy,double mHiggs);
```

The variables `particle`, `pp_ppbar`, `order`, `CM_energy` follow the SusHi numbering

(see examples in the ScannerS distribution). All other variable names follow the usual conventions for the 2HDM, except for the `ModelType` variable (= 1, 2, 3 or 4) where the 3 and 4 types are swapped (3 means the X-type, i.e. lepton specific, and 4 is the Y-type, flipped).

Alternatively you can define your own function to create the `tempsushi.in` file.

c) **Call of SusHi** – Write a line to call `SusHi` by using the following external fortran function directly in the C++ analysis code in the following way

```
sushixsection_(xsecggh_out,errxsecggh_out,xsecbbh_out,
     errxsecbbh_out);
```

where all arguments are `double` precision output variables passed by reference (i.e. where the output is written), and correspond to the gluon fusion cross section and error, and $b\bar{b}$ cross section and error.

Note: Calling SusHi through the interface directly, for each parameter space point, is a bit slow due to the complexity of the calculation. The same applies to SuperIso (though this is faster). Thus for production scans where millions of points are to be generated it is more efficient to produce numerical tables of the observables that these programs compute. In a future version of the manual a detailed description of some tables that have been produced in [2] for the 2HDM will be provided. These tables are stored in the sub-folder `ScannerScore/data/` and the way the function can be traced back by looking at the function `LoadAll2HDMTables(void)` which is in `ScannerScore/ScannerSModels/2HDM/ScannerSTwoHDM.cpp`.

3.4 Hdecay

Currently there is a direct interface to the `Hdecay` calculator [7], for the 2HDM model, with two specific functions to be used in the user analysis (instructions below). However, if the user provides functions to create the `hdecay.in` input file and to read the output files created by `Hdecay`, the interface can be still used for other models, since there is a C++ function

```
void HdecayCalc(void);
```

which can be called to run Hdecay directly in the `ScannerSUser.cpp` analysis file.

The steps to use the interface are:

a) Specify the path to the Hdecay source files in the `makefile`, i.e. in the following line (to de-activate the interface leave this empty with NO white space):

```
Hdecaypath=<Path to directory here>
```

b) Edit at the line where you want to call `Hdecay` and write the following lines:

- **Create the `hdecay.in` file** – In your `ScannerSUser*.cpp` analysis file, write a line calling a function which creates the file `hdecay.in` in your ScannerS working directory. For the 2HDM there is a special function already defined

```
void CreateInputFileHdecay2HDM(int Type,double tanbeta,
     double alpha,double mHlight,double mHheavy,double mA,
     double mCharged,double m12sq);
```

Alternatively you can define your own function to create the `hdecay.in` file.

- **Run `Hdecay`** – For the 2HDM model there is a special function for this step, which stores all branching ratios and decay widths in a map in memory (for easy access in the code – see full list in appendix B). This function must be called **exactly** as in the following line:

```
HdecayCalc2HDM(HdecayA,HdecayHlight,HdecayHheavy,
     HdecayHcharged,HdecayTop);
```

Alternatively, you can call the following generic function to run `Hdecay`, which only creates the usual `Hdecay` output files:

```
void HdecayCalc(void);
```

In this case, you will have to write your own functions to read the output from the files created by `Hdecay` (Note: further interface functions to ease this step will be provided in future ScannerS releases).

- **Access the `Hdecay` output** – For the 2HDM model this is done directly by using the map variables that were populated by `HdecayCalc2HDM`. For example the branching ratio for the decay $A \to b\bar{b}$ is accessed through

```
HdecayA["BR(A -> b bbar)"]
```

The full list of variables for the 2HDM is in appendix B.

WARNING: Be very careful to write the string passed as an argument to the map **exactly** (including spaces) as in the appendix B. A typing mistake will create an new element in the map and will return zero! In future releases a mechanism to avoid this behaviour will be introduced for safety.

3.5 Micromegas (tested with v3.6.9.2)

The MicrOmegas interface works a bit differently from the other interfaces, because the ScannerS code must be placed inside a MicrOmegas project directory. Before running MicrOmegas with ScannerS, you will need to prepare the MicrOmegas project directory for your specific model (see [8]) and then place the ScannerS code inside. Note that all MicrOmegas functions can be called directly in the ScannerS code as you would in a MicrOmegas project.

The steps to use the interface are:

a) Create a new project in the MicrOmegas installation directory and check the model (follow instructions in `lapth.cnrs.fr/micromegas/`).

b) Drop your ScannerS user files, `makefile` and the source directory (`ScannerScore`) in the newly created MicrOmegas project directory (basically all contents of a typical ScannerS project). Only after this step will you be able to compile the code since the makefile links MicrOmegas sources which are assumed to be one directory up.

c) Activate the MicrOmegas interface in the `makefile` by editing the following line (otherwise to de-activate leave this empty with NO white space)

```
MicromegasOn==ON
```

d) **Setting values of model variables** – The first step in the analysis is to pass the values of the model parameters for the current parameter space point, to MicrOmegas. Let's say that, for example, you have implemented a 2HDM inert model with dark matter, and that you have named the masses of the light, heavy, CP odd and charged Higgses in the MicrOmegas project as `Mh,MHH,MAs,MHc` and m_{12}^2 as `m12sqr`, etc... Then, in your `ScannerSUser*.cpp` analysis, you would pass using the MicrOmegas `assignVal` function

```
assignVal("Mh",mHlight);
assignVal("MHH",mHheavy);
assignVal("MAs",mA);
assignVal("MHc",mHcharged);
assignVal("m12sqr",L[7]);
...
```

e) **Perform `MicrOmegas` calculations**– Now that the model parameters are set you can call any function which computes MicrOmegas quantities. For example to compute the dark matter relic density you could use a piece of code like:

```
char cdmName[10];
int err=sortOddParticles(cdmName);
if(err) { cerr<<"Can't calculate "<<cdmName<< endl; exit(-1);}

// Compute dark matter contributions from dark matter particle
double Omega,Xf,cut=0.01,Beps=1e-5;
Omega=darkOmega(&Xf,1,1e-5);
```

For further details check the ScannerS examples and the MicrOmegas manual.

A List of available arrays used in the HigssBounds interface

In this section we provide a full list of the arrays that are stored in an objects of type `HBobject`, which are used to pass the input to HiggsBounds, and can be accessed by the user.

```
//Hadronic input variables
double * Mh; //All objects of this type are vectors
double * MhGammaTot;
int * CP;
double * CS_lep_hjZ_ratio;
double * CS_lep_bbhj_ratio;
double * CS_lep_tautauhj_ratio;
double * CS_lep_hjhi_ratio;
```

```
double * CS_tev_hj_ratio;
double * CS_tev_hjb_ratio;
double * CS_tev_hjW_ratio;
double * CS_tev_hjZ_ratio;
double * CS_tev_vbf_ratio;
double * CS_tev_tthj_ratio;
double * CS_lhc7_hj_ratio;
double * CS_lhc7_hjb_ratio;
double * CS_lhc7_hjW_ratio;
double * CS_lhc7_hjZ_ratio;
double * CS_lhc7_vbf_ratio;
double * CS_lhc7_tthj_ratio;
double * CS_lhc8_hj_ratio;
double * CS_lhc8_hjb_ratio;
double * CS_lhc8_hjW_ratio;
double * CS_lhc8_hjZ_ratio;
double * CS_lhc8_vbf_ratio;
double * CS_lhc8_tthj_ratio;
double * BR_hjss;
double * BR_hjcc;
double * BR_hjbb;
double * BR_hjmumu;
double * BR_hjtautau;
double * BR_hjWW;
double * BR_hjZZ;
double * BR_hjZga;
double * BR_hjgaga;
double * BR_hjgg;
double * BR_hjinvisible;
double ** BR_hjhihi; //This is a MATRIX!!! Access as usual.

//Partonic input variables, which are not also above
double * CS_gg_hj_ratio;
double * CS_bb_hj_ratio;
double * CS_bg_hjb_ratio;
double * CS_ud_hjWp_ratio;
double * CS_cs_hjWp_ratio;
double * CS_ud_hjWm_ratio;
double * CS_cs_hjWm_ratio;
double * CS_gg_hjZ_ratio;
double * CS_dd_hjZ_ratio;
```

```
double * CS_uu_hjZ_ratio;
double * CS_ss_hjZ_ratio;
double * CS_cc_hjZ_ratio;
double * CS_bb_hjZ_ratio;
//effective coupling input variables
double * g2hjss_s;
double * g2hjss_p;
double * g2hjcc_s;
double * g2hjcc_p;
double * g2hjbb_s;
double * g2hjbb_p;
double * g2hjtoptop_s;
double * g2hjtoptop_p;
double * g2hjmumu_s;
double * g2hjmumu_p;
double * g2hjtautau_s;
double * g2hjtautau_p;
double * g2hjWW;
double * g2hjZZ;
double * g2hjZga;
double * g2hjgaga;
double * g2hjgg;
double * g2hjggZ;
double ** g2hjhiZ;//This is a MATRIX!!! Access as usual.

//Charged Higgses
double * MHplus;
double * MHplusGammaTot;
double * CS_lep_HpjHmi_ratio;
double * BR_tWpb;
double * BR_tHpjb;
double * BR_Hpjcs;
double * BR_Hpjcb;
double * BR_Hptaunu;

//Initialisation
int nHzero,nHplus,whichanalyses,whichinput;
```

B List of map variables for the Hdecay interface

The special function for the 2HDM file populates the following list of map variables which can be accessed by typing **exactly** as below [6]:

```
HdecayA["BR(A -> b bbar)"]
HdecayA["BR(A -> tau+ tau-)"]
HdecayA["BR(A -> mu+ mu-)"]
HdecayA["BR(A -> s sbar)"]
HdecayA["BR(A -> c cbar)"]
HdecayA["BR(A -> t tbar)"]
HdecayA["BR(A -> g g)"]
HdecayA["BR(A -> gamma gamma)"]
HdecayA["BR(A -> Z gamma)"]
HdecayA["BR(A -> Z h)"]
HdecayA["BR(A -> A Z)"]
HdecayA["BR(A -> A W+ H-)"]
HdecayA["Width"]
```

```
HdecayHlight["BR(h -> b bbar)"]
HdecayHlight["BR(h -> tau+ tau-)"]
HdecayHlight["BR(h -> mu+ mu-)"]
HdecayHlight["BR(h -> s sbar)"]
HdecayHlight["BR(h -> c cbar)"]
HdecayHlight["BR(h -> t tbar)"]
HdecayHlight["BR(h -> g g)"]
HdecayHlight["BR(h -> gamma gamma)"]
HdecayHlight["BR(h -> Z gamma)"]
HdecayHlight["BR(h -> W+ W-)"]
HdecayHlight["BR(h -> Z Z)"]
HdecayHlight["BR(h -> A A)"]
HdecayHlight["BR(h -> Z A)"]
HdecayHlight["BR(h -> H+ H-)"]
HdecayHlight["BR(h -> W+ H-)+BR(h -> W- H+)"]
HdecayHlight["Width"]
```

[6] The string arguments which describe each decay should be self explanatory

```
HdecayHheavy["BR(H -> b bbar)"]
HdecayHheavy["BR(H -> tau+ tau-)"]
HdecayHheavy["BR(H -> mu+ mu-)"]
HdecayHheavy["BR(H -> s sbar)"]
HdecayHheavy["BR(H -> c cbar)"]
HdecayHheavy["BR(H -> t tbar)"]
HdecayHheavy["BR(H -> g g)"]
HdecayHheavy["BR(H -> gamma gamma)"]
HdecayHheavy["BR(H -> Z gamma)"]
HdecayHheavy["BR(H -> W+ W-)"]
HdecayHheavy["BR(H -> Z Z)"]
HdecayHheavy["BR(H -> h h)"]
HdecayHheavy["BR(H -> A A)"]
HdecayHheavy["BR(H -> Z A)"]
HdecayHheavy["BR(H -> W+ H-)+BR(H -> W- H+)"]
HdecayHheavy["BR(H -> H+ H-)"]
HdecayHheavy["Width"]
```

```
HdecayHcharged["BR(H+ -> c bbar)"]
HdecayHcharged["BR(H+ -> tau+ nu_tau)"]
HdecayHcharged["BR(H+ -> mu+ nu_mu)"]
HdecayHcharged["BR(H+ -> u bbar)"]
HdecayHcharged["BR(H+ -> u sbar)"]
HdecayHcharged["BR(H+ -> c dbar)"]
HdecayHcharged["BR(H+ -> c sbar)"]
HdecayHcharged["BR(H+ -> t bbar)"]
HdecayHcharged["BR(H+ -> t sbar)"]
HdecayHcharged["BR(H+ -> t dbar)"]
HdecayHcharged["BR(H+ -> W+ h)"]
HdecayHcharged["BR(H+ -> W+ H)"]
HdecayHcharged["BR(H+ -> W+ A)"]
HdecayHcharged["Width"]
```

```
HdecayTop["BR(t -> b W+)"]
HdecayTop["BR(t -> b H+)"]
HdecayTop["Width"]
```

References

[1] R. Coimbra, M. O. Sampaio, and R. Santos, *ScannerS: Constraining the phase diagram of a complex scalar singlet at the LHC*, *Eur.Phys.J.* **C73** (2013) 2428, [arXiv:1301.2599].

[2] P. Ferreira, R. Guedes, M. O. P. Sampaio, and R. Santos, *Wrong sign and symmetric limits and non-decoupling in 2HDMs*, arXiv:1409.6723.

[3] F. Mahmoudi, *SuperIso v2.3: A Program for calculating flavor physics observables in Supersymmetry*, *Comput.Phys.Commun.* **180** (2009) 1579–1613, [arXiv:0808.3144]. http://superiso.in2p3.fr/.

[4] P. Bechtle, O. Brein, S. Heinemeyer, G. Weiglein, and K. E. Williams, *HiggsBounds 2.0.0: Confronting Neutral and Charged Higgs Sector Predictions with Exclusion Bounds from LEP and the Tevatron*, *Comput.Phys.Commun.* **182** (2011) 2605–2631, [arXiv:1102.1898]. http://higgsbounds.hepforge.org/HiggsBounds-4-manual.pdf.

[5] P. Bechtle, S. Heinemeyer, O. Stl, T. Stefaniak, and G. Weiglein, *HiggsSignals: Confronting arbitrary Higgs sectors with measurements at the Tevatron and the LHC*, arXiv:1305.1933. http://higgsbounds.hepforge.org/HiggsSignals-1-manual.pdf.

[6] R. V. Harlander, S. Liebler, and H. Mantler, *SusHi: A program for the calculation of Higgs production in gluon fusion and bottom-quark annihilation in the Standard Model and the MSSM*, *Computer Physics Communications* **184** (2013) 1605–1617, [arXiv:1212.3249]. https://sushi.hepforge.org/.

[7] A. Djouadi, J. Kalinowski, and M. Spira, *HDECAY: A Program for Higgs boson decays in the standard model and its supersymmetric extension*, *Comput.Phys.Commun.* **108** (1998) 56–74, [hep-ph/9704448]. http://people.web.psi.ch/spira/hdecay/.

[8] G. Belanger, F. Boudjema, and A. Pukhov, *micrOMEGAs : a code for the calculation of Dark Matter properties in generic models of particle interaction*, arXiv:1402.0787. https://lapth.cnrs.fr/micromegas/.