ScannerS manual: version-1.0.2

Marco O. P. Sampaio,[a] **Rui Santos**[b,c]

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

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

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

E-mail: 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. This first version of the manual provides some information on how to run the code and the usage of the external interfaces. Please refer also to our first publication with the code for details of the numerical strategy [1].

Keywords: Computational Tools, Higgs boson, Beyond Standard Model

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:

- *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.
- 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).

For details on how to use the external interfaces please see Section 3. To have a basic understanding of how the code works it is better to start with the default standalone example in the next section.

1.2 Quick Start: A 2HDM scan

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 potential

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

The steps to follow are:

1. **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.
   - License information and `README` file.
   - The user editable files: `makefile`, `ScannerSInput.nb`, `ScannerSUser.cpp`.

2. **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

```
#...
############################
# 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
#...
```

3. **Execute** `ScannerSInput.nb`: The `ScannerSInput.nb` notebook contains the expression for the scalar potential (`Vscalar`) that the user can adapt

   ```
   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, which are discussed in detail in Sect. 2.1.

   After opening this notebook and selecting *Evaluate Notebook*, an input file, `model.in` is generated in the working directory. This serves as input for the `C++` program (a pre-generated `model.in` file is in the distribution).

4. **Check/edit the User Analysis file –** `ScannerSUser.cpp`:

   The third file to be edited is `ScannerSUser.cpp`. Here the user has 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) final calculations after the scan is over (`UserFinalCalcs`). In this simple example all are empty except the function `UserAnalysis`

   ```
   bool UserAnalysis(PhiRef & Phi,LambdaRef & L,MassRef & Mass,
   MmixingRef & Mixing){
     //////////////////////////////////////////////////////
     // ENTER CODE FOR YOUR TESTS/OUTPUT DURING THE SCAN //
     //////////////////////////////////////////////////////
     ...
   }
   ```
   which prints the values of the Masses (`Mass[i]`), couplings (`L[i]`) and VEVs (`Phi[i]`) to the output file:

```
  ...
  /////////////////////////////////////////
  ////////// Final print out//////////////
  /////////////////////////////////////////

  cout<< "--- Masses ---"<< endl;
  for(size_t i=0; i!=Phi.size();++i){
    cout <<Mass[i]<< endl;
  }
  cout<< "--- Couplings ---"<< endl;
  for(size_t i=0; i!=L.size();++i){
    cout <<  L[i]<< endl;
  }
  cout<< "--- VEVs ---"<< endl;
  for(size_t i=0; i!=Phi.size();++i){
    cout <<  Phi[i]<< endl;
  }
  cout<<endl;

  return true;
}
```

Note that this function returns `true` or `false`, so if the user wishes to reject the point before output (so that the program tries to generate a new point), a condition can be written such as

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

  /////////////////////////////////////////
  ////////// Final print out//////////////
  /////////////////////////////////////////
  ...
}
```

Such a condition could be for example a cross-section limit to be compared with a a predicted value computed from the current parameter space point. Note that the mixing matrix is also available through the variable `Mixing`.

There are two more template functions where the user provides expressions for the boundedness from below conditions (`CheckStability`) and the global minimum condition (`CheckGlobal`). The corresponding expressions are already entered for this 2HDM example.

Finally, there are three template functions to re-parametrise: i) the VEVs that are scanner over (`MyPhiParametrization`), ii) the mixing matrix (`MyInternalMixing`) and iii) to impose conditions among model parameters (`MyCoupMassRelations`). In this example only the first is activated in `ScannerSInput.nb` (see also Sect. 2.1) such that the usual 2HDM parametrisation is used, i.e.

$$\begin{aligned} \texttt{Phi[2]} \equiv v_1 &= v\cos\beta \;\;,\;\; \beta \equiv \texttt{PhiPar[0]} \\ \texttt{Phi[6]} \equiv v_2 &= v\sin\beta \end{aligned} \tag{1.2}$$

with $v = 246$ GeV, as seen in the following stretch of code

```
void 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]);
}
```

These template functions will be discussed in detail in Sect. 2.2.3.

5. **Compile the code and generate 10 points**:

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

```
$ make
$ ./ScannerS --nscan 10
```

or

```
$ ./ScannerS -i model.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`[1]. If you wish to specify a different filename run with the option `-o`, i.e.

```
$ ./ScannerS -i model.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

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

that the format of the output file is completely user defined, so you can change it in `ScannerSUser.cpp` according to your needs.

6. **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)

```
$ make clean
$ make MODE=-DVERBOSE
```

Then run

```
$ ./ScannerS
```

Since the program attempts various points before accepting a valid one, the output contain several summaries for each attempted point. 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[2]

```
------------------
--- 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]= ...
```

[2]We shown here the full numerical decomposition only for the first vector for brevity.

```
--- 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 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 are always scanner over, so they are independent parameters are well.

For this point we have also

```
----------------------
------ 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 Step by step to model/analysis implementation

In this section we describe some further details of how the various parts of the user-defined code works.

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

As explained in the previous section, the potential for the model is introduced in the *Mathematica* input file, `ScannerSInput.nb`. Following the notation of the code, an arbitrary potential `Vscalar` is a real function which can always be written in terms of a number `Nscalar` of canonically normalised fields ϕ`[i]`, as a linear form over a set of `Ncoup` real couplings `L[a]`, i.e.

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

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 renormalisable 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. The dimensions `Nscalar` and `Ncoup` must be set in the beginning of the notebook.

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].	
ϕMin[i], ϕMax[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.
ϕParMin[j], ϕParMax[j] j $= 0, ...,$ NparamsVEVs-1	Ranges for the scan over the parameters ϕPar[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], ϕMixParMax[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], ϕMLParMax[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.

Furthermore, there are several switches that must be set so that the code knows the ranges of the scan and the options available. A full summary of all options available (to be discussed in the next paragraphs) is in Table 1.

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

For field vacuum expectation values (VEVs):

1. 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;
]
```

2. 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.

For the parameters `L[a]` of the potential, and the physical state masses `mass[i]` a similar strategy applies to the corresponding ranges `LMin[a]`, `massMin[i]`, etc... (see Table. 1).

Scan boxes for other parameters: 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 on a hypercubic boxes, but on more generic slices/volumes in the parameter space. These are discussed in detail in Sec. 2.2.3 (see also Table 1 for a short description).

2.2 Defining the analysis –`ScannerSUser.cpp`

After the *Mathematica* notebook `ScannerSInput.nb`, has been executed to generate the input file, the use must define the analysis he/she wishes 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[3].

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:

1. `UserInitCalcs`: The function looks like

   ```
   void UserInitCalcs(void){

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

   }
   ```

   so it is empty by default. The user can enter here initial calculations/operations that are done (only once) before the scan starts. For example, the user may 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<<...`.

2. `UserAnalysis`: This is where the user writes the analysis done on each point that is generated in the scan. The general structure looks like

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

     if(_condition_)
       return false;

     ...

     ////////////////////////////////////////
     ////////// Final print out/////////////
     ////////////////////////////////////////

     std::cout<<...
     ...

     return true;
   }
   ```

[3]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).

The main points to note are the following. First there is a first part where the user is supposed to 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. Then there is a print out part, where the user decides what gets printed in the output file `model.out` if the point was accepted. The last line must always be `return true;` so that the program accepts the point and moves on to generate the next one, if all tests passed.

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

```
void 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 minimum that was generated locally is actually the absolute minimum) 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.

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

```
bool 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.

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

```
bool 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),(1.2), 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. (1.2)).

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

$$\texttt{Phi[0]} = \texttt{f}_0\texttt{(PhiPar[0],...,PhiPar[NparamsVEVs-1])} \tag{2.2}$$

$$\ldots = \ldots \tag{2.3}$$

$$\texttt{Phi[Nreal-1]} = \texttt{f}_{\texttt{Nreal-1}}\texttt{(PhiPar[0],...,PhiPar[NparamsVEVs-1])} \tag{2.4}$$

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. (1.2))

```
void 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

$$\texttt{MixInternal[i][j]} = F_{ij}(\texttt{MixPar[k],Phi[k],PhiPar[k]}) \tag{2.5}$$

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 for example

```
void 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 parameters in the last stage of the generation of a point. This function is `MyCoupMassRelations`. This capability is not so essential, so further examples will be presented in future versions of this manual.

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[4] to de-activate it.

3.1 SuperIso (tested with v3.3)

The SuperIso library [2] 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 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:

1. 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>
```

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

   (a) **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.

   (b) **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 [2]). 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

(UNDER TESTING!)

Both HiggsBounds and HiggsSignals can be linked by indicating the correct path to the

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

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). Note however that the output hasn't yet been tested extesnsively, so use it at your own risk.

3.3 SusHi (tested with v1.1.0)

Currently there is an interface to the `SusHi` cross section calculator [3], 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:

1. 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>
   ```
2. Edit your `ScannerSUser.cpp` analysis file at the line where you want to call `SusHi` and write the following lines:

   (a) **Create the `tempsushi.in` 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.

   (b) **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.

3.4 Hdecay (tested with v6.0.0)

Currently there is a direct interface to the `Hdecay` calculator [4], 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:

1. 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>
   ```

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

   (a) **Create the `hdecay.in` 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.

   (b) **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 A). This function must be called **exactly** as in the following line:

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

   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).

   (c) **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 A.

3.5 Micromegas (tested with v2.4.5.)

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 [5]) 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:

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

2. 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.

3. 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
   ```

4. Edit your `ScannerSUser.cpp` analysis file at the line where you want to call MicrOmegas. A typical call involves (check examples):

   (a) **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 pointm, 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 you would pass using the MicrOmegas `assignVal` function

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

   (b) **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 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 [5]:

```
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"]
```

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

```
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"]
```

```
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"]
```

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] 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/.

[3] 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/.

[4] 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/.

[5] 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/.