DarkPACK: A new package to compute BSM squared amplitudes for dark matter observables

M. Palmiotto\textsuperscript{a}\textsuperscript{*}, A. Arbey\textsuperscript{a,b}, F. Mahmoudi\textsuperscript{a,b}

\textsuperscript{a}Université de Lyon, Université Claude Bernard Lyon 1, CNRS/IN2P3, Institut de Physique des 2 Infinis de Lyon, UMR 5822, F-69622, Villeurbanne, France

\textsuperscript{b}Theoretical Physics Department, CERN, CH-1211 Geneva 23, Switzerland

Abstract

We present here a new package to automatically generate a complete library of 2 to 2 squared amplitudes at leading order in any New Physics models. The package is written in C++ and based on the MARTY software. The numerical library generated using this package is interfaced with SuperIso Relic, and examples of New Physics scenarios are provided.

1 Introduction

The question of the nature of dark matter is an important topic in both particle physics and cosmology (see [1] for a recent review). During decades, the most studied New Physics (NP) scenario providing a dark matter candidate has been the minimal supersymmetric extension of the Standard Model (MSSM), for which several softwares have been developed to compute dark matter observables. However, so far no supersymmetric particle has been discovered in particle physics experiments, and other NP scenarios are worth being investigated. In this context, the software SuperIso Relic [2–4], which aims at studying dark matter observables in the standard cosmological model and in alternative cosmological scenarios, has been until now focused on the MSSM and NMSSM, and is being developed in order to study other NP scenarios.

An effort in the direction of generalising and automating the computations has already been performed through the development of the code MARTY [5], which allows for the automatic calculation of amplitudes, cross-sections and Wilson coefficients in a large variety of NP models. However, a step further is needed to compute dark matter observables, such as dark matter relic density, which can involve thousands of scattering amplitudes.

Other codes publicly available to calculate dark matter observables are DarkSUSY [6,7], MadDM [8,9] and MicrOMEgas [10]. Despite its name, DarkSUSY is model-independent. In fact, provided some inputs such as the
DM mass and its self-annihilation cross section, it is possible to compute the relic density, alongside with other observables related to direct or indirect detection. Moreover, it has a modular structure that makes it easy to link it with other tools.

MadDM and MicrOMEGAs are also model-independent: the former relies on Madgraph [11], and the latter on CalcHEP [12]. It is therefore possible to provide model inputs compatible with those softwares and then all the relevant quantities for relic densities, as for instance the annihilation cross sections, are computed from their respective backend, in order to allow the computation of DM relic density and other DM related observables.

In this paper, we present the package DarkPACK, which aims at automatically generating a numerical library of scattering amplitudes to compute dark matter observables such as relic density, and which is interfaced with SuperIso Relic. In the current version of DarkPACK the generated library contains all the squared amplitudes of 2 NP particles into 2 Standard Model (SM) particles at leading order.

In order to validate the results, we focus on the MSSM which is an adequate benchmark in terms of complexity and compare in particular the results of DarkPACK with those of SuperIso Relic v4, which relies on a self-generated FormCalc [13] FORTRAN code.

This article is structured as follows. In section 2 the compilation instructions of the package are provided. In section 3 we present the philosophy behind the code, i.e. the main goals and the methods used. Furthermore we describe the role of MARTY and of the numerical library, as well as general information such as how to handle numerical inputs and where to find the example programs we provide. The main functions of MARTY that are used to generate the numerical library are given in section 4. In section 5 we describe how to use the numerical library. Functions are listed accordingly to the user’s needs. Further information can be found in the appendices and in the source code, where the functions are described in the comments.

## 2 Setup of the package

DarkPACK can be downloaded at the following address: [https://gitlab.in2p3.fr/darkpack/darkpack-public](https://gitlab.in2p3.fr/darkpack/darkpack-public). In order to use the code, the user needs a working setup of MARTY installed.

In order to compile the library, the user has to run the script `lib_setup.sh` providing three arguments:

- The name of the numerical library (by default `mssm2to2` for the MSSM, and `scalar` for the other example model).

---

1We provide a script which automatically installs MARTY. More information about the installation and usage of MARTY can be found at [https://marty.in2p3.fr](https://marty.in2p3.fr) and in Refs. [5][14][16].
• The number of threads used for the compilation with dynamic linking (use 1 in case of doubt), or the string -nomake for not compiling automatically. Instructions are provided if the user wants to perform a static compilation instead.

• Optionally, the name of another numerical library. This has to be used when compiling a new library for the first time, while the auxiliary files have been used to compile another library previously. By specifying this argument, a search and replace is performed on the files to update them with the new library name.

The content of the package is the following. A numerical library can be found in the default folder (mssm2to2 for the MSSM and scalar for the other example model), and external files for the library in the auxiliary_library folder.

In order to generate the library from scratch\footnote{The MSSM library is provided within the package and is not required to be generated.}, the source file (e.g. MSSM.cpp for the MSSM, and scalar.cpp for the other example model we provide) has to be compiled and the corresponding program launched.

For this purpose the user can run the script ./lib_generate.sh, providing as auxments the name of the target library and the name of the source code:

./lib_generate.sh <library name> <source code name>

and then the generated library has to be compiled with the instructions of the previous paragraph.

Further instructions can be found in the README.md file.

3 Outlook of the code

This code has been developed with the purpose of having a new tool to compute 2 to 2 scattering amplitudes, firstly in the MSSM at the leading order (LO). In this release, we provide the setup for the MSSM and for a model with a scalar DM candidate with a scalar messenger (see e.g. \cite{17,18}).

The idea behind this package is to use MARTY in order to define the desired model and quantities to compute and to export a numerical C++ library. In particular, in this release there are the example files MSSM.cpp and scalar.cpp files that contain the code that computes symbolically all the 2 to 2 sum of the squared amplitudes relevant for the computation of relic density, respectively in the MSSM and in the other BSM model. In the former case, the process list we consider is the same as in SuperIso Relic and can be found by opening the file data/processes_all.psiso. The list can be easily extended in many ways. For instance, it is possible to add
manually some processes in MSSM.cpp or to use what is already there to read processes from text files. More detail about this can be found in the documentation of the function computeAndAddToList, described afterwards. It should be noted that it is not possible, at least in the current version of MARTY, to add a subset of processes in a previously generated library. So, if the user wants to add new quantities to an existing library, a new generation of the library is required.

For what concerns the usage of the numerical library, the raw output of MARTY is not meant to be particularly flexible. Normally it consists in a list of functions that the user can call in a C++ source file. The DarkPACK package provides a super-set of MARTY’s numerical library, which allows for a more intuitive usage and additional features, such as the running of SM parameters. The main features are described below.

In particular, we have been able to enhance the flexibility, automatically generating a hash table (i.e. a std::unordered_map) whose keys are strings unique for each 2 to 2 process and whose values are a std::tuple with all the relevant quantities for computing the sum of the squared amplitudes for it. Therefore, only for the processes present in the hash table it is possible to compute quantities: we describe how to do it in the example files. We would like to point out that this procedure can be used to put in the libraries also other kind of observables, even unrelated to the 2 to 2 processes, such as specific decay widths or Wilson coefficients. Within this framework, more potentialities of MARTY will be exploited in the future.

As far as input values are concerned, in this release we provide algorithms to read them from .lha files [19,20]. Decay widths at the tree-level and mass spectra can be computed using MARTY if needed, otherwise the necessary quantities have to be provided among the input parameters.

In the auxiliary_library/script_mssm2to2 directory some example files are provided, to show how to use the main features of the code in practice. The users are invited to contact the authors if they need some more examples or features. We provide:

- **example_1_single_process.cpp**
  An example file that shows how to initialise a single process and do some calculations.

- **example_2_running.cpp**
  An example file that shows how to easily run the Standard Model parameters.

- **example_3_process_vector.cpp**
  An example file that shows how to efficiently deal with vectors of processes, by computing an inclusive cross section.
• example_4_dWeff.cpp
  An example file that shows how to compute $\frac{dW}{d\cos(\theta)}$.

Similar example files are provided for the scalar model, in the directory auxiliary_library/script_scalar.

4 Main features of the MARTY code

For the sake of simplicity and clarity, we defined the following aliases:

```cpp
using Process = std::vector<mty::Insertion>;
using Processes = std::vector<Process>;
```

In fact, a vector of Insertion contains the information for a specific process. We furthermore define the extension .psiso, for a text file containing the names of 2 to 2 processes in the SuperIso convention, as well as the new type

```cpp
typedef struct
{
  Process process;
  mty::Order order; // Options mty::Order::TreeLevel,
                   // mty::Order::OneLoop
  bool leading_order;
  mty::gauge::Type Wgauge;
} Process2to2ToCompute;
```

that contains a process, information about order to which its amplitude will be computed, and the choice of the gauge for the W boson.

A list of the main functions that we implemented in MARTY is given below:

1. `std::string processName(Process const &proc)`
   This function takes as input a specific process, and returns a string that corresponds to a name for this process. This function can serve many purposes. In DarkPACK we use it to assign a name to the functions in the numerical library.

2. `std::string generateCorrespondance(
   std::vector<Particle> psm,
   std::vector<Particle> pbsm,
   const std::string filename="smBsm.h")`
   This function takes as first argument a vector of Particle that corresponds to the list of the particles to be considered as SM, as second

---

3This needs to be done because a very small number of squared amplitudes can be numerically computed to be negative in the Feynman gauge. Computing them in the unitary gauge is a way to fix this inconsistency.
argument a vector of Particle that corresponds to the list of the particles to be considered as BSM, and as third optional argument a string filename. This function generates the file auxiliary_library/filename with the structure of a C++ header. It will contain the list of the SM and the BSM particles that the numerical library needs to know.

3. int computeAndAddToList( mty::Model &model,
    mty::Library &lib,
    std::vector<Process2to2ToCompute> listofprocs,
    std::string nameSmBsmFile="smBsm.h"
)

This function takes as input:

- a model,
- a library for the output,
- the list of processes to be computed and added in the library,
- the name of the file created with generateCorrespondance.

This function computes the amplitude of each Process2to2ToCompute in the input vector within the specified gauge and to the specified order. If the boolean leading_order is set to true, the information in order is neglected, and the calculation stops at the tree-level if the amplitude is non-zero, otherwise it is performed at one-loop\(^4\) Then such an amplitude is squared and saved in the library lib. This function also provides the creation of some auxiliary files in the numerical library, such as the content of correspondance.h, described in what follows.

4. int addFromFile( mty::Model &model,
    Processes &processes,
    std::vector<std::string> &names,
    std::vector<std::string> &namesSuperiso,
    std::string filename)

This function has been written specifically for the MSSM. The inputs are:

- the model (MSSM),
- a vector of processes,
- two vectors of strings, that will contain the names of the processes in two conventions respectively: the one defined in processName, and the one used in SuperIso,

\(^4\)We do not go to higher loop levels because MARTY does not provide such a feature at the moment.
• a string, which is the input file name.

This function reads the file filename, which is a .psi.iso file. For each process name, this function reconstructs the particle content and puts in processes the process, in names their names in the processName convention, and in namesSuperiso their names in the SuperIso convention. The function returns the number of read processes in case of success.

The other functions are documented in the comments of the code.

5 Main features for the numerical library

We list below the auxiliary functions in the numerical library. In what follows, we call this library bsm2to2.

5.1 Global variables and generic functions

Let us start by describing the main content of the header correspondance.h, included in the other headers listed in the following. Here are defined an enumeration, global variables and functions, in the namespace corr. The most relevant contents of this header are:

1. enum Part_t
   Such an enumeration defines particles.

2. std::vector<std::string> part_names
   This vector is defined to have the name of each particle, corresponding to the enumeration.

In correspondance.h is included also the header params_new.h, where the type struct bsm2to2::Param_t is defined. It inherits the members of the type struct bsm2to2::param_t, different for each library because automatically generated by MARTY, and it adds some quantities useful for the running. This is the type of variables that we use to store data.

5.2 Input reading and manipulation

Since each model has its own parameters, the input management has to be handled by the user accordingly to its needs. In this paragraph, we describe the functions which we defined in order to work within the provided models, i.e. the MSSM and the model with a scalar mediator.

For testing purposes, we added to this function the generation of a file processes_chep.txt. This file has two columns and a row for each process. The first element of the row is the process name in the CalcHEP convention and the second row is the corresponding name in the SuperIso convention.
The headers we use are:

- `leshouchesfromsuperiso.h`: its functions are defined under the namespace `superisosupport`, and they allow us to read from a `.lha` file calling SuperIso’s routines;

- `leshouchesfrommarty.h`: its functions are defined under the namespace `readmodule`, and they allow us to read from a `.lha` file calling the lha extension native in the MARTY installation.

For a general use, the library to refer to in order to modify the input management is `leshouchesfrommarty.h`. The library `leshouchesfromsuperiso.h` is useful while dealing with the MSSM only.

We list here the main functions:

1. `int superisosupport::InitInterfaceStruct(const parameters *const param, mssm2to2::param_t &input)`

   This function takes as inputs a pointer to a variable of type `parameters`, that contains the SuperIso inputs, and an address to a `param_t`. It copies the values contained in the SuperIso inputs to the structure `param_t`. In practice, the SuperIso structure is read using the SuperIso routines[^1] and then this function is called to handle the inputs in the numerical library.

2. `int superisosupport::InitInterfaceStruct_Full(const parameters *const param, mssm2to2::Param_t &input)`

   This function calls `InitInterfaceStruct` and initialises the other members of the `Param_t` variable given as input.

3. `struct parameters superisosupport::ReadLHA(mssm2to2::Param_t &input, const char * name, int *err)`

   This function uses the previous one to read the `.lha` file with the path `filename`, and saves the data in `input`. It returns a `parameters` which is the corresponding structure of the data used in SuperIso Relic. If there are no errors in the process, `err` is set to zero.

4. `bsm2to2::Param_t readmodule::ReadLHA(const std::string filename);`

[^1]: They can be found in `leshouches.c` and `leshouches.h`. 
This function reads the .lha file with the path filename and returns a bsm2to2::Param_t whose members are filled with inputs provided in the input file. If some essential SM input is not provided, it is filled with the default Particle Data Group [21] value.

The other functions are documented in the comments of the headers. Note that no further manipulation is done on the values, except for the running of the Standard Model parameters \(m_t, m_b, \alpha_s\). This means that, if no specific code is written, the input file has to provide quantities as the mass spectrum computed externally.

It is in fact possible to compute the mass spectrum inside the program. In this regard, we invite the interested users to read the MARTY’s documentation to have the full details.\(^7\) In Appendix B, we give the most relevant information on the spectra.

5.3 Handling the running of SM parameters

Below we list the functions for the running for SM quantities, i.e. the strong coupling constant \(g_s\), and the top and the beauty quark masses. This is handled via a class called RunningSM, accessible after including RunningSM.h. Here we list its main public methods and elements:

1. RunningSM(const bsm2to2::Param_t &input)
   This constructor builds the class variable starting from the values stored in a Param_t variable.

2. RunningSM(void)
   This constructor builds the class variable from the default PDG values.

3. enum ParticlesList : short int {UP, DOWN, STRANGE, CHARM, BEAUTY, TOP, EL, MU, TAU, NUE, NUMU, NUTAU, GLUON, W, Z, PHOTON, HIGGS}
   It is an enumeration that contains all the physical particles in the Standard Model.

4. double GetMbPole()
   Returns the b-quark pole mass.

5. double GetTopPoleMass()
   Returns the t-quark pole mass.

6. double AlphaStrong(double Q, double mtop)
   Returns the value of \(\alpha_s\) at the energy Q, taking mtop as the value for the top pole mass.

\(^{7}\)See https://marty.in2p3.fr/doc/marty-manual.pdf for such a documentation.
7. `double GetMcPole()`
   Returns the charm pole mass computed from $m_c(m_c)$.

8. `double GetMbMb()`
   Returns $m_b(m_b)$.

9. `double GetMtopMtop()`
   Returns $m_t(m_t)$.

10. `double GetQuarkMass(enum ParticlesList part, double Qf)`
    Takes a quark as enumeration and an energy scale $Qf$. The return value is the mass of the particle at the scale $Qf$.

11. `void HandleParamRunning(
        bsm2to2::Param_t &input,
        const double Q)
    )`
    This method performs the running of the parameters at the scale $Q$ and saves the results in $input$.

12. `void RunLightQuarks(bool x=true)`
    This method changes the default behaviour of `HandleParamRunning`. After its calling, if $x$ is `true`, the running of the down, up and strange masses is enabled. Otherwise, the running of the down, up and strange masses is disabled.

13. `void RunCharmMass(bool x=true)`
    This method changes the default behaviour of `HandleParamRunning`. After its calling, if $x$ is `true`, the running of the charm mass is enabled. Otherwise, the running of the charm mass is disabled.

### 5.4 Functions related to 2 to 2 processes

We describe here the functions and the methods dedicated to the computation of the sum of 2 to 2 squared amplitudes, cross sections, and contributions to $W_{\text{eff}}$. These functions and methods are accessible by including `process.h`. In this header, the class `Process2to2` is defined. In order to understand how the public methods work, we remark that the main members of this class are the private ones

```cpp
csl::InitSanitizer<int> p[4];
csl::InitSanitizer<bool> ap[4];
```

They are two arrays of size 4, because the class is intended for a process of the kind $1,2 \to 3,4$. $p[i]$ contains the $i$-th field as enumeration, while the $ap[i]$ is `true` or `false`, depending on whether the $i$-th entry is a particle or its antiparticle. This is always specified, even for a particle which is its own
antiparticle. It can be important to know that, when a process is filled, the order of the particles may change. In fact, after the finalisation of a process, an algorithm determines whether the sum of the squared amplitudes of the process is in the library. If it is in, the particles are re-ordered accordingly to the order they appear in the function that computes the sum of the squared amplitudes. This is relevant to verify if some quantities related to kinematic parameters have to be computed: in such a case, it is necessary to calculate them accordingly to the order in which they appear in the Process2to2 variable. Furthermore, the following type is defined:

```cpp
struct Insertion
{
    int field;
    bool part;

    Insertion(int i, bool b=true) {
        field = i;
        part = b;
    }
};
```

in order to specify the particles that enter a process. In practice, it is possible to construct a variable of this type by assigning an integer, that corresponds to the field, or by assigning a two elements list, with an integer and a boolean, where the boolean determines if it is a particle or an antiparticle. The main public methods are the following ones:

1. **Process2to2()**
   - This is a constructor: it creates an empty process.

2. **Process2to2(const std::vector<Insertion> &v);**
   - This constructor constructs the class corresponding to the process with incoming (outgoing) particles whose field is in the first (last) two elements of v.

3. **Process2to2(const std::vector<int> &p, const std::vector<bool> &ap)**
   - This constructor takes as input two vectors of size 4. It constructs the class corresponding to the process with incoming (outgoing) particles whose field is in the first (last) two elements of p. The second argument corresponds to true (false) if the element is a particle (antiparticle).

4. **short int set( short int n,**

In order to specify the particles that enter a process. In practice, it is possible to construct a variable of this type by assigning an integer, that corresponds to the field, or by assigning a two elements list, with an integer and a boolean, where the boolean determines if it is a particle or an antiparticle. The main public methods are the following ones:

1. **Process2to2()**
   - This is a constructor: it creates an empty process.

2. **Process2to2(const std::vector<Insertion> &v);**
   - This constructor constructs the class corresponding to the process with incoming (outgoing) particles whose field is in the first (last) two elements of v.

3. **Process2to2(const std::vector<int> &p, const std::vector<bool> &ap)**
   - This constructor takes as input two vectors of size 4. It constructs the class corresponding to the process with incoming (outgoing) particles whose field is in the first (last) two elements of p. The second argument corresponds to true (false) if the element is a particle (antiparticle).

4. **short int set( short int n,**
const int &ip,
const bool iap)

This function sets the n-th particle with field ip and (anti)particle iap.

5. std::string getName()
   Returns the name of the process.

6. std::string getMname()
   Returns the name of the process according to the convention in MARTY.

7. std::string getSname()
   Returns the name of the process according to the convention in SuperIso Relic.

8. double getMass(short int i)
   Returns the mass of the i-th particle in the process.

9. double getSumSquaredAmpl(
    bsm2to2::Param_t &input,
    const double &sqrts,
    const double &ctheta)

   Returns the sum of the squared amplitudes for the process, with the
   numerical inputs contained in input, the center of mass energy sqrt{s},
   and the cosine of the angle between particle 1 and particle 3 ctheta.

10. double getAvgSquaredAmpl(
    bsm2to2::Param_t &input,
    const double &sqrts,
    const double &ctheta)

    Returns the average of the squared amplitudes for the process, with
    the numerical inputs contained in input, the center of mass energy
    sqrt{s}, and the cosine of the angle between particle 1 and particle 3
    ctheta.

11. double getDiffWeffContrib(
    bsm2to2::Param_t &input,
    const double &sqrts,
    const double &ctheta)

    Returns the contribution to \( \frac{dW_{eff}}{d \cos(\theta)} \) for the process, defined as (see
    e.g. [2]):

    \[
    \left( \frac{dW_{eff}}{d \cos(\theta)} \right)_{1,2 \rightarrow 3,4} = \frac{f_{CP1234}}{S_{f34} \sqrt{sp_{eff}}} \sum_{\text{all d.o.f.}} |M|^2
    \]

    (1)

    where
(a) $f_{CP} = 2$ if $\bar{12} \rightarrow \bar{34}$ is allowed, otherwise it is 1

(b) $S_{F34} = 2$ if 3 and 4 are the same field and the same particle or anti-particle, otherwise it is 1

(c) $p_{ij} = \sqrt{s - (m_i + m_j)^2(s - (m_i - m_j)^2)} \over 2 \sqrt{s}$

(d) $p_{eff} = 1 \over 2 \sqrt{s - 4m_{N1}^2}$

with $m_{N1}$ the mass of the lightest stable new particle.

For the computation the numerical inputs contained in input are used, together with the center of mass energy $\sqrt{s}$, and the cosine of the angle between particle 1 and particle 3 $c\theta$.

12. double getDiffCrossSection(
    bsm2to2::Param_t &input,
    const double &sqrts,
    const double &ctheta)

Returns the differential cross section for the process, with the numerical inputs contained in input, the center of mass energy $\sqrt{s}$, and the cosine of the angle between particle 1 and particle 3 $c\theta$.

13. double getTotalCrossSection(
    bsm2to2::Param_t &input,
    const double &sqrts,
    double *discr=nullptr);

Returns the total cross section for the process, with the numerical inputs contained in input, and the center of mass energy $\sqrt{s}$s. The optional argument, if specified, allocates in discr the relative error of the integral. For details, see section [6].

Other methods of this class are defined in Appendix [C].

For the MSSM we provide the header setofprocs.h, where a class named SetOfProc is defined. Its purpose is the computation of global quantities in a more efficient way in terms of performance. A similar class could be defined in other models: if several amplitudes have to be computed at the same energy while varying the energy over an interval, a similar solution would be really beneficial for the performances. In order to understand the way it works, it would be useful to know that it has the following private members:
RunningSM run;
mssm2to2::Param_t input;
std::vector<Process2to2> p;

Its initialisation takes time, so we recommend to define one of those as a
global variable, initialise it once and create copies locally if needed. Its
public methods are the following:

1. SetOfProc(const mssm2to2::Param_t &param)
   It is the constructor. It copies the content of param into input, then
   it constructs run, and it allocates the vector p with all the possible 2
to 2 processes of the kind SUSY + SUSY to SM + SM whose sum of
the squared amplitudes is available in the library, avoiding duplicates.
Since initialising a single process needs more passages, the creation of
a variable with this constructor takes time. For this we recommend
to create a variable as global and then copy its content in functions if
necessary.

2. void runAtScale(const double Q)
   This method does all it takes to have all the quantities run at the energy
scale Q.

3. double getWeff(const double sqrtS)
   This method returns $g_L^{2W_{eff}}$, with $g_L$ the number of degrees of
freedom of the lightest stable NP particle, at the center of mass energy
sqrtS. In this method, the lightest stable NP particle is determined
from the content of the given input.

4. size_t getN()
   Returns the number of elements allocated in p.

5. Process2to2 getProcess(const size_t i)
   Returns the p[i].

6. mssm2to2::Param_t getInputs()
   Returns input.

6 Comparison with other softwares

We validated our code on 3430 processes of the kind SUSY + SUSY →
SM + SM. These processes are the ones that can be directly found in the
mssm2to2 library in the repository. We compared our results for the sums
of the squared amplitudes and the contribution to $dW_{eff}/d\cos(\theta)$ with the ones from
SuperIso Relic [2–4], and our results for total and differential cross sections
with the ones from micrOMEGAS/CalcHEP [10,12]. In this way, we validated
the behaviour of MARTY’s MSSM library [5] and also we have performed numerical tests.

In terms of time performance, on a 8 core machine, our sample test computes total cross sections with an average time of about $1.2 \times 10^{-4}$s for each one. The comparison of the results of the total cross sections with micrOMEGAs/CalcHEP has shown that there are some cross sections (around 9\%) for which the integration algorithm produces numerically unstable results with any software. This could be due to the presence of resonances, however the full understanding of the underlying reason is not achieved yet. In order to be able to control such a behaviour without affecting too much performance, we integrate the differential cross sections with the Gauss-Legendre method at two different orders and we compute the discrepancy. If the value is not accurate enough we pass to the next polynomial order and we compare it with the previous one. When the 9th order is reached and convergence is not reached yet, the trapezoidal rule is used, since it is a method whose error can be controlled and whose increment in order is done without losing the previous evaluations. Therefore, subsequent evaluations with the trapezoidal rule are made, and if convergence is reached the algorithm stops. If the number of intervals exceeds 256, the last value with the last error is taken as the result. We analysed the behaviour of this method, concluding that the trapezoidal rule made converge 16\% of the cross sections that do not converge with the Gauss-Legendre method, achieving the 9\% of non-convergence mentioned earlier.

The comparison with other softwares has been made by taking care of using the inputs in the same way. In particular, we have set the CKM matrix to the identity matrix, in order to compare the results with the SuperIso Relic/FormCalc code, and the mass of the muon and the electron to zero for the comparison with micrOMEGAs/CalcHEP. The outcome from the comparison with SuperIso Relic is that all the sums of the squared amplitudes are in agreement, except for the heights of some peaks. The comparison with micrOMEGAs/CalcHEP shows that most of the cross sections are in agreement. An example is provided in figure 1. The differences between the results with micrOMEGAs/CalcHEP can be classified in several categories, for which we provide some sample plots:

- By default, CalcHEP uses Simpson rule to compute total cross sections from the differential ones. For some processes such a rule fails to achieve a correct result. To be able to compare CalcHEP’s results with ours, we implemented the trapezoidal rule in CalcHEP to get the same quantities. An example of such a behaviour is shown in figure 2.

- In some other cases, no algorithm gives a coherent result, because

\[8\] The uncertainty on the numerical integration is provided by passing a double* to the integration function.
Figure 1: Total cross section for $C_1 \bar{C}_1 \rightarrow \bar{b} b$ obtained with our package labelled as MARTY and using CalcHEP. The two codes are in agreement.

- as mentioned earlier, the Simpson rule in CalcHEP fails to achieve a correct result;
- our algorithm, while giving a numerically reasonable result, is affected by large uncertainties;
- the trapezoidal rule in CalcHEP converges to a numerically reasonable result, different from ours and with a negligible uncertainty, but it presents some resonance-like peaks that are not supposed to be there.

In figure 3 an example of such a behaviour is shown, and the Feynman diagrams contributing to the corresponding process are shown in figure 4. Amongst all the particles in the internal legs, none of them has a mass that corresponds to the higher energetic peaks shown in the plot. An example of a differential cross section with several peaks is given in figure 5.

- For some processes, our results and the ones of CalcHEP are different at the threshold. However, we are in agreement with SuperIso Relic. An example of this is shown in figures 6 and 7.

Finally we present in figure 8 $W_{\text{eff}}$ obtained with DarkPACK and with SuperIso Relic. As can be seen in the figure, the two softwares are in excellent agreement in the energy range relevant for the calculation of the relic density. We do not show the data for other softwares, since SuperIso Relic is for example in excellent agreement with DarkSUSY.
Figure 2: Total cross section for the process $N_2 N_4 \to H^- W^+$ computed with our code using MARTY, with CalcHEP’s default integration and with a trapezoidal rule integration in CalcHEP. Note that the Simpson’s rule fails to several orders of magnitude.

Figure 3: Total cross section for the process $N_2 N_4 \to h_0 H_0$ computed with our code (labelled as MARTY), with CalcHEP’s default integration and finally with the trapezoidal rule integration in CalcHEP. Note that the Simpson’s rule fails to several orders of magnitude, and in our code the trapezoidal rule on 256 intervals does not converge.
Figure 4: Feynman diagrams contributing to $N_2 N_4 \rightarrow h_0 H_0$ at tree-level.

Figure 5: Differential cross section for the process $N_2 N_4 \rightarrow h_0 H_0$ at $\sqrt{s} = 1.497\text{TeV}$ computed with our code (labelled as MARTY), and with CalcHEP. Note that the two codes are in good agreement.
Figure 6: Sum of the squared amplitudes for the process $N_3 N_3 \rightarrow \bar{d} d$ at $\cos(\theta) = 0$. Our code and SuperIso are in agreement.

Figure 7: Total cross section for the process $N_3 N_3 \rightarrow \bar{d} d$. Our code and CalcHEP disagree in the threshold region.
7 Conclusions

In this paper, we presented a new package to deal with 2 to 2 sums of the squared amplitudes to the leading order in any New Physics scenario. The development of such a code addresses the needs for flexibility which can be more challenging to achieve in already existing tools, such as FormCalc [13] and CalcHEP [12], which are the backends for SuperIso Relic v4 [4] and micrOMEGAs5.0 [10], respectively.

Using DarkPACK, the user can more easily parallelise the calculations, since global variables have been avoided as much as possible, achieving a better time performance. Additional features may be added upon request of the users. We will firstly use this code to replace the backend in SuperIso Relic and add more features and functionalities. In this way, the next version of SuperIso Relic will not be exclusive to supersymmetric models, but the user will have the possibility of defining custom models, within the capabilities of MARTY, and calculations can be performed up to the one-loop level. Eliminating the hard-coded FORTRAN code for the squared amplitudes, SuperIso Relic will also benefit in modularity, so that it will be easier to link it with other software, as well as to increase the number of predictable observables. Furthermore, flavour physics observables can be computed and precision tests can be performed thanks to the generic version of SuperIso [22,25], providing a multi-sectorial setup to study BSM scenarios.
Acknowledgments

We would like to thank G. Uhlrich for his support with MARTY, and G. Bélanger and A. Pukhov for discussions about calculation within MicrOMEGAs and CalcHEP.
A  The SuperIso convention for names

We describe here the SuperIso convention for the names of 2 to 2 reactions. This is a convention that works only for SUSY + SUSY → SM + SM type of reactions. However, we used it in DarkPACK since it is practical and since these reactions are the ones we are mainly interested in. As described in section[4] as well as in the documentation of MARTY, there are also other ways to define processes.

The structure of the name is of the kind p1(bar)p2(bar)p3(bar)p4(bar), where the “bar” is present if the preceding field has to be taken as antiparticle. The list of the particles and their names can be found in Table[1]. The user can check the file processes.psiso for the full list of the processes we tested, named in this convention.

| SUSY name | SUSY particle | SM name | SM particle |
|-----------|---------------|---------|-------------|
| o1, o2, o3, o4 | the four neutralinos | h, hh | light, heavy Higgs |
| c1, c2 | the two charginos | hc | charged Higgs |
| t1, t2 | stop 1 and 2 | h3 | pseudoscalar Higgs |
| b1,b2 | sbottom 1,2 | w, z | W, Z |
| dr,dl | sdown right, left | g | gluon |
| ur,ul | sup right, left | a | photon |
| cr,cl | scharm right, left | d, u, s, c, b, t | quarks |
| sr,sl | sstrange right, left | e, m, l | charged leptons |
| er,el | selectron right, left | n, m, n | neutrinos |
| mr,ml | smuon right, left | | |
| l1,l2 | stau 1,2 | | |
| ne | electron sneutrino | | |
| nm | muon sneutrino | | |
| nl | tau sneutrino | | |
| go | gluino | | |

Table 1: List of the names of the particles in the SuperIso convention.

B  Mass spectrum calculation

Full information on mass spectra and general properties of the numerical libraries generated by MARTY can be found by looking at chapter 7 of its manual[9]. In this section we explain what it is done in the MSSM.cpp file. The relevant part of the code, where Lib is the class that handles the library that will eventually be exported is this one (Listing[2]). In this way, all the physical particles are listed, and their mass terms are all grouped. The parameter that

[https://marty.in2p3.fr/doc/marty-manual.pdf](https://marty.in2p3.fr/doc/marty-manual.pdf)
// Listing all the Physical particles
std::vector <Particle> part_0 =
    mssm.getPhysicalParticles([&](Particle p) {
        return p->isPhysical();
    });
std::vector <Particle> part;
for ( size_t i = 0 ; i != part_0.size() ; i++ )
{
    if (!(IsOfType<GhostBoson>(part_0[i]) ||
         IsOfType<GoldstoneBoson>(part_0[i])))
        part.push_back(part_0[i]);
}
// Re-fixing mass names
for ( size_t i = 0 ; i != part.size() ; i++ )
{
    if (!part[i]->getMass()->getName().empty())
    {
        part[i]->getMass()->setName("m_"+part[i]->getName());
    }
}
// In the following statement mssm is a mty::Model variable
// and lib is a mty::Library variable
lib.generateSpectrum(mssm);

Listing 1: Treatment of the mass spectrum in MARTY.

appear in the struct param_t as m_(name of the particle) is no longer
the bare mass in this way, but the mass that has been already computed by a
spectrum generator.
It is however possible to give the bare masses as starting input values in
a variable struct param_t params and then call
updateSpectrum(params);
if the users wants to create the whole spectrum, or just
updateMassExpressions(params);
if the user wants only to update the mass expressions without performing
the diagonalization.

C Running for multiple calculations at the same energy

In this appendix we want to explain how to efficiently perform the running
when more quantities are computed at the same energy. Our choice of the
whole setup for calculation has been guided by the idea of limiting as much as possible global variables, in order to give the user the possibility to easily parallelise calculation, for instance simply using the C++ standard libraries.

In order to fully understand what comes next, we will refer to what we explained in section 5. A practical example to describe how to use what comes next can be found in the file

auxiliary_library/script_mssm2to2/example_3_process_vector.cpp

A variable of the type Process2to2 has, amongst its private members, the following ones:

```cpp
RunningSM *runptr;
bool isRunDataExternal;
bool isRunningExternal;
```

The default values after construction of the those members are `nullptr` and `false`.

When a function to get the sum of the squared amplitudes or the cross section is called, the default behaviour is that if `runptr` is equal to `nullptr`, a new `RunningSM` class is allocated, the value of `isRunDataExternal` is set to `true`. This allows the destructor of the class to know it has to free the memory pointed by the `RunningSM` member. Furthermore, the calculation of every quantity at a given energy can be performed handling the running via `runptr`.

However, if the user has to compute many quantities of different processes at the same energy, performing the running for each process is a waste of resources. So, a class `RunningSM` can be constructed before creating the processes, and after that one can use the methods `setRunningExternal` and `setRunningData` to perform the running once and on the defined variable. For instance, for the process \( N_1, N_1 \rightarrow Z, Z \):

```cpp
RunningSM run(input);
std::vector<Insertion> v = {corr::N_1, corr::N_1,
corr::Z, corr::Z};
Process2to2 proc(v);
proc.setRunningData(&run);
proc.setRunningExternal();
```

double Ecm=3.0e+3; // Chosing 3 TeV as center of mass energy
run.HandleParamRunning(input, Ecm);
double xsection = proc.getTotalCrossSection(input, Ecm);
```
where `input` is a `Param_t` type variable properly initialised. The same result will be produced by simply using

```cpp
std::vector<Insertion> v = {corr::N_1, corr::N_1,
corr::Z, corr::Z};
```
Process2to2 proc(v);
double Ecm=3.0e+3; // Choosing 3 TeV as center of mass energy
double xsection = proc.getTotalCrossSection(input, Ecm);

as shown in
auxiliary_library/script_mssm2to2/example_1_single_process.cpp

The example file we mentioned at the beginning of this appendix shows how to efficiently compute an inclusive cross section in this way.

Note that with this method the user can choose to use qualifiers in the declaration of the RunningSM variable, to make it for instance local to the thread and be able to coherently parallelise calculations with the C++ standard libraries.
References

[1] A. Arbey and F. Mahmoudi. Dark matter and the early Universe: a review. *Prog. Part. Nucl. Phys.*, 119:103865, 2021.

[2] A. Arbey and F. Mahmoudi. SuperIso Relic: A Program for calculating relic density and flavor physics observables in Supersymmetry. *Comput. Phys. Commun.*, 181:1277–1292, 2010.

[3] A. Arbey and F. Mahmoudi. SuperIso Relic v3.0: A program for calculating relic density and flavour physics observables: Extension to NMSSM. *Comput. Phys. Commun.*, 182:1582–1583, 2011.

[4] A. Arbey, F. Mahmoudi, and G. Robbins. SuperIso Relic v4: A program for calculating dark matter and flavour physics observables in Supersymmetry. *Comput. Phys. Commun.*, 239:238–264, 2019.

[5] G. Uhlig, F. Mahmoudi, and A.e Arbey. MARTY - Modern ARtificial Theoretical phYsicist A C++ framework automating symbolic calculations Beyond the Standard Model. *Comput. Phys. Commun.*, 264:107928, 2021.

[6] P. Gondolo, J. Edsjo, P. Ullio, L. Bergstrom, Mia Schelke, and E. A. Baltz. DarkSUSY: Computing supersymmetric dark matter properties numerically. *JCAP*, 07:008, 2004.

[7] T. Bringmann, J. Edsjo, P. Gondolo, P. Ullio, and L. Bergstrom. DarkSUSY 6: An Advanced Tool to Compute Dark Matter Properties Numerically. *JCAP*, 07:033, 2018.

[8] F. Ambrogi, C. Arina, M. Backovic, Jan Heisig, Fabio Maltoni, Luca Mantani, O. Mattelaer, and G. Mohlabeng. MadDM v.3.0: a Comprehensive Tool for Dark Matter Studies. *Phys. Dark Univ.*, 24:100249, 2019.

[9] C. Arina, J. Heisig, F. Maltoni, L. Mantani, D. Massaro, O. Mattelaer, and G. Mohlabeng. Studying dark matter with MadDM 3.1: a short user guide. *PoS*, TOOLS2020:009, 2021.

[10] G. Bélanger, F. Boudjema, A. Goudelis, A. Pukhov, and B. Zaldivar. micrOMEGAs5.0: Freeze-in. *Comput. Phys. Commun.*, 231:173–186, 2018.

[11] J. Alwall, R. Frederix, S. Frixione, V. Hirschi, F. Maltoni, O. Mattelaer, H. S. Shao, T. Stelzer, P. Torrielli, and M. Zaro. The automated computation of tree-level and next-to-leading order differential cross sections, and their matching to parton shower simulations. *JHEP*, 07:079, 2014.
[12] A. Belyaev, N.D. Christensen, and A. Pukhov. Calchep 3.4 for collider physics within and beyond the standard model. *Computer Physics Communications*, 184(7):1729–1769, 2013.

[13] T. Hahn, S. Paßehr, and C. Schappacher. FormCalc 9 and Extensions. *PoS*, LL2016:068, 2016.

[14] G. Uhlrich, F. Mahmoudi, and A. Arbey. MARTY, a new C++ framework for automated symbolic calculations in Beyond the Standard Model physics. *PoS*, ICHEP2020:928, 2021.

[15] G. Uhlrich, F. Mahmoudi, and A. Arbey. Semi-automated BSM model building procedures in MARTY-1.1 through a 2HDM example. *PoS*, TOOLS2020:042, 2021.

[16] G. Uhlrich, F. Mahmoudi, and A. Arbey. Automatic extraction of one-loop Wilson coefficients in general BSM scenarios using MARTY-1.4. *PoS*, EPS-HEP2021:507, 2022.

[17] M.R. Buckley, D. Feld, and D. Goncalves. Scalar Simplified Models for Dark Matter. *Phys. Rev. D*, 91:015017, 2015.

[18] D. Abercrombie et al. Dark Matter benchmark models for early LHC Run-2 Searches: Report of the ATLAS/CMS Dark Matter Forum. *Phys. Dark Univ.*, 27:100371, 2020.

[19] P.Z. Skands et al. SUSY Les Houches accord: Interfacing SUSY spectrum calculators, decay packages, and event generators. *JHEP*, 07:036, 2004.

[20] B. C. Allanach et al. SUSY Les Houches Accord 2. *Comput. Phys. Commun.*, 180:8–25, 2009.

[21] R. L. Workman and Others. Review of Particle Physics. *PTEP*, 2022:083C01, 2022.

[22] F. Mahmoudi. SuperIso: A Program for calculating the isospin asymmetry of $B \rightarrow K^* \gamma$ in the MSSM. *Comput. Phys. Commun.*, 178:745–754, 2008.

[23] F. Mahmoudi. SuperIso v2.3: A Program for calculating flavor physics observables in Supersymmetry. *Comput. Phys. Commun.*, 180:1579–1613, 2009.

[24] F. Mahmoudi. SuperIso v3.0, flavor physics observables calculations: Extension to NMSSM. *Comput. Phys. Commun.*, 180:1718–1719, 2009.

[25] S. Neshatpour and F. Mahmoudi. Flavour Physics Phenomenology with SuperIso. *PoS*, CompTools2021:010, 2022.