MARTY-1.5
A **M**odern **AR**tificial **T**heoretical ph**Y**sicist
User Manual

Grégoire Uhlrich

May 30, 2022

Contents

List of Figures

List of Tables

List of sample codes

Introduction

MARTY (for **M**odern **AR**tificial **T**heoretical ph**Y**sicist) is a C++ framework developed to accelerate phenomenology Beyond the Standard Model (BSM). When doing phenomenology one wants to build a predictive model, and compare its predictions to experimental data and Standard Model (SM) values. This task is extremely difficult because it needs an important amount of man power to perform the required calculations in general BSM models. Calculating observables require first to be able to evaluate transition amplitudes, cross-sections and Wilson coefficients in effective theories. These calculations can be done pertubatively but represent a real challenge if one wants to do them by hand, considering that a single calculation for a particular model can take weeks and is error prone.

Because of the possibly huge number of terms in amplitude calculations, they must be performed analytically. Computers are not built for that purpose, this is why a program automating theoretical calculations BSM needs a computer algebra system. Existing programs performing one-loop calculations BSM use Mathematica [1], a private and closed software for symbolic manipulations. This is the case of FormCalc [2] and SARAH [3]. Other Mathematica-based programs can perform several other tasks. FeynRules [4] calculates Feynman rules from a Lagrangian, FeynArts [2] can draw Feynman diagrams in the Mathematica interface. There also are codes that have implemented their own computer algebra system like LanHEP [5], CalcHEP [6] and CompHEP [7]. However these codes do not perform one-loop level computations, that are important for phenomenology as many processes are trivial at the tree-level. We may give the example of Flavor Changing Neutral Currents (FCNC) in flavor physics, crucial for phenomenology, that only appear at the one-loop level.

MARTY [8] solves this issue by proposing a free and open-source code, fully written in C++, that automates theoretical calculations at the one-loop level in a very large variety of BSM models. It has its own computer algebra system, CSL (**C**++ **S**ymbolic computation **L**ibrary). All calculations are automated from the Lagrangian of the theory. Feynman rules may first be derived, (squared) amplitudes and Wilson coefficients are calculated at tree-level or at one-loop. All calculations are automated in any BSM model, amplitudes are simplified as much as possible to allow a numerical evaluation that otherwise would be impossible. This evaluation is done by a C++ library generated by MARTY containing all the material necessary to calculate the exact value of the results, depending on the model parameters. MARTY also generates Feynman diagrams graphically with a Desktop application, GRAFED (**G**enerating and **R**endering **A**pplication for **FE**ynman **D**iagrams).

In this manual one may find detailed explanations on core features of MARTY. Keep in mind that there also is the documentation, more interactive and helpful when one already knows how to use MARTY, and the website on which the reader may find more information and in particular all the publications. MARTY permanently uses CSL as computational core to manipulate symbolic expressions. This manual will not go into much details of CSL as

there already is a comprehensive CSL manual.

Chapter 1 represents a review the basic principle of MARTY, including main CSL features. Chapters 2, 3 and 4 present the main interface of MARTY, respectively quantum fields, high energy physics models, and group theory. They are presented outside any actual program context. This context is then treated in chapters 5 for model building, 6 for calculations and 7 for the code generation. These three steps represent the main parts, in that order, of a typical MARTY program. Finally, a word is given about options in chapter 8, built-in MARTY models in chapter 9 and debugging possibilities in chapter 10.

Chapter 1

Basics

1.1 Philosophy

MARTY is an independent code, that has been given all the tools it needs to perform theoretical calculations BSM at the one-loop level.

Modern C++. MARTY has been written in C++ mainly for performance reasons. This language is an old derivative of C but is constantly improving and is the best comprise, in the high energy physics community, between an effective and popular language. The main competitor was python. Analytical calculations require however to manipulate thousands of terms, and we think that python could not offer enough guarantee in terms of execution speed[1].

MARTY is written with the C++17 standard (2017) following modern guidelines [9] and coding standards. C++ recent improvements[2] allow to write code in a safer, simpler, and more optimized way using in particular developments of the standard library [10]. We want to facilitate the development of MARTY in the future, being for us or not.

Generality. MARTY implementations are always very general, i.e. not limited to any particular case. Being for the model definitions or calculations, there is very few limitations to what it can do. Rather than implementing simple and fast algorithms to known cases, MARTY implements more complex and slower ways to get to the same results[3]. This way will however lead to all kind of results, for models we did not even think of when developing the code. We want the context in which MARTY can be used to be as large as possible. Once a fully general algorithm works we may start to optimize it to be more efficient for particular cases. Details on what models can be built in MARTY are given in chapter 5, and on what calculations it can perform in chapter 6. In particular, most of the content in this manual presents special cases of what MARTY can do. When there are real limitations, they are explicitly mentioned.

Independence. As we said in the Introduction, MARTY is independent of Mathematica, but in general of any other framework. We started from scratch in C++, just using the

[1] This is due to the fact that such a program would not much benefit from python pre-compiled functions such as numpy ones because we explicitly manipulate objects that are not numbers, but symbolic mathematical expressions.

[2] In particular since 2011, C++11, C++14 and C++17 standards.

[3] MARTY run-time could indeed be optimized at several places, but is still very efficient.

standard library and built what we needed to create such a code. This means in particular that all aspects of the code are under control. Symbolic manipulations, simplifications, Quantum Field Theory, Group Theory and more can all be found in MARTY's code, modified, corrected, extended. This is a great power implying great responsibilities in terms of development.

1.2 CSL

Symbolic manipulations needed to do physics are done with CSL. In the following is presented a very brief introduction on this computer algebra system, one may find more information in the manual and the documentation.

Throughout this manual, we consider that CSL is properly installed on the computer, with include and namespace statements as presented in sample code 1 in each program.

Sample code 1: CSL program

```
#include <csl.h>

using namespace std; // Standard library namespace
using namespace csl; // CSL namespace

int main() {

    // My program
    return 0;
}
```

1.2.1 The Expr type

Any mathematical expression in CSL can be stored in the Expr type. A number, a constant, a sum, a product, a mathematical function can all be contained in such variable. This type has a simple interface, allowing to print it in the terminal, access elements, and use mathematical operations. To create symbolic expressions, one must call dedicated CSL functions (all with a suffix _s for symbolic). One may then combine them to create more complex expressions with a standard syntax as shown in sample code 2.

Sample code 2: Basics on CSL expressions 1/2

```
Expr a = constant_s("a");
cout << (a*a + sqrt_s(4*a)) endl;
// >> a^2 + 2*a^(1/2)
```

1.2.2 Modifying expressions

Expressions may be modified with CSL interface functions (see sample code 3) or with member functions of the Abstract class, that one can access any time from an expression Expr using -> (expr->function() of expr an Expr variable).

Sample code 3: Basics on CSL expressions 2/2

```
Expr b = constant_s("b");
Expr expr = a*a + sqrt_s(4*a);
Replace(expr, a, b + 1);
cout << expr << endl;
// >> (1 + b)^2 + 2*(1 + b)^(1/2)
DeepExpand(expr);
cout << expr << endl;
// >> 1 + 2*b + b^2 + 2*(1 + b)^(1/2)
```

1.2.3 Accessing sub-expressions

One can easily access sub-expressions in CSL using the `Size()` interface function, and using `operator[]` as shown in sample code 4.

Sample code 4: Access CSL sub-expressions

Let's create a function that prints sub-arguments

```
void printArguments(Expr a)
{
    for (size_t i = 0; i != Size(a); ++i)
        cout << i << " : " << a[i] << endl;
    // Also possible to call a->size()
}
```

And use it

```
Expr a = constant_s("a");
Expr expr = 1 + pow_s(a, 2) + sqrt_s(4*a);
printArguments(expr);
// >> 0 : 1
// >> 1 : a^2
// >> 2 : 2*a^(1/2)
```

Note Using the subscript operator with `[i]` returns another expression of type `Expr`. This procedure can then be repeated recursively to parse the whole expression.

Note `size_t` is simply `long unsigned int`. It is defined to contain size values that cannot be negative and are encoded on 64 bits (on 64 bits architectures).

1.2.4 Tensors

Tensors are a major topic of interest in CSL, as they are ubiquitous in high-energy physics calculations. The principle is presented in figure 1.1. It is very similar to the Quantum Fields that will be presented in chapter 2. There is one unique parent, the abstract tensor, containing all intrinsic properties. The objects entering mathematical expressions, that may be multiple, contain only their specific properties (indices for example) and a pointer to their parent tensor. This allows to save memory, keeping access at all time to any

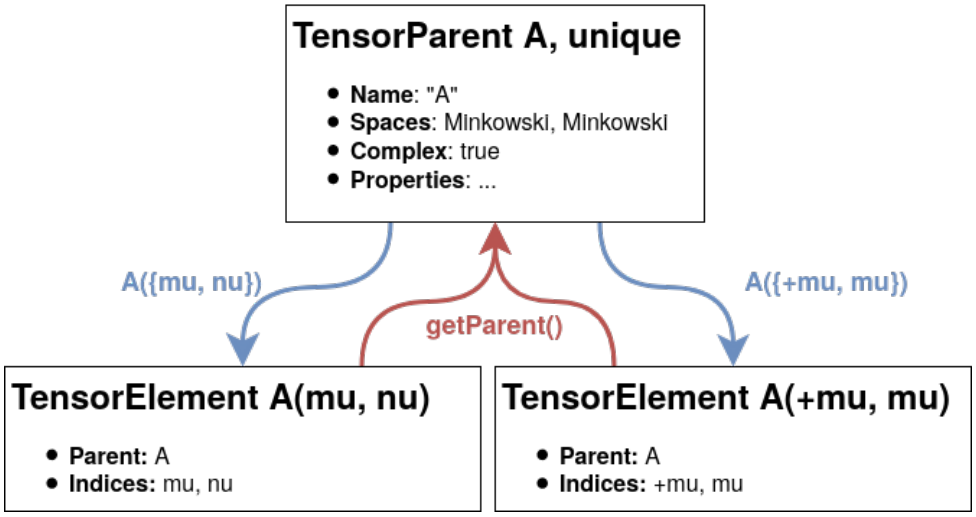


Figure 1.1: Working principle of tensors in `CSL`. The parent (TensorParent) is unique in the program, contains all its intrinsic properties, and can generate symbolic expressions (TensorElement) given some indices.

property through the parent pointer. Indices given to tensor are Index objects. They are simple data container and can be used easily as shown in the following.

TensorParent is not the end of the story, as its life-time must be managed correctly, taking into account the fact that it must not be destroyed if it exists at least one TensorElement. The parent is then encapsulated in a Tensor object, that is a `shared_ptr<TensorParent>` re-implementing some interface.

The reader should keep in mind that the object one manipulates in `CSL` is a `Tensor`. One can give it indices between brackets, and access all member functions of `TensorParent` with `->`. When calling the parent with indices, an expression `Expr` specialized in `TensorElement` is returned. Examples of use are shown in sample code 5. The principle is exactly the same for Quantum fields:

- A Quantum field element, the equivalent of `TensorElement`, is called `QuantumField`.
- A Quantum field parent, the equivalent of `TensorParent`, is called `QuantumFieldParent`.
- The actual parent object the user manipulates, the equivalent of `Tensor`, is called `Particle`.

Sample code 5: Basics on CSL tensors

Creating or getting existing tensors

```
Tensor A("A", {&Minkowski, &Minkowski});
Tensor X("X", &Minkowski);
Tensor g = GetMetric(Minkowski);
```

Creating indices to feed tensors

```
Index mu = GenerateIndex(Minkowski);
Index nu = GenerateIndex(Minkowski);
```

Creating symbolic indexed expressions

```
Expr indexed = A({mu, nu})*(X(+mu)*X(+nu) + g({+mu, +nu}));
cout << indexed << endl;
// >> A_{%mu,%nu}*(X_+%mu*X_+%nu + g_{+%mu,+%nu})
cout << Expanded(indexed) << endl;
// >> A_{+%mu,%mu} + A_{%mu,%nu}*X_+%mu*X_+%nu
```

Note The % symbol means that an index is summed over (repeated) in an expression, and the + symbol means that it is up (for spaces with non-trivial metric like Minkowski that have up and down indices).

1.2.5 Deeper features

This section is a crash course on CSL. To get deeper on how to read and modify symbolic expressions, feel free to take a look at the comprehensive CSL manual. One can find much more details and explanations to really master CSL.

1.3 Basic principles

MARTY is a very general code for high-energy physics. From model building to theoretical calculations at one-loop, everything can be done in MARTY for BSM. Amplitudes, squared amplitudes, Wilson coefficients may be calculated in full generality in a very large variety of BSM models. The idea behind that code is to be able to build a new BSM model easily and perform a detailed phenomenological study within a few lines of C++ and very little time, greatly accelerating the research Beyond the Standard Model.

Such a complex code must encapsulate as many objects as possible and let the user a simple interface to use. This is what is done in MARTY as most of the features can be accessed through a single function call. For interface functions, parameters are simplified as much as possible in order to ask the user a very small quantity of information. A sketch of how MARTY works internally is presented in figure 1.2. One may see in particular that there is no direct communication between the user and calculation modules of MARTY, or CSL the computer algebra system of MARTY. This is the consequence of the fact that calculations are fully automated, without any need for additional information from the user. Note that one can still use all CSL capabilities to modify results as they are symbolic expressions stored in Expr variables.[4]

[4]Results are rarely composed only of symbolic expressions because there is more information to return to the user. One may however always get to symbolic expressions and use CSL to modify them. See chapter 6 for more details.

A standard MARTY program always goes through the same steps.

- **Model loading.** The model can already be ready to use, or need some calculation steps to be done by MARTY. See chapter 5 for information on how to build a model.
- **Setting of options.** Before doing calculations, one may want to change some options to make MARTY do the calculation a certain way. See chapter 8 for more information on possible options.
- **Calculations.** Once the model is loaded and options are set, one may launch the calculation(s) that MARTY must perform. Details are given in chapter 6.
- **Library generation.** After calculating a theoretical quantity analytically, one makes MARTY generate the corresponding C++ code to evaluate the results numerically depending on the model parameters. This procedure is explained is chapter 7.

All these steps are typically very simple to implement and a MARTY program rarely takes more than a hundred lines of code, including line breaks and spaces for readability. The only step that may be very long is to build the model if it is not given by MARTY. A model like the MSSM for example cannot be built from scratch in MARTY in a few lines, even with a simple user interface (see chapter 5).

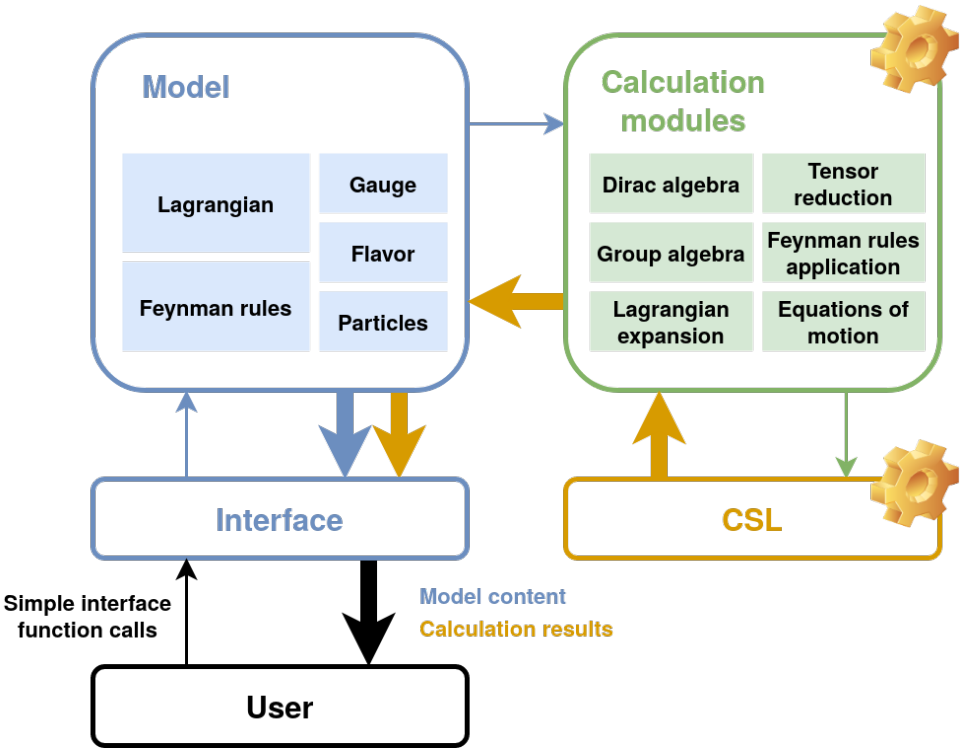


Figure 1.2: Sketch of MARTY basic principle. The user communicates (in C++) with the high-energy physics model to get / modify its content, and to perform calculations. These calculations are done internally by MARTY (using CSL) and are, at least for now, completely separated from the user interface.

Once the numerical C++ code is generated, one may use the resulting values in your favorite phenomenological code to scan the model, detect interesting scenarios with respect to the SM, compare to experimental data etc. At the end of the day, an ultimate tool

would be to have a direct interface between `MARTY` and such codes. It is actually already on the way for SuperIso [11, 12, 13] in flavor physics, and SuperIso Relic [14, 15, 16] in Dark Matter. With such interfaces, one may launch a complete phenomenological analysis of a BSM model at one-loop by basically pressing the Enter key and drinking coffee or watching '*Back to the Future*'.

Chapter 2

Quantum Fields

This chapter is dedicated to MARTY's user interface for quantum fields. The QuantumField object inherits from TensorFieldElement in CSL. In section 1.2 we saw the basic principle of an indexed tensor in CSL, in particular the object TensorElement. A tensor field is a tensor with a space-time point:

$$A_{\mu\nu} \to A_{\mu\nu}(X), \tag{2.1}$$

and a quantum field is a tensor field with more properties. Recalling figure 1.1, the same principle holds for quantum fields. There is simply more properties in a quantum field, and the naming convention is slightly different[1]. The equivalent sketch for the `QuantumField` object is shown in figure 2.1.

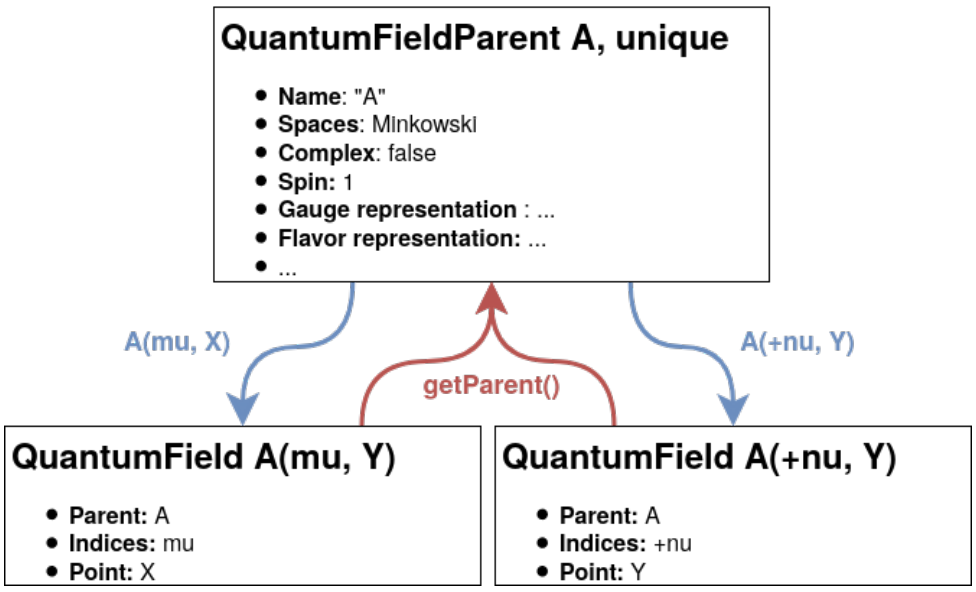


Figure 2.1: Working principle of quantum fields in MARTY. The parent (QuantumFieldParent) is unique in the program, contains all its intrinsic properties, and can generate symbolic expressions (QuantumField) given some indices and a space-time point.

As in the case of simple tensors, the object in the user's hands is not directly the quantum parent, but in this case a Particle object. It is a shared pointer to `QuantumFieldParent`, ensuring a well-managed life-time for the parent. MARTY's user interface always takes

[1] We should have named `QuantumFieldElement` the object entering expressions, but named it `QuantumField` instead because this object is used at many places in MARTY's code and had to be shorter for readability reasons.

`Particle` objects, and the `QuantumFieldParent` interface can be used simply using `->` on a `Particle`.

2.1 Different types of quantum fields

All possible quantum fields in `MARTY` inherit from `QuantumFieldParent` as shown in figure 2.2. The base class is not constructible, and users manipulate a (shared) pointer to this base class. This shared pointer is encapsulated in a `Particle` object and can then reference a Dirac fermion, a vector boson, etc.

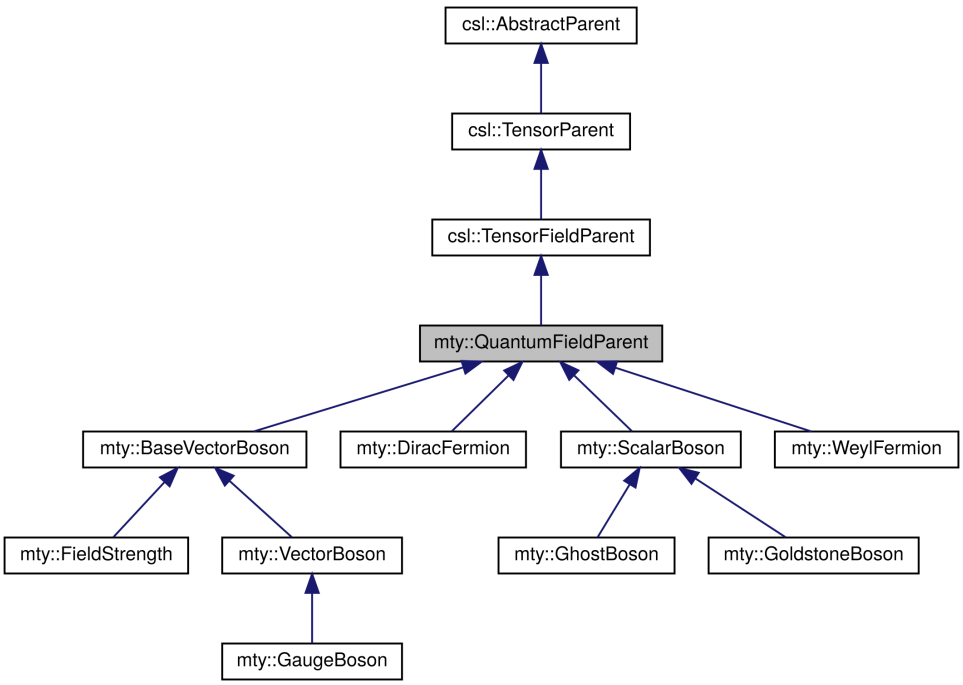


Figure 2.2: Inheritance tree for the QuantumFieldParent object. It first inherits from `CSL` tensors, and then is specialized in the different types of particles in `MARTY`.

2.1.1 Overview

A particle in `MARTY` is one specialization of `QuantumFieldParent`:

- ScalarBoson: Scalar boson, trivial representation of the Lorentz group.
- WeylFermion: Chiral two-component fermion. Weyl fermions are always projected on left or right chiralities in amplitudes. They may be paired to form a Dirac fermion.
- DiracFermion: 2-component Majorana fermion if self-conjugate, 4-component Dirac fermion otherwise. Majorana fermions are embedded in Dirac notation in `MARTY`. Dirac fermions can contain Weyl sub-parts with the definition $\psi = \psi_L \oplus \psi_R$.
- VectorBoson: Spin 1 particle A_μ, associated with a field strength. A vector boson can also have associated ghost and Goldstone bosons.

- GaugeBoson: Specialization of `VectorBoson` that keeps a pointer to the gauged group it is the gauge boson of. A gauge boson in a non abelian gauged group (different from $U(1)$) has a predefined `GhostBoson`.

- FieldStrength: Field strength object for a vector boson (gauge or not) $F_{\mu\nu} = \partial_\mu A_\nu - \partial_\nu A_\mu$.

- GhostBoson: Ghost bosons defined in non abelian gauged group. It is a non-physical anti-commuting bosonic field introduced to quantize properly the theory that appears only at the one-loop level in diagrams. They are linked with their `GaugeBoson` through gauge fixing.

- GoldstoneBoson: Goldstone bosons may be defined to link a scalar boson of the theory as the Goldstone of a vector boson, to have proper gauge fixing conditions.

Some particles may be defined automatically by `MARTY`, like the gauge bosons and ghosts, when defining the gauge group of the theory. Users will however have to define particle content on their own[2], using built-in functions that create all the necessary particles presented above. All particle constructions work the same way. Each time, one must give the name of the particle, the model in which it lives, and additional arguments specific to that particle. In all sample codes in the following, `model` is assumed to be a valid `Model` object, whose gauge is already initialized (see chapter 5 for more details).

2.1.2 Fermions

Three types of spin 1/2 are possible to create in `MARTY`. Weyl, Dirac, and Majorana fermions. Majorana fermions do not have their own builder function, as they do not have their own class either. To create a Majorana one must first build a Dirac fermion, and then specify that it is self-conjugate. Doing so will enable non-trivial contractions in diagrams like $\langle\psi\psi\rangle$ and $\langle\bar{\psi}\bar{\psi}\rangle$, whereas for Dirac fermions only $\langle\psi\bar{\psi}\rangle$ and $\langle\bar{\psi}\psi\rangle$ do not vanish. Sample code 6 presents a summary on how to build fermions.

Sample code 6: Creating fermions

Dirac fermion

```
Particle e = diracfermion_s("e", model);
```

Weyl fermion

```
Particle muL = weylfermion_s("\\mu_L", model, Chirality::Left);
Particle muR = weylfermion_s("\\mu_R", model, Chirality::Right);
```

Majorana fermion

```
Particle maj = diracfermion_s("M", model);
maj->setSelfConjugate(true);
```

See also File fermionicField.h for the documentation of these functions.

[2]Only in a model building context. When using a built-in model, no need to define anything before using the model.

2.1.3 Vectors

Spin 1 particles are often built by MARTY, as they are in general gauge bosons. Most of high-energy physics models like the SM or even BSM do not have additional spin 1 particles. It is still possible in MARTY to create new spin 1 particles, as shown in sample code 7.

Sample code 7: Creating vector bosons

Vector boson

```
Particle A = vectorboson_s("A", model);
```

Field strength

```
Particle F_A = A->getFieldStrength();
```

See also File vectorField.h for the documentation of `vectorboson_s()`.

2.1.4 Scalars

Spin 0 particles are very simple to create as they have no specific property due to their spin. Procedure to create a scalar boson is presented in sample code 8.

Sample code 8: Creating scalars

```
Particle phi = scalarboson_s("\\phi", model);
```

See also File scalarField.h for the documentation of `scalarboson_s()`.

Ghosts and Goldstone bosons can be created explicitly as shown in sample code 9, but in general will be handled automatically by MARTY during the model construction. As those particles are tied to a given `VectorBoson`, no need to define any property. The user only has to give the associated vector, and an optional name.

Sample code 9: Creating ghosts and Golstones

Considering a vector boson `A` as in sample code 7.

Ghost

```
Particle ghost_A = ghostboson_s("c", A);
// Or with name chosen by marty
Particle ghost_A = ghostboson_s(A);
```

See also File ghostField.h for the documentation of `ghostboson_s()`.

Goldstone

```
Particle goldstone_A = goldstoneboson_s("c", A);
// Or with name chosen by marty
Particle goldstone_A = goldstoneboson_s(A);
```

See also File goldstoneField.h for the documentation of `goldstoneboson_s()`.

2.2 Using and modifying a Particle

Once a particle is build following prescriptions of section 2.1, the interface is almost always identical for all the different types of particles. This section will demonstrate how to do basic manipulations on particles and how to get information from them. Keep in mind that you may go at any time on the documentation of the Particle class, and the QuantumFieldParent class that contains all the interface this section presents.

2.2.1 Getting particles

Gauge-related particles

We saw in the previous section how to create new particles. Before going further, it is necessary to explain how to get the particles that `MARTY` creates on its own. Figure 2.3 presents the objects that are automatically created with gauged groups, including naming conventions, in `MARTY`. All default names can be changed by the user. It is however important to know the initial convention to be able to access all objects created automatically.

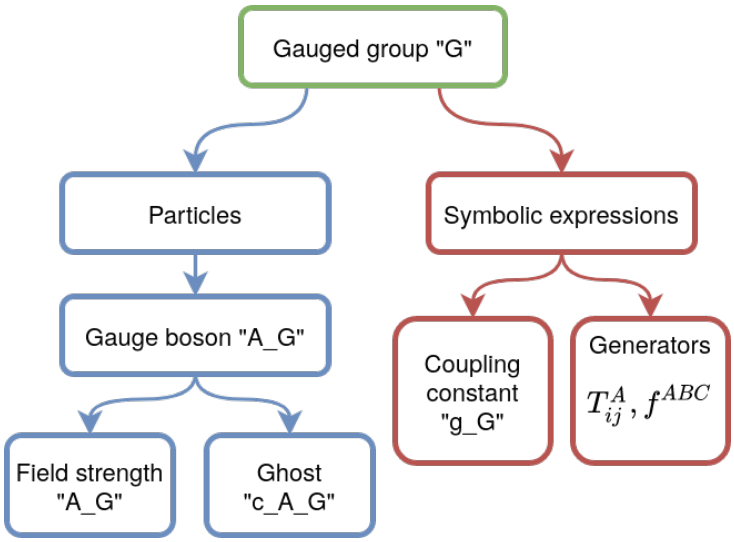


Figure 2.3: Sketch of the main objects a `MARTY` gauged group creates with him. The ghost particle and the generators are not created for the $U(1)$ group as it is abelian.

The way to get different kinds of particles from a model `MARTY` is shown in sample code 10. The field strength cannot be retrieved directly from the model as it has the same name as the vector boson. It is actually not a different particle, simply another structure. This is why the field strength is given by the particle itself, not the model.

Sample code 10: Getting a particle from a model

```
Particle e_L = model.getParticle("e_L"); // Weyl fermion
Particle A = model.getParticle("A_G"); // Vector boson of group "G"
Particle F_A = A->getFieldStrength(); // Field strength of group "G"
Particle c_A = model.getParticle("c_A_G"); // Ghost boson of group "G"
```

Dirac fermion embedding

When creating a Dirac fermion, `MARTY` automatically creates its left and right parts. The three generated particles can talk to each other, and in particular a user can navigate in the triangle they define through simple function calls. This is presented in figure 2.4. Concrete examples in a `MARTY` program are presented in sample code 11.

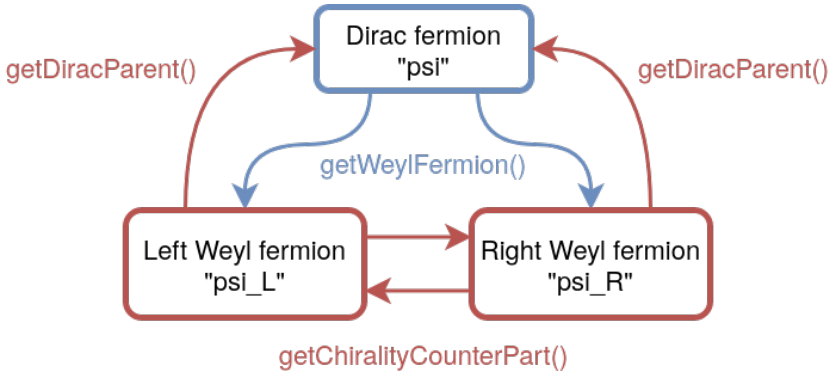


Figure 2.4: Sketch of the relations between the different parts of a Dirac fermion embedding $\psi \equiv \psi_L \oplus \psi_R$. The `getWeylFermion()` function takes as argument a Chirality.

Sample code 11: Dirac fermion embedding

Creating a Dirac fermion

```
Particle psi = diracfermion_s("psi", model);
```

Navigating in the triangle

```
Particle psi_L = psi->getWeylFermion(Chirality::Left);
Particle psi_R = psi_L->getChiralityCounterPart();
Particle other_psi = psi_R->getDiracParent();
```

See also File fermionicField.h to see the documentation.

2.2.2 Basic properties

Basic properties means properties that are not the gauge or flavor representations, treated separately in section 2.2.3. The main properties of quantum fields are presented in table 2.1.

Property	Type	Getter	Setter
Name	`string`	`getName()`	`setName()`
Latex name	`string`	`getLatexName()`	`setLatexName()`
Spin dimension	`int`	`getSpinDimension()`	
Mass	`Expr`	`getMass()`	`setMass()`
Width	`Expr`	`getWidth()`	`setWidth()`
Self-conjugation	`bool`	`isSelfConjugate()`	`setSelfConjugate()`
Physicality	`bool`	`isPhysical()`	`setPhysical()`

Table 2.1: List of properties for quantum fields, with each time the type, getter and setter functions. For setter functions, users must of course give one argument of the right type. See the documentation of class QuantumFieldParent for more information.

Name

Names of particles have two different purposes apart from identifying their owners in expressions. Firstly, names must uniquely define particles to allow a user to identify a particle in a model, at any time, simply given its name. This name has to be short and simple to lighten the interface. Another use of names is in Feynman diagrams, where one prefers in general to see ν_{μ_L} rather than *num*. The problem is that latex expressions are complicated, in that case one should write `"\\nu_{\\mu_L}"` each time referring to the muon neutrino.

This is why regular and latex names are separated since MARTY-1.2. If not specified, the latex name will be identical to the regular one. To avoid multiple calls (`setName()` and `setLatexName()`), it is possible when creating a particle to give both names in the same string literal, separating them by a `;` (spaces around it are ignored). For example, one can define a neutrino with

```
Particle num = weylfermion_s("num ; \\nu_{\\mu_L}", Chirality::Left);
```

For identification purposes `"num"` will then have to be used, but Feynman diagrams will display ν_{μ_L}.

Spin

The spin can of course not be changed, as it would require to change the particle type. The user may access its value at anytime with the `getSpinDimension()` function. Be aware however that as this function returns an integer, one does not get the spin but the spin dimension

$$d = 2j + 1\,, \tag{2.2}$$

for a particle of spin j. Examples of property usage are presented in sample code 12.

Physicality

A non-physical particle cannot be used as external particle in any process. By default ghosts and Goldstone bosons are non-physical and all other particles are, but this property may be changed by the user. Examples of property usage are presented in sample code 12.

Self-conjugation

The self-conjugation property is in general the possibility for the field to contract with itself, not complex conjugated. For a self-conjugate field, contractions like $\langle\phi\phi\rangle$ and $\langle\phi^\dagger\phi^\dagger\rangle$ are enabled in diagrams whereas only $\langle\phi\phi^\dagger\rangle$ and $\langle\phi^\dagger\phi\rangle$ are for other fields, as shown in figure 2.5. This is a general statement. For integer spin particles it is simpler because the self-conjugate property corresponds the the fact that the field is real:

$$\begin{aligned}\phi^\dagger &= \phi \text{ for scalars bosons,}\\ A_\mu^\dagger &= A_\mu \text{ for vector bosons.}\end{aligned} \tag{2.3}$$

There is in this case only one possible contraction.

For fermions however it is more subtle, because the self-conjugate Dirac field is not real. It has 2 degrees of freedom instead of 4 but this is not equivalent to $\psi^\dagger = \psi$, at least not in all realizations. For spin 1/2 particles, the self-conjugate property reads

$$\psi^c \equiv C\bar{\psi} = \psi, \tag{2.4}$$

with the conjugation matrix

$$C \equiv i\gamma^0\gamma^2 \tag{2.5}$$

in the Weyl realization for $\gamma-$matrices, that is the one used in `MARTY`. As the relation is not as simple as $\psi^\dagger = \psi$, we keep both ψ and $\psi^\dagger$ in the Lagrangian, and the four possible contractions presented in figure 2.5 are possible replacing ϕ by ψ. Examples of property usage are presented in sample code 12.

(a) Standard contractions **(b)** Self-conjugate contractions

Figure 2.5: Possible field contractions. For non self-conjugate particles only contractions (a) do not vanish. For self-conjugate particles, all four contractions are non-zero.

Mass and Width

Mass and width arise in propagators when calculating an amplitude. They may be set to any expression, not only constants or zero. One may for example give a mass

$$M_G = \sqrt{\xi} M_A \tag{2.6}$$

to a Goldstone boson related to a vector A, with gauge fixing parameter ξ.

The mass M and width Γ of a particle appear in propagator denominators

$$\frac{1}{p^2 - M^2 + iM\Gamma}. \tag{2.7}$$

The mass also appears in numerators for fermions and vector bosons but we will not detail theses dependencies here. Examples of property usage are presented in sample code 12.

Sample code 12: Quantum fields properties usage

Building particles for the example

```
Particle W = vectorboson_s("W", model);
Particle c_W = ghostboson_s(W);
Particle psi = diracfermion_s("psi", model);
```

The basics

```
cout << c_W->isPhysical() << endl; // 0
cout << W->isPhysical() << endl; // 1
c_W->setPhysical(true);
cout << c_W->isPhysical() << endl; // 1
cout << c_W->getSpinDimension() << "␣" << psi->getSpinDimension()
     << "␣" << W->getSpinDimension();
// >> 1 2 3
```

Mass and width

```
Expr xi = constant_s("xi");
Expr M_W = constant_s("M_W");
Expr G_W = constant_s("G_W");
W->setMass(M_W);
W->setWidth(G_W);
c_W->setMass(sqrt_s(xi) * W->getMass()); // m_{c_W} = sqrt(xi) M_W
cout << c_W->getMass() << endl;
// >> xi^(1/2)*M_W
```

2.2.3 Gauge and Flavor representations

Gauge and flavor representations are crucial for the user to understand as any model building activity with MARTY will require a good knowledge of this aspect.

Gauge representations

MARTY can handle all irreducible representations of all semi-simple groups. These representations are defined by Dynkin labels [17]. These labels are positive integers and the number of labels defining representations of a given group corresponds to the rank ℓ of the algebra. Table 2.2 presents the link between gauged groups and their corresponding algebra.

One may easily find definitions of Dynkin labels for common representations in the literature. Table 2.3 shows sets of labels for the most used representations in physics.

Now that irreducible representations have been defined for non abelian gauged groups, let us treat the special $U(1)$ case. There is no Dynkin label in this case as only a fractional charges define $U(1)$ representations. In order to unify notations when doing model building for $U(1)$ or non abelian gauged groups, a fractional charge is treated as a pair of Dynkin labels, one for the numerator and one for the denominator. For example, in the SM, the electron has Dynkin labels $(-1, 1)$ for the electromagnetic $U(1)$ gauge whereas the up quark has labels $(2, 3)$.

More details on irreducible representations will be given in chapter 4 and on model

Group	Algebra	Rank
$SU(N)$	A_ℓ	$\ell = N - 1$
$SO(2N+1)$	B_ℓ	$\ell = N$
$Sp(2N)$	C_ℓ	$\ell = N$
$SO(2N)$	D_ℓ	$\ell = N$
E_6	E_6	$\ell = 6$
E_7	E_7	$\ell = 7$
E_8	E_8	$\ell = 8$
F_4	F_4	$\ell = 4$
G_2	G_2	$\ell = 2$

Table 2.2: Link between semi-simple groups and their algebras. The rank ℓ of the algebra corresponds to the number of Dynkin labels defining uniquely irreducible representations.

Group	Algebra	Dynkin labels	Dimension
$SU(2)$	A_1	(1)	2 (doublet)
$SU(2)$	A_1	(2)	3 (triplet)
$SU(3)$	A_2	$(1,0)$	3 (triplet)
$SU(3)$	A_2	$(0,1)$	$\bar{3}$ (anti-triplet)
$SU(3)$	A_2	$(1,1)$	8 (octet)
$SU(3)$	A_2	$(2,0)$	6 (sextet)
$SO(4)$	D_2	$(1,0)$	2 (left spinor)
$SO(4)$	D_2	$(0,1)$	$\bar{2}$ (right spinor)
$SO(4)$	D_2	$(1,1)$	4 (vector)
E_6	E_6	$(1,0,0,0,0,0)$	27

Table 2.3: Common representations in high energy physics, with their group, algebra, Dynkin labels and dimensions. Trivial representations (dimension 1) have always labels equal to zero.

building in chapter 5, but let us do a quick introduction on how to define gauge representation for particles in MARTY.

By default, all representations are trivial[3]. Calling the `setGroupRep()` function giving the name of the gauge group and the Dynkin labels between curly braces will automatically change the representation of the particle. An example in a $SU(2)_L \times U(1)_Y$ gauge is given in sample code 13.

Sample code 13: Setting gauge representations

Considering a gauge composed of one $SU(2)$ group named `"L"` and one $U(1)$ group named `"Y"`, one can set the representation of $e_L : (2, -1)$ and $u_R : (1, 2/3)$ writing

```
Particle e_L = weylfermion_s("e_L", model, Chirality::Left);
e_L->setGroupRep("L", {1});
e_L->setGroupRep("Y", {-1}); // denomimator = 1 omitted
```

for the electron and

```
Particle u_R = weylfermion_s("u_R", model, Chirality::Right);
u_R->setGroupRep("Y", {2, 3});
```

for the quark.

Note This example does not exactly correspond to SM conventions for simplicity.

Note When the denominator of a fractional charge is 1 it may be omitted when giving the representation.

See also Chapter 4 for more details on representations, and chapter 5 to see how to build a model in MARTY.

Flavor representations

Flavor in high energy physics has a particular meaning, like the 3 generations of matter particles in the Standard Model. In MARTY it has a slightly more general meaning, as flavor allow users to define as many additional indices as wanted for quantum fields, real or complex. Flavor representations are for now limited to two types:

- **Complex flavors.** Mixes N complex fields, that are considered as a fundamental representation of a $SU(N)$ flavor group.
- **Real flavors.** Mixes N real fields, that are considered as a vector representation of a $SO(N)$ flavor group.

Flavor machinery could be extended in the future is needed. We may consider for example different representations of a same flavor group, with tensors mixing different representations. We think that these cases are very rare and there is then no support in MARTY to do it automatically. Work-arounds are however already possible, like creating different flavors to simulate representations of different dimensions.

Sample code 14 shows how to define flavor representations for particles in a MARTY model. See chapter 5 for more details on model building.

[3]Dimension 1 representation for non abelian gauged group and 0 charge for $U(1)$ group.

Sample code 14: Setting flavor representations

Let's consider a model with two flavor groups. One complex flavor $SU(3)$ named `"C"` and one real $SO(4)$ named `"R"`.

Creating the fields

```
// Real field for the real SO(4) flavor
Particle phi = scalarboson_s("phi", model);
phi->setSelfConjugate(true);

// Complex field for the complex SU(3) flavor
Particle psi = diracfermion_s("psi", model);
```

Setting the representations

```
phi->setFundamentalFlavorRep("R");
psi->setFundamentalFlavorRep("C");
```

Note As we may extend `MARTY` for non fundamental representations in the future, one has to precise through the function name. But for now, no other representation is available.

See also Documentation of QuantumFieldParent and chapter 5 for more details on model building.

2.3 Quantum Fields in expressions

This section presents how to handle quantum fields in symbolic expressions. This is a deeper feature of `MARTY`, and users should not have to do it in general as calculations are fully automated. The idea here is to learn how to create expressions containing quantum field objects, and also read and manipulate them. A user new to this framework may want to skip this part.

2.3.1 Indices

To create symbolic quantum field objects, one must first have indices to give to the particle. Minkowski, Dirac and gauge indices can be gathered simply from the user interface. There is several ways to get indices. The simplest way is presented in sample code 15. The reader should keep in mind why group indices are more complicated to get. There is one vector space per irreducible representation and per gauged group. The model will generate group indices for you given the number of indices you want, the model, the group or group name, and a particle or particle name (the model returns indices in the vector space of the representation of the particle in this particular group).

Sample code 15: Generating indices

Minkowski indices

```
auto mu = MinkowskiIndices(10); // 10 indices
```

Dirac indices

```
auto alpha = DiracIndices(10); // 10 indices
```

Gauge indices, for the representation of the particle "phi" in the gauged group "g"

```
auto A = GaugeIndices(10, model, "g", "phi"); // 10 indices
```

Flavor indices, for the flavor "f"

```
auto I = FlavorIndices(10, model, "f"); // 10 indices
```

Indices can then be used simply using the subscript operator []

```
mu[i]; // is a Minkowski index for i in [0, 10[
alpha[i]; // is a Dirac index for i in [0, 10[
A[i]; // is a Group index for i in [0, 10[
I[i]; // is a Flavor index for i in [0, 10[
```

Note The `auto` keyword allows the user to not care about the type of the index collection: `std::vector<csl::Index>` (or `vector<Index>` without namespaces).

See also The documentation of the Index class.

2.3.2 Space-time point

Quantum fields are tensor fields. They need then a space-time point to live, and MARTY for now is limited to the Minkowski space[4]. One simply generates a space-time point (vector in Minkowski space) following prescription in sample code 16.

Sample code 16: Generating space-time points

The easy way

```
Tensor X = MinkowskiVector("X");
```

The not much harder way

```
Tensor X("X", &Minkowski);
```

Note The second way is fully general and can be used for any vector space (`Minkowski` here).

See also Documentation of Tensor or section 1.2.4 for more details on tensors.

Generating a space-time point is not necessary for quantum fields. It will be if you have to distinguish different points between several fields in the same expression. This is the case when applying the Wick theorem for example, that needs the information of the

[4]This statement does not mean that tensor fields can only live in the Minkowski space. CSL is fully general and there is no such limitation. MARTY calculations are limited to the Minkowski space for now, so quantum fields must live in it.

different space-time points of the fields. However, one may omit the space-time point. In that case, a default one (that is also used in the Lagrangians) is introduced automatically by MARTY.

2.3.3 Creating an expression from a Particle

Now that we saw in section 2.3.1 how to generate indices for quantum fields and in section 2.3.2 how to generate space-time points, we have all we need to create symbolic expressions from Particle objects.

We will consider here the example of a $SU(3)_C \times SU(2)_L$ gauge with a fermion Q in the fundamental representation of both groups, and a vector W in the adjoint representation of the $SU(2)_L$ group. In top of that, let us introduce a complex $SU(3)$ flavor named 'F' for the fermion in order to really have a complete example. Sample code 17 shows how to create a gauge interaction term like the following

$$\mathcal{L} \ni ig\bar{\psi}\not{W}\psi = ig\bar{\psi}^{Iai}_{\alpha}(X)\gamma^{\mu}_{\alpha\beta}W^{A}_{\mu}(X)T^{A}_{ij}\psi^{Iaj}_{\beta}(X), \tag{2.8}$$

with μ a Minkowski index, (i, j) indices in the doublet representation of $SU(2)_L$, a an index in the triplet representation of $SU(3)_C$, A an index in the triplet representation of $SU(2)_L$, and I an index in the 3-dimensional flavor for the fermion. This example is complex on purpose, because it shows the reader how to create arbitrary interaction terms in MARTY, introducing all necessary interface to do so. One may notice that all indices are explicit in MARTY. It may be rather long to write expressions with so many indices, but then cannot be removed apart from specific cases like γ matrices. We chose in MARTY to have one unique and general treatment for all indices. Interface could be improved in the future allowing for example to omit indices in bi-linear diagonal couplings (like $SU(3)_C$ and the flavor indices in equation 2.8), but for now all indices must be given.

An important thing to know is the order of indices defined for quantum fields. If not given in order, MARTY will raise an error and stop the program. The order is the following, to give from left to right:

- **Flavor indices** in the same order than when adding the corresponding flavors to the model.
- **Gauge indices** in the same order than when adding the corresponding gauged groups to the model.
- **Space-time indices**, that concerns for now only spin 1/2 (Dirac index) and spin 1 (Minkowski index) particles.

The term presented in equation 2.8 and in sample code 17 is a gauge interaction, that is of course given automatically by MARTY, but knowing how to build it by hand allows the reader now to create basically any interaction term to add it in the Lagrangian. More details on model building are given in chapter 5.

Sample code 17: Particles to symbolic expressions

We consider here a gauge formed by a $SU(3)$ group `"C"` and a $SU(2)$ `"L"`, with `psi` in the fundamental representation of both groups and in an additional $SU(3)$ flavor `"F"`, and finally `W` in the adjoint of the $SU(2)$.

Generating indices and space-time point

```
auto I = FlavorIndices(1, model, "F");
auto a = GaugeIndices(1, model, "C", "psi");
auto A = GaugeIndices(1, model, "L", "W");
auto i = GaugeIndices(2, model, "L", "psi");
auto mu = MinkowskiIndices(1);
auto al = DiracIndices(2);
Tensor X = MinkowskiVector("X");
```

Getting the two additional tensors we need

```
Tensor gamma = DiracGamma();
Tensor T = GetGenerator(model, "L", "psi");
```

Creating the expression

```
Expr g = constant_s("g");
Expr term = CSL_I * g
    * GetComplexConjugate(psi({I[0], a[0], i[0], al[0]}, X)),
    * W({A[0], +mu[0]}, X)
    * T({A[0], i[0], i[1]})
    * gamma({mu[0], al[0], al[1]})
    * psi({I[0], a[0], i[1], al[1]}, X);
```

Note Here as all fields have the same point X, we could omit it and the default space-time point of `MARTY` would be introduced automatically.

Note There is no need for γ^0 as in `MARTY` $\psi^\dagger$ is considered automatically as $\bar{\psi} = \psi^\dagger\gamma^0$. This saves a lot of unnecessary calculations.

See also Sample code 24 to have more details on how to get generators from a model.

2.3.4 Type system

`CSL` type system does not include `MARTY` objects, and in particular quantum fields. `MARTY` extends the type system to generalize it to any type, even types that may be defined later by the user[5]. While `CSL` type system allows one to find out which type an expression is (number, tensor, sum etc), the extended type system allows one to compare the expression to any given type (including `MARTY` types) and convert back to it (if the type is correct). An example given in sample code 18 shows how to, from a `CSL` expression containing a quantum field, recover the field inside the expression, and its Particle parent.

[5]Of course this generalization has a cost. This cost is to be slightly less simple and less optimized.

Sample code 18: Symbolic expressions to particle

Let's consider that `sfield` is an expression containing a quantum field (this is the case for `term[4]` in sample code 17).

Using CSL, we cannot get the physics information

```
cout << IsIndicialTensor(sfield) << endl;
// >> 1
cout << GetPrimaryType(sfield) << endl;
// >> Indicial
cout << GetType(sfield) << endl;
// >> TensorFieldElement
```

Using the extended type system in MARTY, getting the QuantumField and the Particle

```
if (IsOfType<QuantumField>(sfield)) {
    QuantumField field = ConvertTo<QuantumField>(sfield);
    Particle particle = field.getParticle();
}
```

Warning If one wants to modify the quantum field and the symbolic expression at the same time, one must get a pointer to the field:

```
if (IsOfType<QuantumField>(sfield)) {
    QuantumField *field = ConvertToPtr<QuantumField>(sfield);
    // Here modification of field will affect sfield also
    Particle particle = field->getParticle();
}
```

See also The CSL manual for more information on CSL's type system.

2.3.5 Polarization field

Polarization fields are another quantum field type object in expressions[6]. They represent momentum space fields, with spin explicit spin indices. This gives

$$\phi(X) \to \phi(p) \text{ for scalars,} \tag{2.9}$$
$$A^{\mu}(X) \to \epsilon^{\mu}_{\lambda}(p) \text{ for vectors,} \tag{2.10}$$
$$\psi_{\alpha}(X) \to u_{\alpha\sigma}(p) \text{ for fermions,} \tag{2.11}$$

with λ a spin index for the vector and σ for the fermion. These indices are important to calculate squared amplitudes as polarization sum rules must be applied like

$$\sum_{\lambda} \epsilon^{\mu}_{\lambda}(p)\epsilon^{*\nu}_{\lambda}(p) = -ig^{\mu\nu} \tag{2.12}$$

for the photon for example, and

$$\sum_{\sigma} u_{\alpha\sigma}(p)\bar{u}_{\beta\sigma}(p) = \left(\not{p} + m\right)_{\alpha\beta} \tag{2.13}$$

[6]Polarization fields, at the difference of fermionic quantum fields, commute with each other.

for fermion particles.

Scalar polarization fields do not appear in amplitude results as they have no spin. An option can however be set to tell MARTY to create them, in order to keep track of scalar field insertions in amplitudes. See chapter 8 for more details on options.

All the statements valid for quantum fields in sections 2.3.3 and 2.3.4 are valid for polarization fields, with two main differences:

- The type of expression is not the same, `QuantumField` must then be replaced by `PolarizationField` when testing the type or converting a symbolic expression (see sample code 18).
- Polarization fields are generated also by the same particle, giving a polarization index separately from the other indices, before. An example is presented in sample code 19.

Sample code 19: Polarization fields

Taking the same example as in sample code 17, we create here the same term but with `PolarizationField` objects instead of `QuantumField` objects (useless here but pedagogical)

Creating polarization indices

```
auto pol = Euclid_R3.generateIndices(3);
```

Creating the expression

```
Expr g = constant_s("g");
Expr term = CSL_I * g
* GetComplexConjugate(psi(pol[1], {I[0], a[0], i[0], al[0]}, X)),
* W(pol[1], {A[0], +mu[0]}, X)
* T({A[0], i[0], i[1]})
* gamma({mu[0], al[0], al[1]})
* psi(pol[2], {I[0], a[0], i[1], al[1]}, X);
```

Note Vector spaces for polarization indices do not matter, one may create them, for any particle, from any vector space in the program, like the built-in 3 dimensional space `Euclid_R3` for example.

Warning For scalar bosons one must also give a polarization index, even if the particle has no spin.

See also The documentation of the class PolarizationField.

Chapter 3

Models

3.1 Introduction

Models are very large objects. They basically contain all the theory, and all the user interface for model building (see chapter 5) and calculations (see chapter 6). For this reason, the model is divided in three different parts. The inheritance hierarchy is presented in figure 3.1.

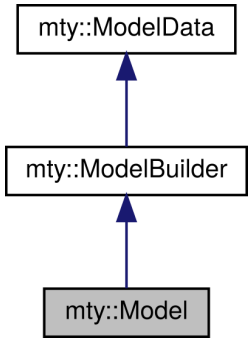


Figure 3.1: Inheritance tree for the Model object. Readers may see the three parts composing a Model. The data container, the model builder, and the final interface with calculation features.

The first interface, ModelData, contains almost all the content of the model: Lagrangian, particles, gauge, flavor, etc. It also has methods to get and modify these different elements. Then ModelBuilder implements all model building features like gauge symmetry breaking, particle replacement, diagonalization etc. Finally, the Model class has methods to perform physical calculations BSM. Feynman rules, amplitudes, squared amplitudes and Wilson coefficients can be calculated from this class.

While the `ModelData` interface is presented in this chapter, `ModelBuilder` and `Model` will be detailed respectively in the chapters about model building 5 and about calculations 6.

3.2 ModelData interface

Class ModelData contains all the basic interface to store and manipulate the content of a high-energy physics model. It contains many methods and we think that its documentation is also well suited to learn all the features. In the following is presented the main manipulations users may have to do through the `ModelData` interface.

Table 3.1 presents the attributes of `ModelData` giving their types and roles in the program. Users may not access them directly but only through the different methods of the class. More details on how to manipulate these attributes are given in the next sections.

For the methods of this class that take a `Particle` as parameter, the user may actually often give different objects, not only a `Particle`. Functions are template, taking any argument that may be given to the `getParticle()` function. In other words when `MARTY` asks the user a `Particle` object, any object may be given that is either directly a `Particle` or an object `MARTY` can use to find a particle through its `getParticle()` methods, like a name, a `QuantumField` object, `QuantumFieldParent`, or even an expression if it contains a quantum field. This simplifies the interface for users, as basically any object representing a field can be given to the template methods. This allows to replace code like

```
model.doSomeThingWithParticle(model.getParticle("phi"));
```

by

```
model.doSomeThingWithParticle("phi");
```

In particular, one can give each time a name `"phi"` for example instead of searching the actual variable containing `phi`.

The same principle holds for gauge and flavor groups and the `getGroup()` methods. This means that a method taking a group as parameter can use a group name instead of a `Group` object.

Attribute name	Type	Purpose
L	`Lagrangian` (doc)	Lagrangian of the model, contains all interaction terms. See section 3.2.3 for more details.
spaceTime	`Space const*` (doc)	Space-time of the theory, for now always `Minkowski`.
gauge	`unique_ptr<Gauge>` (doc)	Gauge group, containing all gauged groups.
flavor	`unique_ptr<Flavor>` (doc)	Flavor, containing all flavor groups
particles	`vector<Particle>` (doc)	List of all particles in the model.
quantumNumbers	`vector<QuantumNumber>` (doc)	List of quantum numbers in the model.
scalarCouplings	`vector<Expr>` (doc)	List of all scalar couplings, in particular gauge couplings.
tensorCouplings	`vector<Tensor>` (doc)	List of all tensor couplings.
momenta	`vector<Tensor>`	List of all momenta used in amplitude calculations.
momentaSquared	`map<pair<size_t,size_t>, Expr>`	Map between pairs of positive integers (i, j) and symbolic quantity. Corresponds to scalar products of momenta $s_{ij} = p_i \cdot p_j$.
gaugeLocked	`bool`	Tells if the gauge is fully initialized (true) and if particle content may be added.

Table 3.1: Attributes of the `ModelData` class with their type, documentation link, and purpose.

3.2.1 Adding / Removing particles

Particles may be added to or removed from a Model. A user will add a particle typically when doing model building (see chapter 5 for more details). Once a particle is added in the model its gauge and flavor representations must not be changed as some default gauge interactions are introduced automatically by MARTY at that moment. Kinetic and mass terms are also introduced automatically, but as one may read in section 3.2.3, masses of particles may still be changed by users. It is also possible to forbid MARTY to introduce any Lagrangian term by giving a boolean when adding the particle. A particle may also be removed from a model. In that case, the particle and all the Lagrangian terms containing it are removed from the theory. These procedures is summarized in sample code 20.

Sample code 20: Adding / Removing particles

Creating particles

```
Particle psi = diracfermion_s("psi", model);
// Setting psi representation ...
Particle phi = scalarboson_s("phi", model);
// Setting phi representation ...
```

Adding the particles

```
model.addParticle(psi); // Adds psi with default interactions
model.addParticle(phi, false); // Adds phi without any interaction
```

Removing a particle

```
model.removeParticle(phi); // phi and all its interactions are removed
```

3.2.2 Managing couplings

It may be important to have access to the couplings present in the model, in case for example one wants to extend the model using them. As shown in table 3.1 there is two kinds of couplings in a MARTY model:

- **Scalar couplings.** Gauge couplings are automatically added in this category when created.
- **Tensor couplings.** Initially empty, may contain any tensor.

Users may get expressions for scalar couplings (Expr objects, typically constants) and tensors for tensor couplings (Tensor objects) from a model, given the name of the coupling. Initially the model only contains gauge couplings, but one can add couplings (scalars or tensors) to the model, and retrieve them later on. As we saw in figure 2.3, gauge couplings are defined initially with a name "g_<group-name>". This procedure is summarized in sample code 21.

Sample code 21: Managing couplings

Getting gauge couplings

```
// Getting the couplings of two gauge groups "Y" and "L"
Expr g_Y = model.getScalarCoupling("g_Y");
Expr g_L = model.getScalarCoupling("g_L");
```

Adding couplings

```
Expr e = constant_s("e");
model.addScalarCoupling(e);
// Defining our own gamma matrix
Tensor my_gamma("gamma", {&Minkowski, &dirac4, &dirac4});
model.addTensorCoupling(my_gamma);
```

Getting tensor couplings

```
Tensor my_gamma_2 = model.getTensorCoupling("gamma");
```

Note In all these example, the first one is probably the most important because there is no other way to get gauge couplings.

See also Section 1.2.4 for more details on tensors, or the CSL manual.

3.2.3 Lagrangian

The Lagrangian is an object almost entirely encapsulated by the `ModelData` class. In other words, a user will probably not directly interact with it, but through the interface functions of the model. Section 3.2.4 shows for example how to add interaction terms to the Lagrangian, using methods of the `ModelData` class. In general, the Lagrangian is just a container of symbolic expressions, and all modifications to it are ordered by Model classes.

Lagrangian meaning in `MARTY`

The Lagrangian contains all kinetic, mass, and interaction terms of the theory. An InteractionTerm in `MARTY` may contain any of the three kinds of terms. The Lagrangian is then just a collection of such objects, divided in three parts:

- **Kinetic terms.** Basically purely informative. No physics in `MARTY` depend on them. They may be however an interesting way to check that model building prescriptions, that `MARTY` also applies to them, are correct.

- **Mass terms.** For built-in models, they do not have any impact on calculations either. However during model building, `MARTY` will look in that part of the Lagrangian to determine masses for particles and matrices to diagonalize. See chapter 5 for more details.

- **Interaction terms.** These terms of course determine the physics, vertices used in Feynman diagrams. They are used once to calculate Feynman rules (see section 6.2).

Amplitude calculations depend on two main features for a given model. Propagators and vertices. As we saw in section 2.2.2, the mass and width of a particle to insert in the propagators are taken from the particle itself, not the model. Changing a mass explicitly as shown in sample code 12 will not change the mass Lagrangian. Mass terms are a wonderful aid for model building (see chapter 5) but will not prevent a user to set masses and widths as wanted for any particle before launching a calculation.

Kinetic terms could in principle affect physics because they determine free equations of motions, the propagators. A scalar Lagrangian like

$$\mathcal{L} = \frac{1}{2}\partial_\mu\phi\partial^\mu\phi - \frac{1}{2}m^2\phi^2 \tag{3.1}$$

implies the following equation of motion for ϕ

$$(\Box + m^2)\phi = 0, \tag{3.2}$$

which in turns gives a propagator of the type

$$\frac{1}{\Box + m^2} \approx \frac{1}{-p^2 + m^2}. \tag{3.3}$$

Kinetic terms could be used to determine the propagators in `MARTY`, but would require unnecessary algebra. Instead all propagators are fixed with a denominator as in equation 3.3, letting the possibility to change propagators explicitly. This feature actually is available in `MARTY` (see documentation of QuantumFieldParent) but has not been tested and is then not detailed in this first manual[1].

Vertices, Feynman rules, are calculated from the interaction Lagrangian. They are computed once (see section 6.2), and after this point the Lagrangian becomes basically just a beautiful, but useless, mathematical expression.

Interaction terms

The InteractionTerm object is used for the three kinds of terms in the Lagrangian. It handles a mathematical expression corresponding to a term in a Lagrangian like

$$ig\bar{\psi}_{i\alpha}\gamma^\mu_{\alpha\beta}W^A_\mu T^A_{ij}\psi_{j\beta}. \tag{3.4}$$

It allows to have a more specialized representation of a term knowing at any moment the fields that are inside. For model building it is important, as `MARTY` must be able to know quickly which particles are in a given term. This may be used to know the type of term (kinetic, mass, interaction), what terms must be modified when replacing particles, and what terms have the same content and must be merged together.

Interaction terms also keep track of all different index contractions in the interaction. When calculating Feynman rules, the Lagrangian expansion is done explicitly using interaction terms. The Wick theorem and most of the algebra is done with a generic term, with only the field content, and the interaction term will apply all factors and index symmetries on the final result. Taking the example of equation 3.4, the calculation of the Feynman rule would inject in the Wick theorem a generic term like

$$\bar{\psi}_{i\alpha}W^A_\mu\psi_{j\beta} \tag{3.5}$$

with only free indices, and the `InteractionTerm` object is asked in the end to recover, in the final result, the initial index structure and factors of the Lagrangian term.

[1] Do not hesitate to contact us if you want to test this feature. Changing a propagator structure could prevent to perform one-loop calculations, but tree-level may be enough in such exotic theories (Lorentz violation theories with modified propagators for example could be implemented with this principle).

3.2.4 Adding Lagrangian terms

There is three ways to add a Lagrangian term in MARTY. The first is automated, just by adding a new particle in the model. In that case MARTY initializes automatically gauge interactions. The second way is to use built-in functions to add common terms, like mass terms. Finally, as section 2.3.3 presented, it is possible to build general interaction terms building the corresponding mathematical expression ourselves.

Built-in interaction terms

There is for now only three types of interaction terms in MARTY one may easily add to a model. Bosonic mass terms

$$\mathcal{L} \ni -s\eta \cdot m^2 \phi^\dagger \phi, \tag{3.6}$$

Dirac or Majorana mass terms

$$\mathcal{L} \ni -\eta \cdot m \bar{\psi}\psi, \tag{3.7}$$

and Weyl mass terms

$$\mathcal{L} \ni -m \left(\bar{\psi}_R \psi_L + \bar{\psi}_L \psi_R\right). \tag{3.8}$$

Each time η is 1 for complex fields, and $\frac{1}{2}$ for real ones, $s = +1$ for scalars and $s = -1$ for vectors. In the Dirac mass terms $\eta = \frac{1}{2}$ corresponds to a Majorana mass. The procedure to add such mass terms is presented in sample code 22. The η and s factors are determined automatically by MARTY.

Sample code 22: Adding mass terms explicitly

For a boson B (scalar or vector)

```
Expr M = constant_s("M");
model.addBosonicMass("B", M);
// Or
model.addBosonicMass("B", "M");
```

For a Dirac or Majorana fermion F

```
Expr m = constant_s("m");
model.addFermionicMass("F", "m");
// Or
model.addFermionicMass("F", "m");
```

For a pair of Weyl fermions (L + R) F_L and F_R

```
Expr m = constant_s("m");
model.addFermionicMass("F_L", "F_R", m);
// Or
model.addFermionicMass("F_L", "F_R", "m");
```

Note When adding a particle to a model, a mass term will be defined automatically if it has a non zero mass. This procedure is useful when one wants to add a mass explicitly during model building for example.

General interactions

To add general interactions, one must give explicitly the expressions to MARTY. It is more involved of course, but completely general and will allow users to build any Lagrangian

they want. The procedure to build these expressions was introduced in section 2.3.3 with an example. Recall that in MARTY the complex conjugate of a fermion, $\psi^\dagger$, is defined directly as $\bar{\psi} = \psi^\dagger\gamma^0$ to avoid dealing with too many unnecessary γ^0 matrices in vertex definitions or calculations. Let us review here the main ingredients for building interaction terms from scratch.

- **Particles.** One must have the `Particle` objects. Either the user-defined ones, or those defined by MARTY that one may get calling the `getParticle()` method. See sample code 10.

- **Indices.** Indices are necessary to call `Particle` objects and get symbolic expressions `Expr`. Sample code 15 shows how to get all the necessary indices and sample code 17 how to use them.

- **Space-time point.** This ingredient is actually not necessary for interaction terms. Users may omit it, MARTY will automatically place all fields to the space-time point defined for the rest of the Lagrangian. Sample code 16 gives more details if one wants to give points anyway.

- **Gauge couplings.** One may have to define new interactions depending on the gauge couplings of the model. The procedure to get gauge couplings has been developed in sample code 21.

- **$\gamma-$matrices.** They are built-in, and may be accessed simply as shown in sample code 23.

- **Generators.** One may have to define new interactions depending one the group generators (T^A_{ij}, f^{ABC} etc) of the model. All generators can be gathered through interface function calls as depicted in sample code 24.

- **Vector spaces.** If one wants to create a new custom tensor, one has to have the vector spaces (`Space`) corresponding to all the tensor indices. This is presented in sample code 25.

- **CSL.** To create new couplings, tensors, one may have to know how to use CSL. The CSL manual will represent an important help, and allow any user to write general interactions. The user in a hurry may want to start with section 1.2.

With these ingredients users should be able to write any unreasonably complicated Lagrangians, starting from the example in sample code 17.

Sample code 23: Getting $\gamma-$matrices

The easy way, using the interface

```
Tensor gamma = DiracGamma(); // gamma matrix
Tensor gamma5 = DiracGamma5(); // gamma5 matrix
Tensor sigma = DiracSigma(); // sigma matrix
Tensor P_L = DiracPL(); // left projector
Tensor P_R = DiracPR(); // right projector
Tensor C = DiracCMatrix(); // Conjugation matrix
```

The not much harder way, using the `dirac4` vector space

```
Tensor gamma = dirac4.gamma; // gamma matrix
Tensor gamma5 = dirac4.gamma_chir; // gamma5 matrix
Tensor sigma = dirac4.sigma; // sigma matrix
Tensor P_L = dirac4.P_L; // left projector
Tensor P_R = dirac4.P_R; // right projector
Tensor C = dirac4.C_matrix; // Conjugation matrix
```

Sample code 24: Getting group generators

A generator is defined for each irreducible representation for each different gauged group. Structure constants f_{ABC} are the generators of adjoint representations, in particular the one of gauge bosons.

Taking the Standard Model example, one has a `"Q_L"` particle in the doublet representation of a $SU(2)$ `"L"` and a gluon `"G"` in the adjoint representation of a $SU(3)$ `"C"`. One may get generators for those representations.

Getting the generators from the model

```
Tensor T_SU2_2 = model.getGenerator("L", "Q_L");
Tensor f_SU3 = model.getGenerator("C", "G");
```

Getting the generators from the interface

```
Tensor T_SU2_2 = GetGenerator(model, "L", "Q_L");
Tensor f_SU3 = GetGenerator(model, "C", "G");
```

Note This procedures are fully general and allow to get any generator in any model.

Sample code 25: Vector spaces

Taking an example with a `"C"` gauged group and an `"F"` flavor group.

From the interface

```
auto gaugeVectorSpace = GetVectorSpace(model, "C", "phi");
auto flavorVectorSpace = GetVectorSpace(model, "F", "phi");
```

From the model

```
auto gaugeVectorSpace = model.getVectorSpace("C", "phi");
auto flavorVectorSpace = model.getVectorSpace("F", "phi");
```

Creating a new tensor with these vector spaces

```
Tensor A("A", {&Minkowski, gaugeVectorSpace, flavorVectorSpace});
// A has one index in Minkowski, one gauge and one flavor index
```

Note The `auto` here deduces the type `Space const*` that is a pointer to a constant `CSL` vector space.

Note Minkowski is a built-in `Space` object so the `&` symbol must be used to get a pointer whereas the spaces returned here are already pointers (no need for `&`).

3.2.5 Fermion number violating interactions

The issue

This section is about fermion number violating interactions, coming from self-conjugate fermions or not. Those vertices require a particular care from `MARTY`'s side, but also form the user which is why a section is dedicated to them. Figure 3.2 presents examples of interactions that may violate fermion number.

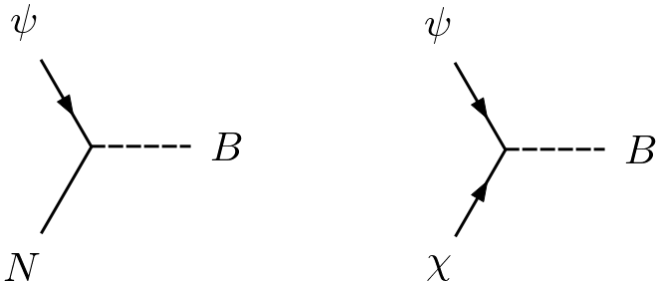


Figure 3.2: Examples of interactions that can lead to fermion number violating processes. ψ is a regular spin $1/2$, N a Majorana and B a boson (scalar or vector).

The issue with such vertices comes in processes such as those depicted in figure 3.3. A naive use of Feynman rules leads to unpleasant fermion bilinears because the fermion number is not conserved along the line, when we always want to have expressions such as

$$\bar{\psi}\Gamma\chi, \tag{3.9}$$

with Γ a combination of gamma matrices with indices flowing from left to right. The reason bilinears must be ordered is not only to have nice expressions, but mostly to be able to simplify them further using well-known tricks. To solve this issue, we follow prescriptions of [18], that defines a set of rules to recover proper fermion bilinears.

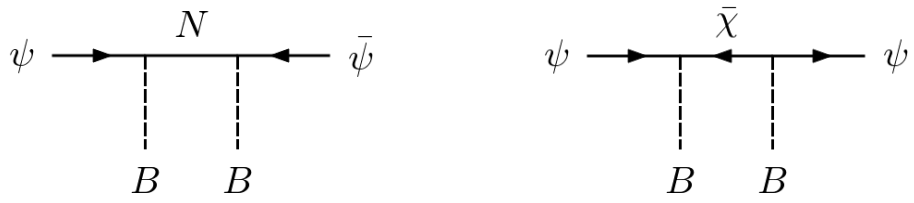


Figure 3.3: Examples of fermion number violating (locally at least) processes. ψ is a regular spin 1/2, N a Majorana and B a boson (scalar or vector).

The conjugation matrix

The conjugation matrix C depends on the $\gamma-$matrix realization. In the Dirac realization a fermion is expressed as $\begin{pmatrix} \psi_L \\ \psi_R \end{pmatrix}$. In this basis, one has

$$C = -i\gamma^0\gamma^2. \tag{3.10}$$

In particular, $C = C^* = -C^T = -C^\dagger = -C^{-1}$. In this basis, a charge conjugated fermion is simply defined by

$$\psi^c \equiv C\bar{\psi}^T. \tag{3.11}$$

A Majorana fermion N is its own charge conjugated particle and the previous equation reads

$$N^c = C\bar{N}^T \equiv N. \tag{3.12}$$

Regarding fermionic external legs, the Conjugation matrix has nice contraction properties with fermion external states $u(p)$ (and $v(p)$ for anti-particles). As C is the charge conjugation matrix, it is defined as a link between particles (u) and anti-particles (v). In particular one has

$$\bar{u} = Cv, \tag{3.13}$$
$$\bar{v} = Cu, \tag{3.14}$$

and equivalently

$$u = \bar{v}C, \tag{3.15}$$
$$v = \bar{u}C. \tag{3.16}$$

C also fulfills properties with gamma matrices transposed namely

$$C\Gamma_i^T C^\dagger \equiv \Gamma'_i = \eta_i\Gamma_i, \tag{3.17}$$

with

$$\eta_i = \begin{cases} +1 & \text{for } \Gamma_i = 1, i\gamma^5, \gamma^\mu\gamma^5 \\ -1 & \text{for } \Gamma_i = \gamma^\mu, \sigma^{\mu\nu}. \end{cases} \tag{3.18}$$

Rules given in [18] consist in defining a fermion line, and inserting conjugation matrices C to recover a nice fermion bilinear such as the one in equation 3.9, while being mathematically equivalent. By ordering the fermion lines, the only subtlety is to obtain the right sign at the end for the diagram in order to keep consistent interference patterns. Users should then be careful about the signs while defining such interactions. This is explained in the next section.

Fermion number violation in MARTY

There is two golden rules in the definition of fermion number violating interactions in MARTY. The first rule is: **Do not use the charge conjugation matrix in vertices with Majorana fermions.** This is because from equation 3.12 we know that a conjugation matrix can be simplified away contracting it with a Majorana fermion. It is possible with MARTY to use charge conjugation matrices with Majorana interactions, but if one has to do it probably means that the vertex expression should be reconsidered, checking that no mistake has been made. Let's see an example of charge conjugation in a vertex with a Majorana N, a Dirac fermion ψ and a vector A

$$\mathcal{L} \ni \lambda A_\mu N \gamma^\mu L^c = \lambda A_\mu N \gamma^\mu C \bar{L}^T. \tag{3.19}$$

Using equations 3.12 and 3.17, one can transform the vertex into

$$\begin{aligned} \mathcal{L} \ni & \lambda A_\mu \bar{N}^T (\gamma^\mu)^T \bar{L}^T \\ = & \lambda A_\mu \left(\bar{N}^T (\gamma^\mu)^T \bar{L}^T \right)^T \\ = & -\lambda A_\mu \bar{L} \gamma^\mu \bar{N} \,, \end{aligned} \tag{3.20}$$

which does not contain any conjugation matrix. Vertices with two Majorana fermions should also follow this rule.

The second rule is: **For fermion number violating interactions between non-Majorana fermions, make sure that the position and sign of C is correct.** One should most of all make sure that conjugation matrices are placed correctly. In particular, in case a fermion number violating vertex or process vanishes incorrectly, conjugation matrices introduced in the Lagrangian should be checked first, in particular considering that

$$C\gamma^\mu \neq \gamma^\mu C, \tag{3.21}$$

$$C_{\alpha\beta} = -C_{\beta\alpha}. \tag{3.22}$$

The way to obtain the symbolic tensor corresponding to the conjugation matrix C in MARTY was presented in sample code 23.

3.2.6 Group theory objects

The ModelData class also handles the gauge of theory. In general, one may obtain from a model the Gauge, Flavor, all GaugedGroups and FlavorGroups, and also the particle representations for the whole gauge (GaugeIrrep), flavor (FlavorIrrep), or a specific group (Irrep).

Manipulating these objects is a much deeper feature of MARTY and should not be needed in a standard use. They represent however an important content of a high energy physics model, and we demonstrate in the following how to access them. Readers will dive into the depths of group theory in MARTY, starting from a high energy physics model, up to deep dark places where abstract group theory implementations are. This will represent the transition to chapter 4. Figure 3.4 presents a summary of the different interfaces presented in this section, and in particular how to get form a high-energy physics model to abstract group theory implementations in MARTY.

Except for representation objects, all groups (flavor or gauged) must be used as **pointers**. This is because their are unique in the program, and must then not be copied.

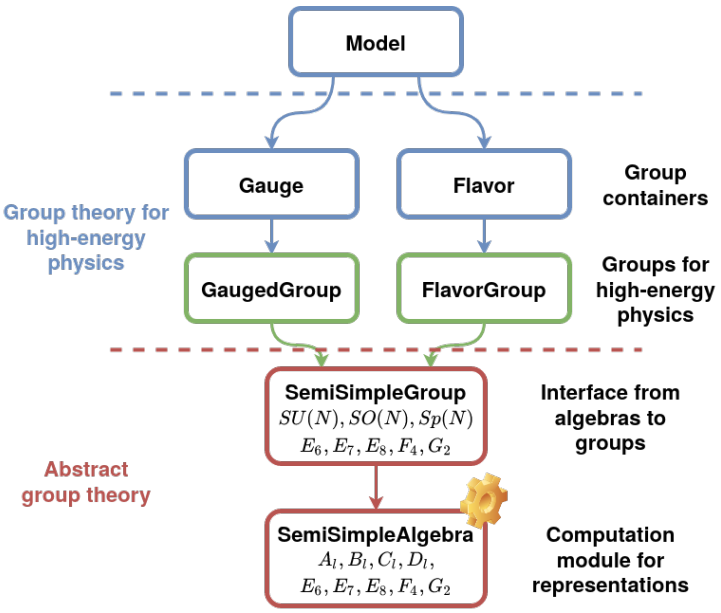


Figure 3.4: Sketch of the successive logical links from a high-energy physics model to abstract group theory implementations in `MARTY`. Representations are calculated by `SemiSimpleAlgebra`, and several interfaces exist to go from abstract representations to gauge and flavor groups, and finally end in the model.

Gauge and Flavor

The `Gauge` contains all the gauge groups, and the `Flavor` all the flavor groups. They may be accessed as shown in sample code 26.

Sample code 26: Gauge and Flavor

Getting the gauge and flavor of a model

```
Gauge *gauge = model.getGauge();
Flavor *flavor = model.getFlavor();
```

Warning Users must always keep pointers to these objects, as they **must not** be copied.

See also Documentation of Gauge and Flavor for more information.

Gauged and Flavor groups

A gauge contains several gauged groups, and a flavor several flavor groups. From their name, one can get these groups from a model, as depicted in sample code 27.

Sample code 27: Gauge and Flavor groups

Considering a model with a $SU(3)$ `"C"` gauged group, and a $SU(3)$ `"F"` flavor.

Getting the gauge and flavor groups from the model

```
GaugedGroup *ggroup = model.getGaugedGroup("C");
FlavorGroup *fgroup = model.getFlavorGroup("F");
```

Warning Users must always keep pointers to these objects, as they **must not** be copied.

See also Documentation of GaugedGroup and FlavorGroup for more information.

Gauge representations

Obtaining the gauge representation of a particle, or in a specific group, are tasks that may be done easily in `MARTY`. One has to give the particle (or its name), and the group (or its name) and the model will return the corresponding representation. More details on representations will be given in chapter 4. Example on how to get particle representations are given in sample code 28.

Sample code 28: Getting representations from particles

Suppose a particle `phi` in a model containing a $SU(3)$ `"C"` gauged group, and a $SU(3)$ `"F"` flavor.

Getting the full gauge and flavor representations

```
GaugeIrrep gaugeRep = model.getGaugeIrrep("phi");
FlavorIrrep flavorRep = model.getFlavorIrrep("phi");
```

Getting a specific group representation (gauged or flavor)

```
Irrep ggroupRep = model.getGroupIrrep("phi", "C");
Irrep fgroupRep = model.getFlavorIrrep("phi", "F");
```

See also Documentation of GaugeIrrep, FlavorIrrep and Irrep for more information.

See also Chapter 4 that will present representations more in details.

Groups and algebras

`GaugedGroup` and `FlavorGroup` objects are an additional abstraction layer with respect to groups from a pure group theory point of view. They contain in particular quantum field theory considerations, that are not needed to define abstract groups. There exist in `MARTY` deeper data structures for groups and algebras. Groups that `MARTY` can define are semi-simple, coming with their semi-simple algebras. SemiSimpleAlgebra implements the deep representation machinery of `MARTY`, and SemiSimpleGroup is an interface to get from algebra to groups we know better ($SU(N)$, $SO(N)$, $Sp(N)$) as figure 3.4 shows. Chapter 4 presents in much more details these objects, but let's introduce here how to get to them from the gauged and flavor groups of the model. This is presented in sample code 29.

Sample code 29: Abstract groups and algebras

Considering a model with a $SU(3)$ `"C"` gauged group, and a $SU(3)$ `"F"` flavor.

Getting gauged and flavor groups

```
GaugedGroup *ggroup = model.getGroup("C");
FlavorGroup *fgroup = model.getFlavor("F");
```

Getting abstract groups from gauged and flavor groups

```
SemiSimpleGroup *group1 = ggroup->getGroup();
SemiSimpleGroup *group2 = fgroup->getGroup();
```

Getting algebras from the groups

```
SemiSimpleAlgebra *algebra1 = group1->getAlgebra();
SemiSimpleAlgebra *algebra2 = group2->getAlgebra();
```

Warning Users must always keep pointers to these objects, as they **must not** be copied.

See also Documentation of file group.h, SemiSimpleGroup and SemiSimpleAlgebra for more information.

See also Chapter 4 for details about deep group theory implementations in `MARTY`.

Chapter 4

Group theory

This chapter is a bit aside from the rest of the manual, as it presents a deeper feature of `MARTY`, that goes beyond the scope of amplitude calculation in quantum field theory. A user only interested in BSM phenomenology may then skip this chapter.

As we saw in section 2.2.3, a quantum field is an irreducible representation of the gauge group. We introduced in particular the link between semi-simple algebras and groups in table 2.2, while table 2.3 presented the definition of the main representations used in physics in terms of Dynkin labels. We will in this chapter talk about how irreducible representations are defined, computed in `MARTY`, and what kind of algebra one may do with them.

Section 4.1 will go in details on what a semi-simple algebra is in `MARTY`, while sections 4.2 and 4.3 will respectively introduce irreducible representations (irreps) and the decomposition of irrep products into sums of irreps. These calculations are similar to what LieART [19] can do, another Mathematica-based program, does. Finally, section 4.4 will recall the link between these abstract group theory considerations and quantum field theory, and in particular the calculations done by `MARTY` with gauge representations.

The Lorentz group A word on the Lorentz group before diving into more abstract considerations. A particle is also an irreducible representation of the Lorentz group, that corresponds to the spin. The Lorentz group $SO(1,3)$ has the same algebra as $SO(4)$. The algebra of $SO(4)$ is

$$\mathfrak{so}(4) = D_2 \cong A_1 \oplus A_1 = \mathfrak{su}(2) \oplus \mathfrak{su}(2). \tag{4.1}$$

In D_2 or $A_1 \oplus A_1$, one must give two Dynkin labels. The common representation are presented in table 4.1.

Particle	Dynkin labels	Dimension
Scalar	$(0,0)$	1
Left Weyl fermion	$(1,0)$	2
Right Weyl fermion	$(0,1)$	2
Dirac fermion	$(0,1) \oplus (1,0)$	$4 = 2 \oplus 2$
Vector	$(1,1)$	4

Table 4.1: Correspondence between Lorentz representations (spin) and Dynkin labels in the algebra $D_2 \cong A_1 \oplus A_1 = \mathfrak{su}(2) \oplus \mathfrak{su}(2)$.

4.1 Semi-simple Lie algebras

4.1.1 Principle

We will not here go into much theoretical details, as this document still is MARTY's manual. Readers wanting to learn more about semi-simple Lie algebras may see [17, 19, 20].

Semi-simple Lie algebras have a common definition, and can be described with the same formalism. They are defined as having have no non-zero abelian ideal. One must find the maximal Cartan sub-algebra, whose dimension is called the rank ℓ of the algebra. There is seven different types of such algebras.

- A_ℓ for $\ell \geq 1$.
- B_ℓ for $\ell \geq 1$.
- C_ℓ for $\ell \geq 1$.
- D_ℓ for $\ell \geq 2$.
- E_ℓ for $6 \leq \ell \leq 8$.
- F_ℓ for $\ell = 4$.
- G_ℓ for $\ell = 2$.

Algebras E_ℓ to G_ℓ are called exceptional while those from A_ℓ to D_ℓ are the algebras of $SU(N)$, $SO(N)$ and $Sp(N)$ groups. They can however all be described with the same formalism. This is what MARTY does to define irreducible representations in all these algebras.

The Cartan sub-algebra defines the so-called simple roots of the semi-simple Lie algebra. An algebra of rank ℓ has exactly ℓ simple roots, and an irreducible representation is defined from them.

4.1.2 Semi-simple Lie algebras in MARTY

In MARTY, such algebra is called SemiSimpleAlgebra. The way to create semi-simple algebras is presented in sample code 30. Irreducible representations will be presented in the next section.

Sample code 30: Semi-simple algebras

Creating algebras

```
auto A2 = CreateAlgebra(algebra::Type::A, 2);
auto B7 = CreateAlgebra(algebra::Type::B, 7);
auto F4 = CreateAlgebra(algebra::Type::F4);
auto G2 = CreateAlgebra(algebra::Type::G2);
```

Note The type deduced by `auto` is `unique_ptr<SemiSimpleAlgebra>`. Simply recall that it is a pointer, and that one must use `->` to access member functions of SemiSimpleAlgebra.

Note Algebra names being very unspecific, prefix `algebra::Type` is needed to access algebra type names.

4.2 Irreducible representations

4.2.1 Highest-weight state

As we saw in the previous section, irreducible representations in a semi-simple Lie algebra are defined from its ℓ simple roots. These roots define a $\ell-$dimensional discrete lattice, in which states $|\psi_i\rangle$ are living. A state is defined with ℓ integer as

$$|\psi_i\rangle \equiv |i_1, i_2, \dots i_\ell\rangle, \quad i_j \text{ integers..} \tag{4.2}$$

An irreducible representation is defined uniquely by its highest-weight state, and from this highest-weight are deduced all other states of the representation $\mathcal{R}$. Each state $|\psi_i\rangle$ has a multiplicity m_i, and the dimension of the irrep is simply

$$d_\mathcal{R} = \sum_{|\psi_i\rangle \in \mathcal{R}} m_i. \tag{4.3}$$

From the highest weight state, one may get all states with multiplicities using annihilation operators recursively. All states in the representation will be found to finally get them all.

4.2.2 The $\mathfrak{su}(2)$ example

Let us consider the simplest example, the physicist's spin in $SU(2)$. The algebra is A_1. We are in a one dimensional, discrete space. Highest weights must have a positive (unique) Dynkin label, and lower states are found by applying the only annihilation operator[1]

$$J^- = \frac{1}{\sqrt{2}}(\sigma^1 + i\sigma^2). \tag{4.4}$$

From the spin 1/2 state, one will get the spin -1/2 and get two 2-dimensional spin 1/2 representation. From the spin 1 highest weight, one will find spin 0 and -1 states to finally get a 3-dimensional representation. This is represented in figure 4.1. Dynkin labels in $\mathfrak{su}(2)$ are just twice the spin value.

This principle is then generalized in ℓ dimensions, and irreducible representations in any semi-simple Lie algebras may be uniquely defined.

[1] This is also the same operator as the W^- boson in the SM, annihilation operator for the weak isospin: $W^- = \frac{1}{\sqrt{2}}(W^1 + iW^2)$.

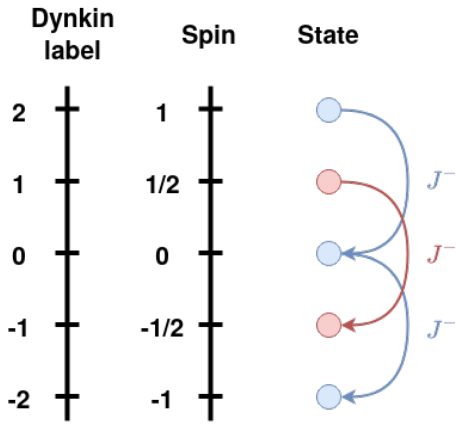


Figure 4.1: $A_1 = \mathfrak{su}(2)$ algebra 1-dimensional space, with spin $1/2$ (red) and spin 1 (blue) representations showed with respectively 2 and 3 states. Correspondence with Dynkin labels is shown. Annihilation operator $J^- = \frac{1}{\sqrt{2}}(\sigma^1 + i\sigma^2)$ action is presented on the different states, starting each time from the highest weight state of the representation.

4.2.3 The $\mathfrak{su}(3)$ example

Let us consider now the most complicated generalization of the previous section, that can still be represented on a sheet of paper, the 2D case. The weight lattice of $A_2 = \mathfrak{su}(3)$ is 2-dimensional and is represented in figure 4.2.

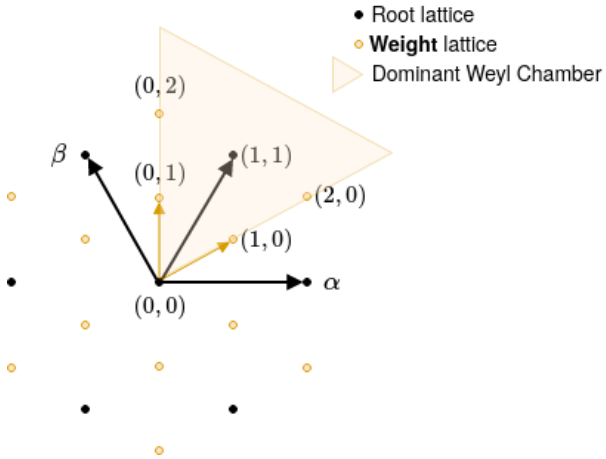


Figure 4.2: Root and weight lattices of $SU(3)$. α and β are the two simple roots of $\mathfrak{su}(3) = A_2$. Dynkin labels of irreducible representations correspond to the position of the highest-weight state on the weight lattice. The dominant Weyl chamber is defined by the set of all positive-weights states. It contains all states that can be highest-weights, i.e. that can define an irreducible representation.

Highest-weights may be any state in the dominant Weyl chamber, i.e. must have positive Dynkin labels. The state $|0, 0, \ldots, 0\rangle$ is always the trivial 1-dimensional representation. A highest weight state in $\mathfrak{su}(3)$ is defined by two Dynkin labels (2D plane), and now there is two different annihilation operators one must apply recursively to the highest weight state to derive all states in an irreducible representation. These two annihilation operators are geometrically along $-\vec{\alpha}$ and $-\vec{\beta}$, with $\vec{\alpha}, \vec{\beta}$ the two simple roots of A_2. Common representations, the quark, anti-quark, and gluon of the strong nuclear force $SU(3)$ symmetry group are presented in figure 4.3.

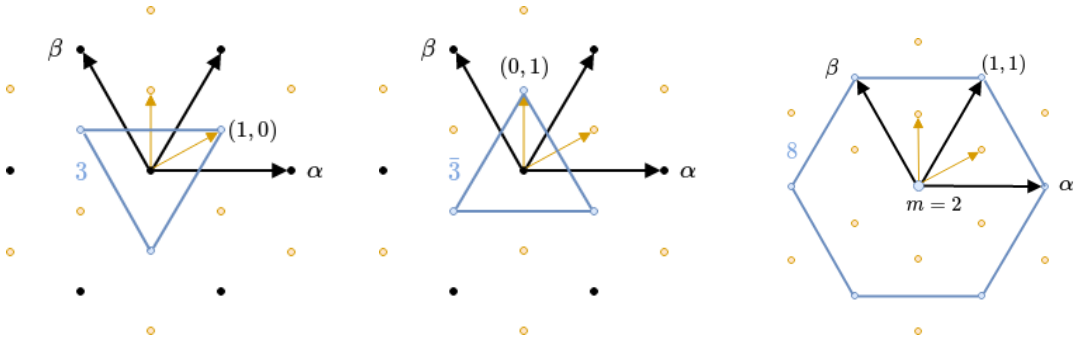


Figure 4.3: Usual QCD representations (quark, anti-quark and gluon respectively) in the weight lattice of $SU(3)$. The coordinates of their highest weight is shown. In the 8-dimensional representation, the state of weight $(0, 0)$ has multiplicity 2. The total number of states is then indeed 8.

4.2.4 Irreducible representations in MARTY

From a semi-simple Lie algebra, one can build any irreducible representation given the Dynkin labels of its highest-weight state, as we said in the previous section. Examples for $SU(2)$ and $SU(3)$ have been given, readers will then have to search in the literature the definition of more exotic representations if needed. The procedure to make MARTY derive an irreducible representation is presented in sample code 31. All the procedure, applying annihilation operators from the highest weight to find all the states, deriving the multiplicities, are calculations done automatically by MARTY, in all semi-simple Lie algebras.

Sample code 31: Irreducible representations

Taking algebras defined in sample code 30.

From the interface

```
Irrep quark = GetIrrep(A2, {1, 0});
Irrep gluon = GetIrrep(A2, {1, 1});
Irrep exotic = GetIrrep(F4, {1, 1, 0, 0});
```

Through member functions

```
Irrep quark = A2->highestWeightRep({1, 0});
Irrep gluon = A2->highestWeightRep({1, 1});
Irrep exotic = F4->highestWeightRep({1, 1, 0, 0});
```

Getting dimensions

```
cout << quark.getDimension() << endl; // 3
cout << gluon.getDimension() << endl; // 8
cout << exotic.getDimension() << endl; // 29172
```

See also Documentation of Irrep for more information on member functions.

4.3 Product decomposition

Product of irreducible representations have an important meaning in particle physics. Each interaction vertex is in fact such a product, with the representations of all the interacting particles. The product can be decomposed in a sum of irreducible representations, the total number of dimensions being conserved. For an interaction to be mathematically valid, the trivial representation must appear in the decomposition. With this procedure one can know for example what type of representation can get out from the annihilation of two particles, or what quark arrangements may result in a color-blind structure (mesons $q\bar{q}$ or baryons qqq). The procedure to decompose a product into a direct sum of irreducible representations can be found in [19][2]. Let us give examples. First, a well-known $SU(2)$ spin

$$2 \otimes 2 = 1 \oplus 3, \tag{4.5}$$

with integers representing the dimensions of the representations. Group theory tells us that combining two $1/2$ spins, one may either get a scalar or a spin 1, not any other spin. If a reader is now wondering what we can get from the collision of two gluons[3], the answer is

$$8 \otimes 8 = 1 \oplus 8 \oplus 8 \oplus 10 \oplus 10 \oplus 27. \tag{4.6}$$

We conclude that we may get a neutral particle, another gluon-type particle or a more exotic representation 10- or 27-dimensional. May be an hint for New Physics the search at the LHC !

With a simple interface, one can make `MARTY` decompose such products. The result is not an `Irrep` but a `SumIrrep`. The interface allows in general to sum, multiply and display any of those objects in any semi-simple Lie algebra. Sample code 32 shows two examples. One may see the link between the $SU(3)$ example and physics. It is the answer to the question *'What quarks arrangements can be color neutral and explain the structure of neutrons and protons ?'*. A neutral state is a trivial $SU(3)$ representation, i.e. the dimension 1. When taking product of quarks, one knows if it can be in a neutral state if the trivial representation appears in the decomposition. The example confirms what we already know, $q\bar{q}$ and qqq are the only states with 2 or 3 quarks that can be color neutral.[4]

[2] Or in `MARTY`'s code, for the brave.

[3] Purely in terms of $SU(3)$ color structure.

[4] Arbitrary combinations of those two building blocks will of course also give neutral states, as tetra-quarks $qq\bar{q}\bar{q}$ and penta-quarks $qqqq\bar{q}$ for example.

Sample code 32: Representation product decomposition

Taking the sample algebras of sample code 30.

In the `A2` algebra

```
Irrep quark = GetIrrep(A2, {1, 0});
Irrep antiquark = GetIrrep(A2, {0, 1});

cout << "3␣x␣3␣=␣" << quark * quark << endl;
// >> 3 x 3 = 3 + 6 (Total dim = 9)
cout << "3␣x␣3c␣=␣" << quark * antiquark << endl;
// >> 3 x 3c = 1 + 8 (Total dim = 9)
cout << "3␣x␣3␣x␣3␣=␣" << quark * quark * quark << endl;
// >> 3 x 3 x 3 = 1 + 8 + 8 + 10 (Total dim = 27)
cout << "3␣x␣3␣x␣3c␣=␣" << quark * quark * antiquark << endl;
// >> 3 x 3 x 3c = 3 + 3 + 6 + 15 (Total dim = 27)
```

In the `F4` algebra

```
Irrep exotic1 = GetIrrep(F4, {1, 0, 0, 0});
Irrep exotic2 = GetIrrep(F4, {0, 0, 0, 1});
cout << exotic1 << endl;
// >> Representation |1,0,0,0> of dimension 52
cout << exotic2 << endl;
// >> Representation |0,0,0,1> of dimension 26
SumIrrep decomposition = exotic1 * exotic2;
cout << "52␣x␣26␣=␣" << decomposition << endl;
// >> 52 x 26 = 26 + 273 + 1053 (Total dim = 1352)
```

See also Documentation of Irrep and SumIrrep.

4.4 Gauge representations

A gauge representation is simply a collection of group representations. The principle is the same as for `Irrep` and `SumIrrep`, but this time one manipulates representations in different groups at the same time in objects GaugeIrrep and SumGaugeIrrep. Taking a sample $SU(3)_C \otimes SU(2)_L$ SM gauge for example, one can take the product of a left-handed quark with a left-handed anti-quark

$$(3,2) \otimes (\bar{3},\bar{2}) = (1 \oplus 8, 1 \oplus 3) = (1,1) \oplus (8,1) \oplus (1,3) \oplus (8,3). \tag{4.7}$$

For gauge representations, one may directly use a high-energy physics model with its interface like we showed in 3.2.6. Sample code 33 presents how to perform the example above, building a Gauge from scratch, independently of any quantum field theory consideration, and calculating the decomposition of $q\bar{q}$ in this $SU(3)_C \otimes SU(2)_L$ gauge.

Sample code 33: Gauge irreducible representations

Building a $SU(3) \times SU(2)$ gauge

```
Gauge gauge;
gauge.addGroup(group::Type::SU, "C", 3);
gauge.addGroup(group::Type::SU, "L", 2);
```

Getting quark $(3, 2)$ and anti-quark $(\bar{3}, \bar{2})$ representations

```
// Two representations between {},
// {1, 0} is SU(3) triplet
// {1}    is SU(2) doublet
GaugeIrrep quark = gauge.getRepresentation({{1, 0}, {1}});
GaugeIrrep antiquark = quark.getConjugatedRep();
cout << quark << endl;
// >> ( 3 , 2 )
cout << antiquark << endl;
// >> ( 3 , 2 )
```

Getting the decomposition of $q\bar{q}$

```
SumGaugeIrrep decomposition = quark * antiquark;
cout << decomposition << endl;
// >> ( 1 , 1 ) + ( 8 , 1 ) + ( 1 , 3 ) + ( 8 , 3 )
if (decomposition.containsTrivialRep()) {
    cout << "Contains␣trivial␣rep␣!" << endl;
    // >> Contains trivial rep !
}
```

Note The `containsTrivialRep()` method for gauge representations is actually used in `MARTY` to check that interactions terms are not loudly violating gauge invariance. They still can violate the gauge even if respecting this condition but it is more subtle to test automatically.

See also Documentation of Gauge, GaugeIrrep and SumGaugeIrrep.

4.5 Dynkin labels for common representations

In this section we present the correspondence between Dynkin labels and the most common irreducible representations (irreps) for all semi-simple groups. In `MARTY`, Dynkin labels allow users to define uniquely irreps when creating particles. Each time, those positive integers must be given between `{}`, as for a $SU(3)$ `"C"` group

```
particle->setGroupRep("C", {1, 0});
```

for the fundamental representation of $SU(3)$ of Dynkin labels $(1, 0)$, typically quarks.

Gauge bosons are in the adjoint representation of their gauged groups, and generally non-trivial representations are the fundamental ones (doublet of $SU(2)$, triplet of $SU(3)$ etc), but this section also presents more exotic irreps. Trivial irreps (dimension 1) have always 0 Dynkin labels $(0, \cdots, 0)$ but need not to be defined in `MARTY`. The correspondence between groups and algebras A_ℓ, B_ℓ, C_ℓ and D_ℓ was given in table 2.2.

4.5.1 $\mathfrak{su}(N)$

$\mathfrak{su}(N)$ is an algebra of rank $N-1$ corresponding to A_{N-1}, meaning that an irrep in that group is uniquely defined by $N-1$ positive integers. Starting from $\mathfrak{su}(2)$ with one label, common representations are presented in table 4.2.

Dynkin labels	Dimension	Common name
$(1, 0, \cdots, 0)$	N	Fundamental
$(0, \cdots, 0, 1)$	N	Anti-fundamental
$(1, 0, \cdots, 0, 1)$	N^2-1	Adjoint

Table 4.2: Dynkin labels for common $\mathfrak{su}(N)$ irreducible representations.

The $\mathfrak{su}(2)$ case

For the $\mathfrak{su}(2)$ algebra, there is only one Dynkin label λ corresponding to the spin j through the relation

$$j = \frac{\lambda}{2}. \tag{4.8}$$

One can then straight-forwardly deduce the Dynkin label for a representation of spin j (dimension $2j+1$) by multiplying by 2.

The $\mathfrak{su}(3)$ case

$\mathfrak{su}(3)$ representations are defined with two Dynkin labels. Table 4.3 presents the correspondence with common $\mathfrak{su}(3)$ irreps.

Dynkin labels	Dimension	Common name
$(1, 0)$	3	Triplet
$(0, 1)$	3	Anti-triplet
$(1, 1)$	8	Adjoint
$(2, 0)$	6	Sextet
$(0, 2)$	6	Anti-sextet
$(2, 1)$	10	Decuplet
$(1, 2)$	10	Anti-decuplet

Table 4.3: Dynkin labels for common $\mathfrak{su}(3)$ irreducible representations.

4.5.2 $\mathfrak{so}(N)$

$\mathfrak{so}(2\ell)$ and $\mathfrak{so}(2\ell+1)$ are algebras of rank ℓ namely D_ℓ and B_ℓ respectively. Particular cases must be mentioned for low ℓ to define properly vector, adjoint, and spinor representations of $\mathfrak{so}(N)$.

Group	Algebra	Dynkin labels	Dimension	Common name
$SO(5)$	B_2	$(1,0)$	5	Vector
$SO(5)$	B_2	$(0,2)$	10	Adjoint
$SO(2\ell+1)$	B_l, $\ell \geq 3$	$(1,0,0,\cdots,0)$	$2\ell+1$	Vector
$SO(2\ell+1)$	B_l, $\ell \geq 3$	$(0,1,0,\cdots,0)$	$\ell(2\ell+1)$	Adjoint
$SO(4)$	D_2	$(1,1)$	4	Vector
$SO(4)$	D_2	$(2,1)$	6	Adjoint
$SO(6)$	D_3	$(1,0,0)$	6	Vector
$SO(6)$	D_3	$(0,1,1)$	15	Adjoint
$SO(2\ell)$	D_ℓ, $\ell \geq 4$	$(1,0,0,\cdots,0)$	2ℓ	Vector
$SO(2\ell)$	D_ℓ, $\ell \geq 4$	$(0,1,0,\cdots,0)$	$\ell(2\ell-1)$	Adjoint
$SO(2\ell)$	D_ℓ	$(0,\cdots,0,1,0)$	$2^{\ell-1}$	Left spinor
$SO(2\ell)$	D_ℓ	$(0,\cdots,0,0,1)$	$2^{\ell-1}$	Right spinor

Table 4.4: Dynkin labels for the simplest $\mathfrak{so}(N)$ irreducible representations. Dimensions have been calculated with `MARTY` as demonstrated in section 4.2.

4.5.3 $\mathfrak{sp}(N)$

$\mathfrak{sp}(2\ell)$ is an algebra of rank ℓ namely C_ℓ. Dynkin labels for the fundamental and adjoint representations of $Sp(N)$ groups are presented in table 4.5.

Group	Algebra	Dynkin labels	Dimension	Common name
$Sp(2\ell)$	C_ℓ	$(1,0,\cdots,0)$	2ℓ	Fundamental
$Sp(2\ell)$	C_ℓ	$(2,0,\cdots,0)$	$\ell(2\ell+1)$	Adjoint

Table 4.5: Dynkin labels for the simplest $\mathfrak{sp}(2N)$ irreducible representations. Dimensions have been calculated with `MARTY` as demonstrated in section 4.2.

4.5.4 E_6

E_6 is an exceptional algebra of rank 6. Dynkin labels for the simplest E_6 representations are presented in table 4.6.

Dynkin labels	Dimension
$(1,0,0,0,0,0)$	27
$(0,1,0,0,0,0)$	351
$(0,0,1,0,0,0)$	2925
$(0,0,0,1,0,0)$	351
$(0,0,0,0,1,0)$	27
$(0,0,0,0,0,1)$	78
$(1,1,0,0,0,0)$	5824
$(1,0,1,0,0,0)$	51975
$(1,0,0,1,0,0)$	7371
$(1,0,0,0,1,0)$	650
$(1,0,0,0,0,1)$	1728

Table 4.6: Dynkin labels for the simplest E_6 irreducible representations. Dimensions have been calculated with `MARTY` as demonstrated in section 4.2.

4.5.5 E_7

E_7 is an exceptional algebra of rank 7. Dynkin labels for the simplest E_7 representations are presented in table 4.7.

Dynkin labels	Dimension
$(1,0,0,0,0,0,0)$	133
$(0,1,0,0,0,0,0)$	8645
$(0,0,1,0,0,0,0)$	365750
$(0,0,0,1,0,0,0)$	27664
$(0,0,0,0,1,0,0)$	1539
$(0,0,0,0,0,1,0)$	56
$(0,0,0,0,0,0,1)$	912

Table 4.7: Dynkin labels for the simplest E_7 irreducible representations. Dimensions have been calculated with `MARTY` as demonstrated in section 4.2.

4.5.6 E_8

E_8 is an exceptional algebra of rank 8. Dynkin labels for the simplest E_8 representations are presented in table 4.8.

Dynkin labels	Dimension
$(1,0,0,0,0,0,0,0)$	3875
$(0,0,0,0,0,1,0,0)$	30380
$(0,0,0,0,0,0,1,0)$	248

Table 4.8: Dynkin labels for the simplest E_8 irreducible representations. Dimensions have been calculated with `MARTY` as demonstrated in section 4.2.

4.5.7 F_4

F_4 is an exceptional algebra of rank 4. Dynkin labels for the simplest F_4 representations are presented in table 4.9.

Dynkin labels	Dimension
$(1,0,0,0)$	52
$(0,1,0,0)$	1274
$(0,0,1,0)$	273
$(0,0,0,1)$	26
$(1,1,0,0)$	29172
$(1,0,1,0)$	8424
$(1,0,0,1)$	1053
$(0,1,1,0)$	107406
$(0,1,0,1)$	19278
$(0,0,1,1)$	4096

Table 4.9: Dynkin labels for the simplest F_4 irreducible representations. Dimensions have been calculated with `MARTY` as demonstrated in section 4.2.

4.5.8 G_2

G_2 is an exceptional algebra of rank 2. Dynkin labels for the simplest G_2 representations are presented in table 4.10.

Dynkin labels	Dimension
$(1,0)$	7
$(0,1)$	14
$(1,1)$	64
$(2,0)$	27
$(0,2)$	77
$(2,1)$	189
$(1,2)$	286
$(2,2)$	729

Table 4.10: Dynkin labels for the simplest G_2 irreducible representations. Dimensions have been calculated with `MARTY` as demonstrated in section 4.2.

Chapter 5

Model Building

This chapter is about advanced features for model building in MARTY. We saw in chapter 2 how to create quantum fields, and in chapter 3 how to add any Lagrangian term by hand. Users then already have what is needed to create general BSM models. In the following are presented more advanced manipulations one can do on MARTY's models to save time and energy, in particular for those having several thousands of terms. This kind of procedures have been used in MARTY to create the MSSM for example, that has more than 7000 Lagrangian terms for its unconstrained version. Writing the Lagrangian by hand was not an option for us. We then decided to write the high-energy Lagrangian, with a $SU(3)_C \times SU(2)_L \times U(1)_Y$ gauge and the SM flavor (3 fermion generations) fully preserved. This Lagrangian is much more simple to get in the literature, and to implement. It uses for example the very compact notation Q_L that contains the three generations of left-handed up and down quarks $u_L, d_L, c_L, s_L, t_L, b_L$. Then, all breaking steps to get to the final low energy Lagrangian have been automated to get to the final low-energy effective Lagrangian, and are then available for any BSM model. These features are presented in the following.

5.1 Recipe

The recipe to build a model in MARTY is always the same. In this section we present a 'fill in the blanks' sample for model building. This is presented in sample code 34. The first step is to create the gauge and flavor groups, as detailed in section 5.2. Then, section 5.3 explains how particle content must be given by the user. Sections 5.5.1, 5.5.2 and 5.5.3 bring up the topic of advanced model building features. Finally, section 5.5.4 will tell users what they have to do and can do at the end of the model building process, before getting into actual calculations that will be presented in chapter 6.

Sample code 34: Model building recipe

Creating the model

```
Model model;
```

Setting the gauge and flavor groups

```
model.addGaugedGroup(args...);
model.addGaugedGroup(args...);
model.addFlavorGroup(args...);
model.init(); // Important to call this function once finished !
```

Adding particle content

```
Particle p1 = builderfunction_s(args...);
p1->setGroupRep(args...);
p1->setFlavorRep(args...);
model.addParticle(p1); // Do not forget to add the particles !
// ...
```

Using advanced model building features

```
// Here really starts the model building game !
model.renameParticle(args...);
model.replace(args...);
model.rotateFields(args...);
model.diagonalizeMassMatrices(args...);
```

Refreshing the model once finished

```
model.refresh(); // Once everything is done, let MARTY do some cleaning
```

Note The refreshing at the end is important. It will allow `MARTY` to find some nice simplifications and merge all the terms that must been merged (same content).

Model building is done by iteration, testing the different steps one after the other. When building a BSM model, we encourage users to often display the state of the model, checking that everything is done correctly by typing

```
cout << model << endl;
```

It allows one to know what `MARTY` is doing, and more importantly what needs to be done to have a meaningful model. For models with many terms, the output may not be contained in a terminal and can be redirected in a file typing for example

```
./myProgram.x > data.txt
```

5.2 Gauge Group

The gauge in a `MARTY` model can be any combination of semi-simple Lie groups. When adding a gauged group to a model, one must give its type, its name, and a dimension when it is relevant. Group types and their respective allowed dimensions are presented in table 5.1. Names must be chosen carefully as many interface functions take the group names as parameter as we have seen in section 2.3.3.

Flavor groups are for now limited to the $N-$dimensional fundamental representations of $SU(N)$ for flavors mixing complex fields, and $SO(N)$ for real ones. One must then sim-

Group	MARTY type	Dimension
$U(1)$	`group::Type::U1`	
$SU(N)$	`group::Type::SU`	$d \geq 2$
$SO(N)$	`group::Type::SO`	$d \geq 2$
$Sp(N)$	`group::Type::Sp`	$d \geq 2$, d even
E_6	`group::Type::E6`	
E_7	`group::Type::E7`	
E_8	`group::Type::E8`	
F_4	`group::Type::F4`	
G_2	`group::Type::G2`	

Table 5.1: Different gauged groups possible in MARTY. The type to give to the `addGaugedGroup()` method is given, as well as the dimension allowed, if there is a dimension to give.

ply give the dimension of the flavor N and a boolean that tells if mixed fields are complex ($SU(N)$) or not ($SO(N)$). Procedure to add gauged and flavor groups is summarized in sample code 35.

Sample code 35: Adding gauged and flavor groups

Taking the Standard Model example: A $SU(3)_C \times SU(2)_L \times U(1)_Y$ gauge, and a $SU(3)_F$ (complex) symmetry flavor group.

Adding gauged groups

```
model.addGaugedGroup(group::Type::SU, "C", 3);
model.addGaugedGroup(group::Type::SU, "L", 2);
model.addGaugedGroup(group::Type::U1, "Y");
```

Adding flavor groups

```
model.addFlavorGroup("F", 3, true); // Complex 3-dimensional flavor
// For a SO(3) real flavor, this command would be
// model.addFlavorGroup("F", 3, false);
```

Recall to initialize the model once finished

```
model.init();
```

Note Again we present the SM example for simplicity but the procedure is fully general and may applied on any groups.

Note As said in section 2.3.3, the definition order of symmetry groups defines also the order of the indices to give to particles that have multiple non-trivial representations. In this case for example, one will have to give color indices before weak isospin ones.

As detailed in chapter 2 gauge bosons and ghosts, coming with their initial kinetic and interaction terms, are created automatically by MARTY. Users then have to create themselves the matter content, as explained in the next section.

5.3 Particle content

Once the gauge is fixed, new particles can be added to the model. Chapter 2 already presented what quantum fields can be defined and how. Let us recall here the basic principle. One first creates a particle, fermionic or bosonic, using a built-in function in `MARTY` (see sample codes 6, 7, 8 and 9 of section 2.1). Then, one has to define the gauge and flavor representations of the particle that are not trivial (see sample codes 13 and 14 of section 2.1). Finally, one may add the particle to the model. As explained in sample code 20 of section 3.2.1, adding a particle will automatically initialize[1]:

- **The kinetic term** of the particle like $\partial_\mu \phi^* \partial^\mu \phi$ for a complex scalar ϕ.
- **The mass term** of the particle if the user defined one through the `setMass()` method of particles (see 2.2.2 for more details).
- **Gauge interactions**, depending on the gauge representations defined by the user **before** adding the particle.

5.4 Completing the Lagrangian

This topic has already been fully explained in section 3.2.4, we will not repeat ourselves here. This is the final step before using the built-in functions in `MARTY` that allow to modify a model, presented in the next section.

5.5 ModelBuilder interface

The class ModelBuilder contains all the necessary interface to modify an existing Lagrangian. The principle is to give the user the possibility to modify all terms in the Lagrangian following a given prescription. Replacement of a mathematical expression, particles, field rotations, mass diagonalization etc can be performed by `MARTY`. The following sections present more in details the different possibilities to this end.

5.5.1 Replacements

There is several kinds of replacements possible in a model. Except simple renaming, replacements are based on the `CSL` machinery, see the CSL manual for more details on how it works under the hood. There is four ways to replace expressions in a model:

- **Renaming particles.** This is the simplest procedure one will see in this section. It is however important to get a model corresponding to standard conventions. It may be done through a single function call, as presented in sample code 36.
- **Replacing scalar expressions.** As in `CSL`, an expression can be replaced with another in the whole Lagrangian, like $a \mapsto b/\cos\theta$.
- **Replacing tensors.** Tensors may also be replaced, but need a particular care to handle indices in the replacement procedure.

[1] Except if specified otherwise by the user.

- **Replacing particles.** Particles may be replaced by any expression. It works like the replacement of tensors but the model needs to know the particle redefinition explicitly. $H^0 \mapsto \frac{h+v}{\sqrt{2}}$ is an example of such replacement.

Sample code 36: Renaming particles

From the `ModelBuilder` interface

```
model.renameParticle("A_L", "W");
```

From the `Particle` object

```
Particle A_L = model.getParticle("A_L");
A_L->setName("W");
```

See also Documentation of Particle and ModelBuilder.

Replacing expressions

Replacing scalar expressions can be done on the full Lagrangian by `MARTY` in a very simple way. One simply has to have all the symbolic expressions needed. For example replacing a coupling g_L of a model by $\frac{e}{\cos\theta_W}$, one needs in the first place the initial coupling g_L of the model. Section 3.2.2 presented in details how to get the built-in gauge couplings of a model.

The example here is taken from the Standard Model. One breaks the $SU(2)_L \times U(1)_Y$ into a $U(1)_{EM}$ symmetry. The two initial gauge couplings g_L and g_Y are identified to a new coupling, the electromagnetic coupling constant e, through the relations

$$g_Y \mapsto \frac{e}{\cos\theta_W}, \tag{5.1}$$

$$g_L \mapsto \frac{e}{\sin\theta_W}. \tag{5.2}$$

This kind of replacements are done easily, as demonstrated in sample code 37.

Sample code 37: Replacing expressions

Taking a $SU(2)_L \times U(1)_Y$ gauge, replacing the two initial coupling constants by a new one e, depending on an angle θ_W.

Getting the initial expressions, and creating the new

```
Expr g_Y = model.getScalarCoupling("g_Y");
Expr g_L = model.getScalarCoupling("g_L");
Expr e = constant_s("e");
Expr theta_W = constant_s("theta_W");
```

Performing the replacements in the Lagrangian

```
model.replace(g_Y, e / cos_s(theta_W));
model.replace(g_L, e / sin_s(theta_W));
```

Note This code is fully general, any kind of scalar replacement can be done in the Lagrangian using the `CSL` machinery.

See also Section 3.2.2 for more details on how to get couplings in a BSM model.

Replacing tensors

Replacing tensor may be more subtle because one has to be careful about indices. Taking a simple example, the replacement of a matrix A_{ij} following

$$A_{ij} \mapsto C_{ij} + B_{ji}, \tag{5.3}$$

with B and C two user-defined matrices. It is not trivial for `MARTY` to know, if indices are not specified for A, which index of A maps to i and which to j. Considering canonicalization of expressions (see the CSL manual), B is simpler than C and is placed first. The replacement for `MARTY` then reads instead

$$A_{ij} \mapsto B_{ji} + C_{ij}. \tag{5.4}$$

You see then that if the indices of the replaced tensor are not given, `MARTY` cannot know in general how to place indices i and j in the final expression. Such an example is presented in sample code 38.

Sample code 38: Replacing tensors

We consider here ϕ and ψ two particles in a given flavor representation with indices i, j, k. We also take for granted all indices and tensors. More details on that in sections 1.2.4 and 2.3.3.

Setting the initial tensor A complex

```
A->setComplexProperty(ComplexProperty::Complex);
```

Building an interaction term $\bar{\psi}_i A_{ijk} \psi_j \phi_k + \text{h.c.}$

```
model.addLagrangianTerm(
        A({i, j, k})
        * GetComplexConjugate(psi({i, alpha}))
        * psi({j, alpha})
        * phi(k),
        true // Adding the hermitian conjugate as well
);
```

Replacing $A_{ijk} \mapsto bB_{ijk} - icC_{jik}$, B and C real tensors

```
model.replace(A({i, j, k}), b*B({i, j, k}) - CSL_I*c*C({j, i, k}));
// The result looks like
// (b*B_{E,F,G} + -i*c*C_{F,E,G})*phi_{G}*psi_{E,a}^(*)*psi_{F,a} + h.c
```

Note The index α for the fermion ψ is a Dirac index.

See also Section 3.2.4 for more details on how to build interaction terms.

See also The CSL manual for more details on tensors and replacement.

Replacing particles

Replacing particles uses the exact same interface as other replacements presented above. The commands are then the same, but users still should know that:

- If the initial expression is a particle, it will be removed from the model, except if it appears also in the final state.
- If the final state contains unknown particles, they will be added to the model, without adding kinetic, mass or interaction terms automatically.

This is presented in sample code 39. For the specific case of vector bosons,

Sample code 39: Replacing particles

Considering a complex scalar field ϕ_i mapped to $\eta_i + i\chi_i$ with χ and η two real scalar fields. i can be an index in any space.

Building the two new fields

```
Particle eta = phi->generateSimilar("eta");
Particle chi = phi->generateSimilar("chi");
eta->setSelfConjugate(true); // real field
chi->setSelfConjugate(true); // real field
```

Replacing the field with the explicit index

```
model.replace(
    phi(i),
    eta(i) + CSL_I * chi(i)
);
```

Replacing the field without specifying indices

```
model.replace(
    phi,
    eta(i) + CSL_I * chi(i)
);
```

Warning The second method works well for one (or zero) index, but may be dangerous when several indices are in the same vector space, for a Field strength for example.

See also Chapter 2 for more details on quantum fields.

Rotations

Field rotations are treated in section 5.5.3 that presents the diagonalization capabilities of MARTY.

5.5.2 Symmetry breaking

Principle

It is possible to break gauge and flavor symmetries in MARTY. This is the most powerful feature for model building, although for now limited to particular cases. Symmetry breaking means two things:

- The broken gauge symmetry can be violated by interaction terms introduced after the symmetry breaking, in the sense that MARTY will not complain.

- Tensors living in vector spaces of the broken group are also broken in pieces. W^I in the $SU(2)_L$ Standard Model gauge for example will be broken in $\{W^1, W^2, W^3\}$.

Limitations for gauge symmetry breaking

The limitations of gauge symmetry breaking are the following[2]. There is for now no sub-grouping support. In other words, one cannot break a group into a subgroup. The group will be completely broken. The second limitation is a not a strong one, but could be unpleasant. When breaking a gauge symmetry, algebra generators are broken. Taking the example of $SU(2)$, generators T^A are expressed as functions of Pauli matrices:

$$T^A_{ij} = \frac{1}{2}\sigma^A_{ij}, \tag{5.5}$$

with

$$\sigma^1 = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \tag{5.6}$$

$$\sigma^2 = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}, \tag{5.7}$$

$$\sigma^3 = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}. \tag{5.8}$$

Structure constants defined as

$$[T^A, T^B] = if^{ABC}T^C \tag{5.9}$$

are $f^{ABC} = \epsilon^{ABC}$ in $SU(2)$, the fully anti-symmetric tensor with $\epsilon^{012} = 1$.

When breaking the gauge representation spaces, the values of generators and structure constants may not be defined. In particular, generators explicitly defined in `MARTY` are those known in the Standard Model (works also in the 2HDM and the MSSM), i.e. those for the doublet and triplet of $SU(2)$, and triplet and octet of $SU(3)$. One still can break a gauge that is not SM-like, but values for generators should then be given by the user to have a numerical result at the end. Otherwise results will depend on symbolic expressions of the type T^1_{02} or f^{124} that are undefined.

Gauge symmetry breaking could easily be improved in future `MARTY`'s developments but is not a priority for now as the present limitations do not prevent one to write any BSM model, simply may complicate this task. If one is particularly interested in such developments, we are available to discuss them !

In practice

In practice, breaking a symmetry in `MARTY` is very simple. This is shown in sample code 40. Broken particles are automatically renamed as said before, a field Φ_I with I in a N-dimensional broken space gives after breaking $\{\Phi_1, \Phi_2, \ldots, \Phi_N\}$. From a programming point of view, a particle of name `phi` will yield broken particles of names `phi_1`, `phi_2` etc. This is important if one wants to access the broken particles later on.

[2]Flavor symmetry breaking has not such limits.

Sample code 40: Symmetry breaking

Considering a `"C"` $SU(3)$ gauge group, and a `"F"` $SU(3)$ flavor group.

Breaking the gauge symmetry

```
model.breakGaugeSymmetry("C");
```

Breaking the Flavor symmetry

```
model.breakFlavorSymmetry("F");
```

Note $U(1)$ symmetries may also be broken. There is no tensor to break, but one tells `MARTY` that future interaction terms may violate the symmetry.

See also Documentation of ModelBuilder.

Sub-grouping in flavor symmetry breaking

Sub-grouping is possible for flavor symmetry breaking. The `CSL` machinery to break tensors exists, but the particle definition in `MARTY` should be changed a bit for the same process to be possible for gauge symmetries. Sub-grouping means keeping a sub-group unbroken. Let's consider the example of a $SU(4)$ flavor symmetry. One may want to break apart the two first components only. This gives for a matrix A_{ij} living in that 4×4 space

$$A_{ij} = \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix} \rightarrow \left(\begin{array}{cc|cc} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \hline \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{array}\right). \tag{5.10}$$

The matrix is then broken into 4 scalars, 2 tensors of dimensions $(1, 2)$, two $(2, 1)$ and one $(2, 2)$. For numbers one gets simply `CSL` constants in expressions. New tensor are named a particular way by `CSL`. The notation is heavy but allows a user to know what is the underlying tensor. The prescriptions to find new names are the following:

- Start with the initial tensor name.
- Add `"_B_"` to signal that the tensor has been broken.
- For each broken dimension add `"i_j"` if the broken tensor corresponds to indices `i` to `j` (or just `"i"` if `i == j`). If the range is maximal (all indices are conserved along the dimension), add `"a"` for 'all'.
- Add `"__"` between each dimension.

Sub-grouping can be done several times and each time these naming conventions are used. For the example in equation 5.10, tensors names would be

- `A_B_0__2_3` and `A_B_1__2_3` for the $(1, 2)$ tensors.
- `A_B_2_3__0` and `A_B_2_3__1` for the $(2, 1)$ tensors.
- `A_B_2_3__2_3` for the $(2, 2)$ tensor.

The exact procedure to break flavor symmetries is presented in sample code 41.

Sample code 41: Sub-grouping flavor symmetries

Considering a $SU(4)$ `"F"` flavor group.

Breaking 4 **in** $2 \oplus 2$ **and then in** $1 \oplus 1 \oplus 2$

```
model.breakFlavorSymmetry("F", {2, 2}, {"F1", "F2"});
model.breakFlavorSymmetry("F1");
```

Breaking 4 **in** $1 \oplus 1 \oplus 2$ **directly**

```
model.breakFlavorSymmetry("F", {1, 1, 2});
```

Breaking 4 **in** $1 \oplus 1 \oplus 1 \oplus 1$

```
model.breakFlavorSymmetry("F");
```

When sub-grouping, one must give the dimensions of all sub-groups (including dimension 1).

Note Users may give names for the new non-trivial flavors. If not given, `"_i"` will be appended to the initial flavor name to distinguish them. If there is one unique non-trivial flavor remaining, the name stays unchanged.

See also Documentation of ModelBuilder for more information.

5.5.3 Diagonalization

Meaningful results in general cannot be obtained without first diagonalizing all mass matrices of the theory. In order to get mass eigenstates one must treat the case of mixings, in mass terms, between different particles. This results in the tree-level spectrum of the theory, with masses and mixings. There is several ways to diagonalize mass matrices depending on what the user wants to do, that range from manual to fully automated solutions.

`MARTY` will complain if it sees mixing terms left in the mass Lagrangian, but will let users do it. Depending on the situation, different kinds of diagonalization are possible, always trying to minimize to size of expressions in the Lagrangian that propagates in Feynman rules and in the end theoretical calculations.

Rotations

Rotations are the first features allowing one to diagonalize mass matrices. There is two ways to rotate fields. Either the field Φ is still in a tensor notation

$$\Phi_i \mapsto U_{ij}\Phi_j, \tag{5.11}$$

or the rotation concerns different fields

$$\begin{pmatrix} \phi \\ \eta \\ \cdots \end{pmatrix} \mapsto \begin{pmatrix} U_{\phi\phi} & U_{\phi\eta} & \cdots \\ U_{\eta\phi} & U_{\eta\eta} & \cdots \\ \cdots & \cdots & \cdots \end{pmatrix} \begin{pmatrix} \phi \\ \eta \\ \cdots \end{pmatrix} \tag{5.12}$$

In both cases, the mixing matrix U must be unitary

$$U^\dagger U = U U^\dagger = 1. \tag{5.13}$$

As the user may want to perform both kinds of rotations, `MARTY` can do both. In any case, what this section present does not diagonalize mixings. It simply applies a rotation to fields in the Lagrangian. If one uses this feature to diagonalize a mass matrix by hand, it should be checked that the left mixing terms in the Lagrangian are indeed zero, even if `MARTY` is not able to know it automatically. Let us consider a simple example to illustrate this point, and take a mass Lagrangian

$$\mathcal{L} \ni -\frac{1}{2}(m_1^2\phi_1^2 + m_2^2\phi_2^2 + 2m_{12}^2\phi_1\phi_2), \tag{5.14}$$

giving a mass matrix

$$M^2 = \begin{pmatrix} m_1 & m_{12} \\ m_{12} & m_2 \end{pmatrix}. \tag{5.15}$$

One may want to rotate away the mixing between ϕ_1 and ϕ_2 by introducing an angle θ

$$\begin{aligned} \phi_1' &\equiv \cos\theta\phi_1 + \sin\theta\phi_2, \\ \phi_2' &\equiv -\sin\theta\phi_1 + \cos\theta\phi_2. \end{aligned} \tag{5.16}$$

This should be an angle defined as diagonalizing the Lagrangian, but `MARTY` will a priori not know it and the resulting Lagrangian is

$$\begin{aligned} \mathcal{L} \ni -\frac{1}{2}\Big[& (\cos^2\theta m_1^2 + \sin^2\theta m_2^2 + 2\cos\theta\sin\theta m_{12})\phi_1'^2 \\ & + (\cos^2\theta m_2^2 + \sin^2\theta m_1^2 - 2\cos\theta\sin\theta m_{12})\phi_2'^2 \\ & + \left(\cos\theta\sin\theta(m_2^2 - m_1^2) + 2m_{12}^2(\cos^2\theta - \sin^2\theta)\right)\phi_1'\phi_2'\Big]. \end{aligned} \tag{5.17}$$

`MARTY` will then end up on such a Lagrangian, eventually[3] read the masses of ϕ_1' and ϕ_2' on diagonal terms and ignore the mixing

$$\left(\cos\theta\sin\theta(m_2^2 - m_1^2) + 2m_{12}^2(\cos^2\theta - \sin^2\theta)\right)\phi_1'\phi_2', \tag{5.18}$$

that should be checked by the user to verify that it is indeed zero, in the user-defined conventions. `MARTY` will display non-diagonal terms in the mass Lagrangian if there are some in the final result when calling the `refresh()` function.

First, let us consider the tensor rotation of equation 5.11. One must provide the rotation matrix U. Sample code 42 presents the way to create a unitary matrix in a given vector space.

[3] If calling the `refresh()` function as precised in section 5.1

Sample code 42: Create a unitary matrix

Creating the unitary tensor "U"

```
Tensor U = Unitary("U", mySpace);
```

Creating indices

```
auto I = mySpace->generateIndices(3);
```

Testing unitarity

```
cout << U({I[0], I[1]})*U({I[1], I[2]}) << endl;
// >> U_{i, j}*U_{j, k}
cout << U({I[0], I[1]})*GetComplexConjugate(U({I[2], I[1]})) << endl;
// >> delta_{i, k}
cout << U({I[0], I[1]})*GetComplexConjugate(U({I[0], I[2]})) << endl;
// >> delta_{i, k}
```

See also Sample code 25 for more details on vector spaces.

See also The CSL manual for explanations on tensors and their properties.

Unitarity is not a mandatory property, but will be very practical in the Lagrangian because simplifications will be done automatically using the unitary property.

Once the user has a unitary matrix, the field rotation as in equation 5.11 is done in one function call, as shown in sample code 43.

Sample code 43: Tensor field rotation

Let us consider a rotation for a field "psi" in a "F" flavor space

Building the unitary matrix

```
auto flavorSpace = GetVectorSpace(model, "F", "psi");
Tensor U = Unitary("U", flavorSpace);
```

Rotating the field

```
model.rotateField("psi", U);
```

The second way to rotate fields is when they are not contained in a tensor, like in equation 5.12. In this case, the user may provide all the mixing matrix components if wanted, or let MARTY define them itself. This is summarized in sample code 44.

Sample code 44: Field rotation

Considering a simple two fields example, `"phi"` and `"eta"`.

Giving a rotation matrix explicitly

```
Expr theta = constant_s("theta");
Expr c = cos_s(theta);
Expr s = sin_s(theta);
model.rotateFields(
        {"phi", "eta"},
        {{c,  s},
         {-s, c}}
        );
```

Note The rotation matrix may be omitted, but one probably wants in this case to read the section 5.5.3 about semi-automatic diagonalization.

Symbolic diagonalization for 2-by-2 matrices

A symbolic diagonalization is possible in MARTY for 2×2 matrices. This is however only recommended for determinant 0 matrices, i.e. with one final massless state. In that case the mixing matrix is simple and may be introduced smoothly in the Lagrangian. For general 2×2 matrices, the result may be rather complicated expressions and a user could prefer the explicit rotation solution presented above. Matrices of higher dimensions cannot be diagonalized symbolically in general, it is then not possible to do it in MARTY.

Diagonalizing a mass matrix symbolically is very simple in MARTY, the procedure being presented in sample code 45.

Sample code 45: Symbolic diagonalization

Consider a 2×2 matrix mixing the B and W^3 bosons in the Standard Model. The matrix has determinant 0 and can be automatically diagonalized typing

```
model.diagonalizeSymbolically("B");
// Or
// model.diagonalizeSymbolically("W^3");
```

Only one particle name is needed, MARTY will find the matrix automatically in the Lagrangian.

If the matrix determinant is known to be zero by the user but difficult to know for MARTY, one can give it this extra piece of information through a boolean to help MARTY simplify mass and mixing matrices

```
// Means "Believe me the determinant is zero,
// use the corresponding simplifications"
model.diagonalizeSymbolically("B", true);
// Or
// model.diagonalizeSymbolically("W^3", true);
```

Note For non-zero determinants, the symbolic diagonalization is not recommended as it will probably introduce very complicated expressions.

Semi-automatic diagonalization

This part presents the way to diagonalize any mass matrix without having to know anything about it. The diagonalization is not performed in the model but later during numerical evaluation. The principle is to introduce arbitrary masses and mixings in the Lagrangian. Results will then depend on it, and diagonalization will be done numerically, in full generality, just before the numerical evaluation. In this section we present how to trigger such diagonalization. See section 7.1.2 to have more information about the actual numerical diagonalization.

There is two types of diagonalizations. The simple one, triggered by a mass matrix acting on a field Φ

$$\Phi^\dagger M \Phi, \tag{5.19}$$

and the bi-diagonalization by matrices acting on two different sets of fields Φ_L and Φ_R

$$\Phi_L^\dagger M \Phi_R + \Phi_R^\dagger M^\dagger \Phi_L \tag{5.20}$$

In the first case, one can define

$$\Phi = U\Phi', \tag{5.21}$$

with U a unitary matrix and a diagonal mass matrix D defined as

$$\Phi^\dagger M \Phi = \Phi' U^\dagger M U \Phi' \equiv \Phi'^\dagger D \Phi'. \tag{5.22}$$

Asking `MARTY` to diagonalize Φ, it will introduce symbolic matrices U and D, keeping the initial M in memory to diagonalize it later, and find values for U and D.

For the second case, one can define

$$\Phi_L = U\Phi'_L, \tag{5.23}$$
$$\Phi_R = V\Phi'_R, \tag{5.24}$$

with U and V unitary matrices and a diagonal mass matrix D defined as

$$\begin{aligned} \Phi_L^\dagger M \Phi_R + \Phi_R^\dagger M^\dagger \Phi_L &= \Phi_L^{\dagger\prime} U^\dagger M V \Phi'_R + \Phi_R^{\dagger\prime} V^\dagger M^\dagger U \Phi'_L \\ &\equiv \Phi_L^{\dagger\prime} D \Phi'_R + \Phi_R^{\dagger\prime} D \Phi'_L. \end{aligned} \tag{5.25}$$

One can see that we get the relation for D

$$U^\dagger M V = V^\dagger M^\dagger U = D. \tag{5.26}$$

The way to obtain U and V is called bi-diagonalization, and requires to solve

$$U^\dagger M M^\dagger U = D^2, \tag{5.27}$$
$$V^\dagger M^\dagger M V = D^2. \tag{5.28}$$

One can check indeed that

$$\begin{aligned} D^2 &= U^\dagger M V V^\dagger M^\dagger U = U^\dagger M M^\dagger U \\ &= V^\dagger M^\dagger U U^\dagger M V = V^\dagger M^\dagger M V. \end{aligned} \tag{5.29}$$

There is two diagonalizations to perform, and one gets finally two mixing matrices U and V for left and right fields, and a diagonal matrix D[4].

[4] D does not actually contain the eigenvalues of M, but the squared roots of eigenvalues of $MM^\dagger$.

The procedure to ask MARTY to perform these diagonalizations is summarized in sample code 46. The principle is very similar to a simple rotation. One simply has to specify a boolean to tell MARTY to remember the mass matrix to be able to diagonalize it later. This is the first step to have a spectrum generator for any BSM model, the next being the library generation, explained in section 7.1.2.

Sample code 46: Semi-automated diagonalization

Simple diagonalize of fields `"phi_1"` and `"phi_2"`

```
model.rotateFields({"phi_1", "phi_2"}, true);
```

Bi-diagonalization of left fields `"psiL_1"` and `"psiL_2"`, and right fields `"psiR_1"` and `"psiR_2"`

```
model.birotateFields({"psiL_1", "psiL_2"}, {"psiR_1", "psiR_2"});
```

Note No need to give a boolean for bi-diagonalization as this feature has been developed only in that particular purpose.

See also Section 7.1.2 to see how to get the spectrum of the theory after diagonalizing symbolically fields following the procedure presented here.

Automatic diagonalization

There is a way to fully automate diagonalization in MARTY. Although we recommend to use semi-automated features, explicit rotations and symbolic diagonalization, one can ask MARTY to look at the mass Lagrangian, diagonalize symbolically the 2×2 matrices it encounters, and using the procedure presented in the previous section for other non-diagonal matrices if the `diagonalizeSymbolically` option is set to `true` by the user (see chapter 8). This is presented in sample code 47.

Sample code 47: Automated diagonalization

Automated diagonalization in a MARTY model

```
model.diagonalizeMassMatrices();
```

Note 2×2 matrices are diagonalized symbolically.

Note Other non-diagonal matrices are diagonalized following the prescriptions of the previous section if the `diagonalizeSymbolically` option is set to `true` by the user (see chapter 8).

5.5.4 Other features

Dirac fermion embedding

When doing model building with Weyl fermions, one may eventually want to have a Dirac fermion notation

$$\psi \equiv \psi_L \oplus \psi_R, \tag{5.30}$$

in particular when a Weyl mass terms arises in the Lagrangian like

$$\mathcal{L} \ni -m \left(\bar{\psi}_L \psi_R + \bar{\psi}_R \psi_L \right) = -m \bar{\psi} \psi. \tag{5.31}$$

It is possible to tell MARTYto create a Dirac fermion, linking it with two existing Weyl fermions (must be left and right). This is presented in sample code 48.

Sample code 48: Dirac fermion embedding

Considering two Weyl fermions `"psi_L"` and `"psi_R"`

```
model.diracFermionEmbedding("psi_L", "psi_R");
```

Note The call of the `refresh()` function in principle should automate it, and recognize Weyl fermions that have a common mass term. It is however possible to do it explicitly.

See also Section 5.1 for the `refresh()` function.

Goldstone boson promotion

When breaking symmetries, Goldstone bosons may appear, that one may want to link to their corresponding vector boson. Gauge fixing will then apply transformations to the Goldstone boson automatically. Sample code 49 presents this simple procedure.

Sample code 49: Goldstone boson promotion

In the Standard Model example, we promote the Higgs degree of freedom G^+ to the Goldstone boson of W.

```
model.promoteToGoldstone("G^+", "W");
```

After that, the Goldstone is linked to the W gauge fixing parameter ξ, in particular through its mass

$$M_G = \sqrt{\xi} M_W. \tag{5.32}$$

See also Section 2.1.4 for more details on Goldstone bosons, and section 6.3 for explanations on gauge fixing in MARTY.

Ghost boson promotion

When breaking symmetries or making replacements, ghost bosons may change independently on their corresponding vectors, and in that case one may want to set the ghost of a particular vector. Sample code 50 presents this simple procedure.

Sample code 50: Ghost boson promotion

In the Standard Model example, we can promote a c^+ to be the ghost boson of W once we defined these two particles.

```
model.promoteToGhost("c^+", "W");
```

After that, the Ghost is linked to the W gauge fixing parameter ξ, in particular through its mass

$$M_G = \sqrt{\xi} M_W. \tag{5.33}$$

See also Section 2.1.4 for more details on ghost bosons, and section 6.3 for explanations on gauge fixing.

Majorana fermion promotion

It is possible to transform a **self-conjugate Weyl fermion** η_L (or η_R) into a **4-component Majorana fermion** η by using the method `promoteToMajorana()`. This is useful for example in the MSSM when defining the gauginos as self-conjugate Weyl fermions to recover the 4-component Majorana definition if needed. The Lagrangian is modified accordingly to simplify what can be, using the properties of the Majorana fermion. More precisely, `MARTY` applies the relevant replacements in the Lagrangian from the definition of the 4-component η to obtain η from η_L or η_R:

$$\eta \equiv \begin{pmatrix} \eta_L \\ \eta_R \end{pmatrix} = \begin{pmatrix} \eta_L \\ C\eta_L^\dagger \end{pmatrix}, \tag{5.34}$$

with the conjugation matrix C, for an initial left-handed Weyl fermion η_L and

$$\eta \equiv \begin{pmatrix} \eta_L \\ \eta_R \end{pmatrix} = \begin{pmatrix} C\eta_R^\dagger \\ \eta_R \end{pmatrix}, \tag{5.35}$$

for an initial right-handed Weyl fermion η_R. Sample code 51 presents this simple procedure.

Sample code 51: Majorana fermion promotion

It is possible to transform a self-conjugate Weyl fermion η_L (or η_R) into a 4-component Majorana fermion η.

```
model.promoteToMajorana("eta_L", "eta");
// Or just
// model.promoteToMajorana("eta_L");
// if the name need not to be modified
```

Chapter 6

Calculations

6.1 General principles

This chapter is about theoretical calculations done by MARTY, its main purpose. In the following are detailed all procedures to get symbolic results from a MARTY program. The numerical evaluation is done outside of MARTY and will be treated separately in chapter 7. Section 6.2 will introduce Feynman rules in MARTY, and how to calculate them. The gauge fixing procedure we be detailed in section 6.3, treating the case of ghosts, Goldstone bosons and the gauge fixing parameter ξ. Finally, sections 6.4, 6.5 and 6.7 will respectively show how to get symbolic results for amplitudes, squared amplitudes and Wilson coefficients in MARTY. All these calculations are available through the Model interface, the last layer of the high energy physics model abstraction in MARTY as we saw in chapter 3.

6.2 Feynman Rules

Feynman rules are stored in the class FeynmanRule. It is mostly encapsulated by the Model class. This means that a user should not have to manipulate explicitly this object. Feynman rules will be automatically computed before a calculation if they have not already been derived.

6.2.1 Get Feynman rules

The procedure to calculate the Feynman rules of a model and get them is presented in sample code 52. There is a simple interface for the user to get the information about the different vertices, being the Feynman diagrams (using *GRAFED*) or the symbolic expressions. One can compute Feynman rules at any time during model building. Next section will introduce notions to be able to read Feynman rules.

Sample code 52: Feynman rules

Launch the calculation of Feynman rules

```
model.computeFeynmanRules();
```

Get Feynman rules

```
// No need to ask the explicit computation when getting rules.
auto rules = model.getFeynmanRules();
```

Displaying the rules

```
Display(rules); // Displaying expressions in standard output
Show(rules); // Shows Feynman diagrams for rules with GRAFED
```

Note The `auto` keyword deduces the type `vector<FeynmanRule>` here, a list of rules.

Note Computing Feynman rules can be done several times if wanted. When getting Feynman rules, they will be calculated only if it is not already done, and then simply returned.

See also Documentation of class FeynmanRule.

6.2.2 Read Feynman rules

It is important to be able for a user to read Feynman rules, and check that they are correct. When doing model building, this is the most important part. An incorrect vertex coming from a misunderstanding of different conventions in the model will end up in wrong results. Before doing any calculation with `MARTY`, we strongly encourage users to check all the Feynman rules that are used.

A Feynman rule is a set of fields that enter the Wick theorem, with which other fields will contract. Here is an example of Feynman rule for a fermion-photon interaction:

```
(0) : Rule for A_mu(p_1) psi_b(p_2)^(*) psi_a(p_3) :
-i*e*gamma_{mu,a,b}
```

One can see the field content of the first line, with corresponding indices and momenta. In the line below stands the expression of the Feynman rule. We see the coupling, the $\gamma-$matrix, but more importantly one may want to know which fermion is incoming, `psi_b(p_2)^(*)` or `psi_a(p_3)`. A Feynman rule does not show the fields it contains but the fields that can be contracted with it. It means that a ϕ^* in the vertex will yield ϕ (and inversely) in the actual Feynman rule. This principle is shown in figure 6.1. We can know be certain that ψ^* in the vertex corresponds to the incoming fermion, and must then be associated with the second index `b` of the γ-matrix.

Let us now consider other Feynman rules examples. First, a scalar QED example, with a $U(1)$ gauge. There is two vertices, a 3-vertex with a derivative of the scalar ϕ and a 4-vertex. Feynman rules are the following

```
(0) : Rule for A_\rho_87(p_1) \phi(p_2) \phi(p_3)^(*) :
i*e*(p_1_\rho_87 + 2*p_2_\rho_87)

(1) : Rule for A_\rho_52(p_1) A_+\rho_53(p_2) \phi(p_3) \phi(p_4)^(*) :
2*i*e^2*delta_{\rho_52,+\rho_53}
```

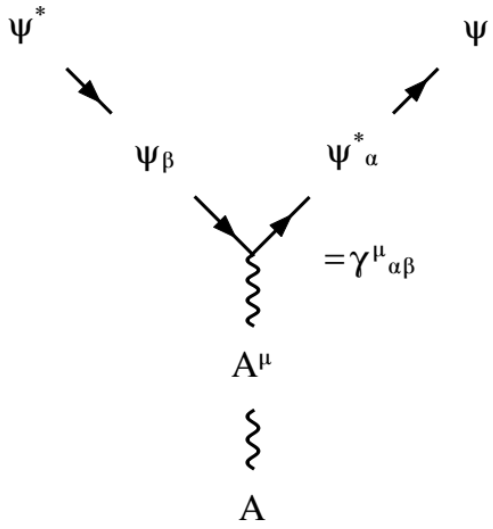


Figure 6.1: Vertex for a Feynman rule, with a fermion-photon interaction example. The fields in the vertex are shown in the middle, and fields using the rule (contracting with it) are on the outside. These outside fields are those defining a Feynman rule, that `MARTY` displays as content of a vertex.

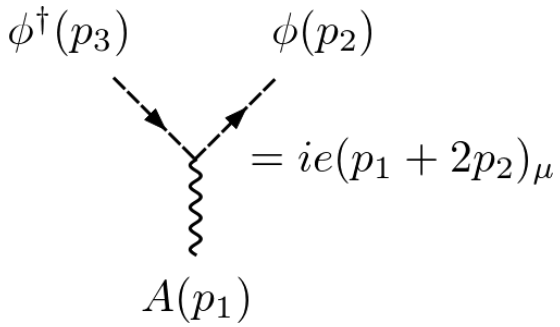


Figure 6.2: 3-vertex rule in Scalar QED. The diagram presented here does not correspond to the inner vertex but to `MARTY`'s Feynman rule, as explained in figure 6.1. All momenta in Feynman rules are directed towards the vertex.

One may see the momentum dependence that arises in the 3-vertex. Figure 6.2 shows in more details this dependence. Momenta in Feynman rules are always considered incoming by default.

A last example of Feynman rules, in a QCD model. One gets for example a fermion-gluon interaction, similar to the QED vertex with a $SU(3)$ generator, and a 4-gluon vertex introducing $SU(3)$ structure constants f^{ABC}. The two rules are the following in `MARTY`:

```
(4) : Rule for G_{h,\tau}(p_1) X_{b,eps}(p_2)^(*) X_{a,del}(p_3) :
i*g*T_{h,a,b}*gamma_{\tau,del,eps}

(5) : Rule for G_{a,+\tau}(p_1) G_{b,+\mu}(p_2) G_{c,\nu}(p_3) G_{d,\rho}(p_4) :
-i*g^2*(f_{a,d,%h}*f_{b,c,%h}*g_{+\mu,+\tau}*g_{\rho,\nu}
+ f_{a,c,%h}*f_{b,d,%h}*g_{+\mu,+\tau}*g_{\rho,\nu}
+ -f_{a,c,%h}*f_{b,d,%h}*delta_{+\mu,\nu}*delta_{\rho,+\tau}
+ -f_{a,b,%h}*f_{c,d,%h}*delta_{+\mu,\nu}*delta_{\rho,+\tau}
+ -f_{a,d,%h}*f_{b,c,%h}*delta_{+\mu,\rho}*delta_{+\tau,\nu}
+ f_{a,b,%h}*f_{c,d,%h}*delta_{+\mu,\rho}*delta_{+\tau,\nu})
```

The 4-vertex reads (renaming indices to a vertex $G^{A\mu}(p_1)G^{B\nu}(p_2)G^{C\rho}(p_3)G^{D\sigma}(p_4)$)

$$\begin{aligned} -ig^2\Big(& f^{adh}f^{bch}g^{\mu\nu}g^{\rho\sigma} + f^{ach}f^{bdh}g^{\mu\nu}g^{\rho\sigma} \\ & - f^{ach}f^{bdh}g^{\mu\sigma}g^{\nu\rho} - f^{abh}f^{cdh}g^{\mu\sigma}g^{\nu\rho} \\ & - f^{adh}f^{bch}g^{\mu\rho}g^{\nu\sigma} + f^{abh}f^{cdh}g^{\mu\rho}g^{\nu\sigma}\Big), \end{aligned} \tag{6.1}$$

that can be factored to recover a well-known rule (can be found in [21] page 511)

$$\begin{aligned} -ig^2\Big(& f^{abh}f^{cdh}\left(g^{\mu\rho}g^{\nu\sigma} - g^{\mu\sigma}g^{\nu\rho}\right) \\ & + f^{ach}f^{bdh}\left(g^{\mu\nu}g^{\rho\sigma} - g^{\mu\sigma}g^{\nu\rho}\right) \\ & + f^{adh}f^{bch}\left(g^{\mu\nu}g^{\rho\sigma} - g^{\mu\rho}g^{\nu\sigma}\right)\Big). \end{aligned} \tag{6.2}$$

6.3 Gauge fixing

Gauge fixing can take place for any vector boson, independently of a gauge group. Taking the most general case of a massive vector boson A^μ of mass M with a Goldstone boson ϕ and a ghost c, one can write down the propagators of these different particles depending on the gauge fixing parameter ξ [21], as presented in figure 6.3. Correspondence between the different gauges available in `MARTY` and the ξ parameter is detailed in table 6.1.

$$\begin{aligned} A^\mu \sim\sim A^\nu &= -i\frac{g^{\mu\nu} - (1-\xi)\frac{p^\mu p^\nu}{p^2-\xi M^2}}{p^2 - M^2}, \\ \phi \text{ ----▶---- } \phi &= \frac{i}{p^2 - \xi M^2}, \\ c \text{ ···▶··· } c &= \frac{i}{p^2 - \xi M^2}. \end{aligned}$$

Figure 6.3: Propagators for a vector boson and its Goldstone and ghost bosons depending on the gauge fixing parameter ξ.

Name	Parameter value	Name in `MARTY`
Feynman 't Hooft	$\xi = 1$	`gauge::Feynman`
Lorenz	$\xi = 0$	`gauge::Lorenz`
Unitary	$\xi = \infty$	`gauge::Unitary`
$\mathcal{R}_\xi$	ξ	`gauge::NotDefined`

Table 6.1: List of the different gauges that one can choose for a particular vector boson in `MARTY`. The $\mathcal{R}_\xi$ gauge lets an explicit ξ dependence in calculation, that should cancel in physical observables.

One can see that setting the gauge choice for a vector boson will simply modify its propagator, and masses of the associated ghost and Goldstone bosons. As physical results must be gauge invariant, doing a calculation in one gauge or another should give the same result. In particular, a calculation in the $\mathcal{R}_\xi$ gauge will be expressed as a function of ξ

but should not depend on it. In the special case of the unitary gauge, the propagator or the vector becomes

$$-i\frac{g^{\mu\nu}-\frac{p^\mu p^\nu}{M^2}}{p^2-M^2}, \tag{6.3}$$

the ghost and Goldstone bosons acquire an infinite mass $\sqrt{\xi}M \to \infty$ and decouple from the theory. In particular, MARTY simply disables these scalars in diagrams for the unitary gauge. Beware however that the latter must not be used for massless vector bosons as the limit $\xi \to \infty$ is ill-defined. The procedure to fix a gauge choice for a vector boson is presented in sample code 53.

Sample code 53: Gauge fixing

Setting the unitary gauge for the W boson and Feynman gauge for the photon A in the Standard Model for example

```
model.setGaugeChoice("W", gauge::Unitary);
model.setGaugeChoice("A", gauge::Feynman);
```

Note The default gauge choice in MARTY is `gauge::Feynman` with $\xi = 1$.

See also Documentation of file gaugedGroup.h.

Users should note that for now Goldstone - ghost interactions as presented in [22] are not taken into account in MARTY. They come from gauge symmetry breaking and are not written in general. If one is interested in processes using such interactions, the corresponding interaction terms should be given explicitly as demonstrated in section 3.2.4.

6.4 Amplitude

The amplitude calculation is the main feature of MARTY, as other quantities (squared amplitudes and Wilson coefficients) require first such calculation.

The calculation is fully automated, and can be launched in a single command line as demonstrated in sample code 55. We however detail in this section the main steps in the derivation of amplitude expressions.

6.4.1 External legs

External legs are the only pieces of information the user has to give beside the order of development. From external legs, MARTY has to find all possible diagrams as presented in figure 6.4.

An external leg carries four pieces of information:

- The underlying quantum field.
- The direction, incoming or outgoing.
- The conjugation, particle or anti-particle.
- The physicality, on-shell or off-shell.

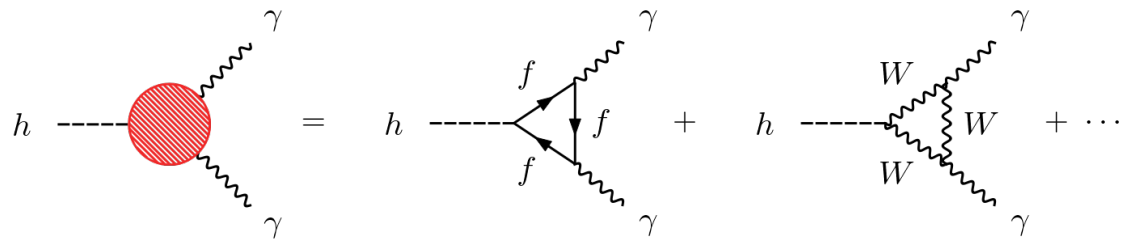


Figure 6.4: Different 1-loop Feynman diagrams possible for the $h \to \gamma\gamma$ process in the SM. There is for example fermion and W-boson triangles.

MARTY has a simple interface to create field insertions that is shown in sample code 54. There are four interface functions (`Incoming()`, `Outgoing()`, `AntiPart()`, `OffShell()`) that can be composed with each other to build the relevant field insertion, starting from a `Particle` type or the name of the field.

Sample code 54: Field insertions

An incoming off-shell fermion `"psi"`

```
Incoming(OffShell("psi"));
// Or
// OffShell(Incoming("psi"));
```

An outgoing anti `phi`

```
Outgoing(AntiPart("psi"));
// Or
// AntiPart(Outgoing("psi"));
```

To give a list of insertions as function parameter, one can put them in curly braces `{}` (here for an electron self-energy calculation):

```
{Incoming(OffShell("e")), Outgoing(OffShell("e"))}
```

See also Documentation of Insertion.

With this information on can now launch an amplitude calculation with MARTY as presented in sample code 55.

Sample code 55: Amplitude calculation

Calculating the transition amplitude for $h \to e\bar{e}$.

Calculate the amplitude from the model

```
auto res = model.computeAmplitude(
        Order::TreeLevel, // or Order::OneLoop
        {Incoming("h"), Outgoing("e"), Outgoing(AntiPart("e"))}
        );
```

Displaying the results

```
Display(res); // Prints symbolic result in standard output
Show(res); // Shows Feynman diagrams with GRAFED
```

Getting the different terms of the amplitude

```
for (size_t i = 0; i != res.size(); i++) {
    Expr &term = res.expression(i);
    // Do something to term
}
```

Note `auto` deduces the type Amplitude, that contains all expressions and diagrams of the process.

See also Documentation of Model and Amplitude.

For now, library generation is only available for scalar quantities. An amplitude (with non-scalar external particles) have indices and cannot be given directly to the library generator presented in chapter 7. To generate C++ code corresponding to an amplitude, one must decompose it in Wilson coefficients, that are scalar quantities in front of different operators. For example the vacuum energy $i\Pi_{\mu\nu}(p)$ of a vector boson A can be decomposed in general as

$$i\Pi_{\mu\nu}(p) = \alpha(p^2, m^2) g_{\mu\nu} + \beta(p^2, m^2) p_\mu p_\nu, \tag{6.4}$$

which corresponds to the amplitude

$$i\mathcal{M} = \alpha(p^2, m^2) \left(\epsilon^*(p) \cdot \epsilon(p)\right) + \beta(p^2, m^2) \left(p.\epsilon^*(p)\right) \left(p \cdot \epsilon(p)\right). \tag{6.5}$$

Calculating Wilson coefficients for this amplitude as explained in section 6.7 will yield $\alpha(p^2, m^2)$ and $\beta(p^2, m^2)$ that can be used for library generation.

6.4.2 Finding diagrams

Finding all the diagrams for a given process can be done in several ways. FeynArts [2] uses a hard-coded set of possible topologies for tree-level and one-loop processes with a given number of external legs. Once done, it tries to fit particles on each leg of each topology. `MARTY` uses a more brutal way. It applies explicitly the Wick theorem, trying all possible Lagrangian interactions for each vertex in the diagram.[1] This can be longer but has the advantage of being fully general, in particular not limited to the one-loop order.

The algorithm finding all diagrams is of course highly optimized to avoid calculating several times the same diagram. Consider the MSSM example that contains about 10^4

[1] The maximum number of vertices is determined by the development order.

interaction terms. Taking a 1-loop amplitude with 3 external particles requires to develop to $\mathcal{L}^3$ in perturbation theory, i.e. to have $(10^4)^3 = 10^{12}$ terms. This is of course not possible to do in a reasonable amount of time on a standard computer, not even considering that for each term arrangement, an algorithm of pair-finding must be applied for the set of fields. The present algorithm in `MARTY` has a very reasonable performance, and is most of the time faster than the actual calculation that follows. It results in a set of possible diagrams, each of which has a corresponding expression, using Feynman rules for all vertices, that must be further simplified. Each diagram comes with a symmetry factor that is automatically determined by `MARTY`'s algorithm, and a possible sign coming from fermion ordering that is also taken care of.

6.4.3 Initial amplitude expression

The amplitude $i\mathcal{M}(i \to f)$ of a transition from an initial state i to a final state f is defined from the S-matrix element

$$\begin{aligned}\langle f|S|i\rangle &\equiv \langle f|(1+i\mathcal{T})|i\rangle \\ &= \langle f|i\rangle + (2\pi)^4\delta^{(4)}\left(\sum_i p_i - \sum_f p_f\right)\cdot i\mathcal{M}(i\to f),\end{aligned} \tag{6.6}$$

with p_i incoming and p_f outgoing momenta. What `MARTY` actually calculates contains the $(2\pi)^4\delta^{(4)}\left(\sum_i p_i - \sum_f p_f\right)$ factor coming from total momentum conservation but it is removed from the result to obtain $i\mathcal{M}(i \to f)$, the term proportional to the identity being not relevant in quantum field theory calculations. Feynman rules are properly inserted at each interaction vertex, and momentum conversation in the diagram is calculated from the LSZ formula that introduces $V_e + V_i + E$ integrals, one for each external vertex (V_e), internal vertex (V_i) and edge (E) in the diagram. These integrals simplify following the rules

$$\int d^4X e^{iX(p-q)} = (2\pi)^4\delta^{(4)}(p-q), \tag{6.7}$$

$$\int d^4q\, \delta^{(4)}(p-q)f(q) = f(p). \tag{6.8}$$

This machinery allows `MARTY` to apply automatically momentum conservation at each vertex in the diagram (external and internal), in particular with equation 6.7. There is always one momentum conservation property at the end of the calculation, that gives the kinematic condition

$$\sum_i p_i - \sum_f p_f = 0. \tag{6.9}$$

The number of edges in a diagram reads

$$E = V - 1 + N_L = V_I + V_E - 1 + N_L, \tag{6.10}$$

with $V = V_I + V_E$ the total number of vertices, and N_L the number of loops. A tree has $V - 1$ edges, and each additional connection will add a loop to the diagram. The LSZ formula introduces then

$$V = V_E + V_I = V_E + V_I - 1 + 1 \tag{6.11}$$

position-space integrals. $V_E + V_I - 1$ of them simplify one by one momentum integrals, and the last one (the +1 left) gives the momentum conservation on the whole diagram. With initially E momentum space integrals, we are left with

$$E - (V_E + V_I - 1) = V_I + V_E - 1 + N_L - (V_E + V_I - 1) = N_L \tag{6.12}$$

momentum integrals in the final expression of the amplitude. The latter must then be simplified, as presented in the following.

6.4.4 Simplify the expression

Group theory

Group theory simplifications include simple tensor contractions and trace of generators like

$$Tr(T^{A_1}T^{A_2}\cdots T^{A_N}). \tag{6.13}$$

We will not here go into many details about the calculation of such quantities. For more explanations, see [23, 24]. Traces can be calculated in all representations of all semi-simple Lie groups. They are decomposed in a combination of invariant fully-symmetric tensors $d^{A_1\cdots A_N}$ and structure constants f^{ABC}. There is additional simplification identities in the defining representation of these groups [20] like

$$T^A_{ij}T^A_{kl} = \frac{1}{2}\left(\delta_{il}\delta_{jk} - \frac{1}{N}\delta_{ij}\delta_{kl}\right) \tag{6.14}$$

in $SU(N)$.

The limitations of group theory simplifications are the following:

- Specific tensors in exceptional and $Sp(N)$ algebras may appear in certain amplitudes. These tensors lack for now simplification properties to have a fully simplified results automatically. This will be improved in the future.
- When calculating a squared amplitude with a color trace, one gets contraction of invariant tensors $d^{ABC\cdots}$ that are not yet simplified automatically by `MARTY`. This limitation can be overcome in defining representations of $SU(N)$ and $SO(N)$ through the application (by `MARTY`) of projection relations as equation 6.14.

Group theory simplifications are actually easy to implement in `MARTY`, the challenge is to find general properties that can be applied in all semi-simple Lie groups. So if one needs any particular simplification that is missing in `MARTY`, we are available to discuss their implementation!

Diracology

Diracology is the algebra with Dirac matrices γ^μ. One has to apply number of simplification properties starting from

$$\{\gamma^\mu, \gamma^\nu\} = 2g^{\mu\nu}, \tag{6.15}$$

calculate traces like

$$Tr(\gamma^\mu\gamma^\nu\cdot), \tag{6.16}$$

and eventually apply Dirac equations in fermion bilinears

$$\not{p}u(p) = mu(p), \tag{6.17}$$
$$\not{p}v(p) = -mv(p), \tag{6.18}$$

to simplify at most fermionic chains. The conjugation matrix

$$C \equiv i\gamma^0\gamma^2 \tag{6.19}$$

also appears in amplitudes due to Majorana fermions or fermion-number violating interactions, and must be simplified using properties of the type

$$C^2 = -1, \tag{6.20}$$
$$C\gamma^\mu = -\gamma^{\mu T}C, \tag{6.21}$$
$$v = C\bar{u}^T. \tag{6.22}$$

More details on that can be found in [25].

All of the algebra presented above is done automatically by `MARTY` in amplitude simplifications. The limitations of these procedures are:

- Only the spin 1/2 algebra is done. In order to have higher spins like 3/2, work would need to be done to implement the relevant algebra simplifications.
- Fermions quadrilinears lack Fiertz identities application [26] to simplify mixed momenta between two fermion currents. This prevents to get 4-fermion Wilson coefficients for now, but is currently being implemented in `MARTY`.

One-loop calculation

As we said in section 6.4.3, a one-loop calculation requires the evaluation of one momentum integrals such as

$$I = \int \frac{d^4q}{i\pi^2} \frac{\prod_{i=1}^{n} q^{\mu_i}}{\prod_{j=0}^{m-1}((q-p_j)^2 - m_j^2)} \tag{6.23}$$

for the rank n $m-$point function, with $p_0 = 0$, $p_{j\geq 1}$ combinations of external momenta, m_j masses of particles in the loop. m is then the number of propagators and n, the rank, the number of momenta in the numerator. Letters are associated to n-point functions, starting with A for the 1-point function, up to E for the 5-point function. Such integrals can be decomposed in different Lorentz structure. Considering the example of the rank 2 3-point function

$$C^{\mu\nu} \equiv \int \frac{d^4q}{i\pi^2} \frac{q^\mu q^\nu}{(q^2 - m_0^2)((q-p_1)^2 - m_1^2)((q-p_2)^2 - m_2^2)}, \tag{6.24}$$

one can write without loss of generality

$$C^{\mu\nu} = C_{00}(p_i, m_j)g^{\mu\nu} + C_{11}(p_i, m_j)p_1^\mu p_1^\nu + C_{22}(p_i, m_j)p_2^\mu p_2^\nu + C_{12}(p_i, m_j)(p_1^\mu p_2^\nu + p_1^\nu p_2^\mu). \tag{6.25}$$

The decomposition is done by `MARTY`, and factors C_{ij} are numerical functions implemented in the Fortran / C library LoopTools [2], used when results are evaluated numerically.

The scalar factors coming from one-loop integrals can have a divergent part that is regularized by taking the dimension $D = 4 - 2\epsilon$. Integrals then take the form

$$I \approx \frac{a}{\epsilon} + b + \mathcal{O}(\epsilon). \tag{6.26}$$

Factors of D coming from Minkowski index contractions must then be kept to determine the local terms they generate when they are multiplied by a divergent integral [27, 28]. For the scalar 1-point function for example, we get the finite part

$$\text{Finite}(DA_0(m^2)) = \text{Finite}((4-2\epsilon)A_0(m^2)) = -2m^2 + 4 \cdot \text{Finite}(A_0(m^2)). \tag{6.27}$$

This is done by `MARTY` automatically, that adds local terms when is necessary.

One loop calculations are limited to

- At most rank 2 1-point function, $A^{\mu\nu}$.
- At most rank 3 2-point function, $B^{\mu\nu\rho}$.
- At most rank 4 3-point function, $C^{\mu\nu\rho\sigma}$.
- At most rank 5 4-point function, $D^{\mu\nu\rho\sigma\lambda}$.
- At most rank 4 5-point function, $E^{\mu\nu\rho\sigma}$.
- Propagators with denominators of the type $\frac{1}{p^2 - X(m)}$. Note that one can absorb in general a factor in front of p^2 in the numerator.

The rank limitations[2] actually follow the integrals LoopTools provides. They could however be relaxed in the future as we could implement reduction formulas for higher-rank integrals, in particular the Passarino Veltman reduction [29, 30]. It can be difficult to adapt the procedure to generalized propagators, as all standard calculations are based on master integrals, provided by libraries like LoopTools. One can still do tree-level calculations in these models, and as they are mostly exotic (Lorentz violating models for example), the tree-level may be sufficient to perform analysis in a phenomenological purpose.

6.5 Squared Amplitude

Cross-sections are the main observables we get in colliders. They are directly proportional to the number of events observed in the various detectors. `MARTY` does not compute directly the cross-sections, but performs the complicated theoretical part. `MARTY` actually calculates the squared amplitude. For incoming particles I with spins j_I and outgoing particles O of spins j_O, the squared amplitude is (as a function of the amplitude $i\mathcal{M}$ that depends on the particle spins)

$$\frac{1}{\prod_I d_I} \sum_{\{j_I\},\{j_O\}} |\mathcal{M}|^2, \tag{6.28}$$

with d_I the spin dimension of the incoming particle I taking into account mass-less effects for spin 1 particles. This quantity is averaged over the spin dimension of the incoming

[2] Rank limitations can also affect some gauges, as for a vector boson Unitary and Lorenz gauges introduce additional momenta in the numerators and denominators of propagators.

particles. Squared amplitudes imply the calculation of traces in Dirac and color spaces (group algebra) that MARTY computes automatically already at the amplitude level. The additional simplifications done by MARTY are the polarization sums for spin 1/2 and spin 1 particles. For spinors $u_\sigma(p)$ (particle) and $v_\sigma(p)$ (anti-particle) with spin σ one has

$$\sum_\sigma u_\sigma(p)\bar{u}_\sigma(p) = \not{p} + m, \tag{6.29}$$

$$\sum_\sigma v_\sigma(p)\bar{v}_\sigma(p) = \not{p} - m. \tag{6.30}$$

$$\tag{6.31}$$

For spin 1 particles, proper quantization is ensured by ghosts (see section 6.3) and the polarization sum for $\epsilon^\mu_\lambda(p)$ of spin λ reads

$$\sum_\lambda \epsilon^\mu_\lambda(p)\epsilon^{\nu *}_\lambda(p) = -g^{\mu\nu} \tag{6.32}$$

for massless particles and

$$\sum_\lambda \epsilon^\mu_\lambda(p)\epsilon^{\nu *}_\lambda(p) = -g^{\mu\nu} + \frac{p^\mu p^\nu}{M^2} \tag{6.33}$$

for vector bosons of mass M.

The result is a scalar depending on momenta and masses of particles in the process. The differential cross-section has always the same form for a given process of amplitude $i\mathcal{M}$

$$d\sigma \equiv K(p_i, m_i) \cdot \left(\frac{1}{\prod_I d_I} \sum_{\{j_I\},\{j_O\}} |\mathcal{M}|^2 \right) d\Pi_{LIPS}, \tag{6.34}$$

with $K(p_i, m_i)$ a factor coming from kinematics, and $d\Pi_{LIPS}$ the Lorentz Invariant Phase Space. Once the amplitude squared has been calculated and simplified, no more computer algebra system is needed to pursue the calculation. This is the quantity that MARTY can compute automatically, as presented in sample code 56. Kinematics considerations and possible integration on external momenta are left to the user using the numerical library generated by MARTY that contains the quantity between brackets in equation 6.34.

Sample code 56: Squared amplitudes

We first need a transition amplitude

```
auto res = model.computeAmplitude(
       Order::TreeLevel, // or Order::OneLoop
       {Incoming("h"), Outgoing("e"), Outgoing(AntiPart("e"))}
       );
```

Square it

```
Expr square = model.computeSquaredAmplitude(res);
```

Displaying the result in standard output

```
cout << square << endl;
```

Note The squared amplitude is a simple CSL expression that can be used directly.

See also Documentation of Model.

Interference and virtual corrections MARTY can in general compute a squared amplitude from two different amplitudes $\mathcal{M}_1$ and $\mathcal{M}_2$, namely

$$\mathcal{M}^2_{cross} = \mathcal{M}_1 \times (\mathcal{M}_2)^* , \tag{6.35}$$

where $\mathcal{M}_1$ and $\mathcal{M}_2$ must have the same external particles. If $\mathcal{M}_1$ and $\mathcal{M}_2$ are different the contribution to the corresponding physical quantity is therefore

$$\mathcal{M}^2_{cross} + \left(\mathcal{M}^2_{cross}\right)^* . \tag{6.36}$$

This feature allows us to calculate selected interference terms ($\mathcal{M}_1$ and $\mathcal{M}_2$ are calculated with the same perturbation theory order but contain different diagrams) or virtual corrections ($\mathcal{M}_1$ and $\mathcal{M}_2$ contain the same diagrams but are calculated at tree-level and one-loop respectively). This is presented in sample code 57.

Sample code 57: Virtual corrections

Consider the $h \to e^+e^-$ process, calculating separately the tree-level and one-loop amplitudes

```
auto ampl_tree = model.computeAmplitude(
    Order::TreeLevel,
    {Incoming("h"), Outgoing("e"), Outgoing(AntiPart("e"))}
);
auto ampl_loop = model.computeAmplitude(
    Order::OneLoop,
    {Incoming("h"), Outgoing("e"), Outgoing(AntiPart("e"))}
);
```

Obtain the tree-level squared amplitude, virtual correction and full 1-loop contribution:

```
Expr square_tree = model.computeSquaredAmplitude(ampl_tree);
Expr M1_M2star = model.computeSquaredAmplitude(ampl_tree, ampl_loop);
Expr virtual_corr = M1_M2star + GetComplexConjugate(M1_M2star);
Expr square_loop = model.computeSquaredAmplitude(ampl_loop);
```

Note The same principle can be applied with amplitudes containing different diagrams to calculate a specific interference term.

See also Documentation of Model.

6.6 Decay widths

The decay width of a particle ϕ can be calculated from the squared amplitude $\phi \to X$ where X can be any final state. The total decay width is therefore expressed as the sum over all the partial decay widths with external states X_i:

$$\Gamma_\phi = \sum_i \Gamma_{\phi \to X_i}, \tag{6.37}$$

where the partial decay widths $\Gamma_{\phi \to X_i}$ are proportional to the squared amplitude of the $\phi \to X_i$ process.

Considering energy conservation, the partial decay width is non-zero if and only if the total mass of the external particles is below the mass of ϕ. If a final state X_i is the set of particles $\{\phi_1, \cdots, \phi_n\}$, the partial decay width can be expressed as

$$\Gamma_{\phi \to X_i} \propto \begin{cases} 0 \text{ if } m_\phi < \sum_j m_{\phi_j}, \\ |\mathcal{M}(\phi \to X_i)|^2 \text{ otherwise.} \end{cases} \tag{6.38}$$

In this section we do not consider the calculation of partial decay widths as they can be obtained straight-forwardly from squared amplitude calculations (see section 6.5).

Since `MARTY-1.5` it is possible to calculate the leading order of the total decay width, namely:

$$\Gamma_\phi^{LO} = \sum_{i,j} \Gamma^{tree}_{\phi \to \phi_i \phi_j}, \tag{6.39}$$

where the partial decay widths are calculated at the tree-level. In the above formula, loop-level diagrams and $1 \to n$ decays with $n \geq 3$ are neglected.

Without third-party libraries providing a higher precision for decay widths, this feature allows `MARTY` to calculate the full spectrum at the tree-level including the widths and only depend on the model input parameters. The procedure to calculate decay widths in `MARTY` is presented in sample code 58.

Sample code 58: Decay widths

In the SM, calculate the Higgs or Z width:

```
Expr Gamma_H = model.computeWidth(Order::TreeLevel, "h");
Expr Gamma_Z = model.computeWidth(Order::TreeLevel, "Z");
```

Note The decay widths are simple `CSL` expressions.

See also Documentation of Model.

It is possible to fully automate the width calculation in a `MARTY` model. When using the automated width calculation, `MARTY` loops over all particles and calculates the width for each of them. When generating a numerical library from this model, the particle widths will be automatically integrated in the spectrum generator and computed in the spectrum calculation. This is presented in sample code 59.

Sample code 59: Automated decay widths

Compute and update the widths of all particles:

```
model.computeModelWidth(Order::TreeLevel);
```

Generate the spectrum as shown in section 7.1.2

```
Library lib('myLibrary');
lib.generateSpectrum(model);
// ... add functions
lib.print();
```

Note With the width calculation and the spectrum library generation, the generated library will automatically update the decay widths for all particles upon spectrum calculation.

See also Chapter 7 for more details on library generation.

6.7 Wilson coefficients

6.7.1 Definitions

Wilson coefficients are complex numbers in front of operator structures in `MARTY`. Here we are considering only the Leading Order (LO), there is then no subtlety in the matching such as Operator mixing and Renormalization Group Equations (RGE) at Next to Leading Order (NLO), that will come in a future version of `MARTY`. In Effective Field Theories (EFT), amplitudes are the matrix element of an effective Hamiltonian

$$\mathcal{H}_{eff} \equiv \sum_i C_i \hat{\mathcal{O}}_i, \tag{6.40}$$

with $\hat{\mathcal{O}}_i$ effective operators and C_i their Wilson coefficients. The transition amplitude between an initial state i and a final state f is defined quantum mechanically as the matrix element of this Hamiltonian:

$$i\mathcal{M}(i \to f) = \langle f|(-i\mathcal{H}_{eff})|i\rangle = -i\sum_i C_i \langle f|\hat{\mathcal{O}}|i\rangle. \tag{6.41}$$

The operator matrix elements $\langle f|\hat{\mathcal{O}}|i\rangle$ may not in general be calculated pertubatively and contains long distance effects. However the BSM dependence lies in the Wilson coefficient and a perturbative calculation is enough to determine its value as explained in [31]. In this case, a matrix element is simply a particular contraction of external fields. The general case for an amplitude with N external fields $\{\Phi_I^{\{A_I\}}\}_I$ with indices $\{A_I\}$ can be written as

$$i\mathcal{M} = -i\alpha \sum_i C_i \cdot T_i^{\{A_1\}\cdots\{A_N\}} \cdot \Phi_1^{\{A_1\}} \cdots \Phi_N^{\{A_N\}}, \tag{6.42}$$

with $T_i^{\{A_1\}\cdots\{A_N\}}$ all different external fields contractions in the resulting amplitude and α a convention dependent constant. Multiplying the result by i Wilson coefficients can be identified in front of different matrix elements.

When asked, `MARTY` will decompose an amplitude in the different external field contractions it encounters and give the coefficients in front, taking into account a global user-defined factor α. There is for now no specific treatment for well-known Wilson coefficients, `MARTY` only simplifies at most the amplitude decomposing it in an operator basis, letting the task to identify Wilson coefficients to the user. The particular case of magnetic operators (fermion-fermion-vector) is treated explicitly in `MARTY` as the operator structure used in the literature cannot appear in amplitude otherwise. This operator reads for a $\psi_1 \to \psi_2 A$ process

$$\bar{\psi}_2 \sigma^{\mu\nu} T^A P_R \psi_1 F^A_{\mu\nu} \tag{6.43}$$

and its chirality counter part

$$\bar{\psi}_2 \sigma^{\mu\nu} T^A P_L \psi_1 F^A_{\mu\nu}, \tag{6.44}$$

with T^A a group generator in general, that can be trivial. $F^A_{\mu\nu}$ is the field strength of A in momentum space

$$F^A_{\mu\nu} \equiv i(q_\mu \epsilon^A_\nu(q) - q_\nu \epsilon^A_\mu(q)) \tag{6.45}$$

for a vector boson of momentum q and polarization vector ϵ, and the σ matrix defined as

$$\sigma^{\mu\nu} \equiv \frac{i}{2}\left[\gamma^\mu, \gamma^\nu\right]. \tag{6.46}$$

6.7.2 Wilson coefficient extraction

In order to obtain Wilson coefficients using `MARTY`, there are four steps to follow:

- Options setup.
- Amplitude calculation, including the decomposition on an operator basis.
- Definition of the operator of which the coefficient must be extracted.
- Extraction of the coefficient.

The first two steps are similar to the amplitude calculation and are presented in sample code 60. For 4-fermion operators, the order of external fermions in the operator basis must be user-defined. From the initial order given when defining the external particles, the final order is defined as a permutation of the initial order. Considering a four fermion process $\psi_1\psi_2 \to \psi_3\psi_4$, a fermion order $(2, 0, 3, 1)$ corresponds to operators of the type

$$(\bar{\psi}_3\Gamma^A\psi_1)(\bar{\psi}_4\Gamma^B\psi_2), \tag{6.47}$$

where $\Gamma^{A,B}$ are generalized couplings. The permutation is defined starting at 0, a valid permutation is therefore a permutation of $(0, 1, 2, 3)$. The fact that particles are incoming or outgoing is not relevant for this ordering.

Sample code 60: Wilson coefficient calculation

Definition of the options for the calculation, in particular the global factor defined in equation 6.42 that must be factored out

```
FeynOptions options;
Expr alpha = ...; // Convention-dependent factor e.g. -G_F/sqrt(2)
options.setWilsonOperatorCoefficient(alpha);
```

Calculation of the $b \to s\gamma$ decay:

```
auto wilsons = model.computeWilsonCoefficients(
    OneLoop,
    {Incoming("b"), Outgoing("s"), Outgoing("A")},
    options
);
```

For 4-fermion processes such as $b \to s\mu\mu$, the order of external fermions **has** to be provided

```
options.setFermionOrder({1, 0, 2, 3});
auto wilsons = model.computeWilsonCoefficients(
    OneLoop,
    {Incoming("b"), Outgoing("s"),
    Outgoing("mu"), Outgoing(AntiPart("mu"))},
    options
);
// b s* mu* mu ordered with (1, 0, 2, 3)
// means operators of the type (s* b)(mu* mu)
```

Display the coefficients

```
Display(wilsons);
```

Note An order can also be provided for 2-fermion operators but is not mandatory.

Once the calculation has been performed, it is possible to extract the coefficients in multiple ways. The first one, simple but not practical, is presented in sample code 61.

Sample code 61: General Wilson coefficient extraction 1/2

Considering the variable `wilsons` result of the `computeWilsonCoefficients()` method, it is possible to extract any coefficient in the list by giving its index:

```
Expr C = wilsons[2].coef.getCoefficient();
        // Coefficient of the third operator
```

While this method is very simple, the index of a particular operator cannot be known in advance in general.

Another method exists to extract reliably any Wilson coefficient. The principle is to give to MARTY the expression of the operator of which the coefficient must be extracted. Considering the example of C_9, the effective Hamiltonian is expressed as

$$\mathcal{H}_{\text{eff}} \ni \frac{-4G_F}{\sqrt{2}} V_{ts}^* V_{tb} \frac{e^2}{16\pi^2} C_9 \left(\bar{s}\gamma^\mu P_L b\right) \left(\bar{\mu}\gamma_\mu \mu\right), \tag{6.48}$$

with the global factor, the Wilson coefficient C_9 and the operator[3]

$$O_9 \equiv \left(\bar{s}\gamma^\mu P_L b\right) \left(\bar{\mu}\gamma_\mu \mu\right). \tag{6.49}$$

The global factor has to be provided in the calculation options as presented in sample code 60, and the operator can be created by the user to specify the coefficient that has to be extracted. To create an operator expression, the procedure is very similar to the creation of Lagrangian terms discussed in section 3.2.4. This is shown in sample code 62. The only new feature needed to create an operator is the extraction of the process momenta[4] to define external fields, e.g.:

$$\left(\bar{s}(p_2)\gamma^\mu P_L b(p_1)\right) \left(\bar{\mu}(p_3)\gamma_\mu \mu(p_4)\right). \tag{6.50}$$

[3]While in the literature operators often carry their own global factors, in MARTY an operator is only the contraction of external fields without additional factor.

[4]This step could be ignored in the future because the momenta are not theoretically required to define operators. For technical reasons however, it is for now necessary to provide them.

Sample code 62: General Wilson coefficient extraction 2/2

Here we consider the process $b(p_1) \to s(p_2)A(p_3)$ and present the extraction of the coefficient of $\bar{s}(p_2)\gamma^\mu b(p_1)A_\mu$.

Obtaining the momenta of the process:

```
auto p = wilsons.kinematics.getOrderedMomenta(); // p is a vector
// Momenta are indexed from 0
```

Obtaining the other required object (similar to section 3.2.4):

```
// The fields
auto b = model.getParticle("b");
auto s = model.getParticle("s");
auto A = model.getParticle("A");
// Additional tensors
auto gamma = dirac4.gamma;
// Indices
auto i  = model.generateIndex("C", "b"); // quark color index
auto al = DiracIndices(2);
auto mu = MinkowskiIndex();
```

Creating the operator

```
Expr Op = GetComplexConjugate(s({i, al[0]}, p[1]))*A(mu, p[2])
       *gamma({+mu, al[0], al[1]})*b({i, al[1]}, p[0]);
```

Extracting the coefficient

```
Expr C = getWilsonCoefficient(wilsons, Op);
```

For common operators, built-in functions exist to create them without having to explicitly construct them from scratch. This is the case for dimension-5[5] and dimension-6 operators with 4 fermions. Dimension-5 magnetic operators are defined for two fermions ψ_1, ψ_2 and a vector boson A as

$$O_{\text{mag}} \equiv \left(\bar{\psi_1}(T^A)\sigma^{\mu\nu}\Gamma\psi_2\right) F_{\mu\nu}, \tag{6.51}$$

with (T^A) the algebra generator when relevant, $F_{\mu\nu}$ the field strength of A and

$$\Gamma \in \left\{\mathbb{1}, \gamma^5, P_L, P_R\right\}. \tag{6.52}$$

To define a magnetic operator, a user therefore only has to provide Γ that is one element picked in a set of 4 elements.

Dimension-6 operators with fermions ψ_1, ψ_2, ψ_3 and ψ_4 are defined by

$$O_{d=6} \equiv T_{ijkl}\left(\bar{\psi_1}^i\Gamma^A\psi_2^j\right)\left(\bar{\psi_3}^k\Gamma^B\psi_4^l\right), \tag{6.53}$$

[5]Dimension-5 operators are strictly speaking dimension-4 operators (2 fermions, 1 vector) because MARTY does not add by default any scalar mass in the operator definition. This mass can be user-defined and we still prefer to specify dimension-5 to avoid confusion with the literature.

with this time

$$\Gamma^A, \Gamma^B \in \Big\{ \begin{array}{l} \mathbb{1}, \gamma^5, P_L, P_R, \\ \gamma^\mu, \gamma^\mu\gamma^5, \gamma^\mu P_L, \gamma^\mu P_R, \\ \sigma^{\mu\nu}, \sigma^{\mu\nu}\gamma^5, \sigma^{\mu\nu} P_L, \sigma^{\mu\nu} P_R \end{array} \Big\}, \tag{6.54}$$

with Γ^A and Γ^B contracting to leave no free Minkowski index. The indices i, j, k, and l in equation 6.53 are color indices, contracted by T_{ijkl} that can be of four kinds:

$$\begin{aligned} T_{ijkl} &= \delta_{ij}\delta_{kl}, \\ T_{ijkl} &= \delta_{il}\delta_{kj}, \\ T_{ijkl} &= \delta_{ik}\delta_{jl}, \\ T_{ijkl} &= T^A_{ij}T^A_{kl}. \end{aligned} \tag{6.55}$$

To define a dimension-6 operator, Γ^A, Γ^B and T_{ijkl} must be provided by the user. To make the choice easy, the several possibilities for Γ and T are stored in the enumerations `mty::DiracCoupling` and `mty::ColorCoupling` respectively (they can be found in `include/builtinOperators.h`). The different enumeration elements are presented in tables 6.2 and 6.3.

Enumeration element	Name	Expression
`DiracCoupling::S`	Scalar	$\mathbb{1}$
`DiracCoupling::P`	Pseudo-scalar	γ^5
`DiracCoupling::L`	Left	P_L
`DiracCoupling::R`	Right	P_R
`DiracCoupling::V`	Vector	γ^μ
`DiracCoupling::A`	Axial	$\gamma^\mu\gamma^5$
`DiracCoupling::VL`	Vector left	$\gamma^\mu P_L$
`DiracCoupling::VR`	Vector right	$\gamma^\mu P_R$
`DiracCoupling::T`	Tensor	$\sigma^{\mu\nu}$
`DiracCoupling::TA`	Tensor axial	$\sigma^{\mu\nu}\gamma^5$
`DiracCoupling::TL`	Tensor left	$\sigma^{\mu\nu} P_L$
`DiracCoupling::TR`	Tensor right	$\sigma^{\mu\nu} P_R$

Table 6.2: Dirac couplings possible to define Lorentz couplings for operators in `MARTY`.

Enumeration element	Name	Expression
`ColorCoupling::Id`	Identity	$\delta_{ij}\delta_{kl}$
`ColorCoupling::Crossed`	Crossed	$\delta_{il}\delta_{kj}$
`ColorCoupling::InvCrossed`	Crossed inversed	$\delta_{ik}\delta_{jl}$
`ColorCoupling::Generator`	Generator	$T^A_{ij}T^A_{kl}$

Table 6.3: Color couplings possible to define operators in `MARTY`. See equation 6.53 for the definition of the indices $ijkl$.

To fully define a magnetic or a dimension-6 operator, it is enough to provide the couplings using the enumerations `mty::DiracCoupling` and `mty::ColorCoupling`. An example

of magnetic operator is given in sample code 63 and dimension-6 operators are discussed in sample code 64.

Sample code 63: (Chromo-)Magnetic operators

Considering the result of a Wilson coefficient calculation in a variable `wilsons`, it is possible to extract the coefficient of a magnetic operator by first creating the operator:

```
auto O_mag1 = chromoMagneticOperator(model, wilsons, DiracCoupling::R);
   // e.g. for C_7
auto O_mag2 = chromoMagneticOperator(model, wilsons, DiracCoupling::S);
   // e.g. for (g-2)
```

Finally, the Wilson coefficient extraction:

```
Expr C7 = getWilsonCoefficient(wilsons, O_mag1);
Expr gm2 = getWilsonCoefficient(wilsons, O_mag2);
```

Note The $\sigma^{\mu\nu}$ term in the operator is embedded in the definition and must not be taken into account in the `DiracCoupling` provided as defined in equation 6.51.

Sample code 64: Dimension-6 operators

Considering the result of a Wilson coefficient calculation in a variable `wilsons`, it is possible to extract the coefficient of a dimension-6 operator by first creating the operator:

```
auto O1 = dimension6Operator(model, wilsons,
   DiracCoupling::L, DiracCoupling::R); // (P_L)x(P_R)
auto O2 = dimension6Operator(model, wilsons,
   DiracCoupling::VL, DiracCoupling::V); // (G^mu P_L)x(G_mu)
```

Finally, the Wilson coefficient extraction:

```
Expr C1 = getWilsonCoefficient(wilsons, O1);
Expr C2 = getWilsonCoefficient(wilsons, O2);
```

For a non-identity color coupling in a group named `"C"`:

```
auto O1_crossed = dimension6Operator(model, wilsons,
   DiracCoupling::L, DiracCoupling::R,
   {"C", ColorCoupling::Crossed}
   ); // (P_L)_ij x (P_R)_ji
Expr C1_crossed = getWilsonCoefficient(wilsons, O1_crossed);
```

Since `MARTY-1.5` it is also possible to create easily dimension-5 operators that are not magnetic operators i.e. of the type

$$O_{\mathrm{d}=5} \equiv \left(\bar{\psi}_1(T^A)\Gamma^\mu\psi_2\right) A_\mu^{(A)}, \tag{6.56}$$

where the generator T^A is implicitly provided by `MARTY` when relevant. The coupling Γ^μ is a vector coupling:

$$\Gamma^\mu \in \{\gamma^\mu, \gamma^\mu\gamma^5, \gamma^\mu P_L, \gamma^\mu P_R\}. \tag{6.57}$$

In `MARTY` such an operator can therefore be defined using one coupling in `(V, A, VL, VR)` that are defined in table 6.2. This is presented in sample code 65.

Sample code 65: Dimension-5 operators

A dimension-5 operator (not magnetic) can be obtained using:

```
auto OV = dimension5Operator(model, wilsons,
    DiracCoupling::V);
auto OA = dimension5Operator(model, wilsons,
    DiracCoupling::A);
```

Finally, the Wilson coefficient extraction:

```
Expr CV = getWilsonCoefficient(wilsons, OV);
Expr CA = getWilsonCoefficient(wilsons, OA);
```

6.7.3 Calculation details

In this section we present some calculation details for the extraction of Wilson coefficients. We simply discuss how the extraction of Wilson coefficients is different from a simple identification of terms in an amplitude and how MARTY addresses this challenge.

Factors and signs of the coefficients

When matching the effective theory calculation on the full theory at leading order, only the sign and combinatorial factor of the coefficient must be derived and the complete calculation in the effective theory is not necessary.

MARTY calculates this factor automatically and inserts it into the decomposition of the amplitude. More precisely, the result of a Wilson coefficient calculation in MARTY corresponds to the decomposition

$$i\eta \times i\mathcal{M} \equiv \sum_i C_i \mathcal{O}_i, \tag{6.58}$$

with $i\mathcal{M}$ the initial amplitude that has been multiplied by i and η which contains the sign and combinatorial factor for the calculation. The sign comes from the order of fermions between the amplitude calculation and the operator defined in the effective Hamiltonian. The combinatorial factor is not equal to one when there are identical particles in the effective operator. Considering an effective operator with n identical particles

$$\mathcal{O}_\phi \equiv \phi^n, \tag{6.59}$$

the calculation in the effective theory is proportional to

$$i\mathcal{M}_{\text{eff}} \propto n!, \tag{6.60}$$

which gives a relation for the Wilson coefficient

$$C \propto \frac{i\mathcal{M}_{\text{full}}}{i\mathcal{M}_{\text{eff}}} \propto \frac{1}{n!}. \tag{6.61}$$

This factor, together with the sign of fermion ordering and the multiplication by i are added automatically by MARTY when calculating Wilson coefficients.

Magnetic operators

Magnetic operators of the type

$$\mathcal{O}_{\text{mag}} \propto \left(\bar{\psi}\sigma^{\mu\nu}\Gamma^{A}\psi\right) F_{\mu\nu}, \tag{6.62}$$

do not appear naturally in amplitude calculations. Instead, they can be identified as combinations of

$$\mathcal{O}_i = (A \cdot p_i)\left(\bar{\psi}\Gamma^{A}\psi\right), \tag{6.63}$$

with p_i momenta of the two fermions $\bar{\psi}$ and ψ. The sign of the contribution to the magnetic coefficient depends on the incoming or outgoing nature of p_i with respect to q which is the momentum of the vector boson.

In particular, extracting the coefficient of a magnetic operator built by hand following the method described in sample code 62 will return 0 as the operator is not present itself in the decomposition. Because of this, the `chromoMagneticOperator()` function discussed in sample code 63 implements the relevant relations to obtain the coefficients of magnetic operators in an easy way.

Dimension-6 operators at one-loop

The extraction of the Wilson coefficients of 4-fermion operators at the loop-level is done on the mass shell for the external fermions and requires the calculation of three types of diagrams:

- Box diagrams implying the calculation of 4-point functions.
- Triangle diagrams implying the calculation of 3-point functions.
- Mass correction diagrams implying the calculation of 2-point functions.

Following [32], external momenta are set to zero in the integral propagators to extract the coefficients (except when the corresponding terms are divergent, e.g. in the 2-point function case). This means that integrals are simplified such as the following

$$\int_k \frac{(\not{k}+m)(\not{k}+m)}{(k^2-m_0^2)((k-p_1)^2-m_1^2)((k-p_2)^2-m_2^2)} \longrightarrow \int_k \frac{k^2+m^2}{(k^2-m_0^2)(k^2-m_1^2)(k-m_2^2)}. \tag{6.64}$$

This simplification is performed for all diagrams except the penguins diagrams with a massless vector boson as a propagator (such as the photon) for which the full integral has to be evaluated with the on-shell external momenta and masses.

`MARTY` takes care of separating the calculation to derive each part in the right way and finally extract the coefficient.

The issue with mass correction diagrams One issue arises with flavor changing one-loop mass correction diagrams with a massless vector boson as mediator, such as the one shown in figure 6.5. For such diagrams, 2-point function flavor-changing counter-terms must be explicitly supplemented in the Lagrangian to obtain a correct result. As `MARTY` does not yet support tree-level flavor-changing propagators, we use a redundancy in the penguin calculation to correct a posteriori the pathological contributions. More

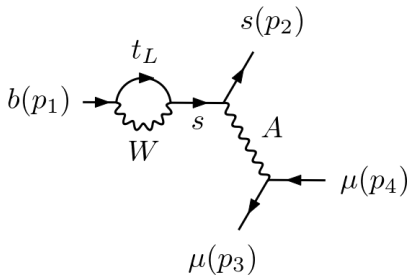


Figure 6.5: Mass correction diagram appearing in the extraction of the Wilson coefficients of 4-fermion operators.

specifically, the penguin amplitude of a $\psi \to \psi A$ process can be decomposed in 6 different operators given by

$$\mathcal{O}i^{(\prime)} \equiv (A \cdot p_i)\bar{\psi}(\gamma^5)\psi \tag{6.65}$$

$$\mathcal{O}_g^{(\prime)} \equiv \bar{\psi}\not{A}(\gamma^5)\psi, \tag{6.66}$$

with $p_i \in \{p_1, p_2\}$ the momenta of the external fermions. When counter-terms are not considered, the contributions to the operators in equation 6.66 are pathological but can be derived from the contributions of the operators in equation 6.65 thanks to a redundancy in the result (see e.g. equation (6.138) of [32]). As the operators in 6.65 are independent of mass corrections they do not suffer from the lack of counter-term and are enough to extract any coefficient for the process.

Consequently, `MARTY` applies a patch in the calculation to solve this particular issue. This is only a temporary solution that will be solved with the introduction of interaction terms with only two legs to implement the appropriate counter-terms. For now, this patch has been validated in multiple examples, different calculations and models.

After the application of the patch, the amplitude becomes proportional to

$$\frac{q^2}{q^2}, \tag{6.67}$$

with a vanishing on-shell $q^2 = 0$ leading to an embarrassing $0/0$ in the numerical program. When calculating diagrams automatically in the general case, it is difficult to identify symbolically such simplifications in general. To prevent this numerical issue, `MARTY` introduces a regulator in the numerator and denominator and the calculation becomes proportional to

$$\frac{0+\epsilon}{0+\epsilon} = 1. \tag{6.68}$$

This regulator is discussed in chapter 7 and is the `reg_prop` parameter in the `param_t` struct.

All the code related to this patch can be found in the files `include/penguinpatch.h` and `src/penguinpatch.cpp`. If one is interested in the exact implementation, the relations used to derive the coefficients of the operators in equation 6.66 can be found in the static `applyPenguinPatch_implementation()` method.

6.7.4 Conclusion on the extraction of Wilson coefficients

When extracting Wilson coefficients, several important issues must be addressed:

- **The operator basis must be clearly defined**. As the bases are convention-dependent, the basis used by `MARTY` will not always corresponds to what can be found in the literature. When extracting a coefficient, one should make sure that the operator basis used to define the coefficient in the literature matches with what `MARTY` calculates, otherwise a work needs to be done to match the two bases and extract the correct contribution.

- **Kinematics must be given on-shell** for magnetic and dimension-6 operators (at least). In particular, a valid set of s_{ij} must be provided respecting momentum conservation and $p^2 = m^2$ for all external particles.

- **The regulator must be set to a non-zero value** for 4-fermion processes with a massless vector boson mediator. Otherwise, the result will diverge. A value of 10^{-3} has been empirically validated to give accurate results.

6.8 Automating calculations

In some phenomenological analysis it is necessary to calculate a large number of similar calculations. In that case, the method using explicit particle names can be tedious and not recommended. Consider calculating all 2-to-2 processes for a model with 50 particles, this is a task that one does not want to do by hand. while the `getParticles()` of a model returns the list of all particles, the method that must be used in that case is `getPhysicalParticles()`. Contrary to the first option, this method removes not physical particles and redundancies. For a Dirac fermion ψ for example the left- and right-handed parts are also considered as particles and will cause redundancy in calculations. The `getPhysicalParticles()` method will take care of it and return a list of independent physical particles. This is also possible to filter even more this list, as summarized in sample code 66.

Sample code 66: Get particles lists from a model

Exhaustive list, but not suited for automating calculations

```
auto particles = model.getParticles();
```

Removing non-physical particles and redundancies

```
auto particles = model.getPhysicalParticles();
```

Filtering physical particles

```
auto fermions = model.getPhysicalParticles(
    [&](Particle p) { return p->isFermionic(); }
);
auto bosons = model.getPhysicalParticles(
    [&](Particle p) { return p->isBosonic(); }
);
auto vectors = model.getPhysicalParticles(
    [&](Particle p) { return (p->getSpinDimension() == 3); }
);
auto Ni = model.getPhysicalParticles(
    [&](Particle p) { return (p->getName()[0] == 'N'); }
);
```

Note The deduced return type is each time `std::vector<mty::Particle>` allowing to iterate over it.

Note The lambda expression given to filter out the result can be any user-defined boolean predicate taking a `Particle` as parameter.

See also The CSL manual for more details on lambda expressions.

Once proper lists of particles have been filtered out from the model, one can iterate over them to automate the calculation of a large number of processes. This is presented in sample code 67.

Sample code 67: Automate a large number of calculations

```
for (auto f : fermions) {
    for (auto v : vectors) {
        auto ampl = model.computeAmplitude(
            Order::TreeLevel,
            {Incoming(f), Incoming(AntiPart(f))),
            Outgoing(v), Outgoing(AntiPart(v))}
        );
        auto squared = model.computeSquaredAmplitude(ampl);
    }
}
```

Chapter 7

Code generation

Theoretical calculations, in particular at the one-loop level, cannot be used as symbolic expressions because they have a very large size and can be complicated. Furthermore, when one uses an automated program the final expression will very often be less simplified than the same result obtained by hand. The advantage is that while a physicist may take weeks to perform the calculation, the program will finish in a few seconds or minutes. The only requirement for the output is that it must be simplified enough to allow one to evaluate it numerically. The numerical evaluation consists in obtaining numbers for the symbolic outputs given the initial values for model parameters. This evaluation is not done in the same program than the symbolic calculation for two reasons:

- The symbolic calculation may be done once and be used several times to scan the parameter space.

- If the numerical evaluation is separated from the main program, it can be independent of `MARTY` and contain compiled functions that run much faster.

For that purpose `MARTY` generates C++ code containing functions that can evaluate the symbolic results calculated by the user.

7.1 Library generation

In this section we describe how libraries can be generated using `MARTY`, i.e. for the results of Wilson coefficients or squared amplitudes for example. This part is done in the main `MARTY` program that performs symbolic manipulations. For details about the generated libraries themselves and the numerical computations, see the next section.

7.1.1 General principles

A C++ library can be generated by `MARTY` as sample code 68 demonstrates. Once created, any scalar quantity can be added to the library as a function returning a complex number. These quantities can be squared amplitudes and Wilson coefficients. For bare amplitudes, one has to decompose it in Wilson coefficients to get all different scalar contributions, as explained in section 6.7.

Sample code 68: Create a C++ library with MARTY

Initializing a library

```
Library lib("library_name", "library_path");
```

Adding functions from symbolic expressions

```
Expr A = someExpr();
Expr B = someOtherExpr();
lib.addFunction("A_function_name", A);
lib.addFunction("B_function_name", B);
```

Make MARTY compile the library

```
lib.build(); // creates and compiles the library
// or simply
lib.print(); // creates the library
```

Make MARTY compile the library using multiple jobs

```
lib.build(4); // compiles with 4 jobs (make -j 4)
```

Note If the path to the library is not given, it is `"."` by default.

See also Documentation of Library in CSL.

One can customize a library as presented in sample code 69, for example adding the path for MARTY and CSL headers and libraries if they are not installed in a standard location, or changing the compiler.

Sample code 69: Customize libraries

Setting the compiler

```
lib.setClangCompiler(); // clang++
lib.setGccCompiler(); // g++
```

Adding include dependencies

```
lib.addIPath("/path/to/include/files");
```

Adding library dependencies

```
lib.addLPath("/path/to/library/files");
```

Note These functions must be called before compiling the library (when calling `.build()`).

Note If one wants to use clang++ instead of g++, the corresponding function should be called to allow MARTY to adapt compiler flags to clang, although it is not recommended.

7.1.2 Spectrum generation

As presented in section 5.5.3, MARTY can diagonalize mass matrices by introducing symbolic masses and mixings in the Lagrangian. Given the numerical values of the initial mass matrices, MARTY can diagonalize in full generality any mass matrix, and get the

spectrum of a BSM model. To add the spectrum generator to a generated library only one line is required as shown in sample code 70.

Sample code 70: Generation of the spectrum generator

Only the spectrum depending on diagonalization

```
lib.applyDiagonalizationData(model);
```

The whole spectrum, including abbreviations

```
lib.generateSpectrum(model);
```

Note The second method includes the first and is therefore more general, generating also the spectrum for abbreviations such as $M_W = \frac{1}{2}gv$ (in the form of a callable function depending on the parameters).

Note If the model widths have been calculated as shown in 6.6 the spectrum generator also includes the particle decay widths.

Let us consider a simple example, a 2×2 mass matrix

$$M = \begin{pmatrix} a & c\tan\theta \\ c\tan\theta & b \end{pmatrix}. \tag{7.1}$$

This mass is diagonalized symbolically through

$$U^\dagger M U = D, \tag{7.2}$$

with

$$U \equiv \begin{pmatrix} U_{11} & U_{12} \\ U_{21} & U_{22} \end{pmatrix}, \tag{7.3}$$

$$D \equiv \begin{pmatrix} m_1 & 0 \\ 0 & m_2 \end{pmatrix}. \tag{7.4}$$

In general, the model parameters such as a, b, c and θ in the example above must be provided as input, and the diagonalization procedure derives the final masses and mixings that are necessary to evaluate perturbative quantities. The numerical diagonalization is performed in the generated library as discussed in section 7.2.3

7.1.3 LHA Reader

There is a module of `MARTY` able to read files following Les Houches Accord (LHA) [33, 34]. The purpose of this module is to be able to read easily a LHA file, being for input or output parameters. Documentations of files lhaData.h, lha.h and lhaBlocks.h are comprehensive allowing one to handle easily this module. Sample code 71 shows the basics about this LHA reader.

Sample code 71: LHA Reader, a SUSY example

Reading a file `example.lha`

```
auto data = lha::Reader::readFile("example.lha");
```

Getting a full block in the data structure

```
auto U = data.getValues("Umix"); // 2x2 chargino mixing
cout << "Umix␣=␣[[" << U[0] << ",␣" << U[1] << "],␣["
     << U[2] << ",␣" << U[3] << "]]\n";
```

Getting a single value in the data structure from its id

```
long double mC1 = data.getValue("MASS", 1000024); // Chargino1 mass
long double mC2 = data.getValue("MASS", 1000037); // Chargino2 mass
cout << "mC1␣=␣" << mC1 << endl;
cout << "mC2␣=␣" << mC2 << endl;
```

See also Documentation of class LHAFileData for more details.

To import this module into a generated library, one must give the path to `MARTY`'s source files (for example `/home/MARTY-1.2/marty`) as presented in sample code 72.

Sample code 72: Importing the LHA module in a library

```
lib.importLHAModule("path-to-marty-sub-directory");
```

Warning One should not be confused between the project root directory and the `marty` sub-directory that has to be given.

7.2 The generated libraries

In this section we describe how to use the libraries that are generated by `MARTY`. They are independent of `MARTY` itself or `CSL` (although they use some header-only part of `CSL`, the `CSL` library has not to be linked).

7.2.1 Layout

Libraries that `MARTY` generates have always the same form, presented in figure 7.1. The

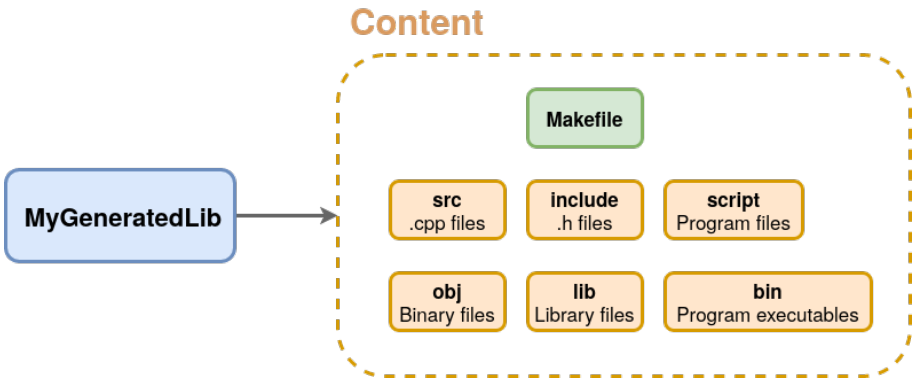


Figure 7.1: Content of the C++ libraries generated by `MARTY`.

Makefile will automatically, by typing `make`, compile all source files in the `src/` directory (spectrum generator, results of symbolic calculations, corresponding header files are in `include/`) and generate object files in `obj/`. The `make` command will also compile user programs placed in `script/`. An example of such a program is generated automatically, called `example_libname.cpp`. One can modify this file (the library is already included), and then type

```
$ make
$ bin/example_libraryname.x
```

to build and execute the program. This procedure can be applied to any `cpp` file put in this directory.

7.2.2 The `param_t` structure

The `param_t` structure contains all the parameters (real or complex numbers) used in the library as public attributes, including all the parameters related to the spectrum generator. `param_t` is defined in the file `include/params.h`. This structure is used as **unique and universal parameter** for all functions generated in the library, i.e. for the results of Wilson coefficients and squared amplitudes.

The parameters are not directly numbers such as `double` but are instead constructed as `csl::InitSanitizer` objects. `InitSanitizer` is a template class designed to prevent uninitialized values to be used in the program. As the number of parameters in the structure quickly grows for complex models, it is easy to forget to initialize one parameter and obtain wrong results. For this reason, the template class `InitSanitizer<T>` encapsulates a parameter (real or complex) and is implicitly convertible to the value it contains. It allows a user to use it in the same way as a standard variable but ensures that the underlying value is initialized when it is used, otherwise an error is raised. This is summarized in sample code 73.

Sample code 73: Generalities on the parameters

Creating the `param_t` struct

```
param_t params; // All parameters are uninitialized
```

Considering two real parameters `m_1` and `m_2` in the structure

```
params.m_1 = 4.5; // initializing m_1
double m_1_conv = params.m_1; // Implicit conversion
params.m_1 = 2*params.m_1 + 1; // Operations are also possible
cout << params.m_1 << endl; // >> 10.0
cout << params.m_2 << endl; // >> Error: m_2 has not been initialized !
```

Converting explicitly to the underlying value

```
double m_1 = params.m_1.get();
```

Note The explicit conversion with `.get()` must be used when the implicit conversion is not possible. For most purposes, the implicit conversion is enough.

The `param_t` struct also has public methods:

- `print()` writes the content of the structure i.e. all parameters and their values. This function can be called with uninitialized parameters for which the value is replaced

by the word `uninitialized`. With a `params_t params` object, it can be called simply with `params.print();`.

- `reset()` resets all parameters to uninitialized values. This is handy to ensure that old values will not be used between two logically distinct uses of the structure. By resetting the structure, it is guaranteed that all parameters used next are properly initialized (otherwise an error is raised). With a `params_t params` object, it can be called simply with `params.reset();`.

Special parameters There are two parameters that can appear in `param_t` independently of the model:

- `Finite`: This parameter is real and is supplemented by `MARTY` in front of each local term in one-loop results, i.e. in front of the finite part coming from local terms in dimensional regularization. This can be set to 1 by default, or 0 to remove local terms. See also the LoopTools manual [2] for more information.

- `reg_prop`: This parameter is real and is added by `MARTY` in propagators to avoid divergences. In most cases this parameter can be set to 0. An example is for Wilson coefficients in 4-fermion processes for which it can be necessary to set a small value for this parameter (such as `1e-3` for example) to avoid a $\frac{0}{0}$ that naturally appears in the result and is difficult to address at the symbolic level.

7.2.3 Spectrum generation

Files `src/global.cpp` and `include/global.h` contain the spectrum generator. The spectrum calculation always goes by three steps:

- **Initialization of input parameters** for the spectrum (mass matrices for the diagonalization or other input parameters of mass functions such as $M_W = \frac{1}{2}gv$ for which g and v must be defined).

- **Calculation of the spectrum** by the library. Input and output parameters are all in the `param_t` structure.

- **Calculation of theoretical quantities** with the spectrum that has been calculated.

The user interface to update the spectrum from the model parameters is presented in sample code 74.

Sample code 74: Spectrum generation

Updating masses and mixings by diagonalizing mass matrices

```
updateDiagonalization(params);
```

Updating (scalar) masses and widths

```
updateMassExpressions(params);
```

Updating the entire spectrum

```
updateSpectrum(params);
```

Note In this example, `params` is a `param_t` object of which all input parameters for the spectrum have been initialized.

Note The `updateSpectrum()` function is equivalent to the combination of `updateDiagonalization()` and `updateMassExpressions()`.

7.2.4 Meta-programming features

While the libraries automatically generated by `MARTY` are hard-coded, there are features allowing to automate several tasks, with the parameters in `param_t` or the functions.

It is possible to loop over the functions in the library. Furthermore, to each parameter in `param_t` and function is associated a name that can be used to recover the object at run-time. In particular, from a character string it is possible to obtain a parameter or a function. These features can prevent having to write hard code to use a large number of parameters or functions. Hence, by choosing wisely the names given to functions and parameters when using `MARTY` to generate the library, it is possible to greatly facilitate the use of the library. The way to loop over functions is presented in sample code 75 and the way to access functions or parameters with their names is shown in sample code 76.

Sample code 75: Looping over functions

`f_G` is the collection of all functions in the library and can be found in the file `include/group_g.h`. It is possible to iterate over it and select functions depending on their names (attribute `name` of type `char const*`).

```
param_t params; // Must be initialized
for (const auto &f : f_G) {
    if (f.name[0] == 'p') // If the function name starts with p
    {
        complex<double> res = f(params);
        // Do something with res
    }
}
```

Note It is also possible to loop over `f_G` with an index, using `f_G.size()` and `f_G[i](params)`.

Sample code 76: Accessing parameters and functions

Knowing the name of a parameter or function, it is possible to recover the underlying object. In the following, we consider a `param_t params` structure with two parameters `m_1` and `m_2`, and a function named `M2` (e.g. calculating a squared amplitude).

Obtaining a function from its name

```
auto f = fmap_G["M2"];
complex<double> res = f(params);
// do something with res
```

Obtaining a parameter from its name

```
params.m_1 = 1;
// for a real param:
auto m_1_from_name = params.realParams["m_1"];
// // for a complex param:
// auto m_1_from_name = params.complexParams["m_1"];
*m_1_from_name = 2;
cout << params.m_1 << endl; // >> 2
```

Note The parameter obtained from the map is a pointer so that it is possible to directly access the object stored in the `param_t` struct. It is therefore necessary to use `*`.

Chapter 8

Options

8.1 The FeynOptions class

This class can be given when asking for an amplitude to customize the output. It is well documented concerning general options and filters that are briefly discussed in this section. Its purpose is to provide a simple way to customize the result for one particular amplitude (or more), avoiding then global options. The basic use of `FeynOptions` is introduced in sample code 77, for detailed explanations see the documentation of FeynOptions.

Sample code 77: Using custom options for amplitudes

Creating the `FeynOptions` object and using it for an amplitude calculation

```
FeynOptions customOptions;
// Set different options, in general looking like:
// customOptions.someSetterFunction(...); // typically filters
// customOptions.someAttribute = ...; // typically local options
auto res = model.computeAmplitude(
    Order::OneLoop,
    {Incoming("h"), Outgoing("A"), Outgoing("A")},
    customOptions // send the options to this function,
                  // always as last parameter
);
```

8.1.1 Local options

Local options are a set of public boolean variables mimicking global options that are detailed in section 8.2. When the `FeynOptions` object is created, all these options are copied from the global ones and can then be modified in the object. This feature allows to have the choice to:

- Modify global options to make the behavior of all following calculations similar.
- Modify local options, in the `FeynOptions` object, to affect only the calculations to which this set of options is given.

For more information about local options, see the the documentation.

8.1.2 Filters

One of the main use cases of the `FeynOptions` class are filters. They are functions, boolean predicates[1] allowing `MARTY` to filter out parts of an amplitude calculation. There are two types of filters:

- Feynman rules filters are applied before a calculation, this prevents a lot of calculations because there is less diagrams to search for, and to calculate. Those filters must take a `InteractionTerm` (see the documentation) as parameter and return a boolean, `true` if the term must be kept.

- Feynman diagrams filters are applied before returning the amplitude result to the user. This option must not be preferred as the calculation goes further before the filter is applied, however it is necessary for complex filtering behaviors. Those filters must take a `FeynmanDiagram` (see the documentation) as parameter and return a boolean, `true` if the diagram must be kept.

Filters are designed to be user-defined, and `MARTY` of course provides some common built-in ones. A filter must return `true` if the object (interaction term, diagram) must be **kept**, and `false` otherwise. Filters are implemented in the file filters.h and defined in `mty::filter`.

Sample code 78: Apply filters in amplitudes

From a lambda expression

```
FeynOptions options;
options.addFilters(
    // Disabling the Z boson
    [&](InteractionTerm const &term) {
        return !term.contains(model.getParticle("Z"));
    },
    // Forwing the W boson in diagrams
    [&](FeynmanDiagram const &diagram) {
        return diagram.contains(model.getParticle("W"));
    }
);
```

Using built-in filters

```
FeynOptions options;
options.addFilters(
    filter::disableParticle("Z"),
    filter::forceParticle("W")
);
```

See also filters.h for more details about built-in filters.

See also Documentation of InteractionTerm and FeynmanDiagram to learn how to get relevant physical information from those objects.

[1] A predicate is a function taking a priori any kind of arguments to return a boolean value, `true` or `false`.

8.1.3 Amplitude selection

In case all contributions must be calculated but the user wants to separate them, it is possible to apply filters on the full amplitude to extract only a part of it in a similar way as in sample code 78. This is shown in sample code 79.

Sample code 79: Select parts of amplitudes

From an amplitude calculation such as the electron self-energy in the SM

```
auto ampl = model.computeAmplitude(
    OneLoop,
    {Incoming("e"), Outgoing("e")}
);
```

Filter out only e.g. the W or Z contributions:

```
auto ampl_W = ampl.filterOut(
    [&](FeynmanDiagram const &diagram) {
        return diagram.contains(model.getParticle("W"));
    }
);
auto ampl_Z = ampl.filterOut(
    [&](FeynmanDiagram const &diagram) {
        return diagram.contains(model.getParticle("Z"));
    }
);
```

Note `ampl_W` and `ampl_Z` are regular amplitude objects that contain a part of `ampl` and do not require any additional calculation.

See also Documentation of FeynmanDiagram to learn how to get relevant physical information from it.

8.2 Global options

All the options presented in the following are boolean variables defined in the file mrtOptions.h or in `CSL`. They can be changed at any time to customize `MARTY`'s calculations and outputs.

8.2.1 `CSL` options

`CSL` also has a similar option system that is detailed in its manual. `CSL` options lie in the namespace `csl::option::` while `MARTY` options are in the namespace `mty::option::`. This will cause ambiguity for the compiler and users will have to specify the library namespace when using an option as demonstrated in sample code 80.

Sample code 80: Set up options

Modifying CSL options

```
csl::option::printIndexIds = false;
```

Modifying MARTY options

```
mty::option::excludeTadpoles = true;
```

Note The library namespace `csl::` and `mty::` have to be specified for options as both libraries use a `option::` namespace.

See also The CSL manual for details about CSL options.

8.3 General options

showDiagrams If `true`, *GRAFED* will be launched after each amplitude calculation by default, except for Feynman rules. If not defaulted, diagrams display can be triggered by calling the function `Show()` as presented in section 6.4.

Default value: `false`.

amputateExternalLegs External legs can be removed from an amplitude calculation. This is equivalent to calculate an effective Feynman rule for a given process, using the vertex in an effective theory for example. Users should know however that as equations of motions cannot be applied without external legs, results may be less simplified than in the standard calculation.

Default value: `false`.

displayAbbreviations By default, the list of all abbreviations used by MARTY will be displayed together with results when calling the `Display()` function presented in section 6.4. This option can be disabled.

Default value: `true`.

displayIntegralArgs As demonstrated in section 6.4.4, MARTY introduces scalar quantities deriving form one-loop integrals, namely quantities such as $C_{ij}(p_i, m_j)$ depending on momenta and masses in the loop. Arguments are numerous and in general prevent one to correctly read a symbolic result. They are then by default not shown, and MARTY simply prints C_{ij}.

Default value: `false`.

diagonalizeSymbolically If set to `true`, irreducible mass matrices with more than two fields will be diagonalized introducing symbolic masses and mixings to help later numerical diagonalization (see section 5.5.3) when calling the `diagonalizeMassMatrices()` method of the ModelBuilder class.

Default value: `false`.

8.3.1 Amplitude calculation options

simplifyAmplitudes All the simplifications presented in section 6.4 can be disabled if one wants to see a bare amplitude expression. Without simplifications, the amplitude corresponds to application of Feynman rules.[2]

Default value: `true`.

simplifyConjugationMatrix The conjugation matrix $C = i\gamma^0\gamma^2$ arises when one uses Majorana fermions or fermion-number violating interaction as chargino-fermion-sfermion interactions in the MSSM. This matrix should not appear in final results for amplitudes, as it can always be simplified out by standard prescriptions. This simplifications can be disabled if one wants to see the explicit conjugation dependence of the amplitude.

Default value: `true`.

discardLowerOrders When calculating an amplitude at the one-loop level, tree-level diagrams are ignored by default. If one wants to get the full results, tree-level included, this option can be set to `false`.

Default value: `true`.

excludeTadpoles If set to `true` tadpole diagrams, i.e. rank 1 1-point functions with one unique external leg, are ignored in amplitudes.

Default value: `false`.

computeFirstIntegral If set to `false`, the tensor reduction introducing the scalar quantities presented in section 6.4.4 is not performed by `MARTY`, and one can get a result with an explicit momentum integral at the one-loop level.

Default value: `true`.

searchAbreviations If set to `false`, many (but not all) abbreviations will be disabled in calculations. This is not recommended, in particular for large results, but can be used in simple toy models to have more verbose results that do not require to read abbreviation values.

Default value: `true`.

abbreviateColorStructures By default color structures left in amplitudes, that could not be simplified, are stored in dedicated abbreviations. This option can be disabled.

Default value: `true`.

[2]Some simple tensor simplifications, index contractions, are still done by `MARTY` in any case.

decomposeInOperators Amplitudes can be decomposed in an operator basis the same way as it is done for the calculation of Wilson coefficients (see section 6.7). By default this option is `false`, and one term of the amplitude corresponds to one Feynman diagram. If `true`, terms are merged and factored by operator structures. For processes with many diagrams and few different operators, it is recommended to set this option to `true`, as it will considerably lower the number of terms in the amplitude. For the calculation of squared amplitudes for example, it can be useful.

Default value: `false`.

applyEquationsOfMotion If set to false, equations of motion for spin 1 and spin 1/2 particles will not be applied. Note that one can also disable equations of motion for a particular field giving it as an off-shell insertion as demonstrated in section 6.4.1.

Default value: `true`.

addLocalTerms If set to `false`, local terms described in section 6.4.4 are not added in the amplitude. This can be used for example to get only the divergent part of an integral, to obtain NLO corrections for example.

Default value: `true`.

Chapter 9

Built-in models

There is in MARTY several built-in models. We did not do extensive tests on all of them yet. One may use them and do calculations in them, but keeping in mind that Feynman rules must should be checked before using the output of MARTY, in particular for large models such as the Standard Model and its extensions. Users may at any time create the models presented in the following, and show the content to be able tu use them as this user manual presents. The method to display a model is presented in sample code 81.

Sample code 81: Display models

Displaying a model in standard output

```
cout << model << endl;
```

Note If the model is very large, one can redirect the standard output to a file writing `./program.x > data.txt` for example.

There is two types of models. For all models up to the 2HDM, they are in the form of C++ code doing model building as presented in chapter 5. These models are in directories

```
marty/include
marty/src
```

for header and source files respectively. Models can also be in the form of C++ code generated automatically. For large models that takes several minutes / hours to built from scratch[1], we generated automatically the C++ code corresponding to the final Lagrangians to save time at the beginning of a program using them. These models are in directories

```
marty/models/include
marty/models/src
```

for header and source files respectively, and for now concerns the pMSSM and uMSSM (respectively phenomenological and unconstrained Minimal Super-symmetric Standard Model). The pMSSM for example takes 15 min to build from scratch, but can be loaded entirely in about 2 sec using the code generated automatically. As the code is not written directly by a human, it can be less pleasant to read.

For simple models there is also `.json` files defining model gauges and particle contents. This feature is not used anymore because MARTY's interface for model building is more practical than in the past. This is why the present manual does not assess the `.json` model files while some of the following models still use this kind of inputs.

[1] From a high energy Lagrangian, breaking symmetries, making replacements etc.

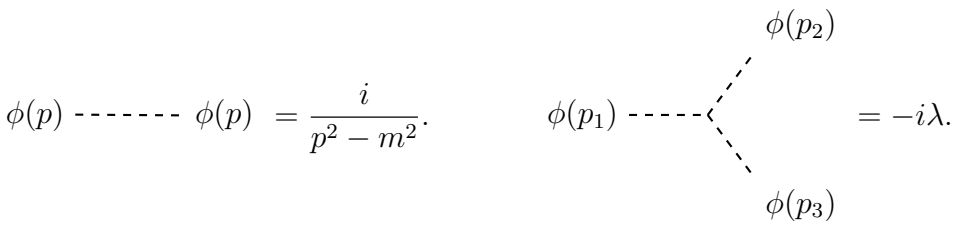
9.1 Simple models

9.1.1 Scalar theory

The theory presented here consists in a unique **scalar** field ϕ, real (self-conjugate). Supposing that the field has a mass m, the Lagrangian is

$$\mathcal{L}_\phi = \frac{1}{2}\partial_\mu\phi\partial^\mu\phi - \frac{1}{2}m^2\phi^2 - \frac{\lambda}{3!}\phi^3. \tag{9.1}$$

Here λ is as a small parameter that we can develop in perturbation theory. The 3! is a combinatorial factor corresponding to the permutation symmetry of the three powers of ϕ in the interaction term. It is conventional and it allows to get a simple Feynman rule for the vertex. Feynman rules for this theory are presented in figure 9.1. A sample using



(a) Propagator **(b)** 3-vertex

Figure 9.1: Feynman rules for the scalar ϕ^3 theory.

this model can be found in

`scripts/sampleScalar.cpp`

9.1.2 Scalar QED

The so-called scalar QED (sQED) is composed in its simpler form by a massive charged scalar (ϕ) coupled to a photon-type particle, i.e. a mass-less spin 1 boson (A^μ) gauging a U(1) symmetry group. The lagrangian reads

$$\mathcal{L}_{\text{sQED}} = (D_\mu\phi)^* D^\mu\phi - m^2\phi^*\phi - \frac{1}{4}F^{\mu\nu}F_{\mu\nu}, \tag{9.2}$$

where

$$D_\mu\phi \equiv (\partial_\mu - igc(\phi)A_\mu)\phi, \tag{9.3}$$
$$F_{\mu\nu} \equiv \partial_\mu A_\nu - \partial_\nu A_\mu \tag{9.4}$$

with g the $U(1)$ gauge coupling, and $c(\phi)$ the charge of ϕ under this symmetry. Writing explicitly the interaction Lagrangian, we get (setting a -1 charge for ϕ)

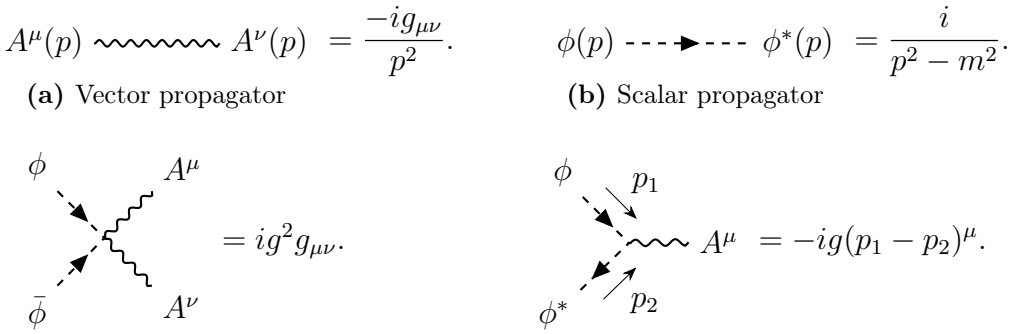
$$\mathcal{L}_{\text{int}} = g^2 A_\mu A^\mu\phi^*\phi + igA^\mu(\phi\partial_\mu\phi^* - \phi^*\partial_\mu\phi). \tag{9.5}$$

Feynman rules for sQED are shown figure 9.2. This model can been found in

`models/files/sQED.json`

and a sample program using it in

`scripts/sampleSQED.cpp`



(a) Vector propagator **(b)** Scalar propagator

(c) 4-vertex **(d)** 3-vertex

Figure 9.2: Feynman rules for the scalar QED model.

9.1.3 QED

The QED in its simpler form describes one charged fermion ψ_e (spin 1/2) carrying an electric charge of -1. The gauge group is composed of one $U(1)$ group. This group is abelian and has one generator, the photon A^μ in our case, that interacts with the electron through the covariant derivative. The Lagrangian is

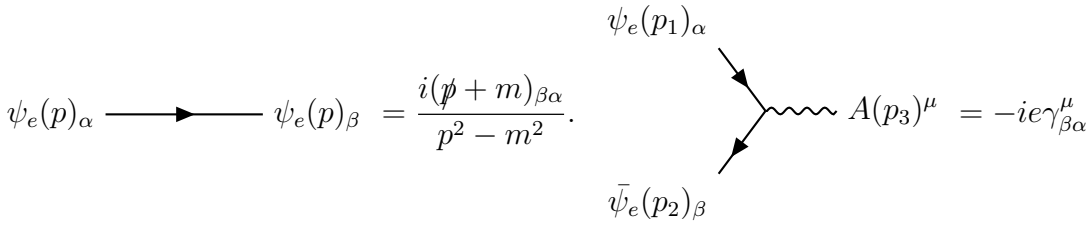
$$\mathcal{L}_{\text{QED}} = \bar{\psi}_e \left(i \not{D} - m_e \right) \psi_e - \frac{1}{4} F_{\mu\nu} F^{\mu\nu}, \tag{9.6}$$

or more explicitly

$$\mathcal{L}_{\text{QED}} = \bar{\psi}_{e\alpha_1} i \gamma^\mu_{\alpha_1\alpha_2} \partial_\mu \psi_{e\alpha_2} - m_e \bar{\psi}_{e\alpha} \psi_{e\alpha} - \frac{1}{4} F^{\mu\nu} F_{\mu\nu} - ie \bar{\psi}_{e\alpha_1} \gamma^\mu_{\alpha_1\alpha_2} A_\mu \psi_{e\alpha_2}, \tag{9.7}$$

with m_e the mass of the electron (the photon is mass-less), e the electromagnetic constant, and $F_{\mu\nu}$ the field strength of A_μ as usual.

The photon has the same propagator as in scalar QED (see figure 9.2). Feynman rules specific to QED are presented in 9.3. QED can been found in `MARTY`'s code in



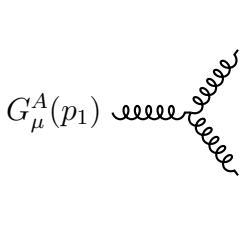
(a) electron propagator **(b)** Vertex

Figure 9.3: Feynman rules for the electron in QED. The photon has the same propagator as in scalar QED. In contrary to scalar QED the interaction vertex does not depend on any impulsion.

```
include/QED.h
src/QED.cpp
```

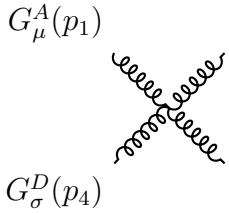
or in a sample code

```
scripts/sampleQED.cpp
```



$$= gf^{ABC}[\quad g_{\mu\nu}(p_1-p_2)_\rho + g_{\mu\rho}(p_3-p_1)_\nu + g_{\nu\rho}(p_2-p_3)_\mu \quad].$$

(a) 3-gluon vertex



$$= ig^2[\quad f^{EAB}f^{ECD}(g_{\mu\sigma}g_{\nu\rho}-g_{\mu\rho}g_{\nu\sigma}) + f^{EAC}f^{EBD}(g_{\mu\sigma}g_{\nu\rho}-g_{\mu\nu}g_{\rho\sigma}) + f^{EAD}f^{EBC}(g_{\mu\rho}g_{\nu\sigma}-g_{\mu\nu}g_{\rho\sigma}) \quad].$$

(b) 4-gluon vertex

Figure 9.4: Feynman rules for gluon auto interactions. All momenta in the 3-vertex are by convention incoming (towards the vertex). The 4-gluon vertex is suppressed by an additional power of the coupling constant g with respect to the 3-gluon vertex.

9.1.4 QCD

$SU(3)$ has eight generators for eight gluons, that interact with quarks (that have a non trivial representation of SU(3)) and themselves. Gluon auto-interactions come from the non-abelian property of the group, i.e. generators do not commute anymore:

$$\left[T^A, T^B\right] \equiv if^{ABC}T^C, \quad f^{ABC} \neq 0, \tag{9.8}$$

with T^A the eight SU(3) generators and f^{ABC} its structure constants. In the abelian case, U(1) for example, there is only one generator that obviously commutes with itself,

$$[T, T] = 0. \tag{9.9}$$

The gluon themselves have then a covariant derivative in order to keep the kinetic term gauge invariant, and in particular

$$G^A_{\mu\nu} \equiv \partial_\mu G^A_\nu - \partial_\nu G^A_\mu + f^{ABC}G^B_\mu G^C_\nu, \tag{9.10}$$

with G^A_μ the gluon field of SU(3) index A going from 1 to 8, f^{ABC} structure constants of the SU(3) group.

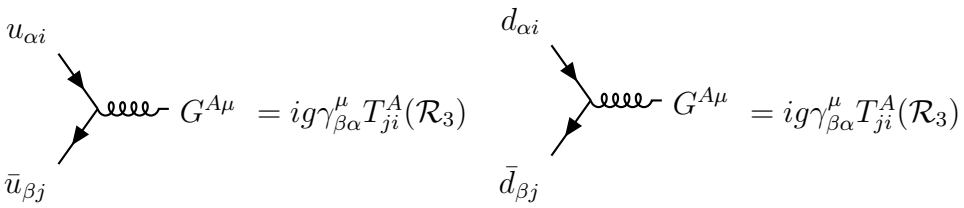
$$\mathcal{L}_{\text{QCD}} = -\frac{1}{4}G^A_{\mu\nu}G^{A\mu\nu} = -\frac{1}{4}\left(\partial_\mu G^A_\nu - \partial_\nu G^A_\mu\right)\left(\partial^\mu G^{A\nu} - \partial^\nu G^{A\mu}\right) \tag{9.11}$$

$$- gf^{ABC}(\partial_\mu G^A_\nu)G^{B\mu}C^{C\nu} \tag{9.12}$$

$$- \frac{1}{4}g^2 f^{EAB}f^{ECD}G^A_\mu G^B_\nu G^{C\mu}G^{D\nu}. \tag{9.13}$$

One can see in the Lagrangian a standard kinetic term, a 3-vertex with a derivative (implies a momentum in the Feynman rule), and a 4-vertex suppressed by an additional power of g. The Feynman rules of these two vertices are presented in figure 9.4.

Feynman rules for fermion gauge interactions are drawn in figure 9.5. We use here only u and d quarks in the triplet representation of $SU(3)$. QCD can been found in `MARTY`'s code in



(a) u gauge interaction **(b)** d gauge interaction

Figure 9.5: Gauge interactions for our custom QCD model. The two different rules are identical and depend on the $SU(3)$ generator of the triplet $\mathcal{R}_3$ representation.

`include/QCD.h`
`src/QCD.cpp`

or in a sample code

`scripts/sampleQCD.cpp`

9.1.5 Electro-weak model

The gauge is

$$\mathcal{G}_{\text{GW}} = SU(2)_L \times U(1)_Y. \tag{9.14}$$

$SU(2)_L$ has three generators that we call W^i_μ by convention (i goes from 1 to 3). The "L" stands for "Left" because only left-handed fermions are coupled to the gauge bosons. $U(1)_Y$ is the hypercharge interaction, similar to the electromagnetism. Its generator is noted B_μ. The initial matter content is presented in table 9.1.

This initial model is broken into a $U(1)_{em}$ symmetry group, recovering Standard model particles, in particular the first generation of fermions and the Higgs boson.

This model is one of the most complicated model whose Lagrangian is still contained in one sheet of paper, so let us present it explicitly. Without taking leptons into accounts,

Particle	$SU(2)_L \times U(1)_Y$
H	$(2,\ 1/2)$
Q_L	$(2,\ 1/6)$
u_R	$(1,\ 2/3)$
d_R	$(1,\ -1/3)$
L_L	$(2,\ -1/2)$
e_R	$(1,\ -1)$

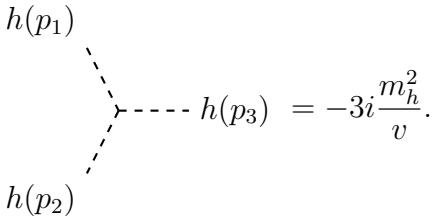
Table 9.1: Matter content of the Electro-weak model implemented in `MARTY`.

the final Lagrangian is

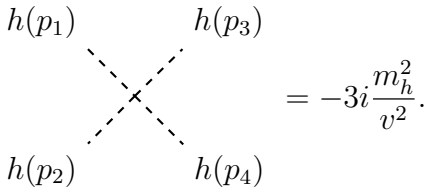
$$
\begin{aligned}
\mathcal{L}_{\text{EW}} = &+ \frac{1}{2}\partial_\mu h \partial^\mu h - \frac{1}{2}m_h^2 h^2 \\
&+ i\bar{u_L}\partial\!\!\!/ u_L + i\bar{u_R}\partial\!\!\!/ u_R - m_u\left(\bar{u_L}u_R + \bar{u_R}u_L\right) \\
&+ i\bar{d_L}\partial\!\!\!/ d_L + i\bar{d_R}\partial\!\!\!/ d_R - m_d\left(\bar{d_L}d_R + \bar{d_R}d_L\right) \\
&- \frac{1}{4}A_{\mu\nu}A^{\mu\nu} \\
&- \frac{1}{2}W^+_{\mu\nu}W^{-\mu\nu} + m_W^2 W^+_\mu W^{-\mu} \\
&- \frac{1}{4}Z_{\mu\nu}Z^{\mu\nu} + \frac{1}{2}m_Z^2 Z_\mu Z^\mu \\
&- \frac{m_h^2}{2v}h^3 - \frac{m_h^2}{8v^2}h^4 \\
&- \frac{m_u}{v}h\bar{u}u \\
&- \frac{m_d}{v}h\bar{d}d \\
&+ \frac{2}{3}e\bar{u}A\!\!\!/ u - \frac{1}{3}e\bar{d}A\!\!\!/ d \\
&+ \frac{2}{3}e\tan\theta_W \bar{u_R}Z\!\!\!/ u_R - \frac{1}{3}e\tan\theta_W \bar{d_R}Z\!\!\!/ d_R \\
&+ \frac{1}{6}e\tan\theta_W \bar{u_L}Z\!\!\!/ u_L - \frac{1}{2}e\cot\theta_W \bar{u_L}Z\!\!\!/ u_L \\
&+ \frac{1}{6}e\tan\theta_W \bar{d_L}Z\!\!\!/ d_L + \frac{1}{2}e\cot\theta_W \bar{d_L}Z\!\!\!/ d_L \\
&+ \frac{ve^2}{2\sin^2\theta_W}W^{+\mu}W^-_\mu h + \frac{e^2}{4\sin^2\theta_W}W^{+\mu}W^-_\mu h^2 \\
&+ \frac{ve^2}{4\cos^2\theta_W\sin^2\theta_W}Z^\mu Z_\mu h + \frac{e^2}{8\cos^2\theta_W\sin^2\theta_W}Z^\mu Z_\mu h^2 \\
&+ e^2 A_\mu A_\nu W^{+\mu}W^{-\nu} - e^2 A_\mu A^\mu W^+_\nu W^{-\nu} \\
&+ ieA_\mu W^+_\nu W^{-\mu\nu} - ieA_\mu W^-_\nu W^{+\mu\nu} + ieW^+_\mu W^-_\nu A^{\mu\nu} \\
&+ e^2\cot^2\theta_W Z_\mu Z_\nu W^{+\mu}W^{-\nu} - e^2\cot^2\theta_W Z_\mu Z^\mu W^+_\nu W^{-\nu} \\
&+ ie\cot\theta_W Z_\mu W^+_\nu W^{-\mu\nu} - ie\cot\theta_W Z_\mu W^-_\nu W^{+\mu\nu} + ie\cot\theta_W W^+_\mu W^-_\nu Z^{\mu\nu} \\
&+ \frac{e^2}{2\sin^2\theta_W}W^+_\mu W^{+\mu}W^-_\nu W^{-\nu} - \frac{e^2}{2\sin^2\theta_W}W^+_\mu W^{-\mu}W^-_\nu W^{+\nu} \\
&+ e^2\cot\theta_W A_\mu Z_\nu W^{+\mu}W^{-\nu} + e^2\cot\theta_W A_\mu Z_\nu W^{-\mu}W^{+\nu} - 2e^2\cot\theta_W A_\mu Z^\mu W^{+\nu}W^-_\nu
\end{aligned}
\tag{9.15}
$$

Feynman Rules for this Lagrangian are presented in figures 9.6, 9.7 and 9.8. A sample program using this model can be found in

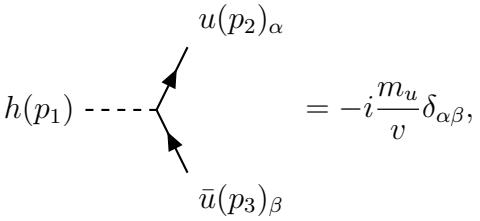
`scripts/sampleElectroWeak.cpp`



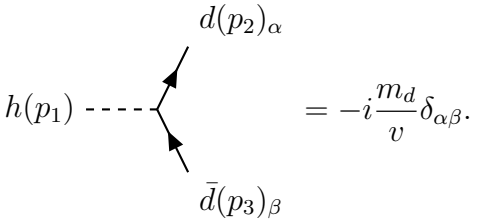
(a) h^3 vertex



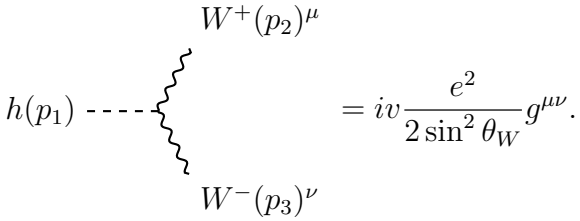
(b) h^4 vertex



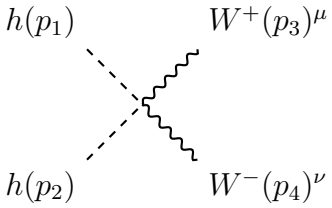
(c) $hu\bar{u}$ vertex



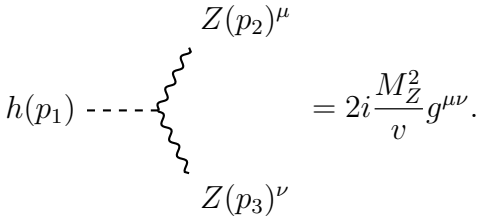
(d) $hd\bar{d}$ vertex



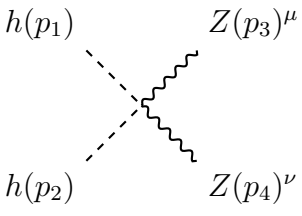
(e) hW^+W^- vertex



(f) hhW^+W^- vertex

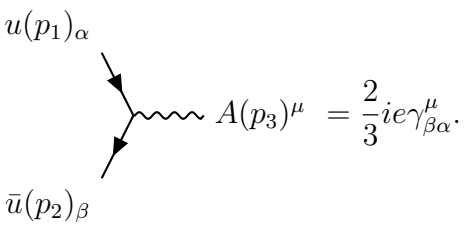


(g) hZZ vertex

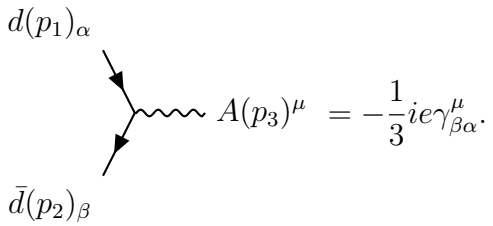


(h) $hhZZ$ vertex

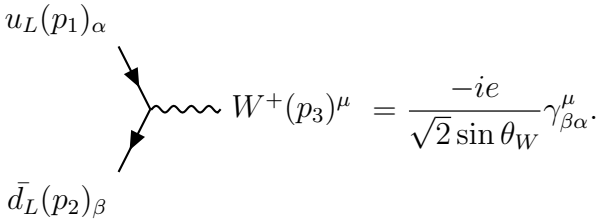
Figure 9.6: Feynman rules for the Higgs interactions in a simple Electro-weak model.



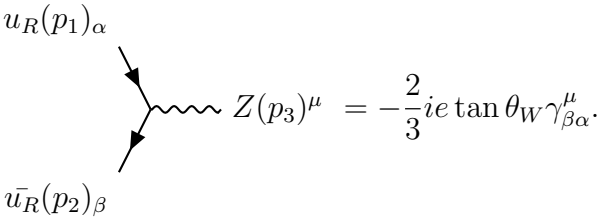
(a) $\bar{u}Au$ vertex



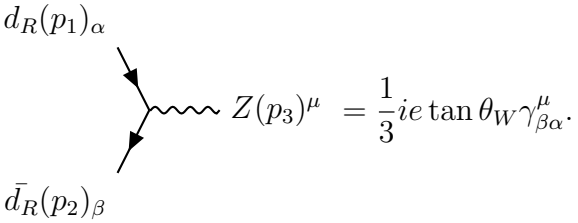
(b) $\bar{d}Ad$ vertex



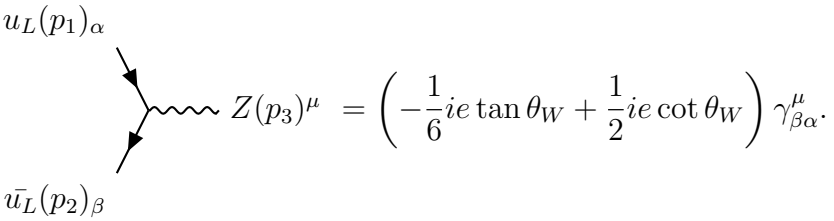
(c) $\bar{u_L} W d_L$ vertex



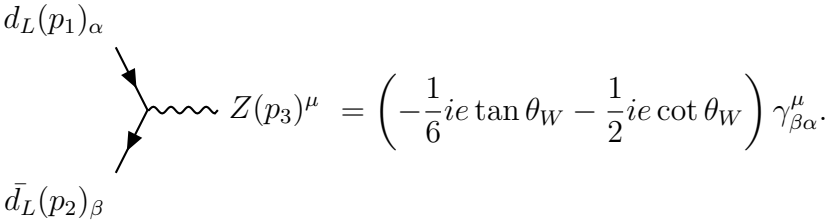
(d) $\bar{u_R} Z u_R$ vertex



(e) $\bar{d_R} Z d_R$ vertex

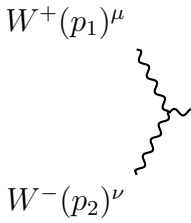


(f) $\bar{u_L} Z u_L$ vertex



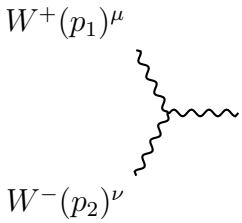
(g) $\bar{d_L} Z d_L$ vertex

Figure 9.7: Feynman rules for fermion gauge interactions in a simple electroweak model with $u_{L/R}$ and $d_{L/R}$ as fermions.



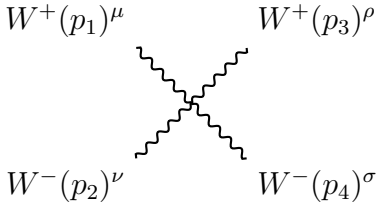
$$= ie\big(g^{\mu\nu}(p_2 - p_1)^\rho + g^{\mu\rho}(p_1 - p_3)^\nu + g^{\nu\rho}(p_3 - p_2)^\mu\big).$$

(a) WWA vertex



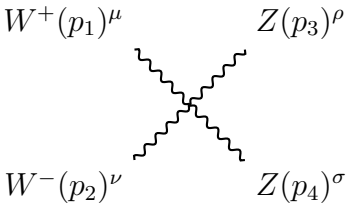
$$= ie\cot\theta_W\big(g^{\mu\nu}(p_2 - p_1)^\rho + g^{\mu\rho}(p_1 - p_3)^\nu + g^{\nu\rho}(p_3 - p_2)^\mu\big).$$

(b) WWZ vertex



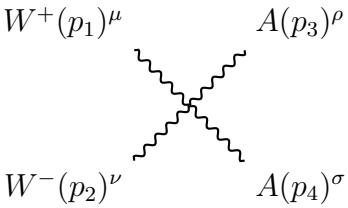
$$= -i\frac{e^2}{\sin^2\theta_W}\big(g^{\mu\nu}g^{\rho\sigma} + g^{\mu\sigma}g^{\nu\rho} - 2g^{\mu\rho}g^{\nu\sigma}\big).$$

(c) $WWWW$ vertex



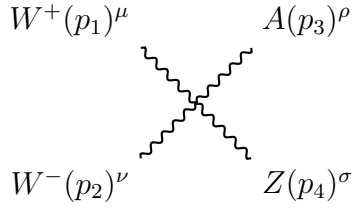
$$= ie^2\cot^2\theta_W\big(g^{\mu\rho}g^{\nu\sigma} + g^{\mu\sigma}g^{\nu\rho} - 2g^{\mu\nu}g^{\rho\sigma}\big).$$

(d) $WWZZ$ vertex



$$= ie^2\big(g^{\mu\rho}g^{\nu\sigma} + g^{\mu\sigma}g^{\nu\rho} - 2g^{\mu\nu}g^{\rho\sigma}\big).$$

(e) $WWAA$ vertex



$$= ie^2\cot\theta_W\big(g^{\mu\rho}g^{\nu\sigma} + g^{\mu\sigma}g^{\nu\rho} - 2g^{\mu\nu}g^{\rho\sigma}\big).$$

(f) $WWAZ$ vertex

Figure 9.8: Feynman rules for gauge bosons interactions in a simple electroweak model.

9.2 Standard Model (SM)

The Standard Model can be found in

```
include/SM.h
src/SM.cpp
```

The final Lagrangian is built explicitly in the broken $SU(3)_C \times U(1)_{em}$ gauge. On can build a Standard Model in `MARTY` following prescriptions given in sample code 82. Users also can access SM input values in the file SM.h, that are used in the Lagrangian.

Sample code 82: The Standard Model

Building a Standard Model

```
SM_Model SM;// Use SM
```

Getting input values

```
Expr alpha_em = sm_input::alpha_em;
Expr M_W = sm_input::M_W;
```

See also Documentation of SM.h.

9.3 2 Higgs Doublet Model (2HDM)

2HDM models consist in adding a second Higgs doublet in the theory, identical to the first one. Their is then 8 Higgs-type degrees of freedom (dof) instead of 4 in the SM. These dof will (after the Electro-weak symmetry breaking) give 4 physical states and three Goldstone bosons:

- h^0 and H^0, scalar particles.
- A^0, a pseudo-scalar particle (eigenvalue -1 with respect to the parity operator).
- $H^\pm$, a charged Higgs (complex, counts for 2 degrees of liberty).

9.3.1 The high-energy Lagrangian

The higgs sector

The two Higgs' are $SU(2)_L$ doublets carrying a $+1$ hyper-charge, Φ_1 and Φ_2 respectively. Their potential is (omitting $SU(2)_L$ indices)

$$\begin{aligned} -V_{\Phi_1,\Phi_2} =& m_{11}^2 \Phi_1^\dagger \Phi_1 + m_{22}^2 \Phi_2^\dagger \Phi_2 - m_{12}^2 \left(\Phi_1^\dagger \Phi_2 + \Phi_2^\dagger \Phi_1\right) + \frac{\lambda_1}{2}\left(\Phi_1^\dagger \Phi_1\right)^2 + \frac{\lambda_2}{2}\left(\Phi_2^\dagger \Phi_2\right)^2 \\ &+ \lambda_3 \left(\Phi_1^\dagger \Phi_1\right)\left(\Phi_2^\dagger \Phi_2\right) + \lambda_4 \left(\Phi_1^\dagger \Phi_2\right)\left(\Phi_1^\dagger \Phi_2\right) + \frac{\lambda_5}{2}\left[\left(\Phi_1^\dagger \Phi_2\right)^2 + \left(\Phi_2^\dagger \Phi_1\right)^2\right]. \end{aligned} \tag{9.16}$$

This potential is the most general one allowing to perform properly a symmetry breaking. It has eight real parameters. Minimizing this potential by searching (classical) solutions of

$$\frac{\partial V_{\Phi_1,\Phi_2}}{\partial \Phi_i^\dagger} = 0, \tag{9.17}$$

and

$$\frac{\partial^2 V_{\Phi_1,\Phi_2}}{\partial\Phi_i\partial\Phi_i^\dagger} > 0, \tag{9.18}$$

for each i that takes the value 0 or 1. Solutions are

$$\Phi_1 = \begin{pmatrix} 0 \\ \frac{v_1}{\sqrt{2}} \end{pmatrix}, \tag{9.19}$$

$$\Phi_2 = \begin{pmatrix} 0 \\ \frac{v_2}{\sqrt{2}} \end{pmatrix}. \tag{9.20}$$

Quantum mechanically, we interpret this minimum as the Vacuum Expectation Value (VEV) of the different fields $\langle\Phi_1\rangle$ and $\langle\Phi_2\rangle$. We then expand around these minima to get degrees of freedom (dof) with no VEV. In our case, it is convenient to define

$$\Phi_i \equiv \begin{pmatrix} \phi_i^+ \\ (v_i + \rho_i + i\eta_i)/\sqrt{2} \end{pmatrix}, \; i = 1, 2. \tag{9.21}$$

We indeed have 8 dof as before: ϕ_i^+ are two complex fields so have four dof, ρ_i and η_i are four real fields, 4 dof also.

Then, expanding V_{Φ_1,Φ_2} in the new fields, one finds 3 and 4 legs interactions between all the scalars, and non diagonal mass matrices. We want in general to diagonalize all of them in a theory. This is because what we can observe in the real world are eigenstates of the free Hamiltonian (without interaction). The free Hamiltonian being composed by kinetic and mass terms, diagonalize it is equivalent to consider only the mass matrix of the system[2].

In our case, there is 3 mass matrices, each mixing two particles. First the ϕ_i^+ matrix M_ϕ^2 defined by

$$\mathcal{L} \ni -\phi_i^-(M_\phi^2)_{ij}\phi_j^+ \tag{9.22}$$

reads

$$M_\phi^2 = \begin{pmatrix} \frac{v_2}{v_1}m_{12}^2 - (\lambda_4+\lambda_5)v_2^2 & -m_{12}^2 + (\lambda_4+\lambda_5)v_1v_2 \\ -m_{12}^2 + (\lambda_4+\lambda_5)v_1v_2 & \frac{v_1}{v_2}m_{12}^2 - (\lambda_4+\lambda_5)v_1^2 \end{pmatrix} \tag{9.23}$$

$$= \left(m_{12}^2 - (\lambda_4+\lambda_5)v_1v_2\right) \cdot \begin{pmatrix} \frac{v_2}{v_1} & -1 \\ -1 & \frac{v_1}{v_2} \end{pmatrix}. \tag{9.24}$$

Its eigenvalues are

$$\lambda^+ = \left(m_{12}^2/(v_1v_2) - \lambda_4 - \lambda_5\right)(v_1^2 + v_2^2) \equiv m_+^2, \tag{9.25}$$

$$\lambda_- = 0. \tag{9.26}$$

The massive state is the charged higgs $H^\pm$ of mass m_+. The mass-less one is two of the three Goldstone bosons of the spontaneous symmetry breaking, $G^\pm$. Then, the mass matrix of η_i defined by

$$\mathcal{L} \ni -\frac{1}{2}\eta_i(M_\eta^2)_{ij}\eta_j \tag{9.27}$$

[2]Kinetic terms are always fully symmetric by a $SO(N)$ (or $SU(N)$ for complex fields). Rotating fields to go in the mass-diagonal basis keeps kinetic terms diagonal as well, and the Hamiltonian is diagonalized.

reads

$$M_\eta^2 = \frac{m_A^2}{v_1^2+v_2^2}\begin{pmatrix} v_2^2 & -v_1v_2 \\ -v_1v_2 & v_1^2 \end{pmatrix}. \tag{9.28}$$

Its eigenvalues are

$$\lambda^+ = \left(m_{12}^2/(v_1v_2) - 2\lambda_5\right)(v_1^2+v_2^2) \equiv m_A^2, \tag{9.29}$$
$$\lambda_- = 0. \tag{9.30}$$

The massive state is the pseudo-scalar A^0 of mass m_A. The mass-less one is the last Goldstone bosons G^0. Finally, the mass matrix of ρ_i defined the same way as for η_i has a non zero determinant and has two non zero eigenvalues, two physical massive states. The matrix M_ρ^2 reads

$$\begin{pmatrix} m_{12}^2\dfrac{v_2}{v_1} + \lambda_1 v_1^2 & -m_{12}^2\dfrac{v_2}{v_1} + \lambda_{345}v_1v_2 \\ -m_{12}^2\dfrac{v_2}{v_1} + \lambda_{345}v_1v_2 & m_{12}^2\dfrac{v_1}{v_2} + \lambda_2 v_2^2 \end{pmatrix}, \tag{9.31}$$

with $\lambda_{345} \equiv \lambda_3+\lambda_4+\lambda_5$. Its eigenvalues may be determined exactly but have complicated expressions. It is convenient to define the rotation angle α that allows to get the two mass eigenstates h^0 and H^0, h^0 lighter by definition. Their expression is

$$h^0 = -\rho_1\sin\alpha + \rho_2\cos\alpha, \tag{9.32}$$
$$H^0 = \rho_1\cos\alpha + \rho_2\sin\alpha. \tag{9.33}$$

The standard model Higgs boson is then, expressed as a function of the two neutral Higgs bosons,

$$h_{\rm SM}^0 = \rho_1\cos\beta + \rho_2\sin\beta \tag{9.34}$$
$$= \cos(\alpha-\beta)H^0 + \sin(\alpha-\beta)h^0, \tag{9.35}$$

where β is defined by $\tan\beta \equiv \dfrac{v_2}{v_1}$. It relates the two VEVs, and is in particular the rotation angle that give Goldstone bosons ($G^\pm$ and G^0) when diagonalizing mass matrices. If one wants to have a theory with a SM-like Higgs boson, there is then two possibilities.

- $\alpha = \beta$. In this case, $h_{\rm SM}^0 = H^0$. Then the lighter Higgs h^0 could be discovered in colliders ($m < 125$ GeV).
- $\alpha = \beta + \pi/2$. Here $h_{\rm SM}^0 = h^0$ and the missing massive state H^0 must be searched at higher energies ($m > 125$ GeV).

To summary transformations from equation 9.21 to physical Higgs (and Goldstone) bosons, let's see transfer matrices explicitly. First, we have defined

$$\begin{pmatrix} \phi_1^+ \\ \phi_2^+ \end{pmatrix} = \begin{pmatrix} \cos\beta & -\sin\beta \\ \sin\beta & \cos\beta \end{pmatrix}\cdot\begin{pmatrix} G^\pm \\ H^\pm \end{pmatrix}. \tag{9.36}$$

Then, for pseudo-scalar bosons, we have

$$\begin{pmatrix} \eta_1 \\ \eta_2 \end{pmatrix} = \begin{pmatrix} \cos\beta & -\sin\beta \\ \sin\beta & \cos\beta \end{pmatrix}\cdot\begin{pmatrix} G^0 \\ A^0 \end{pmatrix}. \tag{9.37}$$

Finally, the two scalar Higgs states are defined through

$$\begin{pmatrix} \rho_1 \\ \rho_2 \end{pmatrix} = \begin{pmatrix} -\sin\alpha & \cos\alpha \\ \cos\alpha & \sin\alpha \end{pmatrix}\cdot\begin{pmatrix} h^0 \\ H^0 \end{pmatrix}. \tag{9.38}$$

Type	U_R	D_R	E_R
I	+	+	+
II	+	-	-
III / Flipped	+	-	+
IV / Lepton specific	+	+	-

Table 9.2: The two first types of 2HDM models. They are defined by the Z_2 charges of right-handed fermions telling if they couple to Φ_1 (charge -1) or Φ_2 (charge +1) through Yukawas.

The fermion sector

The Higgs doublets couple to fermions through Yukawa interactions. For each fermion that has a left part in the doublet representation of $SU(2)_L$ and a right part in the singlet one, one can write the Yukawa Lagrangian:

$$\mathcal{L} \ni -\bar{\psi_L}_i^I \Phi_{1i} y_1^{IJ} \psi_R^J - \bar{\psi_L}_i^I \Phi_{2i} y_2^{IJ} \psi_R^J, \tag{9.39}$$

with y_1 and y_2 two matrices in flavor space (3-dimensional in our case). After the symmetry breaking, these terms yield interactions between fermions and Higgs bosons, and the fermion mass matrices

$$M^{IJ} = y_1^{IJ} \frac{v_1}{\sqrt{2}} + y_2^{IJ} \frac{v_2}{\sqrt{2}}. \tag{9.40}$$

Here in general, y_1 and y_2 cannot be diagonalized simultaneously. In that case, Higgs couplings to fermions will not be diagonal and vertices like $b \to sh^0$ are allowed. In the real world, flavor changing neutral currents (FCNC) are extremely rare (suppressed) and we only observe it in the quark sector. This means that a tree-level FCNC (and in particular on the lepton sector) would be extremely hard to justify in a model that has to fit data. To avoid tree-level FCNC, we have to make assumptions on Yukawa couplings. Either each fermion couples to only one Higgs doublet, or to both if they may be diagonalized simultaneously. Here we consider the first case.

For standard model fermions, there is three different Yukawa couplings. Y^U for U_R, Y^D for D_R (both coupled to Q_L the $SU(2)_L$ doublet) and Y^L for E_R coupled to L_L[3]. The Yukawa Lagrangian reads

$$-\mathcal{L}_{\text{Yuk}} = \sum_{i=1}^{2} \left[\bar{Q}_L \tilde{\Phi}_i Y_i^U U_R + \bar{Q}_L \Phi_i Y_i^D D_R + \bar{L}_L \Phi_i Y_i^L E_R + \text{h.c.} \right], \tag{9.41}$$

where $\tilde{\Phi}_i \equiv i\sigma_2 \Phi_i^* = \begin{pmatrix} \Phi_i^{2*} \\ -\Phi_i^{1*} \end{pmatrix}$. Each Yukawa may come either from Φ_1 or Φ_2. This corresponds to a Z_2 charge: +1 if coupled to Φ_2 only, -1 if coupled to Φ_1 only. Without loss of generality, U_R has always a charge +1. We will define here four types of 2HDM models. They are defined by the Z_2 charges of the three right-handed fermion families as shown table 9.2. Regarding conventions, beware that only types I and II are well defined. For the two other types, names may vary. In particular type III may refer to the one in table 9.2 or to a general flavor mixing 2HDM model.

[3] L_L contains left-handed neutrinos and charged leptons. We do not consider here a right-handed neutrino N_R.

After the symmetry breaking, applying all rotations and the exact Z_2 symmetry to Yukawas, one gets the final Lagrangian

$$\begin{aligned} -\mathcal{L}_{\text{Yuk}} = & \sum_{f=u,d,l} \frac{m_f}{v} \left(\xi_h^f \bar{f} f h^0 + \xi_H^f \bar{f} f H^0 - i \xi_A^f \bar{f} \gamma^5 f A^0 \right) \\ & + \sum_{u,d} \frac{\sqrt{2} V_{ud}}{v} \bar{u} \left(m_u \xi_A^u P_L + m_d \xi_A^d P_R \right) d H^+ + \text{h.c.} \\ & + \sum_{l} \frac{\sqrt{2} m_l}{v} \xi_A^l \bar{\nu}_l l_R H^+ + \text{h.c.}, \end{aligned} \tag{9.42}$$

where $v \equiv \sqrt{v_1^2 + v_2^2}$ and $\xi_{h,H,A}^{u,d,l}$ are trigonometric factors depending on the type of model considered. Table 9.3 shows the values of these parameters as a function of the mixing angles α and β. $P_{L/R}$ are chirality projectors for fermions defines by

$$P_{L/R} \equiv \frac{1 \mp \gamma^5}{2}. \tag{9.43}$$

One important feature to note is the pseudo-scalar coupling of A^0 to fermions (through γ^5) that differs from h^0 and H^0, proportional to the identity. This is due to the fact that A^0 comes with an additional factor of i in couplings (from definition of equation 9.21 and mixing 9.37). When adding left-handed terms of h^0 and H^0, one gets

$$\frac{m_f}{v} \xi_h^f \left(\bar{f}_L h^0 f_R + \text{h.c.} \right) \tag{9.44}$$

$$= \frac{m_f}{v} \xi_h^f \left(\bar{f}_L h^0 f_R + \bar{f}_R h^0 f_L \right) \tag{9.45}$$

$$= \frac{m_f}{v} \xi_h^f \bar{f} \left(P_L + P_R \right) f h^0 \tag{9.46}$$

$$= \frac{m_f}{v} \xi_h^f \bar{f} f h^0, \tag{9.47}$$

and similarly for H^0. However, the factor of i with A^0 induces a sign in the hermitic conjugate and finally one gets

$$\frac{m_f}{v} \xi_A^f \left(i \bar{f}_L A^0 f_R + \text{h.c.} \right) \tag{9.48}$$

$$= \frac{m_f}{v} \xi_A^f \left(i \bar{f}_L A^0 f_R - i \bar{f}_R A^0 f_L \right) \tag{9.49}$$

$$= i \frac{m_f}{v} \xi_A^f \bar{f} \left(P_L - P_R \right) f A^0 \tag{9.50}$$

$$= -i \frac{m_f}{v} \xi_A^f \bar{f} \gamma^5 f A^0. \tag{9.51}$$

	Type I	Type II	Type III (Flipped)	Type IV (Lepton specific)
ξ_h^u	$\cos\alpha/\sin\beta$	$\cos\alpha/\sin\beta$	$\cos\alpha/\sin\beta$	$\cos\alpha/\sin\beta$
ξ_h^d	$\cos\alpha/\sin\beta$	$-\sin\alpha/\cos\beta$	$-\sin\alpha/\cos\beta$	$\cos\alpha/\sin\beta$
ξ_h^e	$\cos\alpha/\sin\beta$	$-\sin\alpha/\cos\beta$	$\cos\alpha/\sin\beta$	$-\sin\alpha/\cos\beta$
ξ_H^u	$\sin\alpha/\sin\beta$	$\sin\alpha/\sin\beta$	$\sin\alpha/\sin\beta$	$\sin\alpha/\sin\beta$
ξ_H^d	$\sin\alpha/\sin\beta$	$\cos\alpha/\cos\beta$	$\cos\alpha/\cos\beta$	$\sin\alpha/\sin\beta$
ξ_H^e	$\sin\alpha/\sin\beta$	$\cos\alpha/\cos\beta$	$\sin\alpha/\sin\beta$	$\cos\alpha/\cos\beta$
ξ_A^u	$-\cot\beta$	$-\cot\beta$	$-\cot\beta$	$-\cot\beta$
ξ_A^d	$\cot\beta$	$-\tan\beta$	$-\tan\beta$	$\cot\beta$
ξ_A^e	$\cot\beta$	$-\tan\beta$	$\cot\beta$	$-\tan\beta$

Table 9.3: Yukawa couplings of u, d, l to the neutral Higgs bosons h^0, H^0, A^0 in the four different models.

9.3.2 Samples

The 2HDM model can be found in

```
include/2HDM.h
src/2HDM.cpp
```

and in a sample code

```
scripts/sample2HDM.cpp
```

Creating a 2HDM model, one has to give the type as presented in sample code 83.

Sample code 83: 2HDM

Creating a 2HDM of a given type

```
TwoHDM_Model<1> THDM_type1;
TwoHDM_Model<2> THDM_type2;
TwoHDM_Model<3> THDM_type3;
TwoHDM_Model<4> THDM_type4;
```

9.4 Minimal Supersymmetric Standard Model (MSSM)

The MSSM follows conventions of [35]. There is two types of models, the unconstrained MSSM (uMSSM) with 105 parameters and a lighter model, the phenomenological MSSM (pMSSM) with 19 parameters.

9.4.1 Unconstrained MSSM

The model-building version can be found in

```
include/MSSM.h
src/MSSM.cpp
```

Ways to build the unconstrained MSSM are presented in sample code 84. The full Lagrangian of the unconstrained MSSM has not yet been generated as C++ code. One can then get it waiting about 2 hours while `MARTY` breaks the full Lagrangian. In an upcoming version we will include the explicit MSSM Lagrangian to allow one in practice to use this model, waiting only a few seconds to load it.

Sample code 84: Unconstrained MSSM

Building the uMSSM from scratch (few hours of calculations)

```
MSSM_Model uMSSM;
```

Loading the final Low Energy uMSSM Model in an upcoming version

```
MSSM_LEM uMSSM;
```

Accessing MSSM input values

```
Expr mu = mssm_input::mu;
Expr tanb = mssm_input::tanb;
```

See also Documentation of file MSSM.h.

9.4.2 Phenomenological MSSM

The model-building version and final version of the model can be found respectively in

```
include/PMSSM.h
src/PMSSM.cpp
```

and

```
models/include/pmssm.h
models/src/pmssm_<spec>.cpp (several files <spec> for this model)
```

Ways to build the phenomenological MSSM are presented in sample code 85.

Sample code 85: Phenomenological MSSM

Building the pMSSM from scratch (few minutes of calculations)

```
PMSSM_Model pMSSM;
```

Loading the final Low Energy pMSSM Model (few seconds to load)

```
PMSSM_LEM pMSSM;
```

Accessing pMSSM input values

```
Expr mu = mssm_input::mu;
Expr tanb = mssm_input::tanb;
```

See also Documentation of file PMSSM.h.

Chapter 10

Debugging MARTY programs

10.1 Common mistakes

This section is about mistakes made by user that lead errors or bad results in a normal run-time of MARTY.

10.1.1 Model Building mistakes

Model building is probably the biggest source of user mistakes because there is many features, implying many possibilities to use MARTY the wrong way.

First, naming conventions in a model require attention, as any object that has not been explicitly defined by the user can be accessed with its name. If one tries to get a particle, a gauged group with a wrong name, MARTY will complain and stop the program. As there is for now no separate treatment of regular and LaTeX names, built-in particle names in MARTY use LaTeX format to display nicely in GRAFED. These errors are easy to fix as MARTY explicitly says that a given name has not been found (for a particle, a group, a coupling etc).

More subtle issues will come from indices in tensors. When adding user-defined Lagrangian terms, one has to give correct indices to tensors and particles is the expression. A valid index structure must respect a few conditions:

- A Lagrangian is scalar, so no free index is allowed in its terms.
- One index must appear twice in a product. More occurrences[1] will induce errors as CSL will not be able to know which indices are contracted with each other.
- If a sum appears in a sub-expression, free indices should be the same in each terms of the sum.

These errors can be slightly more difficult to find as a non valid index structure can cause an error later in the program, and in general will be raised by CSL, not MARTY. In the case an error message concerning indices is raised, we recommend to check all user-defined indexed expressions given to MARTY, even if they do not seem directly related[2]. Index manipulations in CSL and MARTY are rather stable, and an index error is more likely to

[1] An index must also not be alone, otherwise it is a free index.

[2] This of course if there is no obvious cause for the bug.

come from the user-defined program than from the core of CSL or MARTY. In particular, users should be careful about squared expressions such as

$$\mathcal{L} \ni \lambda \left(H_i^\dagger H_i \right)^2 = \lambda H_i^\dagger H_i H_j^\dagger H_j. \tag{10.1}$$

By hand one can replace indices when expanding the product but CSL does not do such renaming automatically in all cases. Considering a Particle H, an Index i and an Expr lambda one should then not write code like

```
Expr prod = GetComplexConjugate(H(i)) * H(i);
model.addLagrangianTerm(
        lambda * prod * prod
        ); // Bad !
```

as it will result in four times the same index "i" in the interaction term, and very probably an error at run-time later on. One must instead use pow_s() or the CSL interface function RenamedIndices() that deep copies an expression, renaming all indices:

```
model.addLagrangianTerm(
lambda * pow_s(GetComplexConjugate(H(i)) * H(i), 2)
); // Good, the pow is handled correctly !
```

or

```
Expr prod = GetComplexConjugate(H(i)) * H(i);
model.addLagrangianTerm(
        lambda * prod * RenamedIndices(prod)
        ); // Good !
```

When doing model building, mistakes about the physics can be made. These errors are no longer ill-defined program behaviors but will lead to bad results. They can be very difficult to find as they depend in particular on the conventions that are used. To reduce the chances do to such mistakes, we encourage users to display regularly the model while modifying it, checking that the state is indeed what one expects. Once the model is finished, Feynman rules must be double checked (when possible) because they are the building blocks of all further calculations done with MARTY.

10.1.2 Calculation mistakes

There is not many mistakes that a user can do while asking MARTY to calculate theoretical quantities, as the interface is reduced to the strict minimum. The only worth addressing mistake in that sense is about field insertions. When giving external legs to calculate amplitudes, a mistake in incoming / outgoing or particle / antiparticle properties is possible. It that case, it will probably result in an absence of diagram, a zero amplitude. If one gets an abnormal null result, it probably comes from wrong insertions. When the amplitude is not equal to zero, it is easier to see the problem when displaying Feynman diagrams with GRAFED, as one will immediately notice that they do not correspond to what was expected.

10.2 GNU Debugger (GDB)

We tried in MARTY's interface to implement as many tests as possible to tell users the problems encountered at run-time. Typically, the program stops if an error has been

found[3]. In that case or if the program crashes because of a state we did not think of, GDB is a good option. In the directory `marty` one can put a program file `program.cpp` in `scripts/` and type

```
$ make program_debug.x
```

and finally launch the program using gdb:

```
$ gdb -ex run bin/program_debug.x
```

The program runs, and when it crashes (being a real crash or one provoked by an error raising in `MARTY`) one can type

```
> where
```

This will show the stack trace, i.e. the successive functions called before the crash, with precise line numbers. This method is very effective to understand a crash. Once found, one may see that the crash comes from a bad use of `MARTY` and be able to solve the problem, or from `MARTY` itself. In the latter case please report the problem to us.

10.3 The author(s)

We always are available for discussion if one encounters an issue with `MARTY`. It is a big code that can do many things, implying that users will have questions and problems. Thanks to the fact that `MARTY` is very general there is not many issues we cannot solve easily simply by discussing with a user. Any needed correction, interface improvement or extension can be considered.

[3]There is in particular no warning in `MARTY`.

Bibliography

[1] Wolfram Research, Inc., "Mathematica, Version 12.1." https://www.wolfram.com/mathematica.

[2] T. Hahn and M. Perez-Victoria, *Automatized one loop calculations in four-dimensions and D-dimensions*, *Comput. Phys. Commun.* **118** (1999) 153 [hep-ph/9807565].

[3] F. Staub, *Exploring new models in all detail with SARAH*, *Adv. High Energy Phys.* **2015** (2015) 840780 [1503.04200].

[4] A. Alloul et al., *FeynRules 2.0 - A complete toolbox for tree-level phenomenology*, *Comput. Phys. Commun.* **185** (2014) 2250 [1310.1921].

[5] A. Semenov, *LanHEP: A Package for the automatic generation of Feynman rules in field theory. Version 3.0*, *Comput. Phys. Commun.* **180** (2009) 431 [0805.0555].

[6] A. Pukhov et al., *CompHEP: A Package for evaluation of Feynman diagrams and integration over multiparticle phase space*, hep-ph/9908288.

[7] CompHEP collaboration, *CompHEP 4.4: Automatic computations from Lagrangians to events*, *Nucl. Instrum. Meth. A* **534** (2004) 250 [hep-ph/0403113].

[8] G. Uhlrich, F. Mahmoudi and A. Arbey, *MARTY – Modern ARtificial Theoretical phYsicist: A C++ framework automating theoretical calculations Beyond the Standard Model*, *Computer Physics Communications* **264** (2021) 107928.

[9] B. Stroustrup and H. Sutter, "C++ Core Guidelines." https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines.

[10] The ISO C++ committee, "C++ Standard Library." https://en.cppreference.com/w/cpp.

[11] F. Mahmoudi, *SuperIso: A Program for calculating the isospin asymmetry of $B \to K^* \gamma$ gamma in the MSSM*, *Comput. Phys. Commun.* **178** (2008) 745 [0710.2067].

[12] F. Mahmoudi, *SuperIso v2.3: A Program for calculating flavor physics observables in Supersymmetry*, *Comput. Phys. Commun.* **180** (2009) 1579 [0808.3144].

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

[14] A. Arbey and F. Mahmoudi, *SuperIso Relic: A Program for calculating relic density and flavor physics observables in Supersymmetry*, *Comput. Phys. Commun.* **181** (2010) 1277 [0906.0369].

[15] 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** (2011) 1582.

[16] 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** (2019) 238 [1806.11489].

[17] P. Cvitanović, *Group Theory*, Princeton University Press (2008).

[18] A. Denner et al., *Feynman rules for fermion-number-violating interactions*, *Nuclear Physics B* **387** (1992) 467.

[19] R. Feger, T.W. Kephart and R.J. Saskowski, *LieART 2.0 – A Mathematica application for Lie Algebras and Representation Theory*, *Computer Physics Communications* **257** (2020) 107490.

[20] P. Cvitanovic, *Group theory for Feynman diagrams in non-Abelian gauge theories*, *Phys. Rev. D* **14** (1976) 1536.

[21] M.D. Schwartz, *Quantum Field Theory and the Standard Model*, Cambridge University Press (Mar, 2014).

[22] J.C. Romão and J.P. Silva, *A resource for signs and feynman diagrams of the standard model*, *International Journal of Modern Physics A* **27** (2012) 1230025.

[23] T. van Ritbergen, A. Schellekens and J. Vermaseren, *Group theory factors for Feynman diagrams*, *Int. J. Mod. Phys. A* **14** (1999) 41 [hep-ph/9802376].

[24] S. Okubo, *Casimir Invariants and Vector Operators in Simple Lie Algebra*, *J. Math. Phys.* **18** (1977) 2382.

[25] A. Denner et al., *Compact Feynman rules for Majorana fermions*, *Phys. Lett. B* **291** (1992) 278.

[26] C.C. Nishi, *Simple derivation of general Fierz-type identities*, *American Journal of Physics* **73** (2005) 1160–1163.

[27] A. Denner, *Techniques for calculation of electroweak radiative corrections at the one loop level and results for W physics at LEP-200*, *Fortsch. Phys.* **41** (1993) 307 [0709.1075].

[28] G. Sulyok, *A closed expression for the UV-divergent parts of one-loop tensor integrals in dimensional regularization*, *Physics of Particles and Nuclei Letters* **14** (2017) 631–643.

[29] G. Passarino and M. Veltman, *One Loop Corrections for e^+e^- Annihilation Into $\mu^+\mu^-$ in the Weinberg Model*, *Nucl. Phys. B* **160** (1979) 151.

[30] R.K. Ellis et al., *One-loop calculations in quantum field theory: From Feynman diagrams to unitarity cuts*, *Physics Reports* **518** (2012) 141–250.

[31] A.J. Buras, *Weak Hamiltonian, CP violation and rare decays*, in *Les Houches Summer School in Theoretical Physics, Session 68: Probing the Standard Model of Particle Interactions*, Jun, 1998 [hep-ph/9806471].

[32] A.J. Buras, *Gauge Theory of Weak Decays: The Standard Model and the Expedition to New Physics Summits*, Cambridge University Press (2020), 10.1017/9781139524100.

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

[34] B. Allanach et al., *SUSY Les Houches Accord 2*, *Comput. Phys. Commun.* **180** (2009) 8 [0801.0045].

[35] S.P. Martin, *A supersymmetry primer*, *Advanced Series on Directions in High Energy Physics* (1998) 1–98.