# Mass builder – an interface tool for automated self energy calculation

**James McKay**[a,1]

[1] Imperial College London

September 27, 2016

**Abstract** This program is designed to organise and simplify the calculation of two loop self energy amplitudes by creating a user friendly interface between the existing available programs. Using the C++ language we generate the appropriate Mathematica scripts to run FeynArts, FeynCalc and TARCER to determine and evaluate the necessary two loop amplitudes. We automatically determine the required basis integrals and generate C++ code to evaluate these using TSIL. In this way the user may go from a FeynArts model file to an evaluated self energy with the ability to select amplitudes on a diagram by diagram basis.

The computation of $\mathcal{O}(10)$ amplitudes simultaneously using tools such as FeynCalc results in extremely long run times as simplifications are being attempted at the symbolic level. On the other hand, keeping track of all terms on a diagram by diagram basis is a serious task by any manual or even semi automated method. We offer an alternative; by completely automating this process we are able to keep track of all terms and evaluate them numerically, which on a modest computing set up is the only way to achieve this task without additional user intervention.

## Contents



[a]e-mail: j.mckay14@imperial.ac.uk

## 1 Introduction

The calculation of radiative corrections at the two-loop level is a computationally challenging task which has been significantly simplified with the introduction of modern tools. Even at the most rudimentary level, determining all possible topologies is a non-trivial task, let alone the simplification of the resulting integral expressions, and finally the evaluation of these integrals. Fortunately, FeynArts, FeynCalc, TARCER and TSIL have made each step of this process far more achievable for a wide range of users.

## 2 Installation

Before beginning the following programs are required

- Mathematica 9.0
- FeynCalc 9.0.1 including a patched distribution of FeynArts 3.9 and TARCER 2.0
- TSIL 1.3
- cmake 3.4.0

the Mass Builder C++ code has been tested using gcc version 4.8.4 and

### 2.1 FeynCalc, FeynArts and TARCER

The easiest way to install FeynCalc, FeynArts and TARCER is via the automated installation method. Open a Mathematica notebook or kernel session and enter

```
Import["https://raw.githubusercontent.com/FeynCalc/
    feyncalc/master/install.m"]
InstallFeynCalc[]
```

when requested to install the latest version of FeynArts say yes, as this will automatically patch the FeynArts installation. If you do not follow this method then it is not possible to run FeynArts and FeynCalc in the same session (as we need to do) as many function names are identical between the packages, so to avoid name shadowing follow the recommend method. For more information see the FeynCalc wiki https://github.com/FeynCalc/feyncalc/wiki.

Check if TARCER has been loaded with the following input

```
./MathKernal
$LoadPhi = True;
$LoadTARCER = True;
$LoadFeynArts = True;
<< FeynCalc/FeynCalc.m
```

if TARCER has not been loaded this will give an error and advise the user to run

```
GenerateTarcerMX
```

which will generate the required files. All packages within Mathematica are now set up.

### 2.2 TSIL

The Two-loop Self-energy Integral Library can be downloaded from http://www.niu.edu/spmartin/TSIL/. Mass Builder has been tested with version 1.3. It may installed anywhere (as Mass Builder will request the path at configuration).

### 2.3 Mass Builder

Mass Builder can be obtained from

```
https://github.com/JamesHMcKay/Mass_builder.git
```

The first step is to run the configuration script to tell the program where your Mathematica kernel and TSIL installations are located, if using Linux this is

```
./scripts/config_linux.sh
```

or

```
./scripts/config.sh
```

for OS X users. The configuration script will ask for the location of the Mathematica Kernal. If using OSX the best option to try first is `osx` (or in some rarer instances `osx64`). On a Linux system this is typically located in the path by default, so try the `math` shorted cut at this prompt. If this path is incorrect, Mass Builder will compile but will not run as it can't access Mathematica. The next prompt will ask for the folder in which the TSIL library and `tsil_cpp.h` are located, this should be entered as the path to the folder without the `/` at the end. For example `/Users/<user_name>/Programs/tsil-1.3`.

After configuration, build as normal using

```
cd build
cmake ..
make
```

to compile the program. The Mass Builder executable is now located in the root directory.

## 3 Quick start guide

This section provides a minimalistic example to demonstrate the core features of this program and test the installation has been successful. The example uses a simple scalar field theory with Lagrangian,

$$\mathcal{L} = -\frac{1}{2}m^2\phi^2 - \frac{g}{3!}\phi^3 - \frac{\lambda}{4!}\phi^4 \tag{1}$$

for which I provide a FeynArts model file and the necessary Mass Builder input files in the `models/Scalar/` directory. For using new models it is recommended to read through the full user guide in Section 4 to understand all available features.

### 3.1 Generate FeynArts diagrams

When first approaching a problem involving radiative loop corrections having a visual list of the involved corrections is helpful. In Mass Builder the number assigned to each radiative process, or *diagram*, is useful information for the user to select which process to include in the calculation. FeynArts has the capability to produce a Feynman diagram for each possible process given a model file, thus for a chosen model we call FeynArts and conveniently save this output into uniquely named Portable Document Format (`.pdf`) files in the folder `models/<model>/FA_diagrams/` (if this empty directory does not exist in your own model directory it must be created first).

For this example I will produce all one and two loop radiative corrections and counter term diagrams. For this run mode we need to use the `-f` flag (figures) and specify both the model and particle we are interested in. This particle name must be as it appears in the FeynArts model file. First we generate all two loop diagrams

```
mkdir models/Scalar/FA_diagrams
./mass_builder -f -m Scalar -p S[1]
```

next we need to specify additional flags,

```
./mass_builder -f -m Scalar -p S[1] -l 1
./mass_builder -f -m Scalar -p S[1] -l 1 -c
./mass_builder -f -m Scalar -p S[1] -c
```

for the one-loop, one-loop counter-terms and two-loop counter terms respectively. This will create four files in the directory `models/Scalar/FA_diagrams/`, each file contains up to nine diagrams with numbers underneath. This is the numbering system we will use when referring to specific amplitudes.

### 3.2 Compute amplitudes

The first non-trivial task performed by Mass Builder is taking the computed amplitude for each process and sorting it into a useful form for generating the TSIL interface. That is, we must extract the required basis integrals and their non-zero coefficients. The details of this algorithm are given in section 5.1. For this example we will compute all diagrams in the list `models/Scalar/diagrams.txt`, see section 4 for details on the format of this file. To set this calculation running we simply enter

```
mkdir models/Scalar/output
./mass_builder -a -m Scalar
```

which will tell Mass Builder to compute all diagrams in this specific list for the Scalar model. Alternatively, if only a few diagrams are required one may enter

```
./mass_builder -a -m Scalar -p S[1] -d 1
```

to compute diagram 1, for example. Additional flags may also be entered here, such as `-c` for counter term diagrams or `-l 1` to use one loop order instead. Finally, one may specify an alternative list rather than the default one using the flag `-i` followed by the path to the list file.

### 3.3 Generate code and evaluate

Once the amplitudes have been computed and written into Mass Builder readable format the next step is to generate the TSIL interface. This is conveniently separate from the previous step because computing the amplitudes is time consuming, so this is only done once. However, one may wish to switch on and off different radiative corrections without having to rerun Mathematica.

Mass Builder keeps track of all diagrams which have been computed so we can easily generate the code for every available diagram using the command

```
./mass_builder -g -m Scalar
```

alternatively one may use their own custom list by adding the additional flag `-i` followed by the path to the list file.

Next the generated C++ code must be compiled using the same commands used to make Mass Builder

```
cd build
make
cd ..
```

Now we are finally able to compute the total amplitude using the command

```
./mass_builder -e -i models/Scalar/input.txt
```

where we must explicitly enter the path to an input file which contains values for the masses and couplings. This will return the self energy

```
One-loop self energy of particle S1 = 0.0316688
Two-loop self energy of particle S1 = -3.92488e-05
```

where the particle name has been converted to a more convenient form, this is the form of the particle name which appears in the generated output files.

## 4 Full user guide

### 4.1 Command line interface

The user interface to Mass Builder is via the command line, where all modes of functionality are available depending on the chosen input flags.

The four main modes of operation are determined by the flags `-a` for computing the amplitudes symbolic expressions, `-g` for generating the TSIL interface code, `-e` for the numerical evaluation of the self energy and `-d` to request FeynArts to draw the Feynman diagrams. At least one of these flags is required and if more than one of these flags is given the program will not run.

In addition to the run mode flag there are several additional flags, some optional and some required depending on the mode of operation. All possible flags are

```
Run modes:
-a    compute amplitudes
-g    generate TSIL interface code
```

**Table 1:** The required flags for each run mode behaviour.

| | | | | |
|---|---|---|---|---|
| -a | -m | | | compute all in diagrams.txt |
| -a | -m | | -i | compute all in specified input list |
| -a | -m | -p | -d | compute specific diagram |
| -g | -m | | | generate code for available diagrams |
| -g | -m | | -i | generate code for all in input list |
| -f | -m | -p | | draw all diagrams |
| -e | | | -i | evaluate self energy |

```
-e    evaluate self energy
-f    generate figures from FeynArts
Additional flags requiring input
-m <model>
-p <particle>
-i  <list>
-l <loop_order>
Optional switches
-o    optimise TSIL interface
-c    evaluate counter terms
-v    display Mathematica output during computation
-w    generate code with detailed terminal output
```

where `-o` will be explained in section 5.2 and `-w` will put a `std::cout` statement for each amplitude computed at runtime for detailed inspection of each contribution to the total self energy, as may be useful for identifying large contributions.

### 4.2 Input

All model specific input is stored in the directory `models/<model_name>/`. The required input files are

- `<model_name>.mod` – FeynArts model file
- `masses.txt` – list of masses and identifiers
- `couplings.txt` – list of couplings
- `diagrams.txt` – list of diagrams to compute

which are all stored in the directory `models/<model_name>/`.

The file `masses.txt` can contain either one or two columns. The first, and required, column must contain a list (in no particular order) of the masses exactly as they appear in the FeynArts model file. The second column, which is highly recommended, should contain a, preferably single character, identifier for each mass in the corresponding row. For example a typical masses file would be

```
# masses.txt
MWp          wp
MWm          wm
MZ           z
MA           a
MChi         c
```

where for even more readable output code one could choose unique one character identifiers for `wm` and `wp` instead.

The file `masses.txt` is simply a one column list of couplings and constants exactly as they appear in the FeynArts model file. This is essential for the generated code to compile and for the user input header to contain options for setting these couplings at runtime via an input file [1]. For example a typical couplings file would be

```
# couplings.txt
lambda
g
```

Finally `diagrams.txt` is a list of diagrams to compute. This is identical to the file entered along with the `-i` option at runtime. This file contains at least two columns, the first specifies the particle name in FeynArts format (such as `S[1]`) and the second the corresponding diagram number (to obtain a list of diagrams for each particle in `pdf` output see section 3.1. An optional column may be added to specify the loop order and if this is to be a counter term diagram (if these options are not set globally with the appropriate flags at runtime), including all columns this file would look like

```
# diagrams.txt
F[5]   1   2
F[5]   1   1
F[6]   2   2c
```

which will tell Mass Builder to compute the first diagram for the particle `F[5]` at one and two loop level, and the second two loop counter term diagram for particle `F[6]`. All numbers are in reference to the numbers given with the diagrams as listed in the `pdf` output from `./mass_builder -f -p <particle> -m <model>`.

### 4.3 Output

All output from the amplitude calculation is stored in the directory `models/<model_name>/output` (this empty directory must be created manually before calculation). For typical usage the contents of the `output` directory is not important as this is an intermediate step between computation of the amplitudes and the generated C++ interface to TSIL.

Between computing the amplitudes and generating the code Mass Builder stores the necessary information for each diagram in `models/<model>/output/`. This information is split into four text files

- `basis_integrals_tag.txt` list of required basis integrals

[1] Constants such as `sw2` which regularly appear in couplings must be set, although the final release version will include an option to enter analytical expressions at this stage so they can be set via more sensible inputs at runtime, we will also include many standard expressions by default.

- `coeff_integrals_tag.txt` list of coefficients of the basis integrals in C++ form
- `coeff_products_tag.txt` list of coefficients of the products in C++ form
- `summation_tag.txt` the amplitude as a sum of basis integrals and coefficients

where `tag` encodes the particle name, diagram and loop order (and if this is a counter-term diagram). When necessary the output is written in C++ style for simple implementation into the final code.

## 5 Algorithm details and code structure

### 5.1 Computing the amplitudes

The amplitudes are calculated one diagram at a time using FeynArts, FeynCalc and TARCER which is run externally to C++ using the Mathematica kernel with automatically generated scripts. The goal in this part of the process is to determine the basis integrals which have non-zero coefficients, and what these coefficients are. This separation into *basis integrals* and *coefficients* is the best way to determine which integrals are required in the final numerical calculation and to produce readable and tidy code.

The algorithm begins by evaluating the amplitude $\mathcal{A}$, I then compute the coefficient of every possible basis integral $\{\mathcal{B}_1, \mathcal{B}_2, \ldots\}$. For the non-zero coefficients, $\{\mathcal{C}_1, \mathcal{C}_2, \ldots\}$ I then construct a trail amplitude of the form $\mathcal{A}_{trial} = \mathcal{C}_1 * \mathcal{B}_1 + \mathcal{C}_2 * \mathcal{B}_2 + \ldots$ . I then take the difference $\mathcal{A} - \mathcal{A}_{trial}$ and check for basis integrals with non-zero coefficients, this will find cross terms that have been double counted in the first step. From within the set of basis integrals with a non-zero coefficient at this stage, $\{\mathcal{B}_i, \mathcal{B}_j, \ldots\}$ , I create new "basis integrals" $\mathcal{B}_{ij} B_i * B_j$ which appears to Mathematica as one object. The final amplitude is constructed as

$$\begin{aligned}\mathcal{A}_{trial} \;=\; & \mathcal{C}_1 * \mathcal{B}_1 + \mathcal{C}_2 * \mathcal{B}_2 + \ldots \\ & - \frac{1}{2}\mathcal{C}_{12} * B_1 * B_2 - \frac{1}{2}\mathcal{C}_{21} * B_2 * B_1 - \ldots \\ & + C_{11} B_1 * B_1 + C_{22} * B_2 * B_2 + \ldots\end{aligned}$$

where $\mathcal{C}_{ij}$ is the coefficient of $\mathcal{B}_i * \mathcal{B}_j$ in the original amplitude $\mathcal{A}$. If this does not equal the original amplitude then the program will throw and error and inform the user, see section 7 for details on possible causes of this scenario. All calculations up to this point are symbolic within the generated Mathematica scripts

#### *5.1.1 Basis integral labelling*

A priori we have no information on the basis integrals required for a particular problem. For an amplitude involving multiple particles there are on order hundreds of possible non-degenerate permutations of basis integrals. Thus, when an amplitude is evaluated in Mathematica we have no generic way of identifying the integrals we need to use to reconstruct the result in the form integral times coefficient. So I begin with all possible non-degenerate basis integrals, and quickly determine which ones have a non-zero coefficient in the resulting amplitude. The computational time required for this process is negligible and is achieved through the use of the `Coefficient[ Amplitude, Integral ]` Mathematica routine. Therefore we use this "brute force" method to reliably determine the basis integrals we require without any notable computational penalty.

During this procedure, and in the resultant generated C++ code, we need a unique identifier for each basis integral. However, if the input masses are strings of more than one character, for example `mHp`, `mA0`, and `mW`, then the obvious way to name the basis integral, $F$(`mHp`,`mHp`,`mA0`,`mA0`,`mW`) would be `F_mHpmHpmA0mA0mW` which along with being difficult to read can led to ambiguous labelling of integrals. For example if one choose the mass labelling to be $(H^-, H^0, \chi) =$ (`mHm`,`mH`,`m`) then we easily have the degeneracy $J$(`mH`,`m`,`mHm`) =`J_mHmmHm`= $J$(`mHm`,`mH`,`m`). When dealing with hundreds of possible permutations it is important to avoid such possibilities, however unlikely they may seem.

To overcome this we assign a unique single character identifier to each mass in the routine `set_id`. This will check for user input, which is the recommend action, or in the absence of this input it will attempt to assign a unique identifier to each mass. However, this alone is not sufficient as the original FeynArts model file, and subsequent expressions will contain the original masses, so we must retain this information along with the unique identifier for each basis integral. Therefore we create a C++ map to map the short name, using the identifiers, to a simple class of type `Bases` which holds the following information

```
class Bases
{
public:
string type = "";
string e1 = " ", e2 = " ", e3 = " ", e4 = " ", e5 = "
    ";
string coefficient = "";
string short_name = "";
Bases() {}
<constructors>
};
```

where we also we provide a constructor for each number of elements (masses). For example the basis integral $V$(`mHp`,`mA0`,`mA0`,`mW`) is initialised as

```
Bases base("V",mA0,mA0,mW);
```

which we then save in `std::map<std::string, Bases>` to the integrals short name.

This set up significantly simplifies the entire algorithm, as we no longer need to pull apart basis integral identiiers, such as `F_abcde` character by character to reconstruct and print out the integral in a useful form for either FeynCalc or TSIL, and indeed this would not be possible if any of the identifiers were not a single character. This also enables a huge flexibility in the mass labelling, in practice one may use whatever name they want for the masses without sacrificing final code readability.

## 5.2 The TSIL interface

The generated C++ interface to TSIL is organised on a diagram by diagram basis. However, during the generation of this code the basis integrals required for all diagrams in the chosen set are amalgamated and reduced to a minimalistic set. This set is evaluated in one function and made globally available to the rest of the functions in the script.

The basis integrals are evaluated using the TSIL libraries. The function used, and the corresponding computation time required, depends on the integral required. In the most general case the `TSIL_Evaluate` function is called with 5 mass parameters which will evaluate most of the possible basis integrals. This is also the most time consuming method, however it is required for any of the $M$ or $V$ integrals. Therefore, when we need to call this function we should make sure to also extract any other basis integrals we require to minimise the number of calls required.

In general the possible basis integrals available from each `TSIL_Evaluate` call forms a set of over 30 elements, owing largely to the symmetries between integrals, each of which is extracted using a unique identifying string. As there is no additional computation overhead for extracting these integrals once they are already calculated, if we *must* use `TSIL_Evaluate` for a $M$ or $V$ integral, then we should simultaneously extract all other required integrals that are useful for our problem.

While each call to `TSIL_Evaluate` can compute over 30 integrals, conversely for each basis integral there are multiple arguments that can be passed to the evaluate routine to get the same integral out. Thus we want to find the optimal parameters to pass to `TSIL_Evaluate` to get the maximum number of useful integrals out of it.

We provide a class capable of taking an input list of basis integrals, and providing a correctly formatted set of calls to the TSIL libraries which minimises the computational time required. This significantly increases the time required to generate the code (up to a couple of minutes), due to the huge sorting problem involved, yet will save time if many evaluate calls are going to be required. To invoke this option the flag `-o` must be passed along with the generate call. An example of generated output is

```
TSIL_SetParameters (&bar,mc2, ma2, ma2 , mc2 , mc2,
    Q2);
TSIL_Evaluate (&bar, s);
Fcaacc =   TSIL_GetFunction (&bar,"M");
Jcaa   =   TSIL_GetFunction (&bar,"Svzy");
Jccc   =   TSIL_GetFunction (&bar,"Svxu");
Taca   = - TSIL_GetFunction (&bar,"Tzvy");
Tcaa   = - TSIL_GetFunction (&bar,"Tvzy");
Tccc   = - TSIL_GetFunction (&bar,"Tvux");
Vaacc  = - TSIL_GetFunction (&bar,"Uyuvz");
Vcaac  = - TSIL_GetFunction (&bar,"Uyuzv");
Vccca  = - TSIL_GetFunction (&bar,"Uxzvu");
```

where all integrals evaluated here have been explicitly requested by the user input. compared to the naive case where each integral is evaluated one at a time using the full 5 parameter input when necessary or alternative faster functions when possible, which is computationally less efficient in any case but quicker to generate.

The generated code, located in `src/self_energy.cpp` takes the following structure

```
TSIL_COMPLEXCPP  <basis integral declarations> ;
TSIL_REAL  <mass declarations>;
TSIL_REAL  <coupling declarations> ;

void DoTSIL(TSIL_REAL s,TSIL_REAL Q2)
{
< TSIL basis integral evaluations >
}

void init(Data data)
{
< set couplings & masses from data >
}

TSIL_COMPLEXCPP  diagram_1()
{
TSIL_COMPLEXCPP C =  <Coefficient>;
return  + C * <basis_integral>;
}

TSIL_COMPLEXCPP  diagram_2()
{
TSIL_COMPLEXCPP C =  <Coefficient>;
return  + C * <basis_integral>;
}

void Self_energy::run_tsil (Data &data)
{
TSIL_COMPLEXCPP SE_particle = diagram_1() + diagram_2
    ();
data.SE["particle"] = real(SE_particle);
}
```

where we have one subroutine to call TSIL and compute the basis integrals, and a subroutine for each

diagram, where the subroutine names will encode the particle name, diagram number and loop order (and if it is a counter term diagram or not). The routine `run_tsil` will fill the self energy map for each available particle (in practice we have a map for both the one and two loop self energies separately, `SE_1` and `SE_2`).

Along with the above source code a header file, `data.hpp`, is also generated in the `include/` directory to hold the model data. This header contains a class definition of type `Data` which is designed to manage the input and output of information from the self energy calculator. This class contains declarations for each coupling defined in `couplings.txt`, and for each mass in `masses.txt`. It also holds a vector of strings with the name `avail_part` containing the short names of all particles for which amplitudes are available, along with two maps of type `map<std::string,double>` `SE_1` and `SE_2` which hold the names of the particles and the one-loop and two-loop self energies respectively . Finally, it includes the functions which read the runtime input of values for the couplings and masses relevant for this model. By dynamically updating this class when generating the self energy interface we enable user input of these quantities and a dynamic mapping interface to other functions in the code.

Before code is generated `self_energy.cpp` is a skeleton necessary for the rest of Mass Builder to compile successfully. If `self_energy.cpp` or `data.hpp` becomes corrupted and the rest of the code no longer compiles, which is likely if `couplings.txt` is missing a variable name, then the skeleton code can be restored by simply running scripts/config.sh again.

The diagrams available to be included in the generated TSIL interface are registered in `models/<model>/output/avail_diagrams.txt` which is updated each time a new diagram is computed (it is also checked for duplicate entries, so no diagram, particle, and type combination appears twice). However, if using the `-i` option with the generate code mode, then it is possible for duplicate diagrams to appear (we choose not to override this possibility to avoid unnecessary interference with user input).

### 5.3 Runtime

The calculation of the amplitudes depends on the performance of the tools we are using. The time taken depends strongly on the type of two-loop topology and the number of distinct particles involved. We find run times range from less than a minute to several hours. Fortunately, with this interface tool once a diagram is computed it need not be computed again.

The optimisation routine can take some time to complete, yet is only really nessecery when many evaluations of the amplitudes are required, in which case some time to set this up will pay off in the long term. For example with optimisation routine employed we reduce the runtime for the calculation of an electroweak triplet model with 123 2-loop diagrams and 5 1-loop diagrams from 5.7 seconds to 1.7 seconds. In this instance the optimisation routine took 2 minutes 37 seconds to complete. Thus after only 40 iterations using the optimisation routine has given an advantage here. In doing so we reduced the number of `TSIL_EVALUATE` calls using all 5 mass parameters from 54 to 29, where the original 54 does not include the partial `ST_evaluate` calls for the $S$ and $T$ functions which we have now combined under the more general calls which are already necessary.

## 6 Applications

### 6.1 Electroweak mass splittings

With Mass Builder we provide an optional executable to demonstrate more advanced use of the program. This example calculates mass splitting in an electroweak triplet using one or two loop self energies. Before making this executable the appropriate TSIL interface code must generated, as it relies on the `Data` class containing particular masses and particle names.

To build this example with 1-loop self energies run the following commands

```
./mass_builder -a -m MDM -i models/MDM/example_1.txt
./mass_builder -g -m MDM -i models/MDM/example_1.txt
cd build
make
make example
```

where this list contains all one-loop diagrams. After building the example executable with the above commands now return the root directory and run

```
./example -i models/MDM/input.txt
```

where the input list here contains the required standard model couplings and masses. This will produce a data file in the model output directory called `mass_splittings.txt`. If you have Python installed this can be plotted simply by running

```
python src/plot_example.py
```

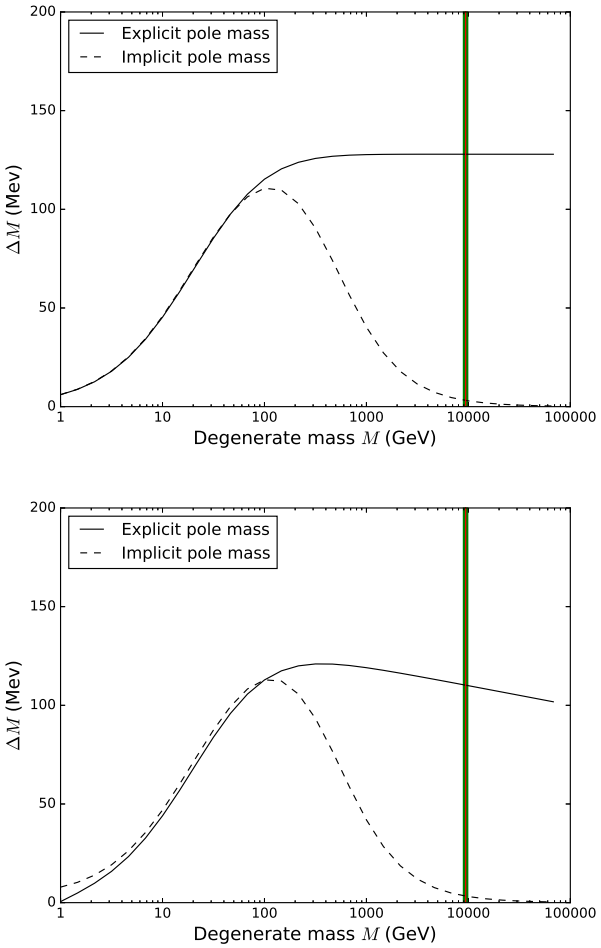


**Fig. 1:** The mass splittings in an electroweak triplet model with a massive photon resulting from one-loop (*top*) and partial two-loop radiative corrections (*bottom*).

### 6.2 Two Higgs doublet model

## 7 Diagnosing errors

In this section I give a brief outline of the possible problems that can be encountered with trying to implement new models in to Mass Builder, along with suggested solutions.

### 7.1 Amplitude computation errors

The most challenging part of the Mass Builder algorithm is the careful extraction of coefficients and basis integrals to reconstruct a given amplitude. If the Mathematica routines encounter computational problems themselves, then this step will either succeed trivially (which may not be immediately apparent to the user) or fail. The first step in a case where a trivial success is suspected (such as empty output files) or an error message appears is to run the same calculation with the `-v` flag to display all Mathematica output at runtime.

The case of trivial success would primarily occur when the Mathematica script is not correctly located. In this case using the `-v` flag will make this immediately clear. Subsequently if the Mathematica kernel has started successfully, it will be evident from this output if the FeynCalc, FeynArts or TARCER packages are not properly loaded.

Once it is confirmed Mathematica is running the next step is to check that all the required masses are given in the `<model>/masses.txt` file. If Mathematica output is being printed to the terminal, then inspection of the final amplitude displayed for any missing masses is a good starting point. If a mass is missing, the entire computation of this amplitude must be repeated[2].

With the trivial case aside, there are unfortunately too many possible reasons why the calculation of the amplitudes might not succeed for reasons beyond what we can control in this interface program. However in almost all cases these problems can be resolved by looking at the Mathematica output and working through the problems for the relevant Mathematica package.

### 7.2 Compilation errors for generated code

This type of error occurs when the amplitudes have supposedly been calculated correctly, the code generated, but either the generated source code `self_energy.cpp` or data header `data.hpp` are corrupted.

The most likely cause of this type of error is an missing coupling or constant from the input files `<model>/couplings.txt` or a mass from `<model>/masses.txt`. This should be easy to identify from the compile time error for an undefined parameter which appears in the generated amplitude but for which Mass Builder has been instructed to create a declaration for. This is rectified by adding the required parameter to the input list and regenerating the code, the amplitudes *do not* need to be recomputed.

In other instances an undefined function may appear in the generated amplitudes. These functions may some Mathematica function associated with FeynCalc or FeynArts. Take for example the CKM mixing matrix which

[2] If an extremely long calculation has been run, and only at the end was it realised a mass was missing, then it *is* possible to avoid repeating it as the amplitude is saved untouched in the Mathematica data file in `output/stage_3.mx` until being overwritten when the next amplitude is computed. However, restarting the algorithm using this requires hacking of the code and should only be a last resort, we do not support such a feature as checking masses before hand should always be done anyway.

may be written as `CKM[i,j]` in the FeynArts model file. Mass Builder is not yet able to deal with *functions* as couplings, but these can be added by the user manually or removed from the FeynArts model file. In the case of the CKM matrix if quark mixing is not required this can be set to the identiy with the line `CKM = IndexDelta` to the FeynArts model file.

Generally, one may add an undefined function into `self_energy.cpp` after code generation, or directly into `generate_code.cpp` if it is going to be used repeatedly. For example, the Mathematica function `Complex[a,b]` is not handled properly by the CForm output, and appears as `Complex(a,b)` in our generated output. Therefore we add the following line to the appropriate location in `generate_code.cpp`

```
<< "TSIL_COMPLEXCPP Complex(double a,double b){
    TSIL_COMPLEXCPP result = a + i*b; return result
    ;}\n"
```

which will print this function definition into the header of `self_energy.cpp`. This solves the problem of this function ever being undefined if it does appear in the generated output, and a similar approach can be followed for any function which appears in the generated output.

### 7.3 Choice of basis integral representation

When all possible basis integrals are constructed in Mass Builder only unique integrals are considered. Integrals which are equal to others, but with a different representation, for example $B(x, y) = B(y, x)$ are not explicitly added to the list. Therefore, if some user operation which requires one of these, but by chance not the one that is in our list, problems can occur if the situation is not handled properly. One case where this may occur is the following.

It is possible, although not recommend and tedious, to construct a TSIL interface without using Mathematica at all. The input may be directly written into `models/output/` in the required form. However caution must be taken to ensure all basis integrals are considered. When the amplitude calculation algorithm is run, only a certain set of basis integrals are considered due to symmetries. When the code is generated, only these basis integrals are searched for in the input files, using the same procedure to construct a comparison set and thus correctly read the short name formats. For example, `Jxyz` may be expected by the code generation routine but `Jxzy` may not be and thus would not be identified and handled correctly.

## 8 Conclusion

I introduce a program designed to organise and simplify the use of two-loop tools for the calculation of self energies. Although entirely an interface tool, this program makes the calculation of multiple two-loop diagrams an accessible task even on modest computing set ups.

This program provides a central structure for carrying out and storing the results of long calculations. By producing an automatically generated interface to the TSIL libraries we enable maximum flexibility for the users choice of precomputed amplitudes to include in a calculation.

The TSIL interface provides an automated method of organising basis integrals into sets which can be evaluated using a single TSIL call, a task near impossible by hand, thus taking advantage of the structure of the TSIL libraries to speed up the calculation of the amplitude. This is especially useful when one is switching between sets of amplitudes to compute, with the optimal combination of evaluation routines changing each time. Even as a standalone feature, this is useful to those who have already obtained a list of required basis integrals from elsewhere and intend to write their own TSIL interface.