Mass builder – an interface tool for automated self energy calculation

James McKay[a,1]

[1] Imperial College London

September 6, 2017

Abstract Mass Builder is designed to *build*, up from the level of a FeynArts model file, a C++ computer code to evaluate renormalised masses. This is achieved by generating the necessary Mathematica and C++ scripts to interface with the existing tools, along with sophisticated intermediary sorting. In doing so it provides a new interface between the symbolic amplitudes provided by FeynArts [1], FeynCalc [2, 3] and TARCER [4] and the numerical evaluation of these amplitudes using TSIL [5].

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 non-trivial, let alone the simplification of the resulting integral expressions, and finally the evaluation of these integrals. Fortunately, FeynArts [1], FeynCalc [2, 3], TARCER [4] and TSIL [5] have made each step of this process far more achievable for a wide range of users.

The interface between the tools available for generic two-loop calculations is only complete up to the stage of a symbolic amplitude. Between FeynArts, FeynCalc and TARCER exists the necessary conversions, yet the final step of numerical evaluation requires significant user intervention. However, for one-loop calculations this process is available with various existing tools. The recently released FeynHelpers [6] serves this purpose by providing analytic one-loop amplitudes, and other existing codes have been able to do this by making use of the LoopTools package [7], such as SARAH [8, 9] interfaced to either SPheno [10] or FlexibleSUSY [11].

The TSIL libraries provide numerical, and in some cases analytical, evaluation of the basis integrals which appear in a two-loop self energy. However, in order to make use of these one must construct a C++ interface to call the TSIL libraries and then evaluate their amplitude. Although the TSIL functions are extremely user-friendly, making use of them from a symbolic Mathematica expression is extremely non-trivial. Therefore we provide Mass Builder which is designed to automate this task by generating the C++ interface.

In addition to providing an automated framework we are also able to split the calculation of many loop diagrams into manageable pieces. 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.

2 Installation

Before beginning the following programs are required

- Mathematica 9.0
- FeynCalc 9.2 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.

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[]
```

(being careful to avoid any spaces which appear in the link when copy-pasting this) 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 (TSIL) can be downloaded from http://www.niu.edu/spmartin/TSIL/. Mass Builder has been tested with version 1.4. It may installed anywhere (as Mass Builder will request the path at configuration).

2.3 Mass Builder

Mass Builder can be downloaded from https://github.com/JamesHMcKay/Mass_builder.git.

Mass Builder is built using cmake with the following commands

```
mkdir build
mkdir output
cd build
cmake -DTSIL_PATH=/path/to/tsil-1.4/ ..
make
```

where you must specify the location of the TSIL directory as a flag to the cmake call. 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 we will produce all one- and two-loop radiative corrections and counter-term diagrams. For this run mode we need to use the -f flag 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. To perform this calculation we run

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

which will tell Mass Builder to compute all diagrams in the default list models/Scalar/diagrams.txt, see section 4 for details on the format of this file. Alternatively, if only a few diagrams are required one may enter

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

to compute diagram one, 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 the whole computation.

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. If code has previously been generated then one must first run ./scripts/clean.sh before the above step, otherwise existing, now incompatible, files will be detected by the cmake system.

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

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

we must run cmake again as it must find the generated source files. 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 = 2.91938e-05
```

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

We also provide detailed output in the file LaTeX_table.tex written to the model's output directory. The columns of this file are particle name, loop order (with a "c" suffix if a counter-term diagram), diagram number and amplitude in GeV.

Finally, with the one-loop amplitudes computed we may determine the required one-loop counter-term coupling using the command

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

This will solve for the counter-term coupling to give the result

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
-b	-m	-p			determine one-loop counter-term

```
Counter-term coupling = -(Power(g,2) + lambda*Power(
    Ms,2))/(32.*Power(Pi,2))
```

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 `-f` 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
-e    evaluate self energy
-f    generate figures from FeynArts
Additional flags requiring input
-m <model>
-p <primary particle>
-q <secondary particle>
-i  <list>
-l <loop_order>
Optional switches
-o    optimise TSIL interface
-c    use counter term diagrams
-v    display the Mathematica input code
-w    generate code with detailed terminal output
-s    maximal file splitting of TSIL interface
-O    calculate physical mass of spin zero field
-r    specify restrictions for FeynArts model
```

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

The `-r` flag will add the text following the flag exactly as is into the FeynArts function `InsertFields[ . . . Restrictions -> { <input> } . . . ]`. This will imply the desired restriction onto the possible set of diagrams generated. This should be used consistently across all commands as the number of allowed diagrams will change, and thus so will the numbering of each diagram.

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.

If a mass is set to zero in the FeynArts model file, with the line `Mass -> 0`, and the user does not wish to replace this with a finite mass for the purposes of the calculation, then the following line must be used in `masses.txt`

```
# masses.txt
null n
```

where `n` can be any identifier as long as it is unique in the list. No further reference to `null` or `n` is required in input file at the numerical evaluation step as Mass Builder will automatically assign zero to any `null` terms appearing in the TSIL interface code.

The file `couplings.txt` is a 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. There is also the option to specify derived couplings and the corresponding analytic expression. All derived couplings and the corresponding relationships must be specified first in the list, as in the example below. The couplings file would typically look like

```
# couplings.txt
d1 (g*g/2+lambda*Ms*Ms/2)
dlambda 0
lambda
g
```

where the counter-term couplings are set to be $d_1 = g^2/2 + \lambda M_S^2/2$ and $d_\lambda = 0$ and the other couplings are left free to be set at run time. In this case `Ms` must be listed in the `masses.txt` file. Any value or relationship defined in the second column of the `couplings.txt` file will override user input at runtime.

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>`.

There are two additional input files one may place in the model directory when a FeynArts contains notation for the couplings and masses that is not supported by Mass Builder by default. The types of notation not supported are functions, that have not been defined in the generated code, such as `Mass[i]` where i is an index. Another common function appears in patched FeynArts model files, during the patching by FeynCalc many symbols are wrapped to avoid clashes with symbols from FeynCalc and will appear as `FCGV["x"]`.

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
- `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

and a Mathematica data file

- `math_data_tag.mx` stores full divergent amplitude for later recall within Mathematica

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.

The Mathematica data file is essential if one wishes to repeat a calculation using the full divergent amplitude. This is necessary for the computation of the tree-level counter-term, where Mass Builder collects all relevant amplitudes for the particle in question and then sums these together before extracting the divergent piece. By keeping this file we loose no information from the original calculation.

4.4 Tree-level counter-term coupling calculation

It is possible to compute the tree-level counter-term coupling given the sum of the loop corrections at one-loop. Since the tree-level counter-term is the only counter-term of one-loop order, we only need to solve one equation to demand no divergences of order $1/\epsilon$. To automatically compute this coupling one first needs to compute all the one-loop amplitudes, and then use the `-b` flag followed by the model and particle identifier, along with the `-v` flag to display the Mathematica output which will print the result to the terminal.

4.5 Example evaluation routines

The self energies are available to external functions via the `data` structure. This is useful for including the results into other routines, or doing further manipulations to the self energies. We provide example source codes to demonstrate different levels of complexity for communicating with the TSIL interface. `Scalar.cpp` is the most basic example of retrieving the one and two-loop self energies. `MSSM.cpp` computes pole masses and compares these via different methods of calculation. `VDM.cpp` will do the same for a vector dark matter model. `EW_triplet.cpp` will do the same again, yet it also includes manually created expressions for the derivatives of the one-loop self energies. This demonstrates how one may add additional integrals by hand that make use of the TSIL libraries.

All example routines are located in the folder `examples/` and are compiled with `make <name>` where `<name>` is the source file name. Note that for each example the corresponding self energies must be generated first, otherwise a null result will be returned. It is straight forward to add similar routines following the syntax used `CMakeLists.txt` for additional targets.

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 $\mathcal{A}$, it then computes the coefficient of every possible basis integral $\{\mathcal{B}_1, \mathcal{B}_2, \ldots\}$. For the non-zero coefficients, $\{\mathcal{C}_1, \mathcal{C}_2, \ldots\}$ it then constructs a trail amplitude of the form $\mathcal{A}_{trial} = \mathcal{C}_1 * \mathcal{B}_1 + \mathcal{C}_2 * \mathcal{B}_2 + \ldots$. The difference $\mathcal{A} - \mathcal{A}_{trial}$ is then evaluated and checked 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, $\{\mathcal{B}_i, \mathcal{B}_j, \ldots\}$, it then creates new "basis integrals" $\mathcal{B}_{ij} = \mathcal{B}_i * \mathcal{B}_j$ which appears to Mathematica as one object, for which coefficients $\mathcal{C}_{ij}$ are evaluated. The final amplitude is then constructed as

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

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 **??** for details on possible causes of this scenario. All calculations up to this point are symbolic within the generated Mathematica scripts.

The basis integrals are then expanded into divergent and finite pieces (see Section 5.3) and the above steps are repeated. It is then possible that the difference $\mathcal{A} - \mathcal{A}_{trial}$ is non-zero, yet contains no basis integrals. In this case this remainder is retained and added to final amplitude as the coefficient `C0` which is simply added to the final sum.

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/clean.sh`.

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 Subtraction of divergences

The amplitudes produced by TARCER are expressed in terms of divergent basis integrals. In a consistent field theory these divergences should be accounted for by divergent counter-term diagrams. Mass Builder offers the ability to compute counter-term diagrams and also compute the analytical form of the two-point tree-level counter-term coupling. The determination of higher order counter-terms and those with more vertices are left to the user to determine via other means. However, in most two-loop calculations only the tree-level counter couplings are required, as these are the only counter-terms which appear in the two-loop order counter-term diagrams. Once we are assured divergences are accounted for, and we have the necessary counter-term couplings, we need to seperate the amplitude into finite and divergent pieces.

The TSIL package provides the evaluation of the finite parts of the basis integrals. However, these basis integrals are not the only finite contributions to the amplitude. For example, if the divergent piece of the basis integral is of order $1/\epsilon$ and the basis integral had a coefficient containing a term linear in ϵ, then this leading divergence becomes a finite contribution that must be included. Thus we must appropriately take $D = 4 - 2\epsilon$ and be careful not to loose any finite contributions. This non-trivial step requires an additional repetition of the algorithm described in Section 5.1 to deconstruct the new, finite, amplitude into a coefficient and basis integral form.

In this section we give details of the leading divergences included from the TSIL documentation and how these are adjusted for sign conventions between the TARCER and TSIL packages. We also present our own notation which the generated TSIL interface is written in terms of. In the following bold face and standard face characters denote the divergent and finite basis integrals respectively. We use A, B and so on to denote the TSIL definitions and TAI, TBI and so on to denote the TARCER definitions.

In TSIL notation we can express the divergent basis integrals as [5]

$$
\begin{aligned}
&\mathbf{A}(x) = A(x) - \frac{x}{\epsilon} + \epsilon A(x) + \mathcal{O}(\epsilon^2) \\
&\mathbf{B}(x,y) = B(x,y) + \frac{1}{\epsilon} + \epsilon B_\epsilon(x,y) + \mathcal{O}(\epsilon^2) \\
&\mathbf{I}(x,y,z) = I(x,y) + \frac{(x+y+z)}{2\epsilon^2} \\
&- \left[A(x) + A(y) + A(z) - (x+y+z)/2\right]/\epsilon \\
&+ A_\epsilon(x) + A_\epsilon(y) + A_\epsilon(z) + \mathcal{O}(\epsilon) \\
&\mathbf{S}(x,y,z) = S(x,y) - \frac{(x+y+z)}{2\epsilon^2} + \\
&\left[A(x) + A(y) + A(z) - (x+y+z)/2 + s/4\right]/\epsilon \\
&+ A_\epsilon(x) + A_\epsilon(y) + A_\epsilon(z) + \mathcal{O}(\epsilon) \\
&\mathbf{T}(x,y,z) = T(x,y) + \frac{1}{2\epsilon^2} - \left[A(x)/x + 1/2\right]/\epsilon \\
&+ (A(x) - A_\epsilon(x))/x + \mathcal{O}(\epsilon) \\
&\mathbf{U}(x,y,z,u) = U(x,y,z,u) + \frac{1}{2\epsilon^2} + \left[B(x,y) + 1/2\right]/\epsilon \\
&+ B_\epsilon(x,y) + \mathcal{O}(\epsilon)
\end{aligned}
$$

where $A_\epsilon(x)$ and $B_\epsilon(x,y)$ are finite functions defined in (2.30) and (2.31) of Martin (2006) [5]. We need the equivalent expressions for the basis integrals in TARCER notation, as this is what we initially work with. Using the relationships between TSIL and TARCER notation [5]

$$
\begin{aligned}
\mathbf{A}(x) &= i\mathbf{TAI}(x) \\
\mathbf{B}(x,y) &= -i\mathbf{TBI}(x,y) \\
\mathbf{I}(x,y,z) &= \mathbf{TJI}_0(x,y,z) \\
\mathbf{S}(x,y,z) &= \mathbf{TJI}_s(x,y,z) \\
\mathbf{T}(x,y,z) &= \mathbf{TJI}_{s,2}(x,y,z) \\
\mathbf{U}(x,y,z,u) &= -\mathbf{TVI}(u,x,z,y)
\end{aligned}
$$

where we have defined

$$
\begin{aligned}
\mathbf{TJI}_{s,2} &= \mathtt{TJI[D,s,\{\{2,x\},\{1,y\},\{1,z\}\}]} \\
\mathbf{TJI}_{s} &= \mathtt{TJI[D,s,\{\{1,x\},\{1,y\},\{1,z\}\}]} \\
\mathbf{TJI}_{0} &= \mathtt{TJI[D,0,\{\{1,x\},\{1,y\},\{1,z\}\}]}
\end{aligned}
$$

we find the relationships

$$
\begin{aligned}
&\mathbf{TAI}(x) = \mathtt{TAI}(x) + \frac{ix}{\epsilon} - i\epsilon A_\epsilon(x) + \mathcal{O}(\epsilon) \\
&\mathbf{TBI}(x,y) = \mathtt{TBI}(x,y) + \frac{i}{\epsilon} + i\epsilon B_\epsilon(x,y) + \mathcal{O}(\epsilon) \\
&\mathbf{TJI}_0(x,y,z) = \mathtt{TJI}_0(x,y,z) + \frac{(x+y+z)}{2\epsilon^2} \\
&- \left[i\mathtt{TAI}(x) + i\mathtt{TAI}(y) + i\mathtt{TAI}(z) - (x+y+z)/2\right]/\epsilon \\
&+ \epsilon A_\epsilon(x) + \epsilon A_\epsilon(y) + \epsilon A_\epsilon(z) + \mathcal{O}(\epsilon) \\
&\mathbf{TJI}_s(x,y,z) = \mathtt{TJI}_s(x,y,z) + \frac{(x+y+z)}{2\epsilon^2} \\
&- \left[i\mathtt{TAI}(x) + i\mathtt{TAI}(y) + i\mathtt{TAI}(z) - (x+y+z)/2 + s/4\right]/\epsilon \\
&+ \epsilon A_\epsilon(x) + \epsilon A_\epsilon(y) + \epsilon A_\epsilon(z) + \mathcal{O}(\epsilon) \\
&\mathbf{TJI}_{s,2}(x,y,z) = \mathtt{TJI}_{s,2}(x,y,z) - \frac{1}{2\epsilon^2} + \left[A(x)/x + 1/2\right]/\epsilon \\
&- (A(x) - A_\epsilon(x))/x) + \mathcal{O}(\epsilon) \\
&\mathbf{TVI}(x,y,z,u) = \mathtt{TVI}(x,y,z,u) + \frac{1}{2\epsilon^2} + \left[B(x,y) + 1/2\right]/\epsilon \\
&+ B_\epsilon(x,y) + \mathcal{O}(\epsilon).
\end{aligned}
$$

These relationships are used to obtain the finite amplitude as an intermediate step in the Mass Builder algorithm. The resultant finite amplitude appearing in the generated TSIL interface is written in terms of TARCER basis integrals, which we express in our own convenient notation, which is related to the TSIL integrals as

$$
\begin{aligned}
\mathcal{A}(x) &\equiv -iA(x) \\
\mathcal{B}(x,y) &\equiv iB(x,y) \\
\mathcal{K}(x,y,z) &\equiv I(x,y,z) \\
\mathcal{J}(x,y,z) &\equiv S(x,y,z) \\
\mathcal{T}(x,y,z) &\equiv -T(x,y,z) \\
\mathcal{V}(u,x,z,y) &\equiv -U(x,y,z,u).
\end{aligned}
$$

These relationships are used to convert the numerical result from the TSIL integrals, which we evaluate in the `DoTSIL` routine, into the form appearing the amplitudes.

5.4 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

Table 2: Parameters, ranges and central values of the test scans of this section, for each scan dimensionality. The ranges for most SM parameters correspond to $\pm 3\sigma$ variations around the 2014 PDG central values [?]. For the Higgs, the range is $\pm 4\sigma$ about the 2014 central value (which encompasses the 2015 4σ range [?]). For the up and down quark masses, we take the central values from the 2014 review, and scan over a range of $\pm 20\%$ around the central values. This is intended to capture the $\pm 3\sigma$ range implied by the likelihoods in PrecisionBit [?], which deal with correlated mass-ratio measurements. The nuclear couplings also incorporate a range of $\pm 3\sigma$ around the best estimates. The dark matter density has an asymmetric range about the central value, as the likelihood that we apply to this parameter is log-normal rather than Gaussian. We refer the reader to Refs. [? ?] for further details and references on the central values and uncertainties associated with the local density and nuclear parameters.

Parameter		Values
Scalar pole mass	m_S	$[45, 10^4]$ GeV
Higgs portal coupling	λ_{hS}	$[10^{-4}, 10]$
Varied in 7 and 15-dimensional scans		
Electromagnetic coupling	$1/\alpha^{\overline{MS}}(m_Z)$	127.940(42)
Strong coupling	$\alpha_s^{\overline{MS}}(m_Z)$	0.1185(18)
Top pole mass	m_t	173.34(2.28) GeV
Higgs pole mass	m_h	125.7(1.6) GeV
Local dark matter density	ρ_0	$0.4^{+0.4}_{-0.2}$ GeV cm^{-3}
Varied in 15-dimensional scans		
Nuclear matrix el. (strange)	σ_s	43(24) MeV
Nuclear matrix el. (up + down)	σ_l	58(27) MeV
Fermi coupling $\times\ 10^5$	$G_{F,5}$	1.1663787(18)
Down quark mass	$m_d^{\overline{MS}}(2\,\mathrm{GeV})$	4.80(96) MeV
Up quark mass	$m_u^{\overline{MS}}(2\,\mathrm{GeV})$	2.30(46) MeV
Strange quark mass	$m_s^{\overline{MS}}(2\,\mathrm{GeV})$	95(15) MeV
Charm quark mass	$m_c^{\overline{MS}}(m_c)$	1.275(75) GeV
Bottom quark mass	$m_b^{\overline{MS}}(m_b)$	4.18(9) GeV

with optimisation routine employed we reduce the run-time 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 1 minutes 45 seconds to complete. Thus after only 26 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 when possible we have combined under the more general calls if they 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-loop (two-loop soon!) self energies. Before making this executable the appropriate TSIL interface code must generated, as it relies on the `Data` class containing particular masses and particles.

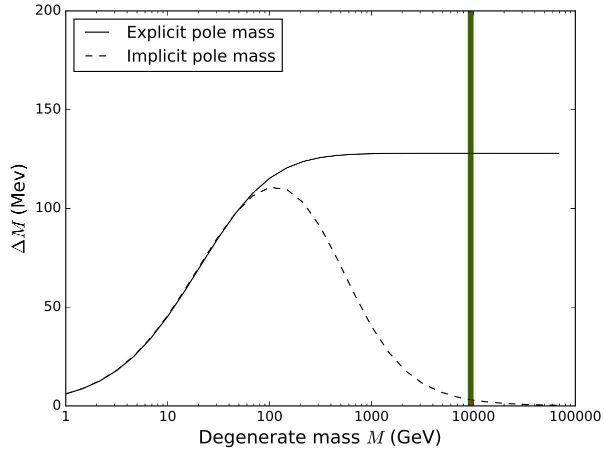

Fig. 1: The mass splittings in an electroweak triplet model with a massive photon resulting from one-loop radiative corrections. The red vertical line (and green one sigma confidence interval) indicates the physically relevant mass scale which would produce the observed relic abundance of dark matter.

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

```
mkdir models/MSSM/output
./mass_builder -a -m MSSM -i models/MSSM/lists/
    example_1.txt
./mass_builder -g -m MSSM -i models/MSSM/lists/
    example_1.txt
cd build
cmake .
make MSSM
```

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

```
./MSSM -i models/MSSM/input.txt
```

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

```
python examples/plot_example.py
```

which will produce a figure called `mass_splittings.eps` in the Mass Builder root directory.

7 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.

References

1. T. Hahn, *Generating Feynman Diagrams and Amplitudes with FeynArts 3*, *arXiv* (2000) hep–ph/0012260v2, [hep-ph/0012260v1].
2. R. Mertig, M. Bohm, and A. Denner, *Feyn Calc - Computer-algebraic calculation of Feynman amplitudes*, *Computer Physics Communications* **64** (1991) 345–359.
3. V. Shtabovenko, R. Mertig, and F. Orellana, *New Developments in FeynCalc 9.0*, *arXiv* (2016) 1601.01167, [arXiv:1601.01167].
4. R. Mertig and R. Scharf, *TARCER - A Mathematica program for the reduction of two-loop propagator integrals*, *Computer Physics Communications* **111** (1998) 13, [hep-ph/9801383].
5. S. P. Martin and D. G. Robertson, *TSIL: A program for the calculation of two-loop self-energy integrals*, *Computer Physics Communications* **174** (2006) 133–151, [hep-ph/0501132].
6. V. Shtabovenko, *FeynHelpers : Connecting FeynCalc to FIRE and*.
7. U. Karlsruhe, *Automated one-loop calculations in four and D dimensions*, .
8. F. Staub, *SARAH 4: A tool for (not only SUSY) model builders*, *Computer Physics Communications* **185** (2014) 1773–1790, [arXiv:1309.7223].
9. F. Staub, *Exploring new models in all detail with SARAH*, *arXiv* (2015) [1503.04200].
10. W. Porod, *SPheno, a program for calculating supersymmetric spectra, SUSY particle decays and SUSY particle production at e+e- colliders*, *Computer Physics Communications* **153** (2003) 275–315, [0301101].
11. P. Athron, J.-h. Park, D. Stöckinger, and A. Voigt, *FlexibleSUSY – A spectrum generator generator for supersymmetric models*, *Computer Physics Communications* **190** (2014) 45, [arXiv:1406.2319].