Mass builder – an interface tool for automated self energy calculation

James McKay

March 21, 2018

Abstract

Mass Builder is designed to *build*, up from the level of a 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], reduced by TARCER [4] and FIRE [5] and the numerical evaluation of these amplitudes using TSIL [6].

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], FeynHelpers [3], FIRE [5], TARCER [4] and TSIL [6] 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 [3] 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 user-friendly, making use of them from a symbolic Mathematica expression is non-trivial. Therefore we provide Mass Builder which is designed to automate this task by generating the C++ interface and automatically managing the whole computation process.

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.

Mass Builder has successfully been used to compute full two-loop self-energies in the minimal dark matter quintuplet model (MDM) and the wino limit of the minimal supersymmetric standard model (MSSM) [13]. It has also been used for a study of the pitfalls of using an iterative procedure for computing electroweak mass splitting [12]. All routines used to produce the results in these studies are available with the Mass Builder distribution. This includes additional code with an interface to the FlexibleSUSY spectrum generator to provide the most precise and consistent input parameters.

1.1 Installation

Mass Builder can be downloaded from https://github.com/JamesHMcKay/Mass_builder.git. 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 1.41
- cmake 3.4.0.

For additional functionality and use of the routines to generate the results appearing in Refs. [12] and [13] FlexibleSUSY 1.7.4 is also required. The Mass Builder C++ code has been tested using gcc versions 4.8.4 and 5.2.0. The Mass Builder executable is built using cmake with the following commands

```
mkdir build
mkdir output
cd build
cmake -DTSIL_PATH=/path/to/tsil-1.41/ ..
make
```

The main executable is now located in the root directory.

1.2 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.

Generate FeynArts diagrams

It is important to check the diagrams that are involved in a self-energy calculation and assign a consistent numbering system to identify each process. FeynArts has the capability to produce a the relevant Feynman diagrams which we store in the folder `models/<model>/FA_diagrams/`. All relevant two-loop self-energy diagrams are generated using the commands

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

while the one-loop and counter-term diagrams can be generating by specifying additional flags `-l 1` and `-c` respectively.

Compute amplitudes

Next we compute the amplitudes and extract the coefficients and required basis integrals, storing these for later use. This is achieved with the commands

```
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.

Alternatively, if only a few diagrams are required one may enter

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

to compute the first two-loop diagram, 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 (two-loop is the default setting). One may also specify an alternative list rather than the default one using the flag `-i` followed by the path to the list file.

Next we need to solve for the one-loop counter-term couplings. This is done automatically using the command

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

This will solve for the counter-term couplings `d1M` and `d1Z` to give the result which is automatically written into the file models/Scalar/couplings.txt[1].

Generate code and evaluate

Once the amplitudes have been computed and stored in the Mass Builder format of basis integrals and coefficients 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. In such a way the generation of code can be done repeatedly, using different combinations of diagrams, without the need to recompute them.

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

[1]We provide additional high-order counter-term couplings, which are not as trivial to compute, in this file as well. However, these are not required for computing the finite part of the two-loop self energy. Instead these are only required when checking that the amplitude are divergence free.

```
./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 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 ..
```

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, which is the name appearing 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.

2 Full user guide

2.1 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. These flags are either *run mode* flags, *input* flags or *option* flags. The input flags, definitions and default values are given in Table 1.

Table 1: The definitions, required input and default values for command line flags used when calling Mass Builder. Input flags must be followed by a string, number or path to an input file. All option flags control boolean parameters, use of the flag will result in the parameter switching to the opposite value from the default.

Input	Definition	Default value
	Run mode flags (specify one and only one)	
`-a`	Compute amplitudes	
`-g`	Generate TSIL interface code	
`-e`	Evaluate self-energies	
`-f`	Generate diagrams from FeynArts	
`-b`	Solve for one-loop counter-term	
	Input flags (must be followed by string or integer)	
`-m` m	Specify the model	`null`
`-p` p_1	Specify a particle in FeynArts style	`null`
`-q` p_2	Second particle for mixing amplitudes	p_1
`-d` n	Specify diagram number	`null`
`-i` *file*	Provide an input list for mode `-a`	`diagrams.txt`
`-i` *file*	Provide an input list for mode `-g`	`output/avail_diagrams.txt`
`-i` *file*	Provide an input list for mode `-b`	`output/avail_diagrams.txt`
`-l` n	Work at n-loop order	2
`-r` *input*	Set restrictions for FeynArts model	`null`
`-k` n	Extract terms of order ϵ^n	0
	Option flags	
`-o`	Optimise TSIL interface	`false`
`-c`	Use counter term diagrams	`false`
`-w`	Print value of each diagram	`false`
`-t`	Use FIRE for tensor reduction	`true`
`expole`	Ignore terms proportional to $1/m_\gamma$	`true`
`onshell`	Set $p^2 \equiv \text{mass}^2$ before computing	`true`

Table 2: The minimum combinations of input flags for each run mode and the resultant behaviour. Each input flag must be specified by the appropriate input, either a string, integer or path to an input file.

Input flags					Resulting behaviour
`-a`	`-m`				Compute all diagrams listed in `diagrams.txt`
`-a`	`-m`			`-i`	Compute all diagrams in the specified input list
`-a`	`-m`	`-p`	`-d`		Compute specified diagram
`-g`	`-m`				Generate code for all available diagrams
`-g`	`-m`			`-i`	Generate code for diagrams in specified input list
`-f`	`-m`	`-p`			Produce FeynArts Feynman diagrams for specified particle
`-e`				`-i`	Evaluate self-energy
`-b`	`-m`	`-p`			Solve for one-loop counter-term

One and only one run mode flag must be given. Additional inputs must be specified depending on the run mode. The requirement can be met in multiple ways, as detailed in Table 2. These flags are always followed by a string, number or path to an input list (following the format given in section 2.2). Option flags control boolean parameters and they are not followed by an input, instead their use results in the parameter switching from the default value, as specified in Table 1.

Restrictions can be specified with the `-r` flag, such as excluding certain particles from a model. This 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. The `-k` flag is used to extract the $\mathcal{O}(\epsilon^n)$ part of the amplitude, given an input n. By default $n = 0$ returns the finite amplitude. The total self-energy should be divergence free and use of $n < 0$ should give a zero amplitude when all counter-terms are appropriately set. Use of $n > 0$ will give unreliable results as we do not carry through all terms of order $\mathcal{O}(\epsilon^m)$ where $m > 0$ in our calculations.

The option `expole` will exclude all terms which have a fictitious IR divergence in the final numerical evaluation. The option `onshell` controls if the external momentum is set equal to the mass before the amplitude is computed in FeynCalc and the tensor integral reduction is carried out. This is set to `true` by default as this is the standard practice and is necessary for some reduction to proceed to the most fundamental basis integrals.

The `-o` flag will be explained in section ??. The `-w` flag 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 and diagnostics.

2.2 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 unique 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 the shortened identifier makes the resulting generated code easier to read. With this is mind one could replace `wm` and `wp` with single character identifiers.

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 all parameters (except the masses specified in masses.txt) exactly as they appear in the FeynArts model file. This is essential for the generated code to have declarations for these parameters and for the user input header to contain options for setting these couplings at runtime via an input file. These parameters can be left free and set at runtime or defined in terms of other parameters. These derived couplings and the corresponding relationships must be specified first in the list, followed by undefined parameters, 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 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 1.2. 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]    {1,2,3}     1
F[6]    2           2c
```

which will tell Mass Builder to compute the first two-loop diagram for the particle `F[5]`, the first, second and third one-loop diagrams for the same particle, and the second two loop counter term diagram for particle `F[6]`. Grouping diagrams together can increase the speed if the diagrams are of a similar topology and contain the same masses on internal legs. If the grouping results in a large number of different masses in the calculation, or combinations of very different topologies, it can excessively increase runtime. 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>`. The file which may be provided at runtime with the `-i` option follows exactly the same format.

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"]`.

2.3 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 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. Whereas in the other files we only store information on the finite, $\mathcal{O}(\epsilon)$, part of the amplitude. Thus no information is lost from the original calculation.

2.4 Interface to external 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.

3 Algorithm details and code structure

3.1 Computing the amplitudes

We calculate the amplitudes either one diagram at a time, or in selected groups, using FeynArts, FeynCalc and FIRE, run from C++ via the Wolfram Symbolic Transfer Protocol

(WSTP). We decompose the resultant symbolic amplitudes into lists of coefficients to be applied to basis integrals, and keep a master list of all the basis integrals required.

The algorithm begins by evaluating the finite part of the amplitude $\mathcal{A}$. It then computes the coefficients $\{C_1, C_2, \ldots\}$ of every possible basis integral $\{\mathcal{B}_1, \mathcal{B}_2, \ldots\}$. For the non-zero C_i, it then constructs a trial amplitude of the form

$$\mathcal{A}_{\text{trial}} = C_1\mathcal{B}_1 + C_2\mathcal{B}_2 + \ldots \tag{2}$$

and checks the difference $\mathcal{A} - \mathcal{A}_{trial}$ for the presence of basis integrals with non-zero coefficients, in order to identify any cross-terms that have been double-counted in the first step. From the set of basis integrals $\{\mathcal{B}_i, \mathcal{B}_j, \ldots\}$ with non-zero coefficients at this stage, the algorithm then creates new 'compound basis integrals' $\mathcal{B}_{ij} = \mathcal{B}_i\mathcal{B}_j$, and presents them to Mathematica as unified objects. We can then instruct Mathematica to extract new coefficients C_{ij} for the compound basis integrals. The final amplitude is then

$$\begin{aligned}\mathcal{A}_{trial} &= C_1\mathcal{B}_1 + C_2\mathcal{B}_2 + \ldots \\ &\quad - \frac{1}{2}C_{12}(\mathcal{B}_1\mathcal{B}_2) - \frac{1}{2}C_{21}(\mathcal{B}_2\mathcal{B}_1) - \ldots \\ &\quad + C_{11}(\mathcal{B}_1\mathcal{B}_1) + C_{22}(\mathcal{B}_2\mathcal{B}_2) + \ldots\end{aligned}$$

where C_{ij} is the coefficient of $\mathcal{B}_i\mathcal{B}_j$ in the original amplitude $\mathcal{A}$. We convert these coefficients into C++ format, and generate numerical routines for evaluating both them and the relevant basis integrals.

3.1.1 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 identifiers, 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.

3.2 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).

3.3 Management 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. 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, then file `couplings.txt` is then automatically updated with the new counter-term coupling.

Three and four-point vertex counter-terms, and higher order terms in the two-point couplings, must be determined via a different method. This can be done by computing one-loop corrections to the vertex, or by computing the two-loop self-energy and demanding that is be divergence free. We use the latter approach as we already compute the self-energies, with example Mathematica routines in the `scripts` directory.

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 **??** to deconstruct the new, finite, amplitude into a coefficient and basis integral form.

There are some minor differences between the basis integral notation in the TSIL and TARCER packages. The notation used in Mass Builder for the finite piece of the basis integrals is a combination of these and is related to the TSIL integrals defined in Ref. [6]

as

$$\begin{aligned}
\texttt{Ax} &\equiv -iA(x) \\
\texttt{Bxy} &\equiv \ iB(x,y) \\
\texttt{Kxyz} &\equiv \ I(x,y,z) \\
\texttt{Jxyz} &\equiv \ S(x,y,z) \\
\texttt{Txyz} &\equiv -T(x,y,z) \\
\texttt{Vuxzy} &\equiv -U(x,y,z,u).
\end{aligned}$$

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

The divergences appearing in the amplitudes as poles in ϵ should arise exclusively from UV divergences. If the theory contains IR divergent amplitudes, for example due to a massless gauge boson, then this should be regulated throughout the calculation using a fictitious mass parameter, m_γ. This parameter should remain in the calculation until the numerical evaluation, where one can take $m_\gamma \to 0$. In some cases taking $m_\gamma = 0$ exactly may result in unexpected behaviour, in which case it is sufficient to choose smaller and smaller values until a limit is reached.

The basis integral $T(x,y,z)$ is not defined for small x, so when x is small or zero (such as $x = m_\gamma^2$) we make the replacement $T(x,y,z) \equiv \bar{T}(x,y,z) - B(y,z)\log(x/Q^2)$ [6]. This will cancel with other terms in the amplitude of the form $A(x)B(y,z) = x\left[\log(x/Q^2) - 1\right]B(y,z)$, and because $\bar{T}(0,y,z)$ is finite, will give a total that is IR safe. This step is necessary even for light quark masses on the eV scale, but generally only when there is a large scale hierarchy present (such as a large external momentum and other masses). If a mass is expected to be small, such as m_γ then it can be given a special status within the Mass Builder package. This replacement then happens automatically during the construction of the amplitude. By default this occurs for any masses with the labels in the set `{ma,mf,md,mu,ms,mb,mc,mm,ml,me}`. This can be changed by locating the array `massesSmall` appearing in `bases.cpp` and `utils.cpp`.

It is also possible to encounter 'fictitious' IR divergences. These can arise from including a finite photon mass when attempting to evaluate non-IR divergent diagrams. In this case the amplitude may contain $\mathcal{O}(1/m_\gamma^2)$ terms. However, the sum of the coefficients of these terms is numerically equivalent to zero (i.e. to within a small factor of the floating-point machine accuracy times the largest individual coefficient). We therefore always see numerically that these terms cancel, even if the integral reduction fails to cancel them symbolically. Thus when $m_\gamma \to 0$, the error from the machine precision eventually becomes huge and looks like a physical divergence. This is currently avoided by separating the amplitude into an $\mathcal{O}(1/m_\gamma^2)$ part and the remainder. Then at evaluation the $\mathcal{O}(1/m_\gamma^2)$ is by default ignored. It can be switched back on using the runtime flag `expole` to change the default behaviour as described in section 2.1. Caution must be used here to be sure that it is indeed a fake divergence and not a physical one.

3.4 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 unique mass parameters. We find run times range from less than a minute to several hours. Time can be reduced by grouping similar diagrams, and leaving the most complex diagrams to be computed individually. However, due to the way Mathematica carries out the symbolic calculations, a poor choice of grouping may result in the calculation taking significantly longer than it would for the sum of the amplitudes alone.

The numerical evaluation of the amplitudes is on the order of seconds but can be reduced using the optimisation method described in section 3.2. This optimisation routine can take some time to complete, and increases dramatically when more masses are present, yet is advantageous when many evaluations of the amplitudes are required. For example with the optimisation routine employed we reduced the numerical evaluation of 123 two-loop diagrams and five one-loop diagrams from 5.7 seconds to 1.7 seconds. In this instance the optimisation routine took under two minutes to complete. Thus after only 26 evaluations the optimisation has been worthwhile. The generation of the TSIL interface code is, excluding the optimisation calculation, effectively instantaneous.

4 Conclusion

I have introduced 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.

5 Installing required packages

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.

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 1.41. It may installed anywhere (Mass Builder will request the path at configuration).

FlexibleSUSY

FlexibleSUSY can be downloaded from https://flexiblesusy.hepforge.org/. Mass Builder has only be tested with version 1.7.4 and is not expected to work with version 2.0 and above. Once FlexibleSUSY is installed the models provided in the `extras/flexiblesusy/` directory should be placed in the appropriate places in the FlexibleSUSY root directory, and then installed using the commands

```
./createmodel --name = "EW_triplet MDM"
./configure --with-install-dir=<mass_builder_root_directory>/flexiblesusy/ --with-
    models="MDM EW_triplet" --disable-librarylink
make install-src
```

and then the flag `-DFS=true` must be used with cmake before building the `splittings` program.

References

[1] T. Hahn, *Generating Feynman diagrams and amplitudes with FeynArts 3*, *Computer Physics Communications* **140** (2001) 418–431, [hep-ph/0012260].

[2] R. Mertig, M. Böhm, and A. Denner, *Feyn Calc - Computer-algebraic calculation of Feynman amplitudes*, *Computer Physics Communications* **64** (1991) 345–359.

[3] V. Shtabovenko, *FeynHelpers: Connecting FeynCalc to FIRE and Package-X*, *Computer Physics Communications* **218** (2017) 48–65, [arXiv:1611.06793].

[4] R. Mertig and R. Scharf, *TARCER - A mathematica program for the reduction of two-loop propagator integrals*, *Computer Physics Communications* **111** (1998) 265–273, [hep-ph/9801383].

[5] A. V. Smirnov, *FIRE5: A C++ implementation of Feynman Integral REduction*, *Computer Physics Communications* **189** (2015) 182–191, [arXiv:1408.2372].

[6] 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].

[7] T. Hahn and M. Pérez-Victoria, *Automated one-loop calculations in four and D dimensions*, *Computer Physics Communications* **118** (1999) 153–165, [hep-ph/9807565].

[8] F. Staub, *SARAH 3.2: Dirac Gauginos, UFO output, and more*, *Comput. Phys. Commun.* **184** (2013) 1792–1809, [arXiv:1207.0906].

[9] F. Staub, *SARAH 4: A tool for (not only SUSY) model builders*, *Computer Physics Communications* **185** (2014) 1773–1790, [arXiv:1309.7223].

[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, [hep-ph/0301101].

[11] P. Athron, J.-h. Park, D. Stöckinger, and A. Voigt, *FlexibleSUSY-A spectrum generator generator for supersymmetric models*, *Computer Physics Communications* **190** (2015) 139–172, [arXiv:1406.2319].

[12] J. McKay, P. Scott, and P. Athron, *Pitfalls of iterative pole mass calculation in electroweak multiplets*, *ArXiv e-prints* (2017) [arXiv:1710.01511].

[13] J. McKay and P. Scott, *Two-loop mass splittings in electroweak multiplets: winos and minimal dark matter*, *ArXiv e-prints* (2017) [arXiv:1712.00968].