Herwig++ Quick Guide
Draft version 0.16

David Grellscheid

August 24, 2005

Contents

I Introduction **3**

1 Installation **4**
1.1 CLHEP 4
1.2 ThePEG 5
1.3 Herwig++ 6

2 Example LEP runs **8**
2.1 The first run `[Herwig++ run]` 8
2.2 Generating `LEP.run` `[Herwig++ read]` 8
2.3 Adding analysis code `[Herwig++.cc]` 9
2.4 Generating `HerwigDefaults.rpo` `[Herwig++ init]` 9

II Special Topics **10**

3 Writing Matrix Elements **11**
3.1 Overview 11
3.2 Preparations 13
3.3 Specify the diagrams and selectors 13
3.4 Specify the kinematics 15
3.5 Calculate the matrix element 16

4 Writing Decayers **17**
4.1 Overview 17
4.2 Preparations 17
4.3 The decay 18
4.4 Configuration setup 20

5 Writing Analysis Handlers **21**
5.1 Overview 21
5.2 Preparations 21
5.3 The analysis code 22
5.4 Configuration setup 24

A Emacs setup **25**

B Example $2 \to 2$ matrix element code **26**

C Example $2 \to 4$ matrix element code **32**

Part I

Introduction

Chapter 1

Installation

Herwig++ uses two underlying libraries (CLHEP and ThePEG) extensively. If these are already installed on your system, skip sections 1.1 and 1.2.

Please read through the sections before typing any commands.

The installation steps were tested with `gcc` series 3.2.3, 3.3, 3.4 and 4.0 on i686 Linux machines, and the library versions mentioned below.

1.1 CLHEP

1.1.1 Download

Visit

```
http://wwwasd.web.cern.ch/wwwasd/lhc++/clhep/
INSTALLATION/clhep-1.9.html
```

and download

```
clhep-1.9.1.2.tgz
```

or any later version. Precompiled binaries are also available.

1.1.2 Installation

```
$ tar xzvf clhep-2.0.1.1.tgz
$ mkdir clhep-build
$ cd clhep-build
$ ../2.0.1.1/CLHEP/configure --prefix=(install-dir)
```

After unpacking the source code, this creates a *separate* build directory outside the source code tree. Now we can run the configure script: replace *(install-dir)* with the path to the directory where you want CLHEP to be installed. The default is `/usr/local`,

other possibilities could be `$HOME` or, if you want to keep everything CLHEP-related neatly in one place, `/opt/CLHEP` or `$HOME/opt/CLHEP`.

```
$ make
$ make check
```

If `make check` runs fine, you're ready to install. For this step, you will need write permission on *(install-dir)*:

```
$ make install
```

After the installation, the source and build directories are not required anymore. Delete them if you're short on space.

1.1.3 Environment variables

For the remaining installations to work, you will need to set the environment variable

`CLHEPPATH=`*(install-dir)*

Check that there is a link called `libCLHEP.so` inside `$CLHEPPATH/lib`. If it is not there, do

```
$ cd $CLHEPPATH/lib
$ ln -s libCLHEP-2.0.1.1.so libCLHEP.so
```

This is required for ThePEG. Instead, you can also define the environment variable

`CLHEPLIB=-lCLHEP-2.0.1.1`

This is especially useful if you do not have write access to the CLHEP directory.

1.2 ThePEG

1.2.1 Download

The website is

`http://shakespeare.thep.lu.se/ThePEG/`

From there, get

`ThePEG-050204.tgz`

or any later snapshot[1].

[1] 050204 won't work with the latest Herwig++ anymore, there has been an architectural change that has not yet been released as a ThePEG snapshot. More up-to date tarballs are available from D.Grellscheid on request.

1.2.2 Installation

```
$ tar xzvf ThePEG-050204.tgz
$ cd ThePEG-050204/ThePEG
$ ./configure --prefix=(install-dir-2)
```

The installation directory can be the same as before, but doesn't need to be.

```
$ make check
```

This will build and check ThePEG in one go. If there are no errors, run

```
$ make install
```

After the installation, the source directory is not required anymore. Delete it if you're short on space.

1.2.3 Environment variables

For the Herwig++ installation to work, you will need to set the environment variable

```
THEPEGPATH=(install-dir-2)
```

1.3 Herwig++

1.3.1 Download

Get the current CVS snapshot[2].

1.3.2 Installation

After unpacking the source code, do

```
$ cd Herwig++
```

Now, you can run

```
$ ./configure --prefix=(install-dir-3)
$ make
$ make check
```

Now you should be able to run

```
$ make install
```

After the installation, the source directory is not required anymore. Delete it if you're short on space.

[2] Alternative tarball available from D.Grellscheid on request, soon also from the Herwig++ website.

1.3.3 Environment variables

To get Herwig++ to run, you'll need to set

`HERWIGPATH=`*(install-dir-3)*

and you'll need to put `$HERWIGPATH/bin` somewhere in your `PATH`.

Now everything should be set up. The next chapter will guide you through a simple generator run, using the LEP setup that comes with the installation.

Chapter 2

Example LEP runs

All the files mentioned in this chapter can be found in `$HERWIGPATH/share/Herwig++`.

2.1 The first run `[Herwig++ run]`

Create a temporary directory to try out Herwig++, and copy all files ending in `.in` from the Herwig installation:

```
$ mkdir herwigtest
$ cd herwigtest
$ cp $HERWIGPATH/share/Herwig++/*.in .
```

Run these two commands to set up a generator file. The commands will be explained further on.

```
$ Herwig++ init
$ Herwig++ read LEP.in
```

The generated `LEP.run` contains a full example setup of a LEP event generator. To run it, try

```
$ Herwig++ run LEP.run -N50
```

The flag `-N` specifies the number of events to generate. After the run, there should be two new files: `LEP.out` which contains an overall cross section, and `LEP.log`, containing the detailed record of the requested 50 events. The log format is described in detail in **(???????)**.

Re-running the above command will give the *same* 50 events again unless the seed of the random number generator is changed by specifying the flag `-seed` *nnnnn*.

If you want to modify generator parameters, you'll need to modify `LEP.in` and regenerate `LEP.run`. The next section shows that in detail.

2.2 Generating `LEP.run [Herwig++ read]`

An event generator like `LEP.run` is created from a repository file which contains the basic physics setup, and a settings file where you can modify repository parameters. The

settings file also specifies which of the generators in the repository should be created. The example files are called `HerwigDefaults.rpo` and `LEP.in`.

```
$ Herwig++ read LEP.in
```

This step will create a new `LEP.run` from `LEP.in` and `HerwigDefaults.rpo`.

Try modifying the beam energy setting in `LEP.in` 20 to 30 GeV away from the Z mass, generate `LEP.run` and produce a few events. Some processes in the logfile should now be mediated by photons instead of Zs.

(More details about LEP.in? Where? Refer to ThePEG interfaces html page?)

2.3 Adding analysis code `[Herwig++.cc]`

The main program `Herwig++` that comes with the installation does not contain any analysis code. The recommended way to add analysis code is to write your own Analysis Handler. See chapter 5 for more detail.

Another option is to include your own code into `int main()` and re-compile your own version of `Herwig++`. Have a look at `Herwig++.cc` and `Herwig.Makefile` **(doesn't exist yet!)** as a guide. More about this and some analysis examples can be found in the slides for the Herwig++ tutorial on
`http://agenda.cern.ch/fullAgenda.php?ida=a031999`.

2.4 Generating `HerwigDefaults.rpo` `[Herwig++ init]`

The version of `HerwigDefaults.rpo` that comes with the installation should be sufficient for most purposes. Many of the settings can be modified inside `LEP.in` (see section 2.2).

You will need this section only if you want to create your own default repository or if you need to adjust a setting that is read-only.

The command to create a repository is

```
$ Herwig++ init
```

This will step through the file `HerwigDefaults.in`, reading in additional files as specified. All required files (extension `.in`) are provided in the `share/Herwig++` directory after installation.

Part II

Special Topics

Chapter 3

Writing Matrix Elements

You can add your own matrix elements to Herwig++. They are compiled as dynamically loadable objects, *i.e.* they do not need to be compiled into the Herwig++ library. Instead, you specify inside `LEP.in` or `HerwigDefaults.in` which object file contains your new matrix element. More about this later.

3.1 Overview

Any new matrix element class you implement should derive from `ThePEG::MEBase` which defines a number of virtual functions that you need to overload. Here, we will just present a short overview. More detailed descriptions follow in later sections.

- Herwig++ expects the function

  ```
  CrossSection MEexample::dSigHatDR()
  ```

  to return the value of the differential cross section

  $$\frac{\mathrm{d}^N\sigma}{\mathrm{d}r_1\,\mathrm{d}r_2\,\ldots\,\mathrm{d}r_N}\,. \tag{3.1}$$

  This is the main interaction point with the outside world for your ME.
  $r_1 \ldots r_N$ are N uniform random numbers in the range $(0, 1)$. Herwig++ provides these for you, but you need to set the N you require inside

  ```
  int MEexample::nDim()
  ```

  There may be a problem at the moment when asking for more than one random number. Workaround: generate the numbers you need in your own code.

- You need to transform $r_1 \ldots r_N$ to the variables you actually need (*e.g.* $\cos\theta$, ϕ) yourself. Also note that the transformation leads to a *Jacobian* factor $\mathcal{J}$ for the cross section:

  $$\frac{\mathrm{d}^N\sigma}{\mathrm{d}r_1\,\mathrm{d}r_2\,\ldots\,\mathrm{d}r_N} = \mathcal{J}\cdot\frac{\mathrm{d}^N\sigma}{\mathrm{d}\cos\theta\,\mathrm{d}\phi\,\mathrm{d}\ldots}\,, \tag{3.2}$$

where

$$\mathcal{J} = \frac{\partial \cos\theta}{\partial r_1} \times \frac{\partial \phi}{\partial r_2} \times \frac{\partial \ldots}{\partial r_{\ldots}}. \tag{3.3}$$

$\mathcal{J}$ can be set and read anywhere within `MEexample` through

```
void MEBase::jacobian(double)
double MEBase::jacobian()
```

You will come across this a few times later on in the chapter.

- The main program provides your matrix element class with the momenta of the incoming particles. You must calculate and assign the outgoing momenta in

  ```
  bool MEexample::generateKinematics(const double * r)
  ```

  The association of particle type with momentum is done through the Feynman diagrams that need to be specified in

  ```
  void MEexample::getDiagrams()
  ```

- Most of the actual matrix element calculation is then done in

  ```
  double MEexample::me2()
  ```

  A range of helper classes from `Herwig::Helicity` simplify this part significantly.

- Since there is usually more than one diagram contributing to the matrix element, the main program needs to know which diagram to put into the event record. It also needs to know how the colour lines flow through the various diagrams. This is done in

  ```
  Selector<MEBase::DiagramIndex>
  MEexample::diagrams(const DiagramVector & diags)
  ```

  and

  ```
  Selector<const ColourLines *>
  MEexample::colourGeometries(tcDiagPtr diag)
  ```

- The remaining step is to set

  ```
  unsigned int MEexample::orderInAlphaS()
  unsigned int MEexample::orderInAlphaEW()
  ```

  to the right numbers for the ME you're writing.

All these functions will be explained in more detail in the following sections.

3.2 Preparations

ThePEG comes with emacs extensions that will help you in creating your own matrix elements. See appendix A for help with the setup. Once the package is installed, start emacs and type `M-x ThePEG-ME-class-files`. This will prompt you for several inputs:

Class name: Conventionally, matrix element class names have the structure `Herwig::MExx2yy2zz`, where `xx` are the incoming particles, `yy` the intermediates and `zz` the outgoing particles that the matrix element deals with. Of course, you can choose any other name. Here, I always assume the class name to be `Herwig::MEexample`.

Base class name: `MEBase` is given as the default here, but for $2 \to 2$ processes, better use `ME2to2Base`. Use the source code documentation of ThePEG and Herwig++ to find out if there are other base classes that you can use.

Include file for base class: the file that contains the declaration for the base class you have chosen.

Persistent? Answer `y` here.

Concrete? Answer `y` here unless you're planning to write a new base class of your own that is not meant to be used directly.

Now, three emacs buffers will be created, named `MEexample.h`, `MEexample.icc` and `MEexample.cc`. They provide the framework that you will need to fill in.

$2 \to 2$ processes are simpler

If you are using the `ME2to2Base` class, it will take care of the event kinematics for you. You can comment out the following four functions in `MEexample.h` and `MEexample.cc`, since they are already implemented in `ME2to2Base`:

```
void setKinematics()
int nDim()
bool generateKinematics(const double * r)
CrossSection dSigHatDR()
```

and you can skip section 3.4. Either way, all further modifications happen only inside `MEexample.cc`.

3.3 Specify the diagrams and selectors

`void MEexample::getDiagrams()`

This is the place to set up all diagrams that contribute to the matrix element. First, create shortcuts to the particles you'll need:

```
tcPDPtr ep = getParticleData(ParticleID::eplus);
tcPDPtr em = getParticleData(ParticleID::eminus);
tcPDPtr Z0 = getParticleData(ParticleID::Z0);
...
```

The diagrams are specified as follows:

```
add(new_ptr((Tree2toNDiagram(2),
em,ep,1,gamma,3,Wminus,3,Wplus,-1)));

add(new_ptr((Tree2toNDiagram(3),
em,nu_e,ep,1,Wminus,2,Wplus,-2)));
```

The argument N to `Tree2toNDiagram(`N`)` corresponds to the number of lines connecting one incoming particle to the other (including the particles temselves). After that, the N particles need to be listed. Internally, they are numbered from 1 to N. The remaining line consists of alternations of a number and a single particle. For example `1,gamma` above means that particle 1 (the electron) is followed by a photon.
(need graph here)
The end of the diagram is indicated by a negative integer, which also acts as a label for this diagram.

`Selector<MEBase::DiagramIndex>` `MEexample::diagrams(const DiagramVector & diags)`

This selector determines which diagram is chosen for display in the event record. It does this by weighting each diagram according to

$$\frac{ME_i^2}{\sum_i ME_i^2}, \tag{3.4}$$

where ME_i^2 is the matrix element squared of diagram i alone. Note that $\sum ME_i^2$ is the sum of the individual squared matrix elements, and *not* the full matrix element. The example code in the appendix shows how to set this up.

`Selector<const ColourLines *>` `MEexample::colourGeometries(tcDiagPtr diag)`

If you have any coloured particles in your diagrams, you need to specify the colour flow here. As an example, take the following diagram:

```
Tree2toNDiagram(2),em,ep,1,gamma,3,Wminus,3,Wplus,
4,d,4,ubar,5,sbar,5,c,-1
```

Particles number 6 and 7 are colour connected, as well as particles 8 and 9. You specify this in a `ColourLines` object as

```
static ColourLines WW2qqqq("6 -7,-8 9");
```

The negative numbers stand for anticolours. Now you need to insert a pointer to the colour lines into the selector and return it:

```
Selector<const ColourLines *> sel;
sel.insert(1.0, &WW2qqqq);
return sel;
```

3.4 Specify the kinematics

You can skip this section if you're writing a $2 \to 2$ matrix element and are using `ME2to2Base` as base class.

`bool MEexample::generateKinematics(const double * r)`

This function lets you specify the momenta for the outgoing particles. You have complete freedom here, as long as you set the outgoing momenta in the end. Here is what it looks like for four outgoing particles:

```
Lorentz5Momentum out2, out3, out4, out5;

(... use meMomenta()[0] and [1] to do your calculations here ...)

meMomenta()[2] = out2;
meMomenta()[3] = out3;
meMomenta()[4] = out4;
meMomenta()[5] = out5;
```

As you can see, the generator holds the momenta for the incoming and outgoing particles in an array. The particle numbering here is *different* from the numbering in the previous section! The incoming particles have index 0 and 1, the outgoing particles are numbered from 2 onwards.

For your calculations you can use the random numbers provided through `r`, or generate random numbers yourself. The `Herwig::Kinematics` class contains a number of useful helper functions. You will need to pay attention to set the right jacobian factors for your cross-section. The $2 \to 4$ code in the appendix contains a complete example.

During a run, this one of the two functions that will be called from the outside, just before `sSigHatDR()`.

`int MEexample::nDim()`

Here you need to specify how many generator-provided random numbers you need in `generateKinematics()`. Make sure you get this right, to prevent an out-of-bounds access via the pointer `r`.

`void MEexample::setKinematics()`

The definition of this function in the base class always seems to be sufficient, you won't need to do anything else here.

3.5 Calculate the matrix element

`double MEexample::me2()`

For the actual calculation of the matrix elements done here, you can get a lot of help from the classes in the `Herwig::Helicity` namespace. First, set up the wavefunctions for the outer particles (we're using $e^+e^- \to W^+W^- \to f\bar{f}\bar{f}f$ as an example):

```
SpinorWaveFunction    em_in( meMomenta()[0], mePartonData()[0], incoming);
SpinorBarWaveFunction ep_in( meMomenta()[1], mePartonData()[1], incoming);
// ...
SpinorBarWaveFunction af4_out(meMomenta()[4], mePartonData()[4], outgoing);
SpinorWaveFunction    ff5_out(meMomenta()[5], mePartonData()[5], outgoing);
```

Then, specify the wave functions for the intermediate particles:

```
VectorWaveFunction interP, interZ, interWp, interWm;
```

Using these wave functions, the actual calculation of a matrix element is as easy as the following few lines. The first line is unfortunately necessary, to get the ThePEG Standard Model pointer to the Herwig++Standard Model object:

```
static const TransientConstRCPtr<Herwig::StandardModel> hwsm
= dynamic_ptr_cast<TransientConstRCPtr<Herwig::StandardModel> >
  (standardModel());

interZ.at(index) = FFZVertex->evaluate(sHat(), 1, Z0, em_in, ep_in);
interP.at(index) = FFZVertex->evaluate(sHat(), 1, Z0, em_in, ep_in);
interWm.at(index) = FFWVertex->evaluate(sHat(), 1, Wminus, ff2_out, af3_out);
interWp.at(index) = FFWVertex->evaluate(sHat(), 1, Wplus, ff5_out, af4_out);

diag[0] = WWWVertex->evaluate(sHat(), interP[k], interWp[i], interWm[j]);
diag[1] = WWWVertex->evaluate(sHat(), interZ[k], interWp[i], interWm[j]);
```

Some details have been omitted here, like summing over the spins, *etc*. Again, you'll find the full example in the appendix.

`CrossSection MEexample::dSigHatDR()`

This is the final place to specify some extra prefactors and return the value of `me2()`. During a run this one of the two functions that will be called from the outside, just after `generateKinematics()`.

Chapter 4

Writing Decayers

Customized decayer modules can be written in a similar fashion to the matrix elements discussed above. They are compiled as dynamically loadable objects that do not need to be compiled into the Herwig++ library. Instead, you specify inside `Decays.in` which object file contains your new decayer. More about this later.

4.1 Overview

Any new decayer class you implement should derive from `ThePEG::Decayer` which defines two virtual functions that you need to overload. Here, we will just present a short overview. More detailed descriptions follow in later sections.

- The function

  ```
  bool ExampleDecayer::accept(const DecayMode & dm)
  ```

  can be used to restrict the use of the new decayer module to specific groups of decays. It should return `true` for all decays it can handle, and `false` otherwise.

- The actual decay is calculated inside

  ```
  ParticleVector ExampleDecayer::decay(const DecayMode & dm, const
  Particle & parent).
  ```

  Here, `dm` contains a description of one specific decay mode, and `parent` is the object that contains the momentum information for the decaying particle.

4.2 Preparations

ThePEG comes with emacs extensions that will help you in creating your own decayers. See appendix A for help with the setup. Once the package is installed, start emacs and type `M-x ThePEG-Decayer-class-files`. This will prompt you for several inputs:

Class name: Conventionally, decayer class names have the structure `Herwig::xyzDecayer`, where `xyz` is a summary of the kind of decays that are handled. Here, I always assume the class name to be `Herwig::ExampleDecayer`.

Base class name: `Decayer` is given as the default here. Use the source code documentation of ThePEG and Herwig++ to find out if there are other base classes that you can use.

Include file for base class: the file that contains the declaration for the base class you have chosen.

Persistent? Answer `y` here.

Concrete? Answer `y` here unless you're planning to write a new base class of your own that is not meant to be used directly.

Now, three emacs buffers will be created, named `ExampleDecayer.h`, `ExampleDecayer.icc` and `ExampleDecayer.cc`. They provide the framework that you will need to fill in. All further modifications happen only inside `ExampleDecayer.cc`.

4.3 The decay

`bool ExampleDecayer::accept(const DecayMode & dm)`

You must use the `accept()` member function to make sure that your new decayer is only used with the decays that it can handle. There are several member functions of the `DecayMode` object `dm` that you can use. The most useful ones are

- `DecayMode::parent()`, which returns a `ParticleData` pointer to the decaying particle,
- `DecayMode::products()`, which returns a multiset of `ParticleData` pointers to the decay products, and
- `DecayMode::orderedProducts()`, which returns a vector of `ParticleData` pointers to the decay products.

You can then use these pointers to check against the modes you want to allow. For more information, see `ThePEG`'s online documentation for the `ParticleData` and `DecayMode` classes.

The accept function can be as simple as

```
return true;
```

for a general decayer. Here's an actual example for a more complicated accept function found in Herwig++:

```
bool EtaPiGammaGammaDecayer::accept(const DecayMode & dm) const
{
  bool allowed = false;

  // check that there are three outgoing particles
  if (dm.products().size() != 3)
    return false;

  // check the incoming particle is an eta or eta prime
  int id = dm.parent()->id();
  if (id == ParticleID::eta || id == ParticleID::etaprime)
    {
      ParticleMSet::const_iterator pit = dm.products().begin();

      // count the outgoing particles by type
      unsigned int npi0 = 0, ngamma = 0;
      for (; pit!=dm.products().end(); ++pit)
        {
          id = (**pit).id();
          if (id == ParticleID::pi0)
            ++npi0;
          else if (id == ParticleID::gamma)
            ++ngamma;
        }

      // accept if products contain one pi0 and two gammas
      if (npi0 == 1 && ngamma == 2)
        allowed = true;
    }
  return allowed;
}
```

ParticleVector ExampleDecayer::decay(const DecayMode & dm, const Particle & parent)

The actual momentum assignment for the decay products happens inside this function. You can use the momentum information that is passed to you with the `parent` object to create product particles in any way that Herwig++ will allow. The output needs to be a vector of `Particle` objects, one for each decay product. If it is impossible to generate the decay, you should return an empty vector. A simple general two-body decayer is shown below. For more examples, look at the decayers that already exist in Herwig++.

```
ParticleVector ExampleDecayer::decay(const DecayMode & dm,
                                     const Particle & parent) const
{
  // we only do two-body decays here
  // this should have been verified in accept() already!
  if (dm.products().size() != 2)
    return ParticleVector();
```

```
  // set up children vector automatically from decay mode
  ParticleVector children = dm.produceProducts();

  // generate two-body decay using Herwig++ Kinematics class
  double cth, phi;
  Kinematics::generateAngles(cth,phi);
  Vector3 unitdir = Kinematics::unitDirection(cth,phi);

  Lorentz5Momentum out0, out1;

  Kinematics::twoBodyDecay(parent->momentum(),
                           children[0]->nominalMass(),
                           children[1]->nominalMass(),
                           unitdir,
                           out0,
                           out1);

  children[0]->set5Momentum(out0);
  children[1]->set5Momentum(out1);

  return children;
}
```

4.4 Configuration setup

First, your new decayer needs to be set up for use in `Decays.in` by including a line such as

```
create Herwig::ExampleDecayer /Defaults/Decays/ExampleDecay01 ExampleDecayer.so
```

You can freely choose the label, here it is `ExampleDecayer01`. This label needs to be specified inside `Hw64Decays.in` after all decay modes that should use the new decayer:

```
[...]
decaymode pi0->gamma,gamma; 0.988 1 /Defaults/Decays/ExampleDecay01
decaymode eta->gamma,gamma; 0.388 1 /Defaults/Decays/ExampleDecay01
[...]
```

Alternatively, you can label your new decayer with a label that is already in use inside `Hw64Decays.in`.

Running the usual steps of `Herwig++ init, read, run` should now produce an event record using your new decay.

Chapter 5

Writing Analysis Handlers

The easiest way to add customized analysis code is to write your own Analysis Handler module. Again, the procedure is very similar to writing matrix elements or decayers. The modules are compiled as dynamically loadable objects that do not need to be compiled into the Herwig++ library. Instead, you specify inside `LEP.in` or `LHC.in` which object file contains your new analysis code. More about this later.

5.1 Overview

Any new analysis class you implement should derive from `ThePEG::AnalysisHandler` which defines three virtual functions at various levels of generality that you can overload. Here, we will just present a short overview. More detailed descriptions follow in later sections.

- First, there is

  ```
  void ExampleAnalysis::analyze(tEventPtr event, long ieve, int loop,
  int state)
  ```

  , where you can include some event-level analysis, or prepare for further analysis steps. By default, this function will hand over the list of final state particles to

- ```
  void ExampleAnalysis::analyze(const tPVector & particles)
  ```

  . The default here is to loop over the list of particles, and for each one of them call

- ```
  void ExampleAnalysis::analyze(tPPtr)
  ```

  , where you can include particle-specific code.

5.2 Preparations

ThePEG comes with emacs extensions that will help you in creating your own analysis handlers. See appendix A for help with the setup. Once the package is installed, start

emacs and type `M-x ThePEG-AnalysisHandler-class-files`. This will prompt you for several inputs:

Class name: Conventionally, analysis handler names have the structure `Herwig::xyzAnalysis`, where `xyz` can be any unique description you choose. Here, I always assume the class name to be `Herwig::ExampleAnalysis`.

Base class name: `AnalysisHandler` is the default here.

Include file for base class: the file that contains the declaration for the base class you have chosen.

Persistent? Answer `y` here.

Concrete? Answer `y` here unless you're planning to write a new base class of your own that is not meant to be used directly.

Now, three emacs buffers will be created, named `ExampleAnalysis.h`, `ExampleAnalysis.icc` and `ExampleAnalysis.cc`. They provide the framework that you will need to fill in. All further modifications happen only inside `ExampleAnalysis.cc`.

5.3 The analysis code

`void ExampleAnalysis::analyze(tEventPtr event, long ieve, int loop, int state)`

The important variable here is `event`, the pointer to the current event data, through which you can get access to all steps of the event record, from the initial hard process to the final states. The other flags are for bookkeeping and event consistency. Look at the source documentation of `AnalysisHandler` for more information. If you're simply interested in the final state particles, there's nothing you need to do in here, the default call to the parent function is fine. As an example, here's a bit of code to extract all neutral pions:

```
void Pi4FermionAnalysis::analyze(tEventPtr event,
                                 long ieve, int loop, int state)
{
  // consistency check. Should be present unless you know what you're doing.
  if ( loop > 0 || state != 0 || !event ) return;

  // find all neutral pions
  tPVector particles;
  for(unsigned int ix=0, nstep=event->primaryCollision()->steps().size();
  ix<nstep; ++ix) {
    ThePEG::ParticleSet part=event->primaryCollision()->step(ix)->all();
    ThePEG::ParticleSet::const_iterator iter=part.begin();
    ThePEG::ParticleSet::const_iterator end=part.end();
    for( ;iter!=end;++iter)
```

```
      if((**iter).id()==ParticleID::pi0)
        particles.push_back(*iter);
  }
  // analyse them
  analyze(particles);
}
```

void ExampleAnalysis::analyze(const tPVector & particles)

Here you can work on the list of particles that were handed over by the previous step. Let's do something simple here:

```
void Pi4FermionAnalysis::analyze(const tPVector & particles) {
  cout << particles.size() << " neutral pions found.\n";
  for ( int i = 0, N = particles.size(); i < N; ++i ) analyze(particles[i]);
}
```

Of course, for a real analysis, it doesn't make much sense to print out the count for each of the possibly thousands of events. You'll most probably want an analysis that takes into account all events in your run together. You can achieve this by setting up new member variables in your analysis handler class, that keep track of the global state of things. Initialize them in your default constructor, modify them during the run in one of the analyze functions, and add a printout statement for them to `ExampleAnalysis::dofinish()`. The next function example will show such an access to a user-written data member.

void ExampleAnalysis::analyze(tPPtr)

Here you can include analysis at the level of single particles. In our example, we will store the momenta of decay products from all $\pi_0 \to ffff$ decays. Look for the histogram line near the end. In a similar fashion, you can include any other helper classes or member variables.

```
void Pi4FermionAnalysis::analyze(tPPtr part)
{
  // check the number of childrem of the pi0
  if (part->children().size() != 4)
    return;
  // storage for the our momenta
  Lorentz5Momentum pe[2],pp[2];
  // count of electrons / positrons
  unsigned int ne(0),np(0);
  int id;
  // find the particles
  unsigned int ix,iy;
  for (ix=0; ix<part->children().size(); ++ix) {
    id = part->children()[ix]->id();
    if (id==ParticleID::eplus) {
      pe[ne]=part->children()[ix]->momentum();
```

```
      ++ne;
    } else if (id==ParticleID::eminus) {
      pp[np]=part->children()[ix]->momentum();
      ++np;
    }
  }
  // continue only if we have 2 e- and 2 e+
  if(ne!=2 || np!=2)
    return;
  // find the invariant masses
  Lorentz5Momentum ptemp;
  for(ix=0;ix<2;++ix) {
    for(iy=0;iy<2;++iy) {
      ptemp=pe[ix]+pp[iy];
      ptemp.rescaleMass();
      // Here we add the information to the histogram member object.
      // The final handling of the histogram is then done in
      // ExampleAnalysis::dofinish()
      histogram_.addPoint(ptemp.mass(),0.25);
    }
  }
}
```

5.4 Configuration setup

Your new analysis module needs to be set up for use in one of the config files, such as `LHC.in` or `LEP.in` by including lines such as

```
mkdir /Defaults/Handlers/Analysis
create Herwig::ExampleAnalysis /Defaults/Handlers/Analysis/Example
insert LEPGenerator:AnalysisHandlers 0 /Defaults/Handlers/Analysis/Example
```

Running the usual steps of `Herwig++ init, read, run` should now produce a run including your analysis.

Appendix A

Emacs setup

After installation of ThePEG (section 1.2), you can copy the provided emacs config file into your emacs directory:

```
$ cd $HOME
$ mkdir -p .emacs.d
$ cp $THEPEGPATH/lib/ThePEG/current/ThePEG.el .emacs.d/
```

Now add the line

```
(load-file "$HOME/.emacs.d/ThePEG.el")
```

to your `.emacs` file. To see if it worked, start emacs and try the command

```
M-x ThePEG-class-files
```

If you see a prompt asking you for a class name, everything should be set up.

Appendix B

Example $2 \to 2$ matrix element code

This example code will be available on the Herwig++ web page soon.

```
// -*- C++ -*-
//
// This is the implementation of the non-inlined, non-templated member
// functions of the MEee2WW class.
//
#include "MEee2WW.h"
#include "ThePEG/Interface/ClassDocumentation.h"
#include "ThePEG/Persistency/PersistentOStream.h"
#include "ThePEG/Persistency/PersistentIStream.h"
#include "ThePEG/Repository/EventGenerator.h"
#include "ThePEG/Handlers/StandardXComb.h"
#include "Herwig++/Models/StandardModel/StandardModel.h"
#include "Herwig++/Helicity/WaveFunction/SpinorWaveFunction.h"
#include "Herwig++/Helicity/WaveFunction/SpinorBarWaveFunction.h"
#include "Herwig++/Helicity/WaveFunction/VectorWaveFunction.h"
#include "ThePEG/PDT/EnumParticles.h"
#include "ThePEG/MatrixElement/Tree2toNDiagram.h"
#include <vector>
#include <numeric>
#include <complex>
using namespace Herwig;
using namespace Herwig::Helicity;

MEee2WW::~MEee2WW() {}

// Set up the diagrams that contribute to the decay.
// (called once during 'Herwig++ read')
void MEee2WW::getDiagrams() const
{
  // define pointers to the particle data objects that are required
  tcPDPtr em = getParticleData(ParticleID::eminus);
  tcPDPtr ep = getParticleData(ParticleID::eplus);
  tcPDPtr nu_e = getParticleData(ParticleID::nu_e);
  tcPDPtr gamma = getParticleData(ParticleID::gamma);
```

```
  tcPDPtr Z0 = getParticleData(ParticleID::Z0);
  tcPDPtr Wminus = getParticleData(ParticleID::Wminus);
  tcPDPtr Wplus = getParticleData(ParticleID::Wplus);

  // s-channel Z0 for W+W- production
  // (2) denotes two incoming (spacelike) lines
  // em, ep : name those two lines
  // 1, Z0 : at the end of line 1 (em), attach Z0
  // 3, Wminus : at the end of line 3 (Z0), attach Wminus
  // 3, Wplus : at the end of line 3, attach Wplus
  // -1 : label for this diagram (must be negative)
  add(new_ptr((Tree2toNDiagram(2), em, ep, 1,    Z0, 3, Wminus, 3, Wplus, -1)));

  // s-channel photon for W+W- production
  add(new_ptr((Tree2toNDiagram(2), em, ep, 1, gamma, 3, Wminus, 3, Wplus, -2)));

  // t channel for W+W- production
  // (3) denotes three spacelike lines to connect the incoming particles
  // em, nu_e, ep : name the three spacelike lines
  // 1, Wminus : at the end of line 1 (em), attach Wminus
  // 2, Wplus : at the end of line 2 (nu_e), attach Wplus
  // -3 : label for this diagram (must be negative)
  add(new_ptr((Tree2toNDiagram(3), em, nu_e, ep, 1, Wminus, 2, Wplus, -3)));

  // t- and u-channel for ZZ production
  add(new_ptr((Tree2toNDiagram(3), em, em, ep, 1, Z0, 2, Z0, -4)));
  add(new_ptr((Tree2toNDiagram(3), em, em, ep, 2, Z0, 1, Z0, -5)));
}

Energy2 MEee2WW::scale() const
{
  return sHat();
}

// calculation of the matrix element squared
// this function is called repeatedly for every iteration in the Monte Carlo
double MEee2WW::me2() const
{
  // the ordering of particles here is always the one given by the
  // outside lines in the Tree2toNDiagram() above.
  // watch out for the index count starting at 0 now!
  // i.e. meMomenta()[0] are the momenta of particle 1 (here: em)

  // setup momenta and particle data for the external wavefunctions
  SpinorWaveFunction     em_in( meMomenta()[0],mePartonData()[0],incoming);
  SpinorBarWaveFunction ep_in( meMomenta()[1],mePartonData()[1],incoming);
  VectorWaveFunction   V1_out( meMomenta()[2],mePartonData()[2],outgoing);
  VectorWaveFunction   V2_out( meMomenta()[3],mePartonData()[3],outgoing);

  // get a pointer to the standard model object in the run
  static const TransientConstRCPtr<Herwig::StandardModel> hwsm
```

```
  = dynamic_ptr_cast<TransientConstRCPtr<Herwig::StandardModel> >
  (standardModel());
if (!hwsm) {
  throw Exception() << "hwsm pointer is null in me2() of MEee2WW"
                    << Exception::abortnow;
}

// get pointers to all required Vertex objects
static const TransientRCPtr<Herwig::Helicity::FFVVertex>
  FFZVertex = hwsm->vertexFFZ();
static const TransientRCPtr<Herwig::Helicity::FFVVertex>
  FFPVertex = hwsm->vertexFFP();
static const TransientRCPtr<Herwig::Helicity::VVVVertex>
  WWWVertex = hwsm->vertexWWW();
static const TransientRCPtr<Herwig::Helicity::FFVVertex>
  FFWVertex = hwsm->vertexFFW();

// get pointers to all required particle data objects
static const tcPDPtr gamma = getParticleData(ParticleID::gamma);
static const tcPDPtr Z0 = getParticleData(ParticleID::Z0);
static const tcPDPtr nu_e = getParticleData(ParticleID::nu_e);
static const tcPDPtr em = getParticleData(ParticleID::eminus);


// ===== sum over helicities to get the matrix element ====== //
// this should match the number of diagrams set up above
static const size_t NUM_OF_DIAGS = 5;

// internal wavefunctions
VectorWaveFunction interP, interZ;
SpinorWaveFunction interF;

// storage for the matrix element entries
vector<double> me(NUM_OF_DIAGS,0.0);
vector<complex<double> > diag(NUM_OF_DIAGS,0.0);

// double counting factor (needed in ZZ production)
double prefactor = (V1_out.id() == ParticleID::Z0)? 0.5 : 1.0 ;

// sum variables
double full_me = 0.0;
Complex mesum = 0.0;

// loop over spinor helicity states
for(unsigned int inhel1 = 0; inhel1 < 2; ++inhel1) {
  em_in.reset(inhel1);

  // loop over spinor helicity states
  for(unsigned int inhel2 = 0; inhel2 < 2; ++inhel2) {
    ep_in.reset(inhel2);
```

```
if(V1_out.id() == ParticleID::Wminus) {
  // evaluate first vertex to get wavefunction of intermediate
  interZ = FFZVertex->evaluate(sHat(),1,Z0,em_in,ep_in);
  interP = FFPVertex->evaluate(sHat(),1,gamma,em_in,ep_in);
}

// loop over vector helicity states
for(unsigned int outhel1 = 0; outhel1 < 3; ++outhel1) {
  V1_out.reset(outhel1);

  // loop over vector helicity states
  for(unsigned int outhel2 = 0; outhel2 < 3; ++outhel2) {
    V2_out.reset(outhel2);

    if (V1_out.id() == ParticleID::Wminus
        && V2_out.id() == ParticleID::Wplus) {

      // first calculate the Z exchange diagram
      diag[0] = WWWVertex->evaluate(sHat(),V1_out,V2_out,interZ);
      // then the photon exchange diagram
      diag[1] = WWWVertex->evaluate(sHat(),V1_out,V2_out,interP);
      // and the t channel nu exchange
      interF = FFWVertex->evaluate(sHat(),1,nu_e,em_in,V1_out);
      diag[2] = FFWVertex->evaluate(sHat(),interF,ep_in,V2_out);

    } else {
      diag[0] = diag[1] = diag[2] = 0.0;
    }

    if (V1_out.id() == ParticleID::Z0
        && V2_out.id() == ParticleID::Z0) {

      // evaluate first vertex to get wavefunction of intermediate
      interF = FFZVertex->evaluate(sHat(),1,em,em_in,V1_out);
      // then the t-channel ZZ production
      diag[3] = FFZVertex->evaluate(sHat(),interF,ep_in,V2_out);
      // the same for the u-channel ZZ production
      interF = FFZVertex->evaluate(sHat(),1,em,em_in,V2_out);
      diag[4] = FFZVertex->evaluate(sHat(),interF,ep_in,V1_out);

    } else {
      diag[3] = diag[4] = 0.0;
    }

    // add square of term i to me[i]
    for (size_t i=0; i<NUM_OF_DIAGS; ++i)
      me[i] += prefactor * std::norm(diag[i]);

    // add (me + me + ...)^2 to full_me
    mesum = std::accumulate(diag.begin(),diag.end(),
                            complex<double>(0.0));
```

```
            full_me += prefactor * std::norm(mesum);
          }
        }
      }
    }

    // save individual diagram matrix elements in XComb()
    // and return total ME  (0.25 for spin average)
    DVector save(NUM_OF_DIAGS);
    for (size_t i = 0; i < NUM_OF_DIAGS; ++i) {
      save[i] = 0.25 * me[i];
    }
    meInfo(save);
    return 0.25 * full_me;
}

unsigned int MEee2WW::orderInAlphaS() const
{
    return 0;
}

unsigned int MEee2WW::orderInAlphaEW() const
{
    return 2;
}

// selection procedure to decide which diagram gets
// shown in event record
Selector<ME2to2Base::DiagramIndex>
MEee2WW::diagrams(const DiagramVector & diags) const
{
    vector<double> single_MEs;
    if ( lastXCombPtr() ) {
      single_MEs.assign(meInfo().begin(),meInfo().end());
    } else {
      cerr << "Last XCombPtr was not there\n";
    }

    Selector<DiagramIndex> sel;
    for ( DiagramIndex i = 0; i < diags.size(); ++i ) {
      // this works only if the diagrams were specified
      // in the same order as the me2() calculations
      //
      // diag -1 is me[0], -2 is 1, -3 is 2, etc.
      int meID = abs(diags[i]->id()) - 1;

      // bounds are checked here, use single_MEs[meID] if that's a problem
      sel.insert(single_MEs.at(meID), i);
    }
    return sel;
}
```

```
// colour lines need to be conected correctly
// according to the chosen diagram
Selector<const ColourLines *>
MEee2WW::colourGeometries(tcDiagPtr diag) const
{
  static ColourLines emptyCL("");
  Selector<const ColourLines *> sel;
  sel.insert(1.0, &emptyCL);
  return sel;
}

// if you define your own member variables, make sure to read/write them here
void MEee2WW::persistentOutput(PersistentOStream & os) const {}
void MEee2WW::persistentInput(PersistentIStream & is, int i) {}

// Definition of the static class description member.
ClassDescription<MEee2WW> MEee2WW::initMEee2WW;

void MEee2WW::Init()
{
  // give a meaningful description here, for the command line interface
  static ClassDocumentation<MEee2WW> documentation
    ("The \\classname{MEee2WW} class is a test ME,"
     "implementing the matrix element for"
     "e+e- to W+W- via Z and photon exchange"
     "and e+e- to ZZ"
     "using helicity amplitude"
     "techniques");
}
```

Appendix C

Example $2 \to 4$ matrix element code

This example code will be available on the Herwig++ web page soon.

```
// -*- C++ -*-
//
// This is the implementation of the non-inlined, non-templated member
// functions of the MEexample class.
//
#include "MEexample.h"
#include "ThePEG/Interface/ClassDocumentation.h"
#include "ThePEG/Persistency/PersistentOStream.h"
#include "ThePEG/Persistency/PersistentIStream.h"
#include "ThePEG/Repository/EventGenerator.h"
#include "ThePEG/Handlers/StandardXComb.h"
#include "ThePEG/Handlers/KinematicalCuts.h"
#include "ThePEG/Utilities/SimplePhaseSpace.h"
#include "Herwig++/Utilities/Kinematics.h"
#include "Herwig++/Models/StandardModel/StandardModel.h"
#include "Herwig++/Helicity/WaveFunction/SpinorWaveFunction.h"
#include "Herwig++/Helicity/WaveFunction/SpinorBarWaveFunction.h"
#include "Herwig++/Helicity/WaveFunction/VectorWaveFunction.h"
#include "ThePEG/PDT/EnumParticles.h"
#include "ThePEG/MatrixElement/Tree2toNDiagram.h"
using namespace Herwig;
using namespace Herwig::Helicity;

// define random number generator (instead of ACDC)
ThePEG::StandardRandom MEexample::rnGenerator = ThePEG::StandardRandom();

MEexample::~MEexample() {}

// Set up the diagrams that contribute to the decay.
// (called once during 'Herwig++ read')
void MEexample::getDiagrams() const
{
  // define pointers to the particle data objects that are required
  tcPDPtr Z0 = getParticleData(ParticleID::Z0);
```

```
  tcPDPtr gamma = getParticleData(ParticleID::gamma);
  tcPDPtr Wplus = getParticleData(ParticleID::Wplus);
  tcPDPtr Wminus = getParticleData(ParticleID::Wminus);
  tcPDPtr ep = getParticleData(ParticleID::eplus);
  tcPDPtr em = getParticleData(ParticleID::eminus);
  tcPDPtr nu_e    = getParticleData(ParticleID::nu_e    );

  // store all possible decays of W- boson
  typedef
    std::vector<pair<ParticleID::ParticleCodes, ParticleID::ParticleCodes> >
    Pairvector;
  Pairvector decaypair;
  decaypair.reserve(9);
  decaypair.push_back(make_pair(ParticleID::d, ParticleID::ubar));
  decaypair.push_back(make_pair(ParticleID::d, ParticleID::cbar));
  decaypair.push_back(make_pair(ParticleID::s, ParticleID::ubar));
  decaypair.push_back(make_pair(ParticleID::s, ParticleID::cbar));
  decaypair.push_back(make_pair(ParticleID::b, ParticleID::ubar));
  decaypair.push_back(make_pair(ParticleID::b, ParticleID::cbar));
  decaypair.push_back(make_pair(ParticleID::eminus,   ParticleID::nu_ebar));
  decaypair.push_back(make_pair(ParticleID::muminus,  ParticleID::nu_mubar));
  decaypair.push_back(make_pair(ParticleID::tauminus, ParticleID::nu_taubar));

  // loop over all possible decay combinations for W- and W+
  // first, the W- decay products
  Pairvector::const_iterator mp = decaypair.begin();
  for (; mp != decaypair.end(); ++mp) {
    tcPDPtr f_minus = getParticleData(mp->first);
    tcPDPtr nu_bar  = getParticleData(mp->second);

    // then the W+ decay products
    Pairvector::const_iterator pp = decaypair.begin();
    for (; pp!= decaypair.end(); ++pp) {
      tcPDPtr f_plus = getParticleData(-(pp->first));
      tcPDPtr nu     = getParticleData(-(pp->second));

      // the s-channel photon exchange
      // (see the other example file for details)
      add(new_ptr((Tree2toNDiagram(2),em,ep,1,gamma,3,Wminus,3,Wplus,
                   4,f_minus,4,nu_bar,5,f_plus,5,nu,-1)));

      // the s-channel Z0 exchange
      add(new_ptr((Tree2toNDiagram(2),em,ep,1,Z0    ,3,Wminus,3,Wplus,
                   4,f_minus,4,nu_bar,5,f_plus,5,nu,-2)));

      // the t-channel neutrino exchange
      add(new_ptr((Tree2toNDiagram(3),em,nu_e,ep,1,Wminus,2,Wplus,
                   4,f_minus,4,nu_bar,5,f_plus,5,nu,-3)));
    }
  }
}
```

```
Energy2 MEexample::scale() const
{
  return sHat();
}

// How many random numbers do we want from ACDC?
// Here, we generate all random numbers ourselves
int MEexample::nDim() const
{
  return 0;
}


void MEexample::setKinematics()
{
  MEBase::setKinematics();
}

// user-defined helper function to determine a random angle
// with a strong forward peak
double MEexample::getCosTheta(Energy m1, Energy m2)
{
  // from Fortran-Herwig; gives strong forward peak
  const Energy2 m1_2 = sqr(m1);
  const Energy2 m2_2 = sqr(m2);

  const double D1 = sHat() - m1_2 - m2_2;
  const double lambda = sqr(D1) - 4 * m1_2 * m2_2;
  if (lambda < 0.0)
    throw ImpossibleKinematics();
  const double D =  D1 / sqrt(lambda);

  const double ctmin = lastCuts().cTHMin();
  const double ctmax = lastCuts().cTHMax();

  const double fraction = (D - ctmax) / (D - ctmin);
  const double r = rnGenerator();
  const double costh = D - (D - ctmin) * std::pow(fraction, r);

  jacobian(jacobian() * (costh - D) * std::log(fraction));
  return costh;
}


// user-defined helper function to generate a mass
// from the Breit-Wigner distribution of a particle,
// restricted to the given limits
double MEexample::getMass(tcPDPtr particle, double min, double max)
{
  if (max < min)
```

```
    throw ImpossibleKinematics();

  const double M     = particle->mass();
  const double Gamma = particle->width();

  // get limits on rho
  const double rhomin = atan2(sqr(min)-sqr(M), M*Gamma);
  const double rhomax = atan2(sqr(max)-sqr(M), M*Gamma);

  const double rho = (rhomax - rhomin) * rnGenerator() + rhomin;

  const double mass2 = M * Gamma * tan(rho) + sqr(M);

  jacobian(jacobian()
           *(sqr(mass2 - sqr(M)) + sqr(M)*sqr(Gamma))
           / (M * Gamma)
           * (rhomax-rhomin)
           );

  return sqrt(mass2);
}

// Here you can use nDim() random numbers in the vector provided
// to generate the internal kinematics. Note that sHat() has
// already been given from the outside.
// nDim() *must* be set correctly, otherwise we have out of bounds access...
//
// we are using our own random number generator internally,
// 'r' will not be accessed
bool MEexample::generateKinematics(const double * r)
{
  try {
    // initialize the jacobian factor
    jacobian(1.0);

    tcPDPtr Wplus  = getParticleData(ParticleID::Wplus);
    tcPDPtr Wminus = getParticleData(ParticleID::Wminus);

    double m_23(0.0), m_45(0.0);

    const double mMin_p = Wplus->massMin();
    const double mMin_m = Wminus->massMin();

    if (sqrt(sHat()) < mMin_p + mMin_m) {
      // not enough phase space
      throw ImpossibleKinematics();
    }

    // get the off-shell masses for the intermediate Ws
    // these mass assignments should really be symmetrized
    const double mMax_p = min( Wplus->massMax(), sqrt(sHat()) - mMin_m );
```

```
m_45 = getMass(Wplus, mMin_p, mMax_p);

const double mMax_m = min( Wminus->massMax(), sqrt(sHat()) - m_45 );
m_23 = getMass(Wminus, mMin_m, mMax_m);


double cth, phi;
// to set phi conveniently, we use
Kinematics::generateAngles(cth,phi);
// cth is reset immediately
cth = getCosTheta(m_23, m_45);

// ===== first, produce the W+ W- pair's momenta ===== //

Vector3 unitdirWW = Kinematics::unitDirection(cth,phi);
Lorentz5Momentum outWm23, outWp45;

// assign the outgoing 5-momenta using one of the Kinematics
// helper functions
Kinematics::twoBodyDecay(meMomenta()[0]+meMomenta()[1],
                         m_23,
                         m_45,
                         unitdirWW,
                         outWm23,
                         outWp45);

// ===== now, flat decay of each W into a pair of particles ===== //

Kinematics::generateAngles(cth,phi);
Vector3 unitdir23 = Kinematics::unitDirection(cth, phi);

Kinematics::generateAngles(cth,phi);
Vector3 unitdir45 = Kinematics::unitDirection(cth, phi);

Lorentz5Momentum out2, out3, out4, out5;

Kinematics::twoBodyDecay(outWm23,
                         meMomenta()[2].mass(),
                         meMomenta()[3].mass(),
                         unitdir23,
                         out2,
                         out3);

Kinematics::twoBodyDecay(outWp45,
                         meMomenta()[4].mass(),
                         meMomenta()[5].mass(),
                         unitdir45,
                         out4,
                         out5);

meMomenta()[2] = out2;
```

```
    meMomenta()[3] = out3;
    meMomenta()[4] = out4;
    meMomenta()[5] = out5;

    // Save the jacobian dPS/dr for later use.
    double jacW = Kinematics::CMMomentum(sqrt(sHat()), m_23, m_45) / sqrt(sHat());

    double jac23 = Kinematics::CMMomentum(m_23, out2.mass(), out3.mass())
      / (8.0 * sqr(Constants::pi) * m_23);

    double jac45 = Kinematics::CMMomentum(m_45, out4.mass(), out5.mass())
      / (8.0 * sqr(Constants::pi) * m_45);

    jacobian(jacobian() * jacW * jac23 * jac45);

    return true;
  }
  catch (const ImpossibleKinematics &) {
    return false;
  }
}

// calculation of the matrix element squared
// this function is called repeatedly for every iteration in the Monte Carlo
double MEexample::me2() const
{
  // the ordering of particles here is always the one given by the
  // outside lines in the Tree2toNDiagram() above.
  // watch out for the index count starting at 0 now!
  // i.e. meMomenta()[0] are the momenta of particle 1 (here: em)

  // setup momenta and particle data for the external wavefunctions
  SpinorWaveFunction     em_in( meMomenta()[0],mePartonData()[0],incoming);
  SpinorBarWaveFunction  ep_in( meMomenta()[1],mePartonData()[1],incoming);

  SpinorWaveFunction     ff2_out(meMomenta()[2],mePartonData()[2],outgoing);
  SpinorBarWaveFunction  af3_out(meMomenta()[3],mePartonData()[3],outgoing);
  SpinorBarWaveFunction  af4_out(meMomenta()[4],mePartonData()[4],outgoing);
  SpinorWaveFunction     ff5_out(meMomenta()[5],mePartonData()[5],outgoing);

  // get a pointer to the standard model object in the run
  static const TransientConstRCPtr<Herwig::StandardModel> hwsm
    = dynamic_ptr_cast<TransientConstRCPtr<Herwig::StandardModel> >
    (standardModel());

  if (!hwsm) {
    throw Exception() << "hwsm pointer is null in me2() of MEexample"
                      << Exception::abortnow;
  }

  // get pointers to all required Vertex objects
```

```
  static const TransientRCPtr<Herwig::Helicity::FFVVertex>
    FFWVertex = hwsm->vertexFFW();
  static const TransientRCPtr<Herwig::Helicity::VVVVertex>
    WWWVertex = hwsm->vertexWWW();
  static const TransientRCPtr<Herwig::Helicity::FFVVertex>
    FFPVertex = hwsm->vertexFFP();
  static const TransientRCPtr<Herwig::Helicity::FFVVertex>
    FFZVertex = hwsm->vertexFFZ();

 // get pointers to all required particle data objects
  static const tcPDPtr gamma = getParticleData(ParticleID::gamma);
  static const tcPDPtr Z0 = getParticleData(ParticleID::Z0);
  static const tcPDPtr Wplus = getParticleData(ParticleID::Wplus);
  static const tcPDPtr Wminus = getParticleData(ParticleID::Wminus);
  static const tcPDPtr nu_e = getParticleData(ParticleID::nu_e);

  // ===== sum over helicities to get the matrix element ====== //

  // store intermediate wavefunctions separately for each
  // of the four incoming helicity combinations
  vector<VectorWaveFunction> interP(4), interZ(4), interWp(4), interWm(4);
  size_t index = 0;

  // s-channel diagrams
  // loop over spinor helicity states
  for(unsigned int inhel0 = 0; inhel0 < 2; ++inhel0) {
    em_in.reset(inhel0);

    // loop over spinor helicity states
    for(unsigned int inhel1 = 0; inhel1 < 2 ; ++inhel1, ++index) {
      ep_in.reset(inhel1);

      // evaluate first vertex to get wavefunction of intermediate
      interP.at(index) = FFPVertex->evaluate(sHat(),1,gamma,em_in,ep_in);
      interZ.at(index) = FFZVertex->evaluate(sHat(),1,Z0,    em_in,ep_in);
    }
  }

  // intermediate W-
  index = 0;
  // loop over spinor helicity states
  for(unsigned int outhel2 = 0; outhel2 < 2; ++outhel2) {
    ff2_out.reset(outhel2);

    // loop over spinor helicity states
    for(unsigned int outhel3 = 0; outhel3 < 2; ++outhel3, ++index) {
      af3_out.reset(outhel3);

      // evaluate vertex to get wavefunction of intermediate W-
      interWm.at(index) = FFWVertex->evaluate(sHat(),1,Wminus,ff2_out,af3_out);
    }
```

```
  }

  // intermediate W+
  index = 0;
  // loop over spinor helicity states
  for(unsigned int outhel4 = 0; outhel4 < 2; ++outhel4) {
    af4_out.reset(outhel4);

    // loop over spinor helicity states
    for(unsigned int outhel5 = 0; outhel5 < 2; ++outhel5, ++index) {
      ff5_out.reset(outhel5);

      // evaluate vertex to get wavefunction of intermediate W+
      interWp.at(index) = FFWVertex->evaluate(sHat(),1,Wplus,ff5_out,af4_out);
    }
  }

  SpinorWaveFunction inter_nu_e;

  // this should match the number of diagrams set up above
  static const size_t NUM_OF_DIAGS = 3;

  // need to watch out for coloured products, increases ME by factor 3
  double prefactor = 1.0;
  if (mePartonData()[2]->id()<10)
    prefactor *= 3.0;
  if (mePartonData()[5]->id()<10)
    prefactor *= 3.0;

  // sum variables
  double full_me = 0.0;
  vector<double> me(NUM_OF_DIAGS,0.0);
  vector<complex<double> > diag(NUM_OF_DIAGS,0.0);

  for (size_t i = 0; i<4; ++i) { // Wplus
    for (size_t j = 0; j<4; ++j) { // Wminus
      for (size_t k = 0; k<4; ++k) { // gamma/Z0/e_nu

        // s-channel photon
        diag[0] = WWWVertex->evaluate(sHat(),interP[k],interWp[i],interWm[j]);

        // s-channel Z0
        diag[1] = WWWVertex->evaluate(sHat(),interZ[k],interWp[i],interWm[j]);

        switch (k) {
        case 0: em_in.reset(0); ep_in.reset(0); break;
        case 1: em_in.reset(0); ep_in.reset( 1); break;
        case 2: em_in.reset( 1); ep_in.reset(0); break;
        case 3: em_in.reset( 1); ep_in.reset( 1); break;
        default: cerr << "switch() error in me2()\n";
        }
```

```
        // t-channel neutrino
        inter_nu_e = FFWVertex->evaluate(sHat(),1,nu_e,em_in,interWm[j]);
        diag[2] = FFWVertex->evaluate(sHat(),inter_nu_e,ep_in,interWp[i]);

        // me[1] will contain |d1|^2 + |d1|^2 + |d1|^2 + |d1|^2 , etc.
        for (size_t ii=0; ii<NUM_OF_DIAGS; ++ii) {
          me[ii] += prefactor * std::norm(diag[ii]);
        }

        // full_me will contain
        //              |d1 + d2 + d3|^2 + |d1 + d2 + d3|^2
        //           + |d1 + d2 + d3|^2 + |d1 + d2 + d3|^2
        full_me +=  prefactor
          * std::norm(std::accumulate(diag.begin(),
                                      diag.end(),
                                      complex<double>(0.0)));
      }
    }
  }

  // save individual diagram matrix elements in XComb()
  // and return total ME  (0.25 for spin average)
  DVector save(NUM_OF_DIAGS);
  for (size_t i = 0; i < NUM_OF_DIAGS; ++i) {
    save[i] = 0.25 * me[i];
  }
  meInfo(save);
  return 0.25 * full_me;
}

// final cross-section including all jacobian factors
CrossSection MEexample::dSigHatDR() const
{
  return me2()*jacobian()/(16.0 * Constants::pi * sHat());
}

unsigned int MEexample::orderInAlphaS() const
{
  return 0;
}

unsigned int MEexample::orderInAlphaEW() const
{
  return 4;
}

// selection procedure to decide which diagram gets
// shown in event record
Selector<MEBase::DiagramIndex>
MEexample::diagrams(const DiagramVector & diags) const
```

```
{
  vector<double> single_MEs;
  if ( lastXCombPtr() ) { // grelli wonders why?
    single_MEs.assign(meInfo().begin(),meInfo().end());
  } else {
    cerr << "Last XCombPtr was not there\n";
  }

  Selector<DiagramIndex> sel;
  for ( DiagramIndex i = 0; i < diags.size(); ++i ) {

    // this works only if the diagrams were specified
    // in the same order as the me2() calculations
    //
    // diag -1 is me[0], -2 is 1, -3 is 2, etc.
    int meID = abs(diags[i]->id()) - 1;

    // bounds are checked here, use single_MEs[meID] if that's a problem
    sel.insert(single_MEs.at(meID), i);
  }
  return sel;
}

// colour lines need to be conected correctly
// according to the chosen diagram
Selector<const ColourLines *>
MEexample::colourGeometries(tcDiagPtr diag) const
{
  static ColourLines empty(" ");
  static ColourLines line23("6 -7");
  static ColourLines line45("-8 9");
  static ColourLines both("6 -7,-8 9");

  Selector<const ColourLines *> sel;
  if        (mePartonData()[2]->id()<10 && mePartonData()[5]->id()<10) {
    sel.insert(1.0, &both);
  } else if (mePartonData()[2]->id()<10 && mePartonData()[5]->id()>10) {
    sel.insert(1.0, &line23);
  } else if (mePartonData()[2]->id()>10 && mePartonData()[5]->id()<10) {
    sel.insert(1.0, &line45);
  } else if (mePartonData()[2]->id()>10 && mePartonData()[5]->id()>10) {
    sel.insert(1.0, &empty);
  } else {
    cerr << "Unknown colour selector!\n";
  }
  return sel;
}

// if you define your own member variables, make sure to read/write them here
void MEexample::persistentOutput(PersistentOStream & os) const {}
void MEexample::persistentInput(PersistentIStream & is, int) {}
```

```
// Definition of the static class description member.
ClassDescription<MEexample> MEexample::initMEexample;

void MEexample::Init()
{
  // give a meaningful description here, for the command line interface
  static ClassDocumentation<MEexample> documentation
    ("The \\classname{MEexample} class implements"
     "the 2->4 matrix element for e+e-->W+W- pair production"
     "with subsequent decay of the Ws into fermions.\n"
     "The Ws can be off-shell; cos theta is generated"
     "with a strong forward peak; no ACDC random numbers are used.");
}
```