Practical 3: Computing the R Ratio, I - IR Regularisation

Today we will compute a similar process as our first practical where we computed $e^+e^- \to \mu^+\mu^-$. This time, we will now compute $e^+e^- \to d\bar{d}g$. This is slightly more complicated. Recall that we care about this process as it forms the numerator of the R ratio which tells us important information about charges and colours of quarks. Since we have an outgoing gluon state, things are more subtle as we can now have soft and collinear divergences, in other words, infrared problems.

We won't be able to compute the full cross-section due to the singularities, and we won't do the higher order corrections yet that are needed for things to be well-defined.

So, how can we handle this process to leading order? We have to perform some cut of phase space to avoid the singularities. The range of accepted phase space is known as the fiducial volume. We can then look inside this region and explore different distributions, like we did in the last practical. In this way, we obtain a histogram for the cross section, similar to what we'd get in a real experiment.

We'll look at the well-resolved limit of the process (i.e. when the gluon is detectable). To get a feel for this, we'll compute the result both in Python and in MadGraph.

Lets open VSCode. In `run.py` we should see a new R-ratio experiment. This involves a new method `rratio()`. Lets take a look at this method. Inside, we define a new `process` variable, an instance of `Matrix_3_epem_ddxg_no_z()`. Inside this class we can see there is a function `matrix()`. This has five calls to generate three wavefunctions for the outgoing particles, and two for the incoming. We then have two more for building the final result (stitching them at the vertices). Since we have two diagrams for this process, we can save some time as we don't need to recompute from everything from scratch each time - the diagrams share some common components. By computing the wavefunctions, all we need to do for the 2nd diagram is compute the subpart that differs, `w[1]`. This sort of time-save would be difficult to do analytically.

We, in `rratio()`, then generate the phase space, which in this case we stick to using flat phase space for now.

We can then numerically integrate. This is an adaptive integrator, meaning over each iteration it remaps the integration variables to concentrate samples where the function is largest. What is the integrand? In this case, we generate a phase space point, compute the smatrix and then evaluate at the phase space point, weight by the Jacobian and add to our evaluation for integration.

However, now we have our fiducial cuts. Thus, code the adds our integrand to be evaluated is enclosed by an if statements with `pass_cuts()`. This function takes an event and decides if this is a configuration we want to consider. As you can see in this method, we explicitly compute the cosine to see if the point is within our bounds. Note that our objects are encoded as Lorentz vectors: operations like the dot product automatically respect the Minkowski metric. We also reject points that are below our gluon cut which we have set to `50.0`. If we don't pass the cut, we don't bother computing the matrix element. We don't have to, since it would be weighted by zero for the Jacobian. This makes our code a little more efficient.

If we run this code, we should see a result for the cross section which is not divergent. If we make our cutoffs smaller, we should see that our cross-section becomes bigger, but not by much. Indeed, we saw via our regularisation calculations that the divergence is logarithmic.

We'd like to compare this to MadGraph. Note that MadGraph usually computes proton collisions, not electrons. Thus the cuts by default are slightly different by default.

Let's generate the event

```
generate e+ e- > d d~ g / z
```

and output to a file

```
output quicktest
```

We can launch are see what happens

```
launch
```

In the run parameters, we will set `True = fixed_ren_scale`. Some important parameters that will affect the result of this process are `ptj`, `etaj`, `drjj`. Running this, we should see a cross section that is something like

```
Cross-section :   0.02761 +- 0.0001198 pb
```

This is much smaller than our python cross section. Why is this? Our cuts were much more restrictive. Thus, our cross section result is highly dependant on the momentum range we consider.

A nice exercise is to try and plot the cross section distribution. We can navigate inside the folder `quicktest/Events/run_01` and find the file `unweighted_events.lhe.gz` to see all the exact events we generated. We can extract this with `gunzip unweighted_events.lhe.gz`, and inspect it with `vim unweighted_events.lhe`. This will contain inside all the events that MadGraph considered in the cross section calculation. You can in theory plot all of these events in a histogram. However, the subtly is parsing the format. We can use the MadGraph parser to do this.

Inside our code, we have a method stub `rratio_analyze_events()`. This imports `lhe_parser`. Inside there is a class `CHEPEventFile()`. You can extend this method to parse the file and import all the data we generated, and analyse the events. Since the `CHEPEventFile` is an iterator, we can loop over it. This spits out one event at a time -

just like a collider! We can also get all the data at once. You can also access properties of the involved particles. You should explore the particle attributes and figure out how to access important properties like momentum. This will allow you to compute additional observables.

The good thing about our analysis function is that we can write it to take either data from our Python code, or from MadGraph event files. This means we need only code the analysis once! In the same vein, we could also add code to our integrator to write our events to file so our generated events are saved as same format as MadGraph.