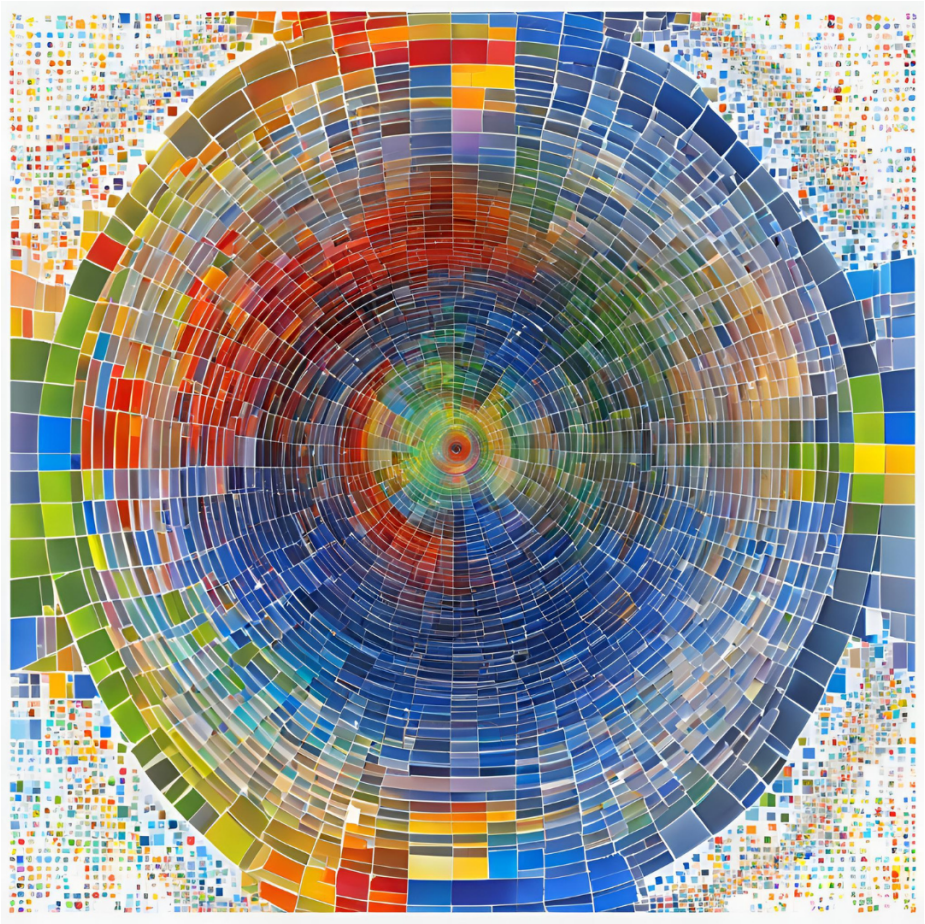
Computational High Energy Physics



Contents

1 Introduction

These notes follow the course 'Computational High Energy Physics' offered by Prof. Dr. Valentin Hirschi in 2025 at the University of Bern. It covers theoretical aspects of implementing a numerical collider in an efficient way, as well as a practical component implementing the various pieces required in order to simulate high energy collisions of the likes at CERN.

Lecture 1: Differential Cross-Sections at Fixed Order

Practical 1: Installing Ubuntu, MadGraph5, and Testing with Python

Setup

To begin with, we will need a working Linux operating system installed on our computer. If you're a Windows user, the easiest way to do this is by using the official Windows Subsystem for Linux (WSL), where the instructions to install can be found here. Once installed, you should be able to run the program Ubuntu, which will open a command prompt Window which allows you to interact with the copy of Ubuntu now running on your computer.

To test to see if your installation is working, you can run the command `ls` to list the contents of the current directory. This will be empty, but should run without errors. You can then make a new folder (directory) named 'chep' by running

```
mkdir chep
```

If you run `ls` again, you should now see this folder in the output. To move into this folder, we can change directory

```
cd chep
```

Next, check to make sure Git is installed, by running

```
which git
```

This should tell you where your git installation is. If this doesn't work, make sure git is installed. If this runs without problem we should be able download the latest version of the course repository by running

```
git clone https://github.com/ValentinHirschi/ComputationalHEP.git
```

and navigate into its directory

```
cd ComputationalHEP/
```

We can run `ls` to see the newly downloaded contents. We can also update this repository week by week by using

```
git pull
```

To edit the code for the course, the simplest way is to use VS Code. To open the repository in VS Code, we can run from the current directory

```
code .
```

This should install and open a VS Code window.

Along with our course code, we will also be running MadGraph5. We install this using

```
wget https://launchpad.net/mg5amcnlo/3.0/3.6.x/+download/MG5_aMC_v3.5.7.tar.gz
```

We can unpack this tarball using

```
tar -xzf MG5_aMC_v3.5.7.tar.gz
```

We should now see the decompressed contents when we run `ls`. We can then remove the zipped file, as we no longer need it

```
rm -rf MG5_aMC_v3.5.7.tar.gz
```

We will now have a new directory that we can navigate into

```
cd MG5_aMC_v3_5_7/
```

We will also need few other things installed to get everything to run properly. To ensure this works, first run

```
sudo apt-get update
```

and enter your password when prompted. Then, we need to install Fortran

```
sudo apt update && sudo apt-get install gfortran
```

and type `Y` to proceed. We also need C++ installed, which can be done with

```
sudo apt install build-essential
```

For our python code that we will run later, we will also need a few packages. Firstly, check `python3 --version`, and make sure it is ≥ 3.12. If not, run

```
sudo add-apt-repository ppa:deadsnakes/ppa -y
sudo apt update
sudo apt install -y python3.12 python3.12-venv python3.12-dev
sudo update-alternatives --install /usr/bin/python3 python3 /usr/bin/python3.12 1
sudo update-alternatives --config python3
```

To install packages, we need to use `pip`. To install this, run

```
sudo apt install python3-pip
python3 -m ensurepip --upgrade
python3 -m pip install --upgrade pip setuptools
```

Then, we install `symbolica`

```
pip install symbolica
```

and `numpy`

```
pip install numpy
```

Our First Process in Madgraph

Lets try and use Madgraph, and see how it works. To run it, use the command

```
./bin/mg5_aMC
```

You should now see the introduction text for MadGraph5. To see the commands that are part of the package, we can type

```
help
```

If you ever get stuck, you can use the `help` command to view documentation and instructions. Lets try and generate the cross section for a simple process, $e^+e^- \to \mu^+\mu^-$. To do this, we will generate the process:

```
generate e+ e- > mu+ mu-
```

The output should say that we have generated one process with two diagrams. These will correspond to scattering via an intermediate photon, and intermediate Z boson. To see these, we can run

```
display diagrams ./
```

This will generate a file with a drawing of the diagrams (you may not be able to open it though on the default Windows installation).

Obviously, if we want to study Quantum Electrodynamics, we don't want to consider the process mediated by the Z boson. We can ignore this process by using instead

```
generate e+ e- > mu+ mu- / z
```

To compute the result, we can firstly output the process

```
output
```

and then begin by typing

```
launch
```

This will begin running the code. We can choose now to adjust some parameters (we have 60 seconds to decide if we want to do this). We won't adjust any of the first options for now, so enter `0`. On the next screen, we would like to edit some run paramters, so enter `2`. This will enter an editor window. You can navigate the text using the arrow keys. To edit the text, we enter 'insert' mode by pressing `i`. To stop editing, press `Esc`. To save and exit this screen, type `:w` to write your edits, then `:q` to exit to the previous screen.

We would like to make a few edits.

Firstly, in the Standard Cuts section, we would like to remove the cutoffs. We should set also `ptl`, the minimum, to `0.0`.

We would also like to set the remove the upper bound on rapidity by setting `etal` to `-1`.

We will also remove the minimum distance between leptons by setting `drll` to `0.0`.

You can then exit to the previous screen, and run the process by entering `0`. This will run and produce a numerical calculation of the cross-section! It should be something like

```
Cross-section :   0.09286 +- 2.585e-05 pb
```

We can now exit Madgraph by typing

```
exit
```

Our First Process in Python

The course code we downloaded contains a Python script that we can use to compute the same process. Lets navigate back up to the project directory with `cd ..` to where our python code will be. Lets open again VS Code by typing `code .`. Inside the `experiments` folder, there is a file called `epem_lplm_fixed_order_LO.py`. This is a Python script that implements the same scattering process that we just computed in MadGraph. We will be trying to understand exactly what this code does, and how to modify it to model processes other than leptonic processes $e^+e^- \to l^+l^-$.

Let's first trying running the code as it is. To see how to do this, we can firstly go back to the terminal, and run

```
./run.py --help
```

This will give us some documentation. It tells us we can run the experiment `epem_lplm_fixed_order_LO`. Let's do this

```
./run.py epem_lplm_fixed_order_LO --seed 3
```

Here, for testing purposes, we specify the random number seed to use in order to get the same result each time. You should see the code run for a few iterations and produce a numerical result for the cross-section, for example

```
Iteration 9: 0.0929257 +- 0.000163142, chi=1.1246
```

This is quite close to the Madgraph value, indeed they lie within each others uncertainty bounds.

Lecture 2: Phase Space Sampling and Cross-Section Perturbative Expansion

Practical 2: Implementing a Phase Space Sampler

This week we will look at how phase space sampling can be implemented in python. We can begin by updating our repository with

```
git pull
```

Make sure you're in the project directory when running this, else you'll get an error. If things aren't working (because you've done some edits), you can run

```
git stash
```

the stash away your changes. You can now `git pull` again to get the latest version. You can then resync your changes on top of the new version using `git stash apply`. In the experiments folder, you should now have a new file, `sampling_experiment.py`. You should now also see that `run.py` has been edited with a new case in `__main__` for the sampling experiment. If we try and run this now, we might find that we're missing dependencies. To add the missing requirements, we can use the requirements file supplied by the repository:

```
python3 -m pip install -r ./CHEP/requirements.txt
```

Lets run the new experiment. To see how to use the new code, we can run

```
python3 run.py sampling_experiment --help
```

This tells us that we have optional run parameters which allows us to specify our random seed. Lets just run the code without a seed. If we wait a moment, we should see some lovely distributions similar to what we have seen in lectures! If the plots don't show (`* UserWarning: FigureCanvasAgg is non-interactive`), try running

```
sudo apt-get install python3-tk
```

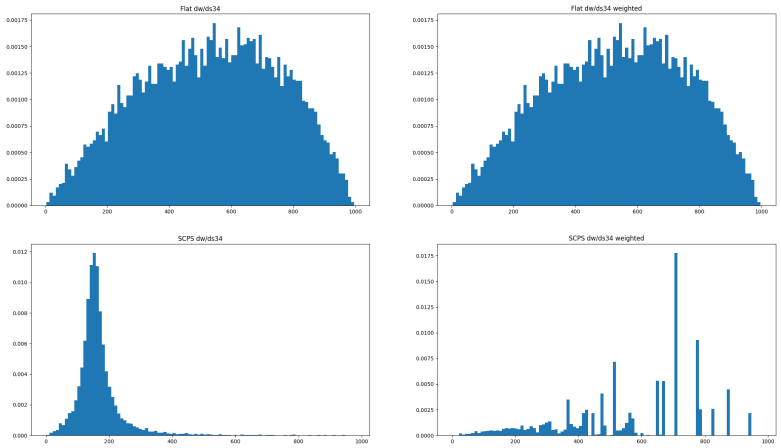
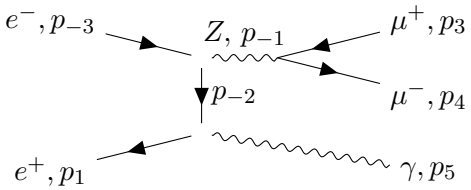


Figure 1: Phase Space Sampling. The top row is "flat space", so the right image is the left multiplied by the Jacobian which is just a constant ("1" for a normalised volume). The bottom row is biased sampling, to pick momenta distributed around the relevant pole of our process, in this case the Z boson with mass 150GeV and width 60GeV.

Lets try and understand what we're looking at, and what the code is doing. Lets open VSCode with `code .`. Inside `sampling_experiment`, firstly, the code picks a model. It then specifies the `topology` - this corresponds to an assignment of momenta to our diagram.



We have a propagator with momentum $p_1 - p_5$ (t-channel). We also have an $s-$channel block. This is specified for our particular process in `get_topology()`. You can `Ctrl+Click` on `get_topology()` to see the assignment explicitly. Inside this function, for the outgoing $s-$channel, we assign momentum 3 to particle ID -12 (μ^+), momentum 4 to 12 (μ^-), and -1 to 23 (Z boson). For the $t-$channel, we assign momentum 1 to ID 11 (positron), 5 to ID 22 (photon), and -2 to the virtual electron. These two channels are joined by the incoming electron vertex, which we assign momentum p_{-3}.

This allows us to encode our graph in terms of relationships between momenta labels and particle types.

We run our numerical collider at centre of mass energy `E_cm=1000.0`. How do we choose our sampling? What is the dimensionality of phase space over which we integrate? Because we specify our particles, the code knows our masses and particle lifetimes. This gives us the resonances. Since we have three outgoing particles, there are 12 DOF. But, they're constrained to be on-shell, reducing this to 9 DOF. Overall momentum conservation gives 4 more constraints, giving us 5 parameters over which we must integrate.

Now, lets run the code. The first output we see is

```
      s-channels:     3(-12) 4(12) > -1(23)
 and t-channels: 1(11) 5(22) > -2(11), -2(11) -1(23) > -3(-11)
 selected path:  [[0], []]
```

These are the pieces of the diagram that we are computing. We should also see next an output of 5 different, random momenta - our 2 incoming and 3 outgoing. We can also see that these generated momenta obey energy conservation - indeed, the next output are the momenta components and the sum over momenta, should be a very small number ($\sim$e$-$12), which is almost zero (with discrepancy due to computer rounding).

Since our supported phase space is compact, we can actually integrate with our phase space measure to find the phase space volume.

We can see exactly how good our integration is over many iterations. For the Single Channel Phase-Space parametrisation, we can see that there is quite a lot of variation between iterations. We can see that for the flat phase-space generator, the variance is quite small. Running the flat space integration gives us basically a constant result. Why is this? Well, the Jacobian of flat space is simply 1/Vol. So it's no surprise that it integrates very easily - it doesn't have to do anything fancy change of coordinates. Thus, run to run, the result is mostly the same.

For the Single Channel Phase-Space parametrisation, we reject a lot of data points due to our sampling. This is the trade off: we have worse statistics in exchange for better sampling.

Let's re-examine the plots. There are 4 plots. The top row is for flat phase space. The left column are our samples. The right column is our sampling weighted by the Jacobean. Unsurprising flat space has a constant Jacobian, so the left and right graphs in the upper row are identical. We are sampling momenta from a uniform distribution in this case, so why is it that the plotted distribution doesn't look uniform? Even though the momentum is uniformly sampled, an arbitrary function of momentum will not look uniform. As an easy example, a uniform sample of points in $[-1, 1] \times [-1, 1]$ will not give a uniform distribution under $f(x, y) = \sqrt{x^2 + y^2}$ - it will be weighted towards large radii as there are more possible points on the circumference.

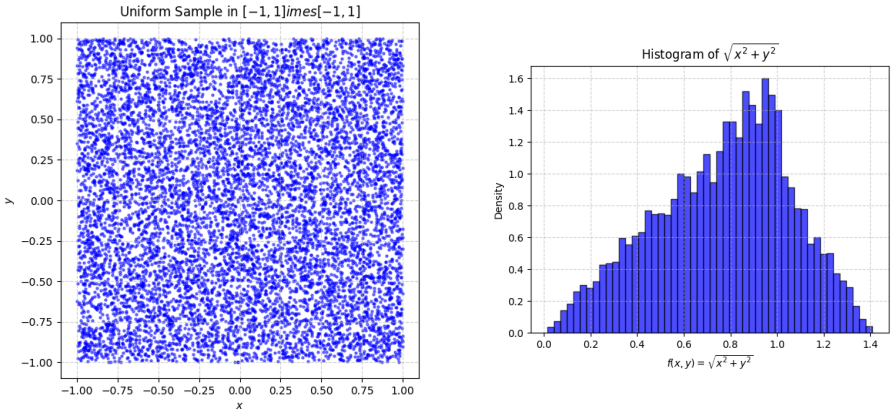


Figure 2: A function on a uniform distribution need not yield a uniform distribution. An easy example is the radius function.

So, the first row makes sense. For the single channel phase space in the bottom row, the distribution on the left is very narrowly distributed to 90GeV. This is because we reject a lot of points in order to only include momenta relevant to the process. In theory, multiplying but the Jacobian, which encodes this change of phase region, should yield in the bottom right a distribution that looks like the top row. Clearly, this doesn't seem to be the case. This is because we don't have enough statistics to accurately reproduce the top row - we have rejected a large number of samples. If we increase the width of the Z boson from 60Gev to say, 6000GeV, we can see the distributions start to match a bit more closely, as we are rejecting fewer points.

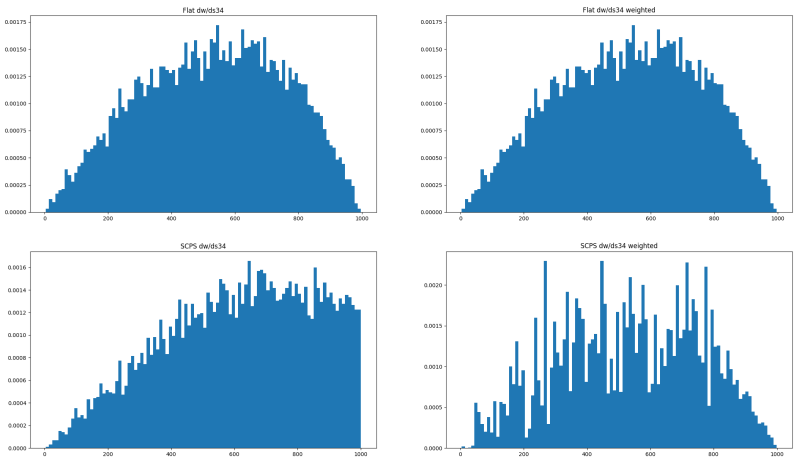


Figure 3: Phase Space Sampling with the Z boson with mass 150GeV and width 6000GeV. The reweighted distribution on the bottom right starts to look more like the top row, as with this width we reject fewer points and thus have better statistics.

Practical 3: IR Regularisation via Cutoffs

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

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

and output to a file

```
1      output quicktest
```

We can launch are see what happens

```
1       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

```
1      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.

Practical 4: The Angular Distribution in Python

For this practical, your task will be to implement the remaining code to reproduce some results for the $e^+e^- \to q\bar{q}g$. Particularly, we would like to reproduce the real emission plot - the distribution of events with respect to angle. Recall that at leading order, with no gluon emission, we expect the relative angle between quarks to be π: they should scatter in opposite directions. As we introduce gluons if we allow gluons in our final state, the angle will be slightly reduced as the gluon will carry some of the momentum. As the gluon becomes soft, the angle will restore to π. We would like to see the exact distribution of angles after implementing a soft cut. Create a new file for our experiment, `rratio_differential.py`. We rename the function to `rratio_differential`. In `run.py`, add this experiment to the cases, as well as the appropriate code to the parser. We will also need to import the file so that `run.py` knows where to find our code,

```
1       from CHEP.experiments.rratio_differential import rratio_differenital
```

We would like to plot a histogram. How do we gather the data we need for plotting? If we look inside our integrand function, we can see the where we generate our data and add to our evaluations. This is where we can start gathering angular data. The simplest way is to add a new argument to our integrand function, which allows it to take an array as input. Note that one should take care when doing this: when we pass an object into a method, depending on the coding language, it can either be a reference to the object's location in memory or copy? If it were a copy, the object will only be modified in the scope of the method, and our original object will be unmodified. This isn't good if we're trying to store data. We want reference for the histogram. This is the default in python - exactly what we want. However, if we want to parallelise, this would make things more difficult as the code would try and modify our array concurrently - though we won't worry about this for now. Let's create an array for our data:

```
1          costheta_histo=[0.0 for _ in range(200)]
```

This gives 200 bins. We could also add an option for number of bins in our experiment run parameters, but for now we will keep things simple. We then edit our integrand constructor to take this vector as an item by adding a parameter `histo`. Inside the integrand method, we can assign our generated phase space point to a set of variables to save the momenta

```
1       p_ep, p_em,p_d,p_db,p_g=ps_point
```

We can then compute the angle between our quarks

```
costhetaqq=p_d.space().dot(p_db.space())/(abs(p_d.space())*abs(p_db.space()))
```

We would then like to assign this to a bin. To do so, we should round the angle down to an appropriate number based on the number of bins. Noting that `int` casting in python is the same as taking the `floor`, then

```
bin_id=int((1+costhetaqq)/2*len(histo))
```

Note that we also need to account for the weight, since we don't have unweighted events. We can save these weights with

```
            histo[bin_id][0]+=wgt
```

A differential distribution is defined for a given observable function $J(\phi)$ of the final-state kinematic configuration ϕ, which in case reads:

$$J(\phi) \equiv \cos\theta_{q\bar{q}}(\phi) := \frac{\vec{p}_q \cdot \vec{p}_{\bar{q}}}{|\vec{p}_q||\vec{p}_{\bar{q}}|} \tag{1}$$

The formal definition of the differential cross-section for this observable function $J(\phi)$ is:

$$\frac{d\sigma}{dJ} := \int d\phi |\mathcal{M}(\phi)|^2 \delta\left(J(\phi) - J\right) \tag{2}$$

In order to solve this integral analytically, one would have to be able to express the observable J in terms of the phase-space parameterization variables supporting ϕ, which is not always possible, especially for complicated observables. Not to mention that one may be interested in monitoring a vast ensemble of different observable at the same time for a given simulation. It is therefore convenient to instead consider a piece-wise constant approximation of the truly continuous differential distribution $\frac{d\sigma}{dJ}$, by constructing histograms instead, with a certain number of bins, each identified with a boundary $b_{i,\min}$ and $b_{i,\max}$. The value of the piece-wise constant function within such an interval, i.e. the "height" h_i of this bin of the histogram, is then:

$$h_i := \int_{b_{i,\min}}^{b_{i,\max}} dJ \left(\frac{d\sigma}{dJ}\right) = \int d\phi |\mathcal{M}(\phi)|^2 \Theta\left[J(\phi) - b_{i,\min}\right] \Theta\left[b_{i,\max} - J(\phi)\right] \tag{3}$$

Notice that it is clear that if the total range of the histogram covers the complete final-state phase-space, then the following propery holds:

$$\sum_i h_i = \sigma_{\text{tot}} \tag{4}$$

This expression (3) is well-suited to be evaluated numerically independently of any choice for the phase space-parameterization. It can essentially be understood as the estimator for the cross-section within a fiducial volume identified by the bin of interest, and during the Monte-Carlo integration, one can simply test whether or not a given sample point belongs to the interior of the bin range.

However, as for any estimator of the central value of an integral, one must normalize the sum of all sample weights by the number of sample points entering that bin, which we must therefore keep track of as follows:

```
        costheta_histo=[(0.0, 0) for _ in range(200)]
```

and change

```
histo[bin_id][0]+=wgt
histo[bin_id][1]+=1
```

We can use this information to appropriately normalise each bin

```
        normalise_histo=[(tot_wgt/n_events) if n_events>0
        else 0 for (tot_wgt,n_events) in costheta_histo ]
```

which we can then plot

```
    bins=np.linspace(-1,1,len(normalise_histo))
    plt.bar(bins,normalise_histo, width=0.01)
    plt.show()
```

As mentioned at the beginning, the resulting distribution will depend on our gluon cut. By setting this to a relatively high energy to remove the soft gluons, for example `GLUON_ENERGY_CUT = 300.0` we can obtain a plot for the angular distribution.

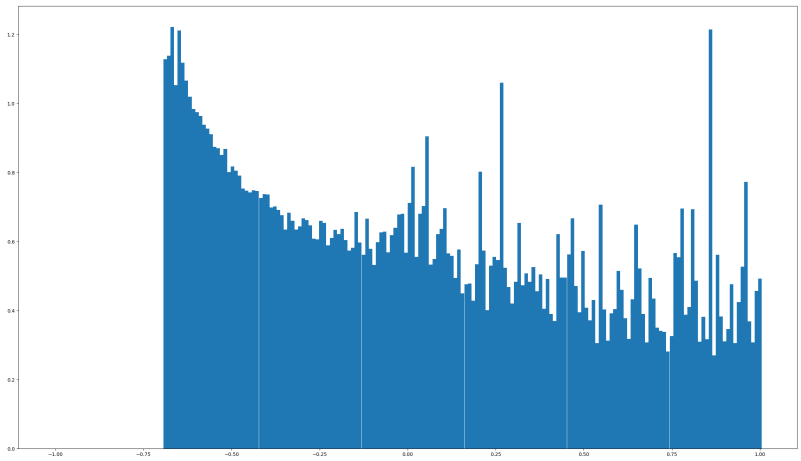


Figure 4: Cross-section angular distribution for the 300 GeV soft gluon cut.

Observe that the distribution close to $\cos(\theta) = -1$ is quite well resolved, while at $\cos(\theta) = 1$ the histogram is much more jagged and noisy. This is an artefact of our sampling: we generate far more points closer to $\theta = \pi$ and thus have better resolution.

Practical 5: Understanding the MadGraph Cuts

Last practical, we finished generating data in our python script to plot a histogram of our angular distribution observable. Firstly, we will make a small comment about this. In our method `pass_cuts()` we have both a gluon cut and a theta cut. Why is it that we need both? Recall that for an outgoing quark of momentum p_1 and outgoing gluon of momentum k, the amplitude we desire has a factor of

$$\frac{1}{p_1 \cdot k} = \frac{1}{E_g E_q (1 - \cos\theta_{gq})} \tag{5}$$

From the left, one may naively assuming that imposing a gluon momentum cut would ensure that the denominator is non-zero. However, of course, this does not prevent the colinear divergence. Likewise, fixing the angle does not stop the soft divergence. Thus, it really is necessary to include both cuts.

We would like to compare our previous results to MadGraph. Let's open it up and compute the process

```
1       generate e+ e- > d d~ g / z
2   display diagrams
```

This is now in memory. We will output

```
1       output CHEP_xcheck
```

to save the generated data for reference. We now `launch CHEP_xcheck` (which is then all that one needs if the `CHEP_xcheck` process has been output already during a prior run of MadGraph). We can access the parameter card by typing `1`. If we would to compare to our Python code, we should double check that each of these parameters are the same. These can be found in `parameters.py`. They should in theory already match what is in MadGraph, but it's good to double check.

We would also like to double check some properties for our run card. Let us type `2` in MadGraph to open it. Firstly, we would like to fix the renormalisation scale

```
1    True = fixed_ren_scale  ! if .true. use fixed ren scale
2        True = fixed_fac_scale  ! if .true. use fixed fac scale
```

We would also like to modify the standard cuts. We would firstly like to change the transverse momentum, `ptj`. Note that this transverse momentum refers to the beam axis momentum, rather than the component of the gluon transverse to the quarks. Fixing the minimum value guarantees that we won't have a soft divergences (since it bounds the momentum from below), however this won't fix our collinear divergence. Next we consider the rapidity of the jets, `etaj`. Note that in relativistic physics, angles are not reference

frame invariant. Thus, instead of performing cuts based on angles, MadGraph uses frame-independent pseudorapidity differences (which are equal to rapidity differences in the case of massless particles), which are a function of the angles, but ideal quantities to observe in a collider experiment since one cannot reconstruct the longitudinal boost connecting the center-of-mass frame of the collision to the laboratory frame. The pseudorapidity given by

$$\eta = -\log\left(\tan\left(\frac{\theta}{2}\right)\right), \tag{6}$$

where θ is the angle between the particle spatial momentum and the positive direction of the beam axis. However, the `ptj` cut already implicitly places a similar angular cut as the pseudorapidity cut would, so it would be of no use for the purpose of further screening infrared singularities. The rapidity is measured relative to the beam axis, and thus it is only enforcing that our jets are angled away from the beam axis.

What is more interesting is `drjj`, the separation between two jets, or the pseudorapidity distance. This is what forces us away from the collinear regime. It is given by

$$\Delta R_{j_1 j_2} = \sqrt{\Delta\eta^2_{j_1 j_2} + \Delta\phi^2_{j_1 j_2}}, \tag{7}$$

where $\eta_{j_1 j_2}$ is the pseudorapidity difference between the two jets, and is the azimuthal angle difference between the two jets. This is preferred to just using the angle because it's much easier than computing $\cos\theta$ experimentally. A value of 0.4 (the default) corresponds to an angle of separation of approximately 10°. If we remove this cut and run the experiment in MadGraph, then can see that we get a singular cross section. Indeed, the process takes a long time to run and each iteration is not within the uncertainty bounds of the previous iteration. This suggests a problem with convergence. Of course, keeping the cut ensures that the computation runs smoothly as expected.

Given that MadGraph uses `drjj` for its cuts, we will modify our code in Python to also use this for our cuts. This is not so difficult to do, as we already have the appropriate operations for Lorentz vectors implemented. Your task is to modify the code to compute `drjj` for our generated phase space points, and determine what the appropriate value of the `drjj` cut that corresponds to our current cuts in terms of θ. This will allow us to cross check both our data generation and analysis in the next practical.

Practical 6: Crosschecking our Generation and Analysis

Today we will generate data in Python and use both our analysis and MadAnalysis to see if we get the same cross section and angular distribution. We will also try generating some data in MadGraph, and run both analyses to see if they match.

Firstly, lets install MadAnalysis. To ensure everything runs smoothly, lets create a new virtual environment. First, we navigate to the `MG5_aMC_v3_5_7` folder and install the dependancies via

```
sudo apt install python3-six
sudo apt install texlive texlive-latex-extra texlive-fonts-recommended
```

The last will ensure we can render our MadAnalysis results as a pdf.

Let's run MadGraph

```
./bin/mg5_aMC
```

and run

```
install MadAnalysis5
```

For this practical, it is best to place our directory `ComputationalHEP` into the MadGraph directory `MG5_aMC_v3_5_7`. This can be done with the `mv` command.

- *Our first task in order to be able to compare the MadAnalysis with our analysis is to ensure both programs are using the same cuts. This means we should implement the cut based on the value of ΔR_{ij}, rather than $\cos\theta$, in our code in the same way MadGraph does.*

We can then run both programs to see if we get the same cross-section. This should hopefully agree. However, this doesn't prove our code to be correct - we may have two compensatory mistakes in our generation and analysis that negate eachother to give us the same answer as MadGraph. To make sure this isn't the case, we can generate data with our code then analyse in MadAnalysis, and generate data with MadGraph and then analyse with our code. If these match, then this is a great sign!

- *So, our next task will be to write a function to import data and analyse generated by MadGraph.*

Recall that we can obtain a dataset from MadGraph by running

```
    generate e+ e- > d d~ g / z
    output quicktest
    launch
```

This will generate a dataset and compute the cross section. We can then

```
    cd quicktest
    cd Events
    cd run_01
    ls
```

and should see a file `unweighted_events.lhe.gz`. This contains the data generated by MadGraph, in compressed form. Inside our `run.py`, we should add a new case to `__main__` for our analysis of MadGraph files. Already, there is a parser defined,

`parser_rratio_analyze_events_experiment`. You should create a new experiment file, similar to `rratio_differential.py`, where instead of generating phase space points in `integrand()`, you import the MadGraph data. Of course, we will need to parse the data in a form that is compatible with our code. The data is stored in a `.lhe` file, which is a complicated format. Fortunately, we can take advantage of MadGraph's own parser. To do this, take a look at `utils/lhe_parser.py`. Note the code

```
    try:
    if os.getenv('MG_ROOT_PATH') is not None:
        sys.path.insert(0, str(os.getenv('MG_ROOT_PATH')))
    else:
        sys.path.append('../')
    from madgraph.various.lhe_parser import EventFile, Event, Particle  # type: ignore
except Exception as e:
    print(f"Could not load Magraph lhe parser: {e}. Specify madgraph root path
    with env. variable MG_ROOT_PATH.")  # nopep8
    sys.exit(1)
from CHEP.utils import CHEP_TEMPLATES
```

This suggests that if we place our `CHEP` folder inside the MadGraph folder, our code will be able to use the built-in Madgraph parser in order to interpret the file.

We then have a daughter class of MadGraph's own `EventFile`,

```

class CHEPEventFile(EventFile):
    def __init__(self, file_path, *args, mode='r', **kwargs):
        super().__init__(file_path, *args, mode=mode, **kwargs)
```

```
        if mode == 'w':
            self.banner = EventFile(os.path.join(CHEP_TEMPLATES,
            "template_events_file.lhe"), mode='r').get_banner()  # nopep8
            self.banner.write(self, close_tag=False)
            self.write("\n"*10)
        self.banner = self.get_banner()
```

We can then run our code with, for example 10 iterations,

```
    python3 run.py rratio_differential -ni 10
```

Once this is implemented, running our analysis on the MadGraph data should give us a similar result to the data we generated ourself!

- *The final task will be to analyse our generated data with MadAnalysis.*

Note that in `rratio_differential.py` we can see a line of code already

```
        event_file = CHEPEventFile("./epem_ddxg.lhe", mode='w')
```

Note that the `integrand()` method takes this as an input argument. We can see in this method that if the phase space point passes our cuts, we already write to the event file. So, after running our code we should already see this file in our directory. There is also code

```
        unweighted_event_file = CHEPEventFile("./epem_ddxg.lhe", mode='r')
    unweighted_event_file.unweight("./unweighted_epem_ddxg.lhe")
    unweighted_event_file.close()
```

and thus we should also see an unweighted version in the directory.

Inside `ComputationalHEP/CHEP/templates`, you should find a template specification file `run.ma5` This is a file that we can run with MadGraph to automatically generate our cross sections and analysis without having to manually type each command in MadGraph. We should open this file

```
    vim ComputationalHEP/CHEP/templates/run.ma5
```

Inside there are lines

```
    import ./epem_ddxg.lhe as weightedEvents
  import ./unweighted_epem_ddxg.lhe as unweightedEvents
```

By changing these to directory and name of our generated files (if they're not already in the current directory), we can import and run MadAnalysis on them with (from with the folder `ComputationalHEP`)

```
1       ../HEPTo
2   ols/madanalysis5/madanalysis5/bin/ma5 CHEP/templates/run.ma5
```

Hopefully, you get cross-sections and histograms that look like those produced by your analysis!