Introduction to Parton Showers and Matching

Tutorial for summer schools

1 Introduction

In this tutorial we will discuss the construction of a parton shower, the implementation of on-the-fly uncertainty estimates, and of matrix-element corrections, and matching at next-to-leading order. At the end, you will be able to run your own parton shower for e^+e^- →hadrons at LEP energies and compare its predictions to results from the event generator Sherpa (using a simplified setup). You will also have constructed your first MC@NLO and POWHEG generator.

2 Getting started

You can use any of the docker containers for the school to run this tutorial. Should you have problems with disk space, consider running `docker containers prune` and `docker system prune` first. To launch the docker container, use the following command

```
docker run -it -u $(id -u $USER) --rm -v $HOME:$HOME -w $PWD <container name>
```

You can also use your own PC (In this case you should have PyPy and Rivet installed). Download the tutorial and change to the relevant directory by running

```
git clone https://gitlab.com/shoeche/tutorials.git && cd tutorials/ps/
```

For simplicity, this tutorial uses PyPy, a just-in-time compiled variant of Python. If you are unfamiliar with Python, think of it as yet another scripting language, such as bash, but way more powerful. A peculiar feature of Python, and indeed its biggest weakness, is that code is structured by indentation. That means you need to pay careful attention to all the spaces in this worksheet. Missing spaces, or additional ones may render your code entirely useless at best. The worst case scenario is that it will still run, but produce the wrong answer.

Some important ingredients of any QCD calculation have been predefined for you. This includes four vectors and operations on them, the running coupling, α_s, and a particle container. We also provide an implementation of the analysis, which you will use at the end of the tutorial to compare predictions with Sherpa. All this so you can fully focus on your parton shower!

Get started by creating a file called `shower.py`. First we need to import the predefined methods

```
import math as m
import random as r

from vector import Vec4
from particle import Particle, CheckEvent
from qcd import AlphaS, NC, TR, CA, CF
```

This will import all above mentioned classes, some important QCD constants, and functions from the math and random library, which come with the pypy installation itself.

The basic ingredients of parton showers are

- the splitting functions,
- the splitting kinematics,
- the veto algorithm.

Let us tackle them one by one.

3 The splitting functions

We will use the splitting functions from Eq. (3.22) in arXiv:1411.4085[1]. First we define an abstract base class, which allows to store the flavors of the splitting function

```
class Kernel:

    def __init__(self,flavs):
        self.flavs = flavs
```

Next we define the first real splitting function, the $q \to qg$ kernel.

```
class Pqq (Kernel):

    def Value(self,z,y):
        return CF*(2./(1.-z*(1.-y))-(1.+z))

    def Estimate(self,z):
        return CF*2./(1.-z)

    def Integral(self,zm,zp):
        return CF*2.*m.log((1.-zm)/(1.-zp))

    def GenerateZ(self,zm,zp):
        return 1.+(zp-1.)*m.pow((1.-zm)/(1.-zp),r.random())
```

This class has several member functions needed at different stages of the event generation. The first is called `Value`, it is used to compute the actual value of the kernel once the full final-state kinematics is known in terms of two variables z and y, which are defined in terms of the quark momentum, p_i, the gluon momentum, p_j, and a "spectator" momentum, p_k, as

$$\begin{aligned} z &= \frac{p_i p_k}{p_i p_k + p_j p_k} \\ y &= \frac{p_i p_j}{p_i p_j + p_i p_k + p_j p_k} \end{aligned} \tag{1}$$

For now, you can think of them as two of the three parameters that define the kinematics of the gluon emission. Ask yourself why the splitting function does not depend on a third parameter.

The next function is `Estimate`. It is used to overestimate the splitting kernel during the veto algorithm. If you are unfamiliar with the veto algorithm, you may want to read about it in Sec. 4.2 of hep-ph/0603175. In the `Estimate` function, we have dropped the dependence on y, and we have approximated the numerator of the splitting function by 2. Think about how the veto algorithm works and ask yourself why the approximation is valid.

The following function simply defines the integral of the overestimate, needed for the selection of a point in the veto algorithm. It depends on the minimal and maximal values of z that can be attained while conserving four-momentum in the branching. Can you derive these boundaries in terms of the transverse momentum? Hint: The kinematics are listed in Eq. (2).

The function `GenerateZ` selects a value for z between the minimum and the maximum, following the probability distribution given by `Estimate`. Together with the point selected for the evolution variable of the shower, and an azimuthal angle, this fixes the kinematics needed for constructing the emission.

You are now in a position to construct the two remaining types of splitting functions, for $g \to gg$ and for $g \to q\bar{q}$ splittings. Name the corresponding classes `Pgg` and `Pgq`. If you are unsure about how to implement them, you may peek at the file `.shower.py`, which contains a reference implementation. Did you get the color factors straight? No? Ask your tutor, why!

[1] Equation (3.22) in arXiv:1411.4085 defines V_{qg} and V_{gg}. We denote the same splitting functions as P_{qq} and P_{gg}

4 The kinematics

Let us now start constructing the parton shower itself.

```
class Shower:

    def __init__(self,alpha,t0):
        self.t0 = t0
        self.alpha = alpha
        self.alphamax = alpha(self.t0)
        self.kernels = [ Pqq([fl,fl,21]) for fl in [-5,-4,-3,-2,-1,1,2,3,4,5] ]
        self.kernels += [ Pgq([21,fl,-fl]) for fl in [1,2,3,4,5] ]
        self.kernels += [ Pgg([21,21,21]) ]
```

This is simply the constructor, which takes two arguments: The strong coupling, and a cutoff scale. Both are stored in the shower object, and a maximum of the strong coupling is computed. We also instantiate a list of splitting kernels, where the $q \to qg$ and $g \to q\bar{q}$ kernels come in multiple copies, one for each possible quark flavor.

Next we set up the parton-shower kinematics. In terms of the above defined variables, z and y, the momenta of the three partons involved in a branching process can be calculated as

$$\begin{aligned} p_i &= z\,\tilde{p}_{ij} + (1-z)y\tilde{p}_k + k_\perp \\ p_j &= (1-z)\,\tilde{p}_{ij} + zy\tilde{p}_k - k_\perp \\ p_k &= (1-y)\tilde{p}_k \end{aligned} \tag{2}$$

The momenta $\tilde{p}_{ij}$ and $\tilde{p}_k$ are the emitter and spectator momenta before the branching process, and $k_\perp$ is a transverse momentum that fulfills the relation $\vec{k}_\perp^2 = 2\tilde{p}_{ij}\tilde{p}_k\, y\, z(1-z)$. This is implemented in our parton shower as follows

```
    def MakeKinematics(self,z,y,phi,pijt,pkt):
        Q = pijt+pkt
        rkt = m.sqrt(Q.M2()*y*z*(1.-z))
        kt1 = pijt.Cross(pkt)
        if kt1.P() < 1.e-6: kt1 = pijt.Cross(Vec4(0.,1.,0.,0.))
        kt1 *= rkt*m.cos(phi)/kt1.P()
        kt2cms = Q.Boost(pijt).Cross(kt1)
        kt2cms *= rkt*m.sin(phi)/kt2cms.P()
        kt2 = Q.BoostBack(kt2cms)
        pi = z*pijt + (1.-z)*y*pkt + kt1 + kt2
        pj = (1.-z)*pijt + z*y*pkt - kt1 - kt2
        pk = (1.-y)*pkt
        return [pi,pj,pk]
```

Note that we have added an azimuthal angle, called ϕ, to the arguments of this function. It will be selected at random between 0 and 2π. Can you find out why we need to boost `kt2`? Why is there a check on the magnitude of `kt1`?

In any branching process, we also need to decide on the new color flow of the final state. The conventional approach is to use the leading color approximation and insert a new color for each emitted gluon, while the flow in $g \to q\bar{q}$ splittings is defined by color conservation. This is implemented by the following method:

```
    def MakeColors(self,flavs,colij,colk):
        self.c += 1
        if flavs[0] != 21:
            if flavs[0] > 0:
```

```
            return [ [self.c,0], [colij[0],self.c] ]
        else:
            return [ [0,self.c], [self.c,colij[1]] ]
    else:
        if flavs[1] == 21:
            if colij[0] == colk[1]:
                if colij[1] == colk[0] and r.random()>0.5:
                    return [ [colij[0],self.c], [self.c,colij[1]] ]
                return [ [self.c,colij[1]], [colij[0],self.c] ]
            else:
                return [ [colij[0],self.c], [self.c,colij[1]] ]
        else:
            if flavs[1] > 0:
                return [ [colij[0],0], [0,colij[1]] ]
            else:
                return [ [0,colij[1]], [colij[0],0] ]
```

The lists `colij` and `colk` represent the colors of the emitter and spectator parton in a bi-fundamental representation, where the first item is the "color" (3) and the second is the "anti-color" ($\bar{3}$). The return value is a list of color assignments for the two daughter partons after the branching. Prove that the above method covers all possible parton splittings `flavs[0]` $\rightarrow$ `flavs[1] flavs[2]`.

5 The veto algorithm

Now we code the heart and soul of any parton shower: The veto algorithm. The basic idea is to generate ordering variables (which we call t) according to the estimated branching probability, and to accept the point generated in t, z and ϕ according to the ratio between true branching probability and overestimate. So first of all we need to loop over all possible values of t that are generated:

```
def GeneratePoint(self,event):
    while self.t > self.t0:
```

Next we set the starting value to the cutoff scale

```
        t = self.t0
```

We iterate over all splitter-spectator pairs that are adjacent in color space

```
        for split in event[2:]:
            for spect in event[2:]:
                if spect == split: continue
                if not split.ColorConnected(spect): continue
```

And we iterate over all kernels that can branch the splitter into two

```
                for sf in self.kernels:
                    if sf.flavs[0] != split.pid: continue
```

We need to compute the z-boundaries in terms of the ordering variable and the maximally available energy, which is given by the invariant mass of the splitter-spectator pair. In order to do so, we need to interpret the ordering variable in terms of kinematical quantities. We choose to order our shower in transverse momentum, therefore t is related to z and y as $t = 2\tilde{p}_{ij}\tilde{p}_k\, y\, z(1-z)$, leading to $z_\pm = (1 \pm \sqrt{1 - 2t_0/(\tilde{p}_{ij}\tilde{p})})/2$. If the allowed z-range is empty, we skip this pair

```
                    m2 = (split.mom+spect.mom).M2()
                    if m2 < 4.*self.t0: continue
                    zp = .5*(1.+m.sqrt(1.-4.*self.t0/m2))
```

Next we generate a trial emission using Eq. (4.9) in hep-ph/0603175. To keep things simple, we overestimate both the splitting function, and the strong coupling, using the value we pre-calculated in the constructor of our shower

```
g = self.alphamax/(2.*m.pi)*sf.Integral(1.-zp,zp)
tt = self.t*m.pow(r.random(),1./g)
```

If the emission is the one with highest ordering variable, we memorize its parameters

```
if tt > t:
    t = tt
    s = [ split, spect, sf, m2, zp ]
```

The above method lets all splitter-spectator pairs and all splitting functions compete with each other by generating individual values of t. Can you explain why the result is the same as if we had considered the overall integral and then selected one of the branchings according to their relative contribution?

Now that we found the "winner" in the branching competition, we can construct the splitting kinematics. First we update the current state of the shower. That is, if we happen to veto the emission, the next trial emission must start at the current value of t, not at the original one.

```
self.t = t
```

There is a chance that no winner was found at all. In this case the shower must terminate. Otherwise we can go ahead and generate a value for z

```
if t > self.t0:
    z = s[2].GenerateZ(1.-s[4],s[4])
```

Using both z and t we can compute y. This requires again the interpretation of the evolution variable in terms of a kinematic quantity. We choose to order our shower in transverse momentum, therefore t is related to z and y as $t = 2\tilde{p}_{ij}\tilde{p}_k\, y\, z(1-z)$

```
    y = t/s[3]/z/(1.-z)
```

The emission may still not be kinematically possible, therefore we need to check the value of y (Can you think of a reason why $y > 1$ could happen?)

```
    if y < 1.:
```

Next we perform the accept/reject procedure. The true value of the splitting function is computed together with the value of the strong coupling, taken at the transverse momentum of the branching. We divide by the corresponding estimates. This defines the weight which is used for acceptance test against a random number. Note: The additional term $(1-y)$ is the phase-space factor from Eq. (5.20) in hep-ph/9605323.

```
f = (1.-y)*self.alpha(t)*s[2].Value(z,y)
g = self.alphamax*s[2].Estimate(z)
if f/g > r.random():
```

We finally obtained a valid phase space point! Now we pick a value for the azimuthal angle, and we construct the splitting kinematics and the color flow. A new particle is created and inserted into the event record, while splitter and spectator are updated with their new kinematics and (in the case of the splitter) their new color.

```
phi = 2.*m.pi*r.random()
moms = self.MakeKinematics(z,y,phi,s[0].mom,s[1].mom)
cols = self.MakeColors(s[2].flavs,s[0].col,s[1].col)
event.append(Particle(s[2].flavs[2],moms[1],cols[1]))
s[0].Set(s[2].flavs[1],moms[0],cols[0])
s[1].mom = moms[2]
```

We're done with one emission!

```
                        return
```

This was the hardest part. Now we simply iterate the branching procedure in order to resum logarithmic corrections from the hard scale of the problem down to the infrared cutoff. This is implemented by the `Run` method below

```
    def Run(self,event,t):
        self.c = 1
        self.t = t
        while self.t > self.t0:
            self.GeneratePoint(event)
```

We finally need to use our shower. Let us write a simple test program

```
import sys
from matrix import eetojj
from durham import Analysis

alphas = AlphaS(91.1876,0.118)
hardxs = eetojj(alphas)
shower = Shower(alphas,t0=1.)
jetrat = Analysis()

r.seed(123456)
for i in range(10000):
    event, weight = hardxs.GenerateLOPoint()
    t = (event[0].mom+event[1].mom).M2()
    shower.Run(event,t)
    if not CheckEvent(event):
        print "Something went wrong:"
        for p in event:
            print p
    sys.stdout.write('\rEvent {0}'.format(i))
    sys.stdout.flush()
    jetrat.Analyze(event,weight)
jetrat.Finalize("myshower")
print ""
```

Several new aspects of event generation come into play at this point. One is the generation of hard configurations using a matrix-element generator. We will not cover the related Monte-Carlo methods in this tutorial, but if you are interested, you can have a look at the file `matrix.py`, which contains a full-fledged ME generator for $e^+e^- \to q\bar{q}$ at fixed beam energy.

We instantiate the running coupling at two-loops with a reference value of $\alpha_s(m_Z) = 0.118$, and we set the parton-shower cutoff to 1 GeV. We generate 10000 events. They are checked for color and momentum conservation and analyzed using a Durham jet algorithm. The results of the analysis will be stored in a file called `myshower.yoda`. The histograms can be plotted and viewed using

```
rivet-mkhtml myshower.yoda
firefox rivet-plots/index.html
```

You may be interested in a comparison between the very simple code you just created and a state of the art parton-shower program. You can either generate the corresponding results yourself with Herwig, Pythia or Sherpa (using the Rivet analysis `LL_JetRates`), or you can use results from Sherpa that were generated with parameter settings analogous to those that you have been using above

```
rivet-mkhtml -s --mc-errs -c plots.conf Sherpa.yoda myshower.yoda
```

Congratulations! You just coded your first final-state parton shower.

6 Renormalization scale uncertainty

We would now like to estimate the renormalization scale uncertainty associated with the parton-shower prediction. There are two possibilities to tackle the problem: The naive option is to run the shower for three different renormalization scales. This has the apparent disadvantage that with changing renormalization scale the emission probability will change, and therefore the events will not be correlated between the different samples. The better option is to perform the uncertainty estimate on-the-fly, which can be done using the reweighting technique from arXiv:0912.3501 / arXiv:1211.7204.

Revisit the veto algorithm as introduced in the lectures on Monte-Carlo event generators. Convince yourself that changing the branching probability from $f(t)$ to $f'(t)$ implies the corrective weights

$$\frac{f'(t)}{f(t)} \quad \text{for accepted branchings ,} \qquad \frac{g(t)-f'(t)}{g(t)-f(t)} \quad \text{for rejected branchings .}$$

In this context, $g(t)$ denotes the overestimate of the splitting function.

The weights are tied to individual partons. They must be computed for each trial emission between the current starting scale of the evolution and the scale at which the next branching occurs. In particular, if parton one is chosen to branch at t_1 and parton two is chosen to branch at $t_2 < t_1$, then the accept/reject weights must be accumulated between the starting scale and t_1 for *both* parton one and two, i.e. `not` between t_2 and t_1 for parton two. Can you explain why?

We start by adding two functions to the `Particle` class (file `particle.py`) that implement this idea:

```
def GetWeight(self,w,t):
    if len(self.wgt) == 0: return
    for i in range(len(self.wgt[0])):
        if self.wgt[0][i][0] < t: break
        for j in range(len(w)):
            w[j] *= self.wgt[j][i][1]
```

This function accumulates the weights generated at scales larger than `t` and adds them to a set of weights `w`, which is expected to have the same number of entries as the weight container `self.wgt` of the current parton. We assume that the branchings have been generated with decreasing `t`, such that the entries of the weight container are ordered.

```
def AddWeight(self,id,t,w):
    self.wgt += [[]]*(id+1-len(self.wgt))
    self.wgt[id].append([t,w])
```

This function stores a new weight `w` at scale `t` for a variation identified by the integer `id`. If needed, it initializes the corresponding weight container. We declare the basic weight container by adding the following line to the constructor of the `Particle` class:

```
    self.wgt = []
```

Next we implement the reweighting itself. First we modify the constructor of the `Shower` class such as to store a sequence of scale factors

```
def __init__(self,alpha,t0,mur2fac=1.):
    self.mur2facs = [1/mur2fac,mur2fac] if mur2fac != 1. else []
```

Next we define a method that performs the weight computation for a specific parton. Its arguments must contain the parton, the scale, the value of the splitting function and its overestimate and a flag that states whether the branching was accepted or rejected. We exploit the fact that the weights arise from different values of the strong coupling alone. We iterate over all scale factors in `self.mur2facs` and store each weight in the corresponding weight container of the current parton.

```
def Reweight(self,parton,t,g,h,accept):
    for i in range(len(self.mur2facs)):
        f = g*self.alpha(self.mur2facs[i]*t)/self.alpha(t)
        parton.AddWeight(i,t,f/g if accept else (h-f)/(h-g))
```

The reweight function must be called for each trial emission that is kinematically allowed. To this end we modify the `GeneratePoint` function. We replace the ultimate `return` statement by

```
                self.Reweight(s[0],t,f,g,1)
                return
            else:
                self.Reweight(s[0],t,f,g,0)
```

Convince yourself that this will compute and store the weights correctly.

The next step is to accumulate the weights and store them in the shower for subsequent event analysis. We modify the main loop of the shower as follows:

```
def Run(self,event,t):
    self.c = 1
    self.t = t
    self.w = [ 1. for mur2fac in self.mur2facs ]
    while self.t > self.t0:
        self.GeneratePoint(event)
        self.AddWeight(event,self.t)
    self.AddWeight(event,self.t0)
```

Note that we added a call to the (yet to be defined) function `GetWeight` at the end, which accounts for the collection of all reject weights between the last shower branching and the cutoff scale. Why is this necessary?

Finally, we must implement the collection of weights at scale `t` as an iteration over all partons in the event. Once the weights are retrieved, the weight containers of the partons are reset.

```
def AddWeight(self,event,t):
    for parton in event:
        parton.GetWeight(self.w,t)
        parton.wgt = []
```

We are now ready to perform a simulation with on-the-fly estimates of the renormalization scale uncertainty. First we need to instantiate the shower with the desired scale factor and inform the analysis that we will have a number of additional weights per event:

```
shower = Shower(alphas,t0=1.,mur2fac=2.)
jetrat = Analysis(len(shower.mur2facs))
```

For each event we call the analysis with the set of weights computed by the shower:

```
jetrat.Analyze(event,weight,shower.w)
```

This will produce the additional files `myshower.1.yoda` and `myshower.2.yoda`, which correspond to the down- and up-variation of the renormalization scale. Verify that your implementation of the on-the-fly variation is correct by comparing these two results against an explicit computation with reduced/increased renormalization scale. Can you see the benefit of the on-the-fly variation? Are there additional advantages apart from the fact that the predictions are correlated?

7 Matrix element correction

In order to improve the radiation pattern of the hardest parton shower emission, we derive a correction to our splitting functions from the spin-averaged squared matrix element for $\gamma^* \to q\bar{q}g$. If we denote the quark, anti-quark and gluon momentum by p_1, p_2 and p_3, and define $s_{ij} = 2\,p_i p_j$, the ratio to the $\gamma^* \to q\bar{q}$ squared matrix element reads

$$\frac{|M_{q\bar{q}g}|^2}{|M_{q\bar{q}}|^2} = 8\pi\alpha_s\,C_F\,\left(\frac{s_{23}}{s_{13}} + \frac{s_{13}}{s_{23}} + \frac{2s_{12}Q^2}{s_{13}s_{23}}\right)\,. \tag{3}$$

Convince yourself that, using this expression, the corrective weight is given by

$$\frac{(1-x)\left((1-x)^2 + \left(\dfrac{x}{1-z}\right)^2\right)}{1-x(1+z)+x^2\left(z+\left(\dfrac{z}{1-z}\right)^2\right)}\,, \quad \text{where} \quad x = (1-z)(1-y)\,. \tag{4}$$

We implement this in our parton shower using the following function

```
def MECorrection(self,z,y):
    x = (1.-z)*(1.-y)
    num = (1.-x)*(pow(1.-x,2)+pow(x/(1.-z),2))
    den = 1.-x*(1.+z)+x*x*(z+pow(z/(1.-z),2))
    return num/den
```

The correction is applied in addition to the standard acceptance weight. We need to ensure that events with more than two partons are unaltered, hence we insert this statement right before the accept/reject step.

```
if len(event)==4:
    f *= self.MECorrection(z,y)
```

Make sure no additional modifications are necessary. If you are unsure what to check for, ask your tutor.

In order to enable on-the-fly renormalization scale variations in combination with matrix-element corrections, you will need to modify the `Reweight` function of the parton shower. Make the necessary changes and cross-check your results using an explicit variation of the renormalization scale.

8 POWHEG matching

Integrating the matrix element in Eq.(3) over the real-emission phase space in $4-2\varepsilon$ dimensions gives

$$\frac{\mathrm{d}\sigma^{(\mathrm{R})}_{q\bar{q}g}}{\mathrm{d}\sigma_{q\bar{q}}} = \frac{\alpha_s(\mu^2)}{2\pi}\,C_F\,\left(\frac{2}{\varepsilon^2} + \frac{3+2L}{\varepsilon} + \frac{19}{2} - \pi^2 + 3L + L^2\right)\,, \tag{5}$$

where $L = \log(\mu^2/s)$. The corresponding virtual corrections are given by

$$\frac{\mathrm{d}\sigma^{(\mathrm{V})}_{q\bar{q}g}}{\mathrm{d}\sigma_{q\bar{q}}} = \frac{\alpha_s(\mu^2)}{2\pi}\,C_F\,\left(-\frac{2}{\varepsilon^2} - \frac{3+2L}{\varepsilon} - 8 + \pi^2 - 3L - L^2\right)\,. \tag{6}$$

We use these analytic expressions to implement a $\bar{\mathrm{B}}$-function for our POWHEG simulation in `matrix.py`

```
def GeneratePOWHEGPoint(self):
    lo = self.GeneratePoint()
    mu = self.ecms
    l = m.log(mu*mu/(lo[0][2].mom+lo[0][3].mom).M2())
    V = [ -2., -3. - 2.*l, -8.+m.pi*m.pi - 3.*l - l*l ]
```

```
    I = [ 2., 3. + 2.*l, 19./2.-m.pi*m.pi + 3.*l + l*l ]
    if V[0]+I[0] != 0.:
        print "Pole check failed (double pole)"
    if V[1]+I[1] != 0.:
        print "Pole check failed (single pole)"
    K = self.alphas(mu*mu)/(2.*m.pi)*4./3.*(V[2]+I[2])
    return ( lo[0], lo[1]*(1.+K) )
```

It is obvious that the ε^2- and ε-poles cancel between Eq. (5) and Eq. (6), therefore we do not need to perform the pole cancellation test above. However, in more complicated NLO calculations, such a cross-check becomes vital, as it provides a simple yet powerful tool to validate the result.

We can now combine our matrix-element corrected parton shower and the NLO-reweighted matrix-element generator into a full-fledged POWHEG simulation by replacing `hardxs.GenerateLOPoint()` with `hardxs.GeneratePOWHEGPoint()` in `shower.py`.

Generate events using this POWHEG simulation and store the analysis results in a separate `yoda` file by replacing `jetrat.Finalize("myshower")` with `jetrat.Finalize("mypowheg")`. Compare to the parton shower predictions. What can you say about the Catani-Seymour kernels in relation to the full real-emission correction? Could you have anticipated this result based on the expressions above?

9 MC@NLO matching

Next we turn to the MC@NLO matching method. Matrix-element corrections will not be necessary in this case, as the difference between the parton shower and the full real-emission correction is implemented by the hard function. You should therefore disable the matrix element corrections in `shower.py`.

We benefit from the fact that our parton shower is based on the Catani-Seymour dipole splitting functions. This allows us to identify the MC counterterms for our MC@NLO with the Catani-Seymour dipole subtraction terms for $e^+e^- \to q\bar{q}$. We implement both the subtraction terms and the real-emission corrections in `matrix.py`:

```
def Real(self,lome,p1,p2,p3,mu):
    s12 = 2.*p1*p2
    s13 = 2.*p1*p3
    s23 = 2.*p2*p3
    R = lome*(s23/s13+s13/s23+2.*s12*(s12+s13+s23)/(s13*s23))
    cpl = 8.*m.pi*self.alphas(mu*mu)*4./3.
    return cpl*R

def RSub(self,fl,pa,p1,p2,p3,mu):
    s12 = 2.*p1*p2
    s13 = 2.*p1*p3
    s23 = 2.*p2*p3
    y132 = s13/(s12+s13+s23)
    y231 = s23/(s12+s13+s23)
    if y132 < self.amin or y231 < self.amin:
        return [ ]
    z1 = s12/(s12+s23)
    z2 = s12/(s12+s13)
    p13t = p1+p3-y132/(1.-y132)*p2
    p1t = 1./(1.-y231)*p1
    D132 = 1./y132 * (2./(1.-z1*(1.-y132))-(1.+z1))
    D231 = 1./y231 * (2./(1.-z2*(1.-y231))-(1.+z2))
    D132 *= self.ME2(fl,s12+s13+s23,(pa+p13t).M2())
    D231 *= self.ME2(fl,s12+s13+s23,(pa+p1t).M2())
    cpl = 8.*m.pi*self.alphas(mu*mu)*4./3.
    return [ cpl*D132, cpl*D231 ]
```

The argument list of these functions will become clear when we construct the complete hard event. For the moment, it shall suffice to say that `p1`, `p2` and `p3` label the quark, anti-quark and gluon momenta, respectively. Note that we test for a minimum cut on the scaled virtualities y_{132} and y_{231} in the dipole subtraction terms. The corresponding cutoff, `self.amin` should be set to `1.e-10` in the constructor. When a point is generated that falls below this cutoff, we will skip it due to potential numerical problems in the cancellation of real correction and subtraction terms.

Now we are ready to construct the full hard event. We perform the phase-space integration using a forward-branching algorithm, with kinematics defined by the parton shower. In a first step, you should therefore copy the function `MakeKinematics` from `shower.py` to `matrix.py`. Next we need to generate the phase-space variables y and z in Eq. (1) and an azimuthal angle. We also need to compute the corresponding weight. We use priors of $1/y^\alpha$ and $1/z^\beta$, where α and β are adjustable exponents denoted by `self.ye` and `self.ze` in the code. They should be fixed to `.5` and `.01` in the constructor.

```
def GenerateHPoint(self,lo,mu):
    y = pow(r.random(),1./(1.-self.ye));
    w = pow(y,self.ye)/(1.-self.ye);
    z = pow(r.random(),1./(1.-self.ze));
    w *= pow(z,self.ze)/(1.-self.ze);
    phi = 2.*m.pi*r.random()
    w *= (1.-y)/(16.*m.pi*m.pi);
```

Next we determine which of the two leading-order partons radiates, and we construct the kinematics of the full 3-parton final state. This is done based on the leading-order momentum and flavor configuration, which is passed to the `GenerateHPoint` function along with the leading-order weight in form of the variable `lo`.

```
    ij = r.randint(0,1)
    pe = self.MakeKinematics(z,y,phi,lo[0][2+ij].mom,lo[0][3-ij].mom)
```

We can now compute the real emission matrix element and the real subtraction terms, using the functions `Real` and `RSub`. At this point we also perform the cut on small scaled virtualities

```
    Dijk = self.RSub(lo[0][2].pid,lo[0][0].mom,pe[2*ij],pe[2-2*ij],pe[1],mu)
    if len(Dijk) == 0:
        return ( lo[0], 0. )
    R = self.Real(lo[2],pe[2*ij],pe[2-2*ij],pe[1],mu)
```

As the last step in this function, we assemble the H event and its weight. Note that `lo[0]/lo[2]` gives the weight of the leading-order phase space, check the `GeneratePoint` method for details.

```
    return ( [
        lo[0][0],lo[0][1],
        Particle(21,pe[1],[2,1]),
        Particle(lo[0][2].pid,pe[2*ij],[1,0]),
        Particle(lo[0][3].pid,pe[2-2*ij],[0,2])
    ], lo[1]/lo[2]*w*(R-Dijk[0]-Dijk[1]) )
```

To complete the MC@NLO, we need a generator for S events. It is implemented along the lines of the POWHEG generator in Sec. 8, except that the integrated real correction is replaced by the two integrated Catani-Seymour dipole terms:

```
def GenerateSPoint(self,lo,mu):
    l = m.log(mu*mu/(lo[0][2].mom+lo[0][3].mom).M2())
    V = [ -2., -3. - 2.*l, -8.+m.pi*m.pi - 3.*l - l*l ]
    I132 = [ 1., 3./2. + l, 5.-m.pi*m.pi/2. + 3./2.*l + l*l/2. ]
    I231 = [ 1., 3./2. + l, 5.-m.pi*m.pi/2. + 3./2.*l + l*l/2. ]
    if V[0]+I132[0]+I231[0] != 0.:
        print "Pole check failed (double pole)"
```

```
        if V[1]+I132[1]+I231[1] != 0.:
            print "Pole check failed (single pole)"
        K = self.alphas(mu*mu)/(2.*m.pi)*4./3.*(V[2]+I132[2]+I231[2])
        return ( lo[0], lo[1]*(1.+K) )
```

Finally we combine S and H event generation in a single function, which is to be called from `shower.py`:

```
    def GenerateMCNLOPoint(self):
        lo = self.GeneratePoint()
        if r.random() < self.ws:
            nlo = self.GenerateHPoint(lo,self.ecms)
            return ( nlo[0], nlo[1]/self.ws )
        nlo = self.GenerateSPoint(lo,self.ecms)
        return ( nlo[0], nlo[1]/(1.-self.ws) )
```

The parameter `self.ws` determines the fraction of H events to be generated. Usually this would be determined based on the relative variance of the S and H event sample. To keep things simple, we fix this number to `.25` in the constructor.

On the parton-shower end of the MC@NLO matching, we need to respect the reduced starting scale in the evolution of hard events. This scale is identified as the transverse momentum of the branching in the more likely of the two corresponding parton-shower histories. We first determine these two histories, using the fact that their weights correspond to the exact NLO subtraction terms. Then we compute the transverse momentum of the emission in the more likely history based on the phase-space variables z and y. Both tasks are performed by the code snippet below, which is inserted right before the call to `shower.Run(event,t)` in `shower.py`.

```
    if len(event) > 4:
        Dijk = hardxs.RSub(event[3].pid,event[0].mom,
                           event[3].mom,event[4].mom,
                           event[2].mom,91.1876)
        s12 = event[3].mom*event[4].mom
        s13 = event[3].mom*event[2].mom
        s23 = event[4].mom*event[2].mom
        if Dijk[0]/(Dijk[0]+Dijk[1]) > r.random():
            z1 = s12/(s12+s23)
            t = s13*z1*(1.-z1)
        else:
            z2 = s12/(s12+s13)
            t = s23*z2*(1.-z2)
```

Can you explain what needs to be done in processes of higher jet multiplicity?

Generate events using your MC@NLO simulation and store the analysis results in a separate `yoda` file by replacing `jetrat.Finalize("mypowheg")` with `jetrat.Finalize("mymcatnlo")` in `shower.py`. Compare all predictions and judge the matching uncertainty. Could you have anticipated the result based on the expressions you implemented?

The implementation of the MC@NLO method in this tutorial may seem more cumbersome than that of the POWHEG method. But the simplicity of our POWHEG generator is due to the knowledge of the analytic expression of the Born-local integrated real correction. This expression is hardly ever known in practice, and the implementation of POWHEG becomes in fact more complicated than that of MC@NLO. You may want to read about this in Sec. 4.1 of arXiv:1411.4085.