Introduction to LL Resummation

Tutorial for summer schools

1 Introduction

In this tutorial we will discuss the implementation of leading logarithmic resummation for simple additive observables in $e^+e^- \to$hadrons, such as thrust, fractional energy correlation, etc. At the end, you will have gained an understanding of the connection between the analytic approach to resummation and a simple parton shower.

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/ll/
```

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. All this so you can fully focus on the resummation!

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

```
import math as m
import random as r

from vector import Vec4
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 the NLL resummation are

- the parametrization of the observable,
- the radiator function for single-gluon emission,
- the multiple emission correction.

Let us tackle them one by one.

3 The parametrization of the observable

Following the Caesar formalism in hep-ph/0407286, we denote the momenta of the hard partons as p_1 and p_2. Additional soft emissions are denoted by k, and the observable we wish to compute by v. In general, the observable will be a function of both the hard and the soft momenta, $v = V(\{p\},\{k\})$, while in the soft approximation it reduces to a function of the soft momenta alone, $v = V(\{k\})$. In the rest frame of the two hard legs, which span the radiating color dipole, we can parametrize the momentum of a single emission as

$$k = z_1 p_1 + z_2 p_2 + k_T \;, \qquad \text{where} \qquad k_T^2 = 2p_1p_2\, z_1 z_2 \;. \tag{1}$$

We define the rapidity of the emission in this frame as $\eta = \ln(z_1/z_2)/2$. Can you show that $\eta = \ln(z_1 Q/k_T)$, where $Q = \sqrt{2p_1p_2}$? The observable, computed as a function of k when radiated collinear to the hard parton, l, can now be written as

$$V(k) = \left(\frac{k_{T,l}}{Q}\right)^a e^{-b\,\eta} \;. \tag{2}$$

Additive observables can be calculated in the presence of multiple soft gluons as a simple sum, $V(\{k\}) = \sum_i^n V(k_i)$. To compute the LL resummed prediction, we only need the coefficients a and b. For thrust, they are given by $a = 1$ and $b = 1$. Can you derive these coefficients? We encode them in the constructor of our class that will perform the resummation. We also store a reference to the strong coupling.

```
class LL:

    def __init__(self,alpha,a,b):
        self.alpha = alpha
        self.a = a
        self.b = b
```

4 The radiator function

Next we need to compute the resummed prediction itself. This calculation can be broken down into a single-emission and a multiple emission part as

$$\Sigma(v) = \int_v^1 dv'\, \frac{d\sigma}{dv'} = e^{-R(v)}\,\mathcal{F}(v) \;. \tag{3}$$

The single-emission radiator function is given as an integral over the appropriately sectorized collinear splitting function, as explained, for example, in Eq. (2.17) of hep-ph/0407286. If we choose the integration variables for the two-dimensional phase-space integral as $z = 1 - z_1$ and

$$t = k_T^2\,(1-z)^{-\frac{2b}{a+b}} \tag{4}$$

we can write $R(v)$ as

$$R(v) = 2\int_{Q^2 v^{\frac{2}{a+b}}}^{Q^2} \frac{dt}{t} \left[\int_0^1 dz\, \frac{\alpha_s\big(k_T^2\big)}{2\pi}\,\frac{2\,C_F}{1-z}\,\Theta\big(\eta\big) - \frac{\alpha_s(t)}{\pi}\,C_F B_q\right] \;. \tag{5}$$

In this form, the physical interpretation is apparent. The first term in the square brackets is the leading term in the $q \to qg$ splitting function, integrated over z in the region $\eta > 0$. The coefficient B_q is subleading term, integrated over the entire z-range. Can you derive its numerical value? Why does the Θ function appear in this expression? Can you visualize its impact on the physical phase space of the gluon-emission process?

The solution to Eq. (5) is typically written in terms of $\lambda = \alpha_s\beta_0 L$, where $L = -\ln v$. One can define $R(L)$ in terms of a double logarithmic piece, $r(L)$, and a single logarithmic piece, $T(L)$

$$R(v) = 2C_F\left(r(L) + B_q T\left(\frac{L}{a+b}\right)\right)\ , \tag{6}$$

The double logarithmic part is separated into a leading and a sub-leading contribution according to $r(L) = Lr_1(\alpha_s L) + r_2(\alpha_s L)$. You may want to derive the leading contribution using the one-loop running of the strong coupling and the definition in hep-ph/0407286, Eq. (2.21). The result is

$$r_1(\alpha_s L) = \frac{1}{2\pi\beta_0\lambda b}\left((a-2\lambda)\ln\left(1-\frac{2\lambda}{a}\right) - (a+b-2\lambda)\ln\left(1-\frac{2\lambda}{a+b}\right)\right)\ , \tag{7}$$

We will not discuss the sub-leading contribution to $r(L)$ and the sub-leading logarithmic term $T(L)$ in this tutorial. If you like, have a look at hep-ph/0407286 and the file `.nll.py` to see how they can be added. Let us now compute the leading logarithmic result

```
def r1(self,as0,b0,a,b,L):
    l = as0*b0*L
    return 1./(2.*m.pi*b0*l*b)*\
        ((a-2.*l)*m.log(1.-2.*l/a)
         -(a+b-2.*l)*m.log(1.-2.*l/(a+b)))

def r(self,as0,b0,a,b,L):
    return L*self.r1(as0,b0,a,b,L)

def R(self,v):
    L = m.log(1./v)
    return 2.*CF*self.r(self.as0,self.b0,self.a,self.b,L)
```

That was the hardest part!

5 Assembling the pieces

We can now compute the spectrum in Eq. (3) at LL accuracy. This is implemented by the `Run` method of our `LL` class. First we store the reference value of the strong coupling in `self.as0`.

```
def Run(self,event,t):
    self.as0 = self.alpha(t)
```

Next we define the constant $b_0 = \beta_0/(2\pi)$ which is used in the calculation

```
    self.b0 = self.alpha.beta0(5)/(2.*m.pi)
```

We will choose the value v of the observable in a Monte-Carlo fashion by generating it uniformly on a logarithmic scale from 0.01 to 1.

```
    self.v = pow(10.,-2.*r.random())
```

Finally, we compute the resummed prediction $\Sigma(v)$ and return its value. Since the function $\mathcal{F}(v)$ is purely NLL, we do not need to include it

```
    return 2. * m.exp(-self.R(self.v))
```

Now we use the calculation to fill a histogram. Let us write a simple test program

```
import sys, matrix, shapes

alphas = AlphaS(91.2,0.118,0)
hardxs = matrix.eetojj()
shower = LL(alphas,a=1.,b=1.)
shapes = shapes.ShapeAnalysis()

for i in range(1000000):
    event, weight = hardxs.GenerateLOPoint()
    w = shower.Run(event,pow(91.2,2))
    if i%100==0: sys.stdout.write('\rEvent {0}'.format(i))
    sys.stdout.flush()
    shapes.FillHistos(shower.v,weight,[w])
shapes.Finalize('ll')
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. Though not strictly necessary for this exercise, we start from the exact LO matrix element in $e^+e^- \to$hadrons when computing $\Sigma(v)$. 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 one loop with a reference value of $\alpha_s(m_Z) = 0.118$. We generate one million events. The results of the analysis are stored in a file called `ll.1.yoda`. The histograms can be plotted and viewed using

```
rivet-mkhtml ll.1.yoda
firefox rivet-plots/index.html
```

Congratulations! You just coded up your first LL resummation.

6 Implementing resummation as a parton shower

Next we would like to establish the connection between analytic resummation and a parton shower. To this end we will write a simple angular ordered shower without momentum conservation. In fact, as we are working at leading logarithmic accuracy, we only need to generate a single emission in this shower. Get started by creating a file called `llps.py`. Again, we need to import the predefined methods

```
import math as m
import random as r

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

The basic ingredients of parton showers are

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

As we will ignore momentum conservation, we do not need to define the kinematics. This point will be discussed in the next tutorial.

7 The splitting function

First we define the only relevant splitting function at this point, the $q \to qg$ kernel. Since we are working at leading logarithmic accuracy, we only include its leading part in the forward rapidity region, such that $P(z) \to 2C_F/(1-z)\Theta(\eta)$.

```
class Pqq:

    def __init__(self,alpha):
        self.alpha = alpha

    def Value(self,z,t,Q2,v,a,b):
        kt2 = t*pow(1.-z,2.*b/(a+b))
        if pow(1.-z,2) < kt2/Q2: return 0.
        as_soft = self.alpha(kt2)/(2.*m.pi)
        return as_soft*CF*2./(1.-z)

    def Estimate(self,z,eps):
        if z > 1.-eps: return 0.
        as_max = self.alpha(1.)/(2.*m.pi)
        return as_max*CF*2./(1.-z)

    def Integral(self,eps):
        as_max = self.alpha(1.)/(2.*m.pi)
        return as_max*CF*2.*m.log(1./eps)

    def GenerateZ(self,eps):
        ran = r.random()
        return 1.-m.pow(eps,ran)
```

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 splitting variable is known. Can you explain the origin of the `if` condition in this function?

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 assumed an upper limit $1-\varepsilon$ on the z integral, which stems from the phase-space sectorization. Can you derive this limit as a function of the parton-shower cutoff t_c?

The following function defines the integral of the overestimate, needed for the selection of a point in the veto algorithm. It depends on the maximal value of $z_{max} = 1 - \varepsilon$ that can be attained.

The function `GenerateZ` selects a value for z between the minimum and the maximum, following the probability distribution given by `Estimate`. Can you explain the return value of this function?

8 The veto algorithm

We are now in place to set up our parton shower. First we define a constructor that stores a few relevant variables and functions, in particular the coefficients a and b which parametrize the observable, the cutoff v_c of the evolution as a fraction of the observable v, and the strong coupling

```
class Shower:

    def __init__(self,alpha,vc,a,b):
        self.a = a
        self.b = b
        self.vc = vc
        self.kernel = Pqq(alpha)
```

Next we define a function that returns the cutoff in terms of the evolution variable

```
    def TC(self):
        return self.q2*pow(self.vc*self.v,2./(self.a+self.b))
```

Can you derive the relation between v_c and t_c implemented in this function?

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 and z 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.TC():
```

Next we set the starting value to the cutoff scale

```
            t = self.TC()
```

We iterate over all splitter-spectator pairs

```
            for split in event[2:4]:
                for spect in event[2:4]:
                    if spect == split: continue
```

We need to compute the z-boundaries in terms of the ordering variable and the maximally available energy. Here you can make use of the results derived earlier

```
                    eps = pow(self.TC()/self.q2,(self.a+self.b)/(2.*self.a))
```

Next we generate a trial emission using Eq. (4.9) in hep-ph/0603175

```
                    G = self.kernel.Integral(eps)
                    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, eps ]
```

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. 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

```
            if t == self.TC(): return 1.
```

Otherwise we can go ahead and generate a value for z

```
            z = self.kernel.GenerateZ(s[2])
```

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.

```
            f = self.kernel.Value(z,t,self.q2,self.v,self.a,self.b)
            g = self.kernel.Estimate(z,s[2])
            if f/g > r.random():
```

We finally obtained a valid phase space point! If the current value of the observable is larger than the predefined value to be computed, we reject this point, otherwise we accept it.

```
                vi = pow(t/self.q2,(self.a+self.b)/2.)
                if vi > self.v:
                   return 0.
                return 1.
```

This was the hardest part. Now we simply wrap everything with a `Run` method.

```
    def Run(self,event,t):
        self.t = t
        self.q2 = t
        self.v = pow(10.,-2.*r.random())
        return 2. * self.GeneratePoint(event)
```

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

```
import sys, optparse, matrix, shapes

alphas = AlphaS(91.2,0.118,0)
hardxs = matrix.eetojj()
shower = Shower(alphas,vc=0.001,a=1.,b=1.)
shapes = shapes.ShapeAnalysis()

for i in range(1000000):
    event, weight = hardxs.GenerateLOPoint()
    w = shower.Run(event,pow(91.2,2))
    if i%100==0: sys.stdout.write('\rEvent {0}'.format(i))
    sys.stdout.flush()
    shapes.FillHistos(shower.v,weight,[w])
shapes.Finalize('llps')
print ""
```

This part is very similar to the analytic resummation. Before running the program, do not forget to change the name of the output histogram to something new, say `'nll'`, to obtain a file `llps.1.yoda`. You can then compare the analytic prediction and the parton-shower result by running

```
rivet-mkhtml ll.1.yoda llps.1.yoda
firefox rivet-plots/index.html
```

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

9 Things to try next

Here are a few suggestions to modify your calculations

- You may want to change the reference value of the strong coupling or perform the calculation with a fixed value of the strong coupling. Don't forget to write the results to a separate histogram output file.
- You may want to check what happens when you change the dependence of the observable on the variables k_T and η in Eq. (2). For example, change $b \to 0.5$ and rerun the generator. Note that while the limit $b \to 0$ is well defined in Eq. (7), you cannot set $b = 0$ directly. Instead you may want to study what happens when $b = 10^{-3}$, $b = 10^{-6}$, etc.