The Drell–Yan process: pp $\to \ell\bar{\ell}$

Alexander Huss

July 19, 2023

Contents

1 Introduction

We will implement the process pp $\to \ell\bar{\ell}$. This gives us access to a simple hadron-collider process that constitutes a "Standard Candle" at the LHC and has a wide range of applications.

2 Cross section

2.1 Partonic cross section

The squared Matrix Element is essentially the same as the one we looked at in the $\mathrm{e}^+\mathrm{e}^- \to \mu^+\mu^-$ case. To make life a little easier for us, we integrated out the kinematics of the $\mathrm{Z}/\gamma^* \to \ell\bar{\ell}$ system. In particular, this means that we're no longer sensitive to the G_2 function in the $\mathrm{e}^+\mathrm{e}^-$ case that was parity odd.

We begin by defining separate *lepton* and *hadron* structure functions

$$L_{\gamma\gamma}(\hat{s}) = \frac{2}{3}\,\frac{\alpha\,Q_\ell^2}{\hat{s}} \tag{1}$$

$$L_{\mathrm{ZZ}}(\hat{s}) = \frac{2}{3}\,\frac{\alpha\,(v_\ell^2 + a_\ell^2)}{\hat{s}}\,\left|\frac{\hat{s}}{\hat{s} - M_{\mathrm{Z}}^2 + \mathrm{i}\Gamma_{\mathrm{Z}} M_{\mathrm{Z}}}\right|^2 \tag{2}$$

$$L_{\mathrm{Z}\gamma}(\hat{s}) = \frac{2}{3}\,\frac{\alpha\,v_\ell Q_\ell}{\hat{s}}\,\frac{\hat{s}}{\hat{s} - M_{\mathrm{Z}}^2 + \mathrm{i}\Gamma_{\mathrm{Z}} M_{\mathrm{Z}}} \tag{3}$$

and ($N_c = 3$)

$$\mathcal{H}^{(0)}_{\gamma\gamma}(\hat{s}) = 16\pi\,N_c\,\alpha\,\hat{s}\,Q_q^2 \tag{4}$$

$$\mathcal{H}^{(0)}_{\mathrm{ZZ}}(\hat{s}) = 16\pi\,N_c\,\alpha\,\hat{s}\,(v_q^2 + a_q^2) \tag{5}$$

$$\mathcal{H}^{(0)}_{\mathrm{Z}\gamma}(\hat{s}) = 16\pi\,N_c\,\alpha\,\hat{s}\,v_q Q_q \tag{6}$$

that we can use to assemble the *partonic* cross section:

$$\hat{\sigma}_{\bar{q}q\to\ell\bar{\ell}}(p_a, p_b) = \frac{1}{2\hat{s}}\,\frac{1}{36}\left\{L_{\gamma\gamma}(\hat{s})\mathcal{H}^{(0)}_{\gamma\gamma}(\hat{s}) + L_{\mathrm{ZZ}}(\hat{s})\mathcal{H}^{(0)}_{\mathrm{ZZ}}(\hat{s}) + 2\mathrm{Re}\Big[L_{\mathrm{Z}\gamma}(\hat{s})\mathcal{H}^{(0)}_{\mathrm{Z}\gamma}(\hat{s})\Big]\right\} \tag{7}$$

2.1.1 Implementation

We'll use a simple class to save and retrieve Standard Model parameters including some convenience functions. We next implement the different structure functions (we need the additional quark id `qid` to distinguish up-type from down-type quarks as they have different couplings to the Z boson.

```python
def L_yy(shat: float, par=PARAM) -> float:
    return (2./3) * (par.alpha/shat) * par.Ql**2
def L_ZZ(shat: float, par=PARAM) -> float:
    return (2./3.) * (par.alpha/shat) * (par.vl**2+par.al**2) * abs(par.propZ(shat))**2
def L_Zy(shat: float, par=PARAM) -> float:
    return (2./3.) * (par.alpha/shat) * par.vl*par.Ql * par.propZ(shat).real
def H0_yy(shat: float, qid: int, par=PARAM) -> float:
    return 16.*math.pi * 3. * par.alpha*shat * par.Qq(qid)**2
def H0_ZZ(shat: float, qid: int, par=PARAM) -> float:
    return 16.*math.pi * 3. * par.alpha*shat * (par.vq(qid)**2+par.aq(qid)**2)
def H0_Zy(shat: float, qid: int, par=PARAM) -> float:
    return 16.*math.pi * 3. * par.alpha*shat * par.vq(qid)*par.Qq(qid)
```

We can now use those structure functions to implement the partonic cross section

```python
def cross_partonic(shat: float, qid: int, par=PARAM) -> float:
    return (1./2./shat) * (1./36.) * (
            L_yy(shat, par) * H0_yy(shat, qid, par)
        +   L_ZZ(shat, par) * H0_ZZ(shat, qid, par)
        +2.*L_Zy(shat, par) * H0_Zy(shat, qid, par)
    )
```

2.2 Hadronic cross section and lepton-pair observables

The hadronic cross section is obtained by convoluting the partonic one with the parton distribution functions:

$$\sigma_{AB\to\ell\bar{\ell}}(P_A, P_B) = \sum_{a,b} \int_0^1 \mathrm{d}x_a \; f_{a|A}(x_a) \int_0^1 \mathrm{d}x_b \; f_{b|B}(x_b) \; \hat{\sigma}_{ab\to\ell\bar{\ell}}(x_a P_A,\, x_b P_B)\,, \tag{8}$$

where the indices a and b run over all possible partons inside the hadrons A and B, respectively. In the case of the Drell–Yan process at lowest order, the two possible "channels" are: $(a,b) \in \{(q,\bar{q}),\, (\bar{q},q)\}$.

We have already integrated out the lepton decay kinematics but have still access to the information of the intermediate gauge boson, $q^\mu = (p_a + p_b)^\mu = (x_a P_A + x_b P_B)^\mu$. To get the differential cross section, we need to differentiate the above formula, which amounts to injecting delta distributions for the observables we're after. Two variables suitable are the invariant mass, $M_{\ell\ell} = \sqrt{q^2}$, and the rapidity, $Y_{\ell\ell} = \frac{1}{2}\ln\Bigl(\frac{q^0+q^3}{q^0-q^3}\Bigr)$, of the di-lepton system. Being differential in these two observables, we're left with no more integrations at LO and the entire kinematics is fixed:

$$\frac{\mathrm{d}^2\sigma_{AB\to\ell\bar{\ell}}}{\mathrm{d}M_{\ell\ell}\mathrm{d}Y_{\ell\ell}} = f_{a|A}(x_a)\; f_{b|B}(x_b)\; \frac{2\,M_{\ell\ell}}{E_{\mathrm{cm}}^2}\; \hat{\sigma}_{ab\to\ell\bar{\ell}}(x_a P_A,\, x_b P_B)\Bigr|_{x_{a/b}\equiv\frac{M_{\ell\ell}}{E_{\mathrm{cm}}}\mathrm{e}^{\pm Y_{\ell\ell}}} \tag{9}$$

2.2.1 Implementation

Let's implement the hadronic differential cross section

```python
def diff_cross(Ecm: float, Mll: float, Yll: float, par=PARAM) -> float:
    xa = (Mll/Ecm) * math.exp(+Yll)
    xb = (Mll/Ecm) * math.exp(-Yll)
    s = Ecm**2
    shat = xa*xb*s
    lum_dn = (
         par.pdf.xfxQ(+1, xa, Mll)*par.pdf.xfxQ(-1, xb, Mll)
        +par.pdf.xfxQ(+3, xa, Mll)*par.pdf.xfxQ(-3, xb, Mll)
        +par.pdf.xfxQ(+5, xa, Mll)*par.pdf.xfxQ(-5, xb, Mll)
        +par.pdf.xfxQ(-1, xa, Mll)*par.pdf.xfxQ(+1, xb, Mll)
        +par.pdf.xfxQ(-3, xa, Mll)*par.pdf.xfxQ(+3, xb, Mll)
        +par.pdf.xfxQ(-5, xa, Mll)*par.pdf.xfxQ(+5, xb, Mll)
        ) / (xa*xb)
    lum_up = (
         par.pdf.xfxQ(+2, xa, Mll)*par.pdf.xfxQ(-2, xb, Mll)
        +par.pdf.xfxQ(+4, xa, Mll)*par.pdf.xfxQ(-4, xb, Mll)
        +par.pdf.xfxQ(-2, xa, Mll)*par.pdf.xfxQ(+2, xb, Mll)
        +par.pdf.xfxQ(-4, xa, Mll)*par.pdf.xfxQ(+4, xb, Mll)
        ) / (xa*xb)
    return par.GeVpb * (2.*Mll/Ecm**2) * (
         lum_dn * cross_partonic(shat, 1, par)
        +lum_up * cross_partonic(shat, 2, par)
        )
```

3 Playground

3.1 Export source code

We can export the python source code to a file `main.py`:

```python
import lhapdf
import math
import cmath
import numpy as np
import scipy
class Parameters(object):
    """very simple class to manage Standard Model Parameters"""

    #> conversion factor from GeV^{-2} into picobarns [pb]
    GeVpb = 0.3893793656e9

    def __init__(self, **kwargs):
        #> these are the independent variables we chose:
        #>  *  sw2 = sin^2(theta_w) with the weak mixing angle theta_w
        #>  *  (MZ, GZ) = mass & width of Z-boson
        self.sw2  = kwargs.pop("sw2", 0.22289722252391824808)
        self.MZ   = kwargs.pop("MZ", 91.1876)
        self.GZ   = kwargs.pop("GZ", 2.495)
        self.sPDF = kwargs.pop("sPDF", "NNPDF31_nnlo_as_0118_luxqed")
        self.iPDF = kwargs.pop("iPDF", 0)
        if len(kwargs) > 0:
            raise RuntimeError("passed unknown parameters: {}".format(kwargs))
        #> we'll cache the PDF set for performance
        lhapdf.setVerbosity(0)
        self.pdf = lhapdf.mkPDF(self.sPDF, self.iPDF)
        #> let's store some more constants (l, u, d = lepton, up-quark, down-quark)
        self.Ql = -1.;    self.I3l = -1./2.;  # charge & weak isospin
        self.Qu = +2./3.; self.I3u = +1./2.;
        self.Qd = -1./3.; self.I3d = -1./2.;
        self.alpha = 1./132.2332297912836907
        #> and some derived quantities
        self.sw = math.sqrt(self.sw2)
        self.cw2 = 1.-self.sw2  # cos^2 = 1-sin^2
        self.cw = math.sqrt(self.cw2)
    #> vector & axial-vector couplings to Z-boson
    @property
    def vl(self) -> float:
        return (self.I3l-2*self.Ql*self.sw2)/(2.*self.sw*self.cw)
    @property
    def al(self) -> float:
        return self.I3l/(2.*self.sw*self.cw)
    def vq(self, qid: int) -> float:
        if qid == 1:  # down-type
            return (self.I3d-2*self.Qd*self.sw2)/(2.*self.sw*self.cw)
        if qid == 2:  # up-type
            return (self.I3u-2*self.Qu*self.sw2)/(2.*self.sw*self.cw)
        raise RuntimeError("vq called with invalid qid: {}".format(qid))
    def aq(self, qid: int) -> float:
        if qid == 1:  # down-type
            return self.I3d/(2.*self.sw*self.cw)
        if qid == 2:  # up-type
            return self.I3u/(2.*self.sw*self.cw)
        raise RuntimeError("aq called with invalid qid: {}".format(qid))
    def Qq(self, qid: int) -> float:
        if qid == 1:  # down-type
            return self.Qd
        if qid == 2:  # up-type
            return self.Qu
        raise RuntimeError("Qq called with invalid qid: {}".format(qid))
    #> the Z-boson propagator
    def propZ(self, s: float) -> complex:
        return s/(s-complex(self.MZ**2,self.GZ*self.MZ))
#> we immediately instantiate an object (default values) in global scope
PARAM = Parameters()

def L_yy(shat: float, par=PARAM) -> float:
    return (2./3) * (par.alpha/shat) * par.Ql**2
def L_ZZ(shat: float, par=PARAM) -> float:
    return (2./3.) * (par.alpha/shat) * (par.vl**2+par.al**2) * abs(par.propZ(shat))**2
def L_Zy(shat: float, par=PARAM) -> float:
    return (2./3.) * (par.alpha/shat) * par.vl*par.Ql * par.propZ(shat).real
def HO_yy(shat: float, qid: int, par=PARAM) -> float:
    return 16.*math.pi * 3. * par.alpha*shat * par.Qq(qid)**2
def HO_ZZ(shat: float, qid: int, par=PARAM) -> float:
    return 16.*math.pi * 3. * par.alpha*shat * (par.vq(qid)**2+par.aq(qid)**2)
def HO_Zy(shat: float, qid: int, par=PARAM) -> float:
    return 16.*math.pi * 3. * par.alpha*shat * par.vq(qid)*par.Qq(qid)
def cross_partonic(shat: float, qid: int, par=PARAM) -> float:
    return (1./2./shat) * (1./36.) * (
            L_yy(shat, par) * HO_yy(shat, qid, par)
        +   L_ZZ(shat, par) * HO_ZZ(shat, qid, par)
        +2.*L_Zy(shat, par) * HO_Zy(shat, qid, par)
    )
def diff_cross(Ecm: float, Mll: float, Yll: float, par=PARAM) -> float:
    xa = (Mll/Ecm) * math.exp(+Yll)
    xb = (Mll/Ecm) * math.exp(-Yll)
    s = Ecm**2
    shat = xa*xb*s
    lum_dn = (
         par.pdf.xfxQ(+1, xa, Mll)*par.pdf.xfxQ(-1, xb, Mll)
        +par.pdf.xfxQ(+3, xa, Mll)*par.pdf.xfxQ(-3, xb, Mll)
        +par.pdf.xfxQ(+5, xa, Mll)*par.pdf.xfxQ(-5, xb, Mll)
        +par.pdf.xfxQ(-1, xa, Mll)*par.pdf.xfxQ(+1, xb, Mll)
        +par.pdf.xfxQ(-3, xa, Mll)*par.pdf.xfxQ(+3, xb, Mll)
        +par.pdf.xfxQ(-5, xa, Mll)*par.pdf.xfxQ(+5, xb, Mll)
        ) / (xa*xb)
    lum_up = (
         par.pdf.xfxQ(+2, xa, Mll)*par.pdf.xfxQ(-2, xb, Mll)
        +par.pdf.xfxQ(+4, xa, Mll)*par.pdf.xfxQ(-4, xb, Mll)
        +par.pdf.xfxQ(-2, xa, Mll)*par.pdf.xfxQ(+2, xb, Mll)
        +par.pdf.xfxQ(-4, xa, Mll)*par.pdf.xfxQ(+4, xb, Mll)
        ) / (xa*xb)
    return par.GeVpb * (2.*Mll/Ecm**2) * (
         lum_dn * cross_partonic(shat, 1, par)
        +lum_up * cross_partonic(shat, 2, par)
        )
if __name__ == "__main__":
    Ecm = 8e3
    for Yll in np.linspace(-3.6, 3.6, 100):
        dsig = scipy.integrate.quad(lambda M: diff_cross(Ecm,M,Yll), 80., 100.)
        print("{:e} {:e} {:e}".format(Yll,dsig[0],dsig[1]))
    tot_cross = scipy.integrate.dblquad(lambda M,Y: diff_cross(Ecm,M,Y), -3.6, +3.6, lambda M: 80., lambda M: 100. )
    print("total xs = {} pb".format(tot_cross))
```

by using the `tangle` command

```
(org-babel-tangle)
```