Hard Process

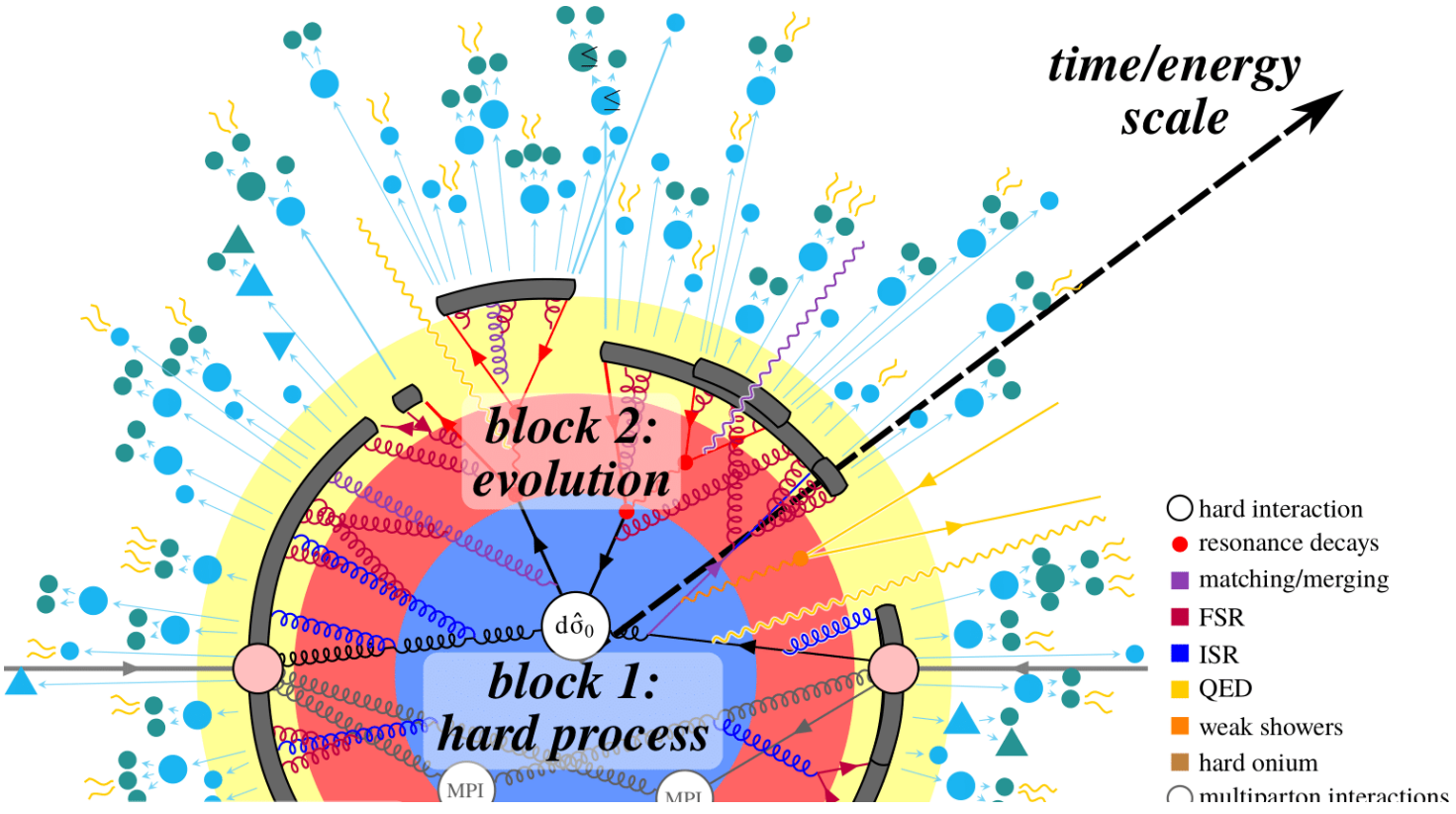
Written by:

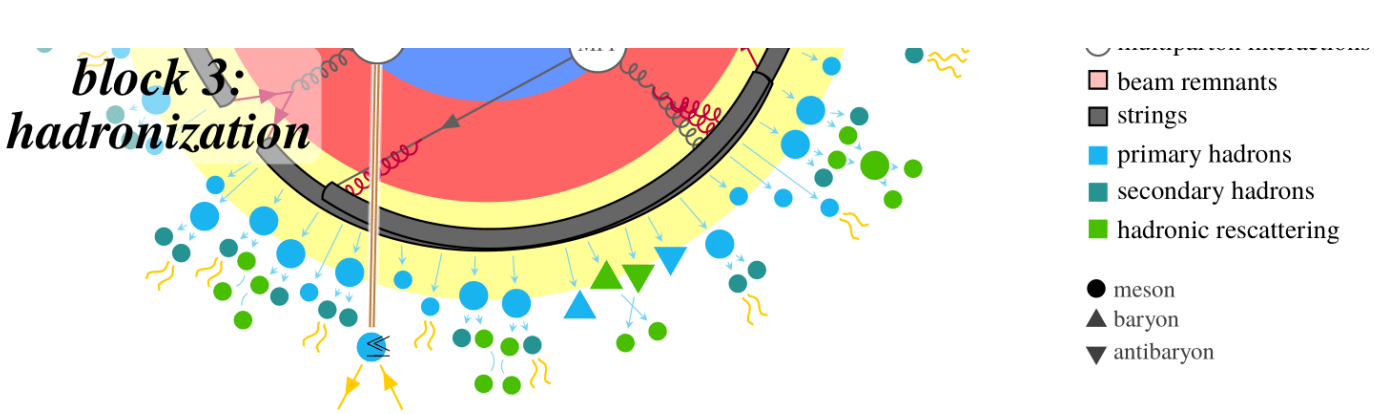
- Philip Ilten (University of Cincinnati)

The starting point for almost any high energy particle physics MC event generator is the hard process, where the signal of interest to the user is generated. For example, this could be $gg \to H$, $q\bar{q} \to Z$, or $gg \to t\bar{t}$. Whatever is chosen becomes the starting point for the entire event. In Pythia, the event generation roughly proceeds through the following steps.

1. Hard process is generated.
2. Resonances prouced in the hard process, like a Higgs or Z are decayed.
3. Perturbative evolution of the hard process via parton showers, initial and final state, is performed.
4. Additional scatterings from the beam, multi-parton interactions (MPI) are interleaved into the perturbative evolution.
5. Partons from the shower are combined together into hadrons through the phenomenoligcal process of hadronization.
6. These hadrons are decayed.
7. Other non-perturbative effects are included, like hadronic rescattering or Bose-Einstein correlations. Sometimes these processes are interleaved with the decays.

Condensing this down even further, we have three major steps: hard process, evolution, and hadronization as shown below.





In this tutorial, we work through how the hard process can be calculated numerically from scratch.

Requirements

In this notebook, we would like to make some comparisons to Pythia to check whether we are getting reasonable answers. To do this, we need to set up our environment. First, we install and import the `wurlitzer` module. This allows programs that have C-like backends to write their output to the Python console. In short, this allows the output of Pythia to be displayed in this notebook.

```
# Redirect the C output of Pythia to the notebook.
!pip install wurlitzer
from wurlitzer import sys_pipes_forever

sys_pipes_forever()
```

```
Requirement already satisfied: wurlitzer in /usr/local/lib/python3.11/dist-packages
```

Next, we need to install the Pythia module.

```
# Install and import the Pythia module.
!pip install pythia8mc
import pythia8mc as pythia8
```

```
Collecting pythia8mc
  Downloading pythia8mc-8.315.0-cp311-cp311-manylinux_2_17_x86_64.manylinux2014_x86
Downloading pythia8mc-8.315.0-cp311-cp311-manylinux_2_17_x86_64.manylinux2014_x86_6
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 29.9/29.9 MB 28.9 MB/s eta 0:00:00
Installing collected packages: pythia8mc
Successfully installed pythia8mc-8.315.0
```

We also have a few local utilities that we need to represent vectors and particle data.

```
# Download the `vector` and `particle` modules.
!wget -q -N https://gitlab.com/mcgen-ct/tutorials/-/raw/main/.full/mc/vector.py
!wget -q -N https://gitlab.com/mcgen-ct/tutorials/-/raw/main/.full/mc/particle.py
```

```
# Import the necessary classes.
from vector import FourVector, Vector, Matrix
from particle import ParticleDatabase, ParticleData
```

We will also need some particle data. We will use the Pythia particle data, which can be read by the `pdb` class we just imported.

```
# Download the Pythia particle database.
!wget -q -N https://gitlab.com/mcgen-ct/tutorials/-/raw/main/.full/mc/data/ParticleData

# Create a particle database we can use throughout this notebook.
pdb = ParticleDatabase()
```

Finally, we need a random number generator. We could use one of the RNGs implemented in [rng.ipynb](), but instead we will use the default `numpy` RNG. We also need the `math` module.

```
# Import the `numpy` and `math` modules.
import numpy as np
import math

# Create an RNG, with a seed of 10.
rng = np.random.default_rng(10)
```

Introduction

In particle physics we oftentimes collide two particles together, and then want to calculate the probability of some given final state. For example, we collide an electron and a positron and want to calculate the probability that we produce a muon and an anti-muon in the final state, $e^-e^+ \to \mu^-\mu^+$. With quantum mechanics we can calculate the probability of transitioning from state A at time $-\infty$ to state B at time $+\infty$. In other words, the initial state A is non-interacting in the far past and the final state B is non-interacting in the far future. This probability of transitioning from A to B is,

$$\left|{}_{+\infty}\langle B|A\rangle_{-\infty}\right|^2 = \left|{}_{+\infty}\langle b_1 \dots b_m|a_1 \dots a_n\rangle_{-\infty}\right|^2$$

using Dirac notation. In the example $e^-e^+ \to \mu^-\mu^+$, $|A\rangle$ is the two particle state $|e^-e^+\rangle$ and $\langle B|$ is the two particle state $\langle\mu^-\mu^+|$. To calculate this probability, either $|A\rangle$ needs to be evolved to time $+\infty$ or $\langle B|$ needs to be evolved to time $-\infty$. This is done with the scattering-matrix S,

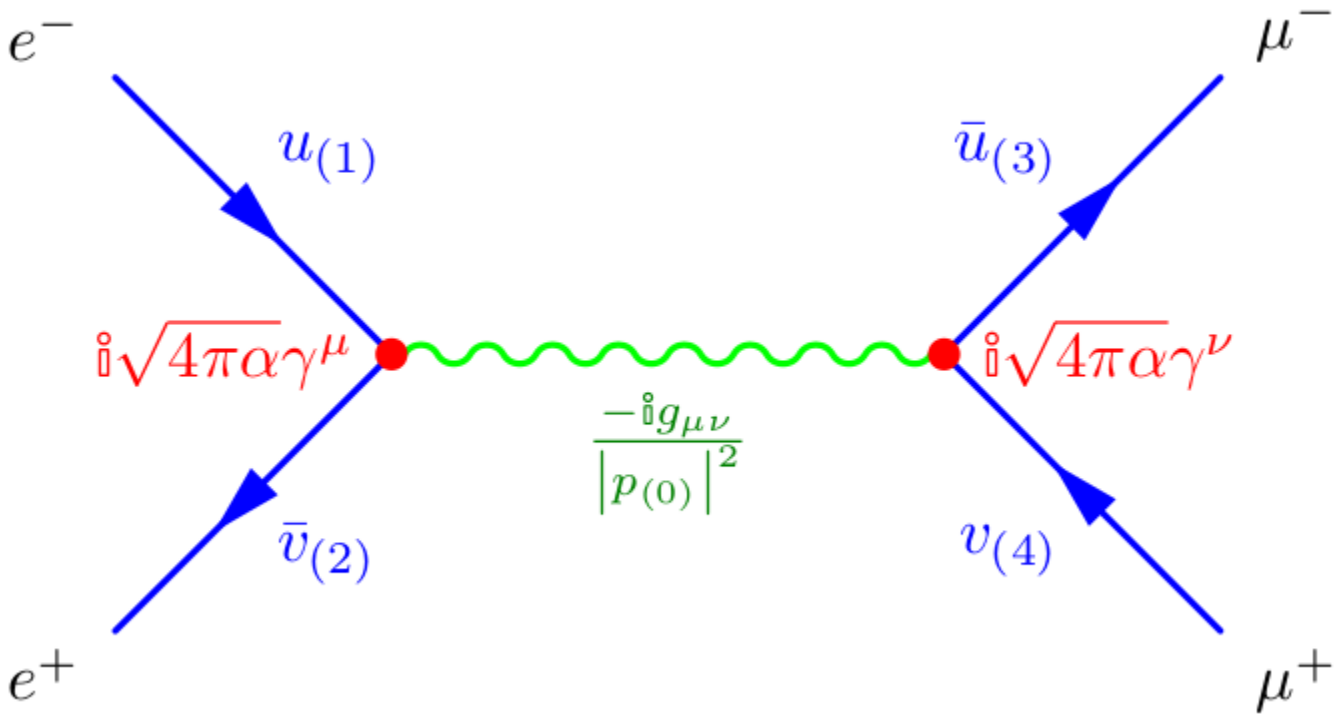
$$|\Psi\rangle_{+\infty} = S^\dagger|\Psi\rangle_{-\infty}$$

which takes a state Ψ in the far future to the far past, and $S^\dagger S = 1$. The scattering-matrix can be factorised into a non-interacting component and an interacting component,

$$\left|{}_{-\infty}\langle B|S|A\rangle_{-\infty}\right| = {}_{-\infty}\langle B|A\rangle_{-\infty} + \mathcal{M}_{A\to B}\mathcal{P}$$

where the non-interacting term is zero if the initial and final state are different, including momenta, and one if identical. The interacting term is written as a matrix element, $\mathcal{M}$, and a four-momentum conservation factor that depends on normalization conventions, $\mathcal{P}$. This matrix element encodes the particle interactions for the process.

The matrix element for a process can be calculated using functional integration, as realized by Richard Feynman. There is a monumental amount of theory behind all this that we are not discussing here, but the end result is quite beautiful and can be visually represented via Feynman diagrams.



Each component of the Feynman diagram represents a mathematical object. In this diagram there are three components: external particle lines in blue, an internal propagator in green, and interaction vertices in red.

$$\mathcal{M} = \mathrm{i}\left[\bar{u}_{(3)}\mathrm{i}\sqrt{4\pi\alpha}\gamma^\mu v_{(4)}\right]\frac{-\mathrm{i}g_{\mu\nu}}{|p_{(0)}|^2}\left[\bar{v}_{(2)}\mathrm{i}\sqrt{4\pi\alpha}\gamma^\nu u_{(1)}\right]$$

After contracting our indices μ and ν we have the following.

$$\mathcal{M} = \frac{-4\pi\alpha}{|p_{(0)}|^2}\left[\bar{u}_{(3)}\gamma^\mu v_{(4)}\right]\left[\bar{v}_{(2)}\gamma_\mu u_{(1)}\right]$$

In this diagram, the electron and positron annihilate into an off-shell photon, which then produces a muon and anti-muon final state. There are infinitely more diagrams which can be drawn with the same initial and final state as this diagram, but these diagrams have more factors of α, the fine structure constant. Since $\alpha \simeq 1/137$ is much less than one, these diagrams contribute very little to the summed matrix element, and can oftentimes be neglected. This is what we call a leading order diagram, as these diagrams represent a perturbative expansion in α of the interaction between the initial and final state.

Each line in the diagram, corresponding to a particle, is assigned a momentum. The electron has momentum $p_{(1)}$, the positron $p_{(2)}$, the photon $p_{(0)}$, the muon $p_{(3)}$, and the anti-muon $p_{(4)}$. The photon momentum is determined by conservation of energy and momentum, $p_{(0)} = p_{(1)} + p_{(2)}$. Additionally, each incoming or outgoing particle is assigned a helicity, $\lambda_{(i)}$, which is the spin of the particle projected along the direction of the particle momentum. Fermions are spin $\frac{1}{2}$ and can take on two helicity eigenvalues, either $+1$ or -1. The fermion lines, e.g., the electron/positron and muon/anti-muon lines, are Dirac spinors which represent the wave function of the particle.

Our goal is to numerically represent this matrix element, and then integrate a cross-section. Throughout this notebook we will work in the Weyl basis to define our Dirac spinors u and v, and the Dirac matrices γ^μ.

Dirac Matrices

The Dirac matrices, γ^μ are defined as,

$$\gamma^0 = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} \quad \gamma^1 = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & -1 & 0 & 0 \\ -1 & 0 & 0 & 0 \end{pmatrix} \quad \gamma^2 = \begin{pmatrix} 0 & 0 & 0 & -\mathrm{i} \\ 0 & 0 & \mathrm{i} & 0 \\ 0 & \mathrm{i} & 0 & 0 \\ -\mathrm{i} & 0 & 0 & 0 \end{pmatrix} \quad \gamma^3 = \begin{pmatrix}$$

and can have their index raised and lowered with the Minkowski metric,

$$g_{\mu\nu} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & -1 \end{pmatrix}$$

just like a four-vector. The repeated index in the matrix element definition for $e^- e^+ \to \mu^- \mu^+$ indicates that a summation should be performed over μ from 0 to 3.

Exercise: implement the Dirac matrices

Using the Weyl basis above, implement the Dirac matrices in the skeleton class below. Here, the `FourVector` class transforms under the Minkowski metric using the notation `~v` for `FourVector v`.

```
class DiracMatrices(FourVector):
    """
    This class provides the Dirac matrices. Note that this class
    inherits from the 'FourVector' class. This is because the Dirac
    matrices also transform under the Minkowski metric, just like
```

```
    standard four-vectors.
    """

    def __init__(self, v0=None, v1=None, v2=None, v3=None):
        """
        Initialize the Dirac matrices. Ideally this would not be mutable.
        """
        g0 = Matrix(
            [0.0, 0.0, 1.0, 0.0],
            [0.0, 0.0, 0.0, 1.0],
            [1.0, 0.0, 0.0, 0.0],
            [0.0, 1.0, 0.0, 0.0],
        )
        g1 = Matrix(
            [0.0, 0.0, 0.0, 1.0],
            [0.0, 0.0, 1.0, 0.0],
            [0.0, -1.0, 0.0, 0.0],
            [-1.0, 0.0, 0.0, 0.0],
        )
        g2 = Matrix(
            [0.0, 0.0, 0.0, -1.0j],
            [0.0, 0.0, 1.0j, 0.0],
            [0.0, 1.0j, 0.0, 0.0],
            [-1.0j, 0.0, 0.0, 0.0],
        )
        g3 = Matrix(
            [0.0, 0.0, 1.0, 0.0],
            [0.0, 0.0, 0.0, -1.0],
            [-1.0, 0.0, 0.0, 0.0],
            [0.0, 1.0, 0.0, 0.0],
        )
        FourVector.__init__(self, g0, g1, g2, g3)
```

Exercise: use the Dirac matrices

Once a `DiracMatrix` object, say `dm`, the individual matrices can be accessed by the index operator, `[i]`. Since we will be wanting to check a few outputs, let us write a little `show` method which makes this a little easier.

```
# Create the Dirac matrices.
dm = DiracMatrices()

# Loop over the Dirac matrices and print.
for i in range(0, 4):
    print(f"------ dm[{i}] = \n{dm[i]}]")
```

```
 ------ dm[0] =
        0.0         0.0         1.0         0.0
        0.0         0.0         0.0         1.0
        1.0         0.0         0.0         0.0
```

```
       0.0        1.0        0.0        0.0]
------ dm[1] =
       0.0        0.0        0.0        1.0
       0.0        0.0        1.0        0.0
       0.0       -1.0        0.0        0.0
      -1.0        0.0        0.0        0.0]
------ dm[2] =
       0.0        0.0        0.0    (-0-1j)
       0.0        0.0         1j        0.0
       0.0         1j        0.0        0.0
   (-0-1j)        0.0        0.0        0.0]
------ dm[3] =
       0.0        0.0        1.0        0.0
       0.0        0.0        0.0       -1.0
      -1.0        0.0        0.0        0.0
       0.0        1.0        0.0        0.0]
```

Dirac Spinors

In the Weyl basis, the Dirac spinors can be written as,

$$u(p,\lambda) = \begin{pmatrix} \kappa(p,\lambda)\sqrt{p^0 - \lambda q} \\ \kappa(p,\lambda)\sqrt{p^0 + \lambda q} \end{pmatrix} \quad v(p,\lambda) = \begin{pmatrix} -\lambda\kappa(p,-\lambda)\sqrt{p^0 + \lambda q} \\ \lambda\kappa(p,-\lambda)\sqrt{p^0 - \lambda q} \end{pmatrix}$$

where q is $\sqrt{p^1p^1 + p^2p^2 + p^3p^3}$, κ is given by,

$$\kappa(p,\lambda) = \begin{cases} \xi \begin{pmatrix} q + p^3 \\ \mathrm{i}p^2 + p^1 \end{pmatrix} & \text{for } \lambda = +1, \begin{pmatrix} 0 \\ 1 \end{pmatrix} \quad \text{as } p^3 \to -q, p^2 = 0, p^1 \to +0 \\ \xi \begin{pmatrix} \mathrm{i}p^2 - p^1 \\ q + p^3 \end{pmatrix} & \text{for } \lambda = -1, \begin{pmatrix} -1 \\ 0 \end{pmatrix} \quad \text{as } p^3 \to -q, p^2 = 0, p^1 \to +0 \end{cases}$$

the normalisation ξ is,

$$\xi = \frac{1}{\sqrt{2\left(q^2 + qp^3\right)}}$$

and Einstein notation has been used for the momentum p, e.g., p^0 is the energy, p^1 is the x-component, p^2 is the y-component, and p^3 is the z-component. Note that both u and v are vectors of length four, but they do not transform under Lorentz transformations. The bar of a Dirac spinor, or anti-particle Dirac spinor, is given by $\bar{u} = u^\dagger\gamma^0$, where $u^\dagger$ is the conjugate transpose of u.

Exercise: particle data

Before we implement a particle class that can return Dirac spinors, we should make sure we understand how to work with the particle database. Access a muon and print all the relevant data that is stored for it.

```
# Get the muon.
pd = pdb["mu-"]

# Loop over its members and print.
for key, val in pd.__dict__.items():
    print(f"--\n{key}: {val}")
```

```
--
pid: 13
--
name: mu-
--
mass: 0.10566
--
tau: 658654.0
--
spin: 2
--
charge: -3
--
colour: 0
--
anti:    pid: -13
  name: mu+
  mass: 0.10566
   tau: 658654.0
  spin: 2
charge: 3
colour: 0
```

Exercise: implement a particle with Dirac spinors

We could implement a class just to store the Dirac spinors, but really we want Dirac spinors as associated with a particle that has some momentum and helicity. To that end, let us define a `Particle` class using the skeleton below. Everything is already filled in, except the `w` method which returns the Dirac spinor.

```
class Particle:
    """
    This class represents a particle.
    """

    def __init__(self, data, p, h):
        """
        Initialize the `Particle` class, given `data` of type
        `ParticleData` for that particle type, the momentum
        four-vector `p`, and the helicity `h`.

        Additional members are also available.
        c: color for this particle.
        a: anti-color for this particle.
```

```
    t: production vertex for this particle.
    parents:  list of parents for this particle.
    children: list of children for this particle.
    """
    from math import sqrt

    self.data = data
    self.p = +p
    if self.p[0] < 0:
        self.p[0] = sqrt(sum([pj**2 for pj in p[1:]]) + data.mass**2)
    self.h = float(h)
    self.c = 0
    self.a = 0
    self.t = FourVector(0, 0, 0, 0)
    self.parents = []
    self.children = []

def __str__(self):
    """
    Return a string to print this particle.
    """
    return ("%6s: %r\n" * 3 + "h, c, a: %.2e, %i, %i\n") % (
        "data",
        self.data,
        "p",
        self.p,
        "t",
        self.t,
        self.h,
        self.c,
        self.a,
    )

def w(self):
    """
    Return the Dirac spinor for this particle.
    """
    from math import sqrt

    if self.data.spin != 2:
        return None
    # Check if particle or anti-particle.
    s = -1 if self.data.pid < 0 else 1
    p = sqrt(sum([pj**2 for pj in self.p[1:]]))
    # Handle if |p| == p[3].
    if p + self.p[3] == 0:
        xi = 1.0
        if s * self.h == 1:
            kappa = [0, 1]
        elif s * self.h == -1:
            kappa = [-1, 0]
        else:
            kappa = [0, 0]
```

```
        # Handle otherwise.
        else:
            xi = 1.0 / sqrt(2.0 * p * (p + self.p[3]))
            if s * self.h == 1:
                kappa = [p + self.p[3], self.p[2] * 1.0j + self.p[1]]
            elif s * self.h == -1:
                kappa = [self.p[2] * 1.0j - self.p[1], p + self.p[3]]
            else:
                kappa = [0, 0]
        hp = xi * sqrt(self.p[0] + self.h * p)
        hm = xi * sqrt(self.p[0] - self.h * p)
        # Return the anti-particle spinor.
        if s == -1:
            return Vector(
                -self.h * kappa[0] * hp,
                -self.h * kappa[1] * hp,
                self.h * kappa[0] * hm,
                self.h * kappa[1] * hm,
            )
        # Return the particle spinor.
        else:
            return Vector(kappa[0] * hm, kappa[1] * hm, kappa[0] * hp, kappa[1] * hp)

    def wbar(self):
        """
        Return the bar Dirac spinor for this particle.
        """
        w = ~self.w()
        w[0], w[1], w[2], w[3] = w[2], w[3], w[0], w[1]
        return w
```

Cross-Section

We now have almost all the ingredients needed to calculate the probability for the process $e^- e^+ \to \mu^- \mu^+$. In the end, what we actually calculate is the cross-section: the process probability, per unit time, over the particle flux of the initial state. The cross-section has units of area, typically given in barn which is $10^{-28}\ \mathrm{m}^2$. The differential cross-section for a two-to-two scattering is given by,

$$\mathrm{d}\sigma = \left(\frac{\hbar c}{8\pi}\right)^2 \frac{\mathcal{S}}{(p_{(1)}^0 + p_{(2)}^0)^2} \sqrt{\frac{\sum_{i=1}^3 (p_{(3)}^i)^2}{\sum_{i=1}^3 (p_{(1)}^i)^2}} \left(\mathcal{M}^* \mathcal{M}\right) \sin\theta \, \mathrm{d}\theta \, \mathrm{d}\phi$$

where this is differential in two variables, ϕ and θ, and $\mathcal{S}$ is $\frac{1}{2}$ if the outgoing particles are identical, 1 otherwise. Assuming a frame where both the incoming electron and positron are oriented along the z-direction and have equal and opposite momenta, the momenta for the outgoing muon and anti-muon can be defined as,

$$p_{(3)} = (p_{(1)}^0,\ q \sin\theta \cos\phi,\ q \sin\theta \sin\phi,\ q \cos\theta)$$

$$p_{(4)} = (p^0_{(1)}, \ -q\sin\theta\cos\phi, \ -q\sin\theta\sin\phi, \ -q\cos\theta)$$

in terms of ϕ and θ where $q = \sqrt{p^0_{(1)}p^0_{(1)} - m_{(3)}m_{(3)}}$.

But, we still need to be able to perform the integral over ϕ and θ. This is where Monte Carlo integration enters. This is a very powerful numerical technique that can be used to integrate an n-dimensional function. See integrate.ipynb for more details. The integral for a two-variable function is,

$$\int_{x_{\min}}^{x_{\max}} \int_{y_{\min}}^{y_{\max}} f(x,y)\,\mathrm{d}x\,\mathrm{d}y \approx \langle f \rangle (x_{\max} - x_{\min})(y_{\max} - y_{\min})$$

where $\langle f \rangle$ is the average value for $f(x,y)$ when randomly sampled with uniform x and y. Consider an $f(x,y)$ which is 1 if within the unit circle, and 0 otherwise. We can find the integral by randomly sampling an x from -1 to 1, and similarly a y from -1 to 1, and evaluating $f(x,y)$. We can do this a large number of times and calculate the average value for $f(x,y)$. As we sample more and more, this average value, multiplied by 4 will approach the area of the unit circle.

Exercise: integrate a circle

Let us first try the first example of integrating a circle to calculate π. First, write a 2D MC integration class, following the skeleton below.

```
class Integrator:
    """
    This class integrates a two variable function.
    """

    def __init__(self, rng, f, xmin, xmax, ymin, ymax):
        """
        Initialize the integrator, given a random number generator `rng`,
        function `f`, a minimum x `xmin`, a maximum x `xmax`,
        a minumum y `ymin`, and a maxumimum y `ymax.
        """
        self.rng = rng
        self.f = f
        self.xmin = xmin
        self.xmax = xmax
        self.ymin = ymin
        self.ymax = ymax
        self.xdif = xmax - xmin
        self.ydif = ymax - ymin

    def mc(self, n=1000):
        """
        Perform MC integration for given number of sampling points `n`.
        """
        t = 0
```

```
        for i in range(0, n):
            x = self.xmin + rng.uniform() * self.xdif
            y = self.ymin + rng.uniform() * self.ydif
            t += self.f(x, y)
        return t / float(n) * self.xdif * self.ydif
```

Now, integrate a circle and calculate π.

```
# Define the circle function.
def circle(x, y):
    """
    Return 1 if `x` and `y` in a unit circle, 0 otherwise.
    """
    from math import sqrt

    # 1 if inside, 0 if outside.
    f = sqrt(1 - x**2)
    return 0 if abs(y) > f else 1


# Circle integration.
integrator = Integrator(rng, circle, -1, 1, -1, 1)
print(integrator.mc())
```

```
3.108
```

Exercise: calculating the differential cross-section

Create a class `Annihilate` which has members corresponding to the four particles of $e^-e^+ \to \mu^-\mu^+$ using the skeleton below. Specifically, define the `xs` method which implements the $2 \to 2$ cross-section formula. Note that the matrix element has already been defined.

```
class Annihilate:
    """
    This class defines the cross-section function needed to calculate
    the integrated cross-section of e+ e- -> mu+ mu-.
    """

    def __init__(self, p1, p2, p3, p4):
        """
        Initialize the
        """
        from math import pi

        self.p1 = p1
        self.p2 = p2
        self.p3 = p3
        self.p4 = p4
        self.dmu = DiracMatrices()
```

```
        self.dmu = DiracMatrices()
        self.dml = ~self.dmu
        # Calculate the cross-section prefactor ((hbar c)/(8 pi))^2 in
        # units m^2 GeV^2.
        self.xspre = (1.97326979e-16 / (8 * pi)) ** 2
        # Calculate the matrix-element prefactor (-4 pi alpha).
        self.mepre = -4 * pi / 137.0

    def me(self):
        """
        Return the matrix element given the state of the internally
        represented particles.
        """
        p0 = self.p1.p + self.p2.p
        return (
            self.mepre
            / p0**2
            * sum(
                [
                    (self.p3.wbar() * self.dmu[mu] * self.p4.w())
                    * (self.p2.wbar() * self.dml[mu] * self.p1.w())
                    for mu in range(0, 4)
                ]
            )
        )

    def xs(self, phi, theta):
        """
        Return the cross-section in m^2 for a given phi and theta.
        """
        from math import sqrt, cos, sin

        ct = cos(theta)
        st = sin(theta)
        q = sqrt(self.p1.p[0] ** 2 - self.p3.data.mass**2)
        p = sqrt(sum([self.p1.p[mu] ** 2 for mu in range(1, 4)]))
        self.p3.p[0] = self.p1.p[0]
        self.p3.p[1] = q * st * cos(phi)
        self.p3.p[2] = q * st * sin(phi)
        self.p3.p[3] = q * ct
        self.p4.p = ~self.p3.p
        me = self.me()
        try:
            me2 = me.real**2 + me.imag**2
        except:
            me2 = me**2
        return self.xspre * me2 * q / p * st / (self.p1.p[0] + self.p2.p[0]) ** 2
```

Exercise: helicity cross-section

Using the `Integrator` and `Annihilate` classes, calculate the cross-section for each helicity

configuration for the process $e^-e^+ \to \mu^-\mu^+$. Give the electron momentum of 100 GeV, and the positron -100 GeV along the z direction.

```
from math import pi

# Create the momenta.
p1 = FourVector(-1.0, 0.0, 0.0, 100)
p2 = FourVector(-1.0, 0.0, 0.0, -100)
p3 = FourVector(0.0, 0.0, 0.0, 0.0)
p4 = FourVector(0.0, 0.0, 0.0, 0.0)

# Create the particles.
pp1 = Particle(pdb["e-"], p1, 1)
pp2 = Particle(pdb["e+"], p2, 1)
pp3 = Particle(pdb["mu-"], p3, 1)
pp4 = Particle(pdb["mu+"], p4, 1)

# Create the annihilation object and integrator.
a = Annihilate(pp1, pp2, pp3, pp4)
i = Integrator(rng, a.xs, 0.0, 2 * pi, 0, pi)

# Loop over the helicities and integrate.
for h1 in [-1, 1]:
    pp1.h = h1
    for h2 in [-1, 1]:
        pp2.h = h2
        for h3 in [-1, 1]:
            pp3.h = h3
            for h4 in [-1, 1]:
                pp4.h = h4
                # Print the result.
                print("%2i %2i %2i %2i %8.1e" % (h1, h2, h3, h4, i.mc(1000) / 1e-31))
```

```
-1 -1 -1 -1  1.6e-26
-1 -1 -1  1  3.0e-20
-1 -1  1 -1  2.9e-20
-1 -1  1  1  1.6e-26
-1  1 -1 -1  1.3e-15
-1  1 -1  1  2.2e-09
-1  1  1 -1  2.2e-09
-1  1  1  1  1.2e-15
 1 -1 -1 -1  1.2e-15
 1 -1 -1  1  2.3e-09
 1 -1  1 -1  2.1e-09
 1 -1  1  1  1.2e-15
 1  1 -1 -1  1.6e-26
 1  1 -1  1  2.8e-20
 1  1  1 -1  2.9e-20
 1  1  1  1  1.6e-26
```

Comparisons with Pythia

Now that we have a cross-section calculated out, it is useful to compare this against Pythia to check our result. To do that, we need the total cross-section averaged over initial helicity configurations and summed over final.

```
def xs_total(rng, p):
    """
    Integrate the e+ e- -> mu+ mu- cross-section, averaged over initial
    helicities, and summed over final helicities, provided a given
    momentum for the e+ and e-. Units are in millibarn.

    rng: random number generator.
    p:   e- momentum along the z axis.
    """
    from math import pi

    # Create the momenta.
    p1 = FourVector(-1.0, 0.0, 0, p)
    p2 = FourVector(-1.0, 0.0, 0, -p)
    p3 = FourVector(0.0, 0.0, 0.0, 0.0)
    p4 = FourVector(0.0, 0.0, 0.0, 0.0)

    # Create the particles.
    pp1 = Particle(pdb["e-"], p1, 1)
    pp2 = Particle(pdb["e+"], p2, 1)
    pp3 = Particle(pdb["mu-"], p3, 1)
    pp4 = Particle(pdb["mu+"], p4, 1)

    # Create the process and integrator.
    a = Annihilate(pp1, pp2, pp3, pp4)
    i = Integrator(rng, a.xs, 0.0, 2 * pi, 0, pi)

    # Loop over the helicities and sum.
    total = 0
    for h3 in [-1, 1]:
        pp3.h = h3
        for h4 in [-1, 1]:
            pp4.h = h4
            xs = []
            for h1 in [-1, 1]:
                pp1.h = h1
                for h2 in [-1, 1]:
                    pp2.h = h2
                    # Convert from m^2 to mb.
                    xs += [i.mc(1000) / 1e-31]
            total += sum(xs) / len(xs)
    return total
```

We also need to configure Pythia to match the process we are calculating, as well as the constants that we are using.

```
def xs_pythia(p):
    """
    Calculate the e+ e- -> mu+ mu- cross-section from Pythia in mb.

    p: e- momentum along the z axis.
    """
    # Create a Pythia object.
    py = pythia8.Pythia("", False)

    # Set printing to a minimum.
    py.readString("Print:quiet = on")
    # Set the beam types.
    py.readString("Beams:idA = 11")
    py.readString("Beams:idB = -11")
    # Set the beam specification by momentum.
    py.readString("Beams:frameType = 3")
    py.readString("Beams:pzA = %r" % p)
    py.readString("Beams:pzB = -%r" % p)
    # Turn off any PDF for the lepton.
    py.readString("PDF:lepton = off")
    # Turn off everything after the hard process.
    py.readString("PartonLevel:all = off")
    # Don't run alpha.
    py.readString("SigmaProcess:alphaEMorder = 0")
    # Set alpha to what we are using. We have to force it.
    py.settings.parm("StandardModel:alphaEM0", 1.0 / 137.0, True)
    # Turn on our process.
    py.readString("WeakSingleBoson:ffbar2ffbar(s:gm) = on")

    # Initialize Pythia and generate.
    py.init()
    acc = 0
    for i in range(0, 10000):
        py.next()
        # Check we have a muon.
        if py.process[5].idAbs() == 13:
            acc += 1
    # Return the cross-section.
    return acc / py.infoPython().nAccepted() * py.infoPython().sigmaGen()
```

Exercise: compare with Pythia

For e^- and e^+ momementa of $5, 10, 50, 100$, and 1000 GeV compare our cross-section to that computed by Pythia.

```
for p in [5.0, 10.0, 50.0, 100.0, 1000.0]:
    print(f"xs_total  ({int(p):6d} GeV) = {xs_total(rng, p):.2e} mb")
    print(f"xs_pythia ({int(p):6d} GeV) = {xs_total(rng, p):.2e} mb")
```

```
xs_total  (     5 GeV) = 8.73e-07 mb
xs_pythia (     5 GeV) = 8.61e-07 mb
xs_total  (    10 GeV) = 2.18e-07 mb
xs_pythia (    10 GeV) = 2.19e-07 mb
xs_total  (    50 GeV) = 8.63e-09 mb
xs_pythia (    50 GeV) = 8.79e-09 mb
xs_total  (   100 GeV) = 2.14e-09 mb
xs_pythia (   100 GeV) = 2.17e-09 mb
xs_total  (  1000 GeV) = 2.19e-11 mb
xs_pythia (  1000 GeV) = 2.16e-11 mb
```