Hazma

Release 1.0

Jun 17, 2020

Contents:

CHAPTER 1

Installation

hazma was developed for python3. Before installing hazma, the user needs to install several well-established python packages: cython, scipy, numpy, and scikit-image. Theses are easily installed by using PyPI. If the user has PyPI installed on their system, then these packages can be installed using:

```
pip install cython, scipy, numpy, scikit-image, matplotlib
```

hazma can be installed in the same way, using:

```
pip install hazma
```

This will download a tarball from the PyPI repository, compile all the c-code and install hazma on the system. Alternatively, the user can install hazma by downloading the package from Hazma repo. Once downloaded, navigate to the package directory using the command line and run either:

```
pip install .
```

or:

```
python setup.py install
```

Note that since hazma makes extensive usage of the package cython, the user will need to have a c and c++ compiler installed on their system (for example gcc and g++ on unix-like systems or Microsoft Visual Studios 2015 or later on Windows). For more information, see the Cython installation guide.

CHAPTER 2

Usage

The user has several options for tapping into the resources provided by `hazma`. The easiest is to use one of the built-in simplified models, where a user only needs to specify the parameters of the model. If the user is working with a model which specializes on one of the simplified models, they can define their own class and inherit from one of the simplified models, obtaining all of the functionality of the built in models (such as final state radiation (FSR) spectra, cross sections, mediator decay widths, etc.) while supplying the user with a simpler, more specialized interface to the underlying models. For a detailed explanations of how these two options are done, see the *basic usage* section below. Another option is for the user to define their own model. To do this, they need to define a class which contains functions for the gamma-ray and positron spectra, as well as the annihilation cross sections and branching fractions. In the *advanced usage* section, we provide a detailed example for this case.

2.1 Basic Usage

2.1.1 Using Simplified Models

Here we give a compact overview of how to use the built in simplified models in `hazma`. All the models built into `hazma` have identical interfaces. The only difference in the interfaces is the parameters which need to be specified for the particular model being used. Thus, we will only show the usage of one of the models. The others can be used in an identical fashion. The example we will use is the `KineticMixing` model. To create a `KineticMixing` model, we use:

```
# Import the model
>>> from hazma.vector_mediator import KineticMixing
# Specify the parameters
>>> params = {'mx': 250.0, 'mv': 1e6, 'gvxx': 1.0, 'eps': 1e-3}
# Create KineticMixing object
>>> km = KineticMixing(**params)
```

Here we have created a model with a dark matter fermion of mass 250 MeV, a vector mediator which mixes with the standard model with a mass of 1 TeV. We set the coupling of the vector mediator to the dark matter, $g_{V\chi} = 1$ and set the kinetic mixing parameter $\epsilon = 10^{-3}$. To list all of the available final states for which the dark matter can annihilate into, we use:

```
>>> km.list_annihilation_final_states()
['mu mu', 'e e', 'pi pi', 'pi0 g', 'pi0 v', 'v v']
```

This tells us that we can potentially annihilate through: $\bar{\chi}\chi \to \mu^+\mu^-, e^+e^-, \pi^+\pi^-, \pi^0\gamma, \pi^0 0V$ or VV (the two-mediator final state). However, which of these final states is actually available depends on the center of mass energy. We can see this fact by looking at the annihilation cross sections or branching fractions, which can be computed using:

```
>>> cme = 2.0 * km.mx * (1.0 + 0.5 * 1e-6)
>>> km.annihilation_cross_sections(cme)
{'mu mu': 8.94839775021393e-25,
 'e e': 9.064036692829845e-25,
 'pi pi': 1.2940469635262499e-25,
 'pi0 g': 5.206158864833925e-29,
 'pi0 v': 0.0,
 'v v': 0.0,
 'total': 1.9307002022456507e-24}
>>> km.annihilation_branching_fractions(cme)
{'mu mu': 0.46347940191883763,
 'e e': 0.4694688839980031,
 'pi pi': 0.06702474894968717,
 'pi0 g': 2.6965133472190545e-05,
 'pi0 v': 0.0,
 'v v': 0.0}
```

Here we have chosen a realistic center of mass energy for dark matter in our galaxy, which as a velocity dispersion of $\sigma_v \sim 10^{-3}$. We can that the VV final state is unavailable, as it should be since the vector mediator mass is too heavy. In this theory, the vector mediator can decay. If we would like to know the decay width and the partial widths, we can use:

```
>>> km.partial_widths()
{'pi pi': 0.0018242846671063036,
 'pi0 g': 2.1037425397685694,
 'x x': 79577.47154594581,
 'e e': 0.007297139521307648,
 'mu mu': 0.007297139521307642,
 'total': 79579.5917070493}
```

If we would like to know the gamma-ray spectrum from dark matter annihilations, we can use:

```
>>> photon_energies = np.array([cme/4])
>>> km.spectra(photon_energies, cme)
{'mu mu': array([2.94759389e-05]),
 'e e': array([0.00013171]),
 'pi pi': array([2.20142244e-06]),
 'pi0 g': array([2.29931655e-07]),
 'pi0 v': array([0.]),
 'v v': array([0.]),
 'total': array([0.00016362])}
```

Note that we only used a single photon energy because of display purposes, but in general the user can specify any number of photon energies. If the user would like access to the underlying spectrum functions so they can call them repeatedly, they can use:

```
>>> spec_funs = km.spectrum_functions()
>>> spec_funs['mu mu'](photon_energies, cme)
[6.35970849e-05]
>>> mumu_bf = km.annihilation_branching_fractions(cme)['mu mu']
```

(continues on next page)

(continued from previous page)

```
>>> mumu_bf * spec_funs['mu mu'](photon_energies, cme)
[2.94759389e-05]
```

Notice that the direct call to the spectrum function for $\bar{\chi}\chi \to \mu^+\mu^-$ doesn't given the same result as `km.spectra(photon_energies, cme)['mu mu']`. This is because the branching fractions are not applied for the `spec_funs = km.spectrum_funcs()`. If the user doesn't care about the underlying components of the gamma-ray spectra, the can simply call:

```
>>> km.total_spectrum(photon_energies, cme)
array([0.00016362])
```

to get the total gamma-ray spectrum. The reader may have caught the fact that there is a gamma-ray line in the spectrum for $\bar{\chi}\chi \to \pi^0\gamma$. To get the location of this monochromatic gamma-ray line, the user can run:

```
>>> km.gamma_ray_lines(cme)
{'pi0 g': {'energy': 231.78145156177675, 'bf': 2.6965133472190545e-05}}
```

This tells us the process which produces the line, the location of the line and the branching fraction for the process. We don't include the line in the total spectrum since the line produces a Dirac-delta function. In order to get a realistic spectrum including the line, we need to convolve the gamma-ray spectrum with an energy resolution. This can be achieved using:

```
>>> min_photon_energy = 1e-3
>>> max_photon_energy = cme
>>> energy_resolution = lambda photon_energy : 1.0
>>> number_points = 1000
>>> spec = km.total_conv_spectrum_fn(min_photon_energy, max_photon_energy,
...                                  cme, energy_resolution, number_points)
>>> spec(cme / 4)  # compute the spectrum at a photon energy of `cme/4`
array(0.001718)
```

The `km.total_conv_spectrum_fn` computes and returns an interpolating function of the convolved function. An important thing to note here is that the `km.total_conv_spectrum_fn` takes in a function for the energy resolution. This allows the user to define the energy resolution to depend on the specific photon energy. Such a dependence is common for gamma-ray telescopes. Next we present the positron spectra. These have an identical interface to the gamma-ray spectra, so we only show how to call the functions and we suppress the output

```
>>> from hazma.parameters import electron_mass as me
>>> positron_energies = np.logspace(np.log10(me), np.log10(cme), num=100)
>>> km.positron_spectra(positron_energies, cme)
>>> km.positron_lines(cme)
>>> km.total_positron_spectrum(positron_energies, cme)
>>> dnde_pos = km.total_conv_positron_spectrum_fn(min(positron_energies),
...                                               max(positron_energies),
...                                               cme,
...                                               energy_resolution,
...                                               number_points)
```

The last thing that we would like to demonstrate is how to compute limits. In order to compute the limits on the annihilation cross section of a model from a gamma-ray telescope, say EGRET, we can use:

```
>>> from hazma.gamma_ray_parameters import egret_diffuse
# Choose DM masses from half the electron mass to 250 MeV
>>> mxs = np.linspace(me/2., 250., num=10)
# Compute limits from e-ASTROGAM
```

(continues on next page)

(continued from previous page)

```
>>> limits = np.zeros(len(mxs), dtype=float)
>>> for i, mx in enumerate(mxs):
...     km.mx = mx
...     limits[i] = km.binned_limit(egret_diffuse)
```

Similarly, if we would like to set constraints using e-ASTROGAM, one can use:

```
# Import target and background model for the e-ASTROGAM telescope
>>> from hazma.gamma_ray_parameters import gc_target, gc_bg_model
# Choose DM masses from half the electron mass to 250 MeV
>>> mxs = np.linspace(me/2., 250., num=10)
# Compute limits from e-ASTROGAM
>>> limits = np.zeros(len(mxs), dtype=float)
>>> for i, mx in enumerate(mxs):
...     km.mx = mx
...     limits[i] = km.unbinned_limit(target_params=gc_target,
...                                   bg_model=gc_bg_model)
```

2.1.2 Subclassing the Simplified Models

The user might not be interested in the generic simplified models built into `hazma`, but instead a more specialized model. In this case, it makes sense for the user to subclass one of the simplified models (i.e. create a class which inherits from one of the simplified models.) As and example, we illustrate how to do this with the Higgs-portal model (of course this model is already built into `hazma`, but it works nicely as an example.) Recall that the full set of parameters for the scalar mediator model are:

1. m_χ: dark matter mass,
2. m_S: scalar mediator mass,
3. $g_{S\chi}$: coupling of scalar mediator to dark matter,
4. g_{Sf}: coupling of scalar mediator to standard model fermions,
5. g_{SG}: effective coupling of scalar mediator to gluons,
6. g_{SF}: effective coupling of scalar mediator to photons and
7. Λ: cut-off scale for the effective interactions.

In the case of the Higgs-portal model, the scalar mediator talks to the standard model only through the Higgs boson, i.e. it mixes with the Higgs. Therefore, the scalar mediator inherits its interactions with the standard model fermions, gluons and photon through the Higgs. In the Higgs-portal model, the relevant parameters are:

1. m_χ: dark matter mass,
2. m_S: scalar mediator mass,
3. $g_{S\chi}$: coupling of scalar mediator to dark matter,
4. $\sin\theta$: the mixing angle between the scalar mediator and the Higgs,

The remaining parameters can be deduced from these using:

$$g_{Sf} = \sin\theta, g_{SG} = 3\sin\theta, g_{SF} = -\frac{5}{6}\sin\theta, \Lambda = v_h.$$

Below, we construct a class which subclasses the scalar mediator class to implement the Higgs-portal model.

```python
from hazma.scalar_mediator import ScalarMediator
from hazma.parameters import vh

class HiggsPortal(ScalarMediator):
    def __init__(self, mx, ms, gsxx, stheta):
        self._lam = vh
        self._stheta = stheta
        super(HiggsPortal, self).__init__(mx, ms, gsxx, stheta, 3.*stheta,
                                          -5.*stheta/6., vh)

    @property
    def stheta(self):
        return self._stheta

    @stheta.setter
    def stheta(self, stheta):
        self._stheta = stheta
        self.gsff = stheta
        self.gsGG = 3. * stheta
        self.gsFF = - 5. * stheta / 6.

    # Hide underlying properties' setters
    @ScalarMediator.gsff.setter
    def gsff(self, gsff):
        raise AttributeError("Cannot set gsff")

    @ScalarMediator.gsGG.setter
    def gsGG(self, gsGG):
        raise AttributeError("Cannot set gsGG")

    @ScalarMediator.gsFF.setter
    def gsFF(self, gsFF):
        raise AttributeError("Cannot set gsFF")
```

There are a couple things to note about our above implementation. First, our model only takes in m_χ, m_S, $g_{S\chi}$ and $\sin\theta$, as desired. But the underlying model, i.e. the `ScalarMediator` model only knows about m_χ, m_S, $g_{S\chi}$, g_{Sf}, g_{SG}, g_{SF} and Λ. So if we update $\sin\theta$, we additionally need to update the underlying parameters, g_{Sf}, g_{SG}, g_{SF} and Λ. The easiest way to do this is using getters and setters by defining $\sin\theta$ to be a `property` through the `@property` decorator. Then every time we update $\sin\theta$, we can also update the underlying parameters. The second thing to note is that we want to make sure we don't accidentally change the underlying parameters directly, since in this model, they are only defined through $\sin\theta$. We an ensure that we cannot change the underlying parameters directly by overriding the getters and setters for `gsff`, `gsGG` and `gsGG` and raising an error if we try to change them. This isn't strictly necessary (as long as the user is careful), but can help avoid confusing behavior.

2.2 Advanced Usage

2.2.1 Adding New Gamma-Ray Experiments

Currently `hazma` only includes information for producing projected unbinned limits with e-ASTROGAM, using the dwarf Draco or inner $10^\circ \times 10^\circ$ region of the Milky Way as a target. Adding new detectors and target regions is straightforward. A detector is characterized by the effective area $A_{\text{eff}}(E)$, the energy resolution $\epsilon(E)$ and observation time T_{obs}. In `hazma`, the first two can be any callables (functions) and the third must be a float. The region of interest is defined by a `TargetParams` object, which can be instantiated with:

```
>>> from hazma.gamma_ray_parameters import TargetParams
>>> tp = TargetParams(J=1e29, dOmega=0.1)
```

The background model should be packaged in an object of type `BackgroundModel`. This light-weight class has a function `dPhi_dEdOmega()` for computing the differential photon flux per solid angle (in $\mathrm{MeV}^{-1}\mathrm{sr}$) and an attribute `e_range` specifying the energy range over which the model is valid (in MeV). New background models are defined by passing these two the

```
>>> from hazma.background_model import BackgroundModel
>>> bg = BackgroundModel(e_range=[0.5, 1e4],
...                      dPhi_dEdOmega=lambda e: 2.7e-3 / e**2)
```

Gamma-ray observation information from Fermi-LAT, EGRET and COMPTEL is included with `hazma`, and other observations can be added using the container class `FluxMeasurement`. The initializer requires:

1. The name of a CSV file containing gamma-ray observations. The file's columns must contain:
    1. Lower bin edge (MeV)
    2. Upper bin edge (MeV)
    3. $E^n d^2\Phi/dE\,d\Omega$ (in $\mathrm{MeV}^{n-1}\mathrm{cm}^{-2}\mathrm{s}^{-1}\mathrm{sr}^{-1}$)
    4. Upper error bar (in $\mathrm{MeV}^{n-1}\mathrm{cm}^{-2}\mathrm{s}^{-1}\mathrm{sr}^{-1}$)
    5. Lower error bar (in $\mathrm{MeV}^{n-1}\mathrm{cm}^{-2}\mathrm{s}^{-1}\mathrm{sr}^{-1}$)

    Note that the error bar values are their y-coordinates, not their relative distances from the central flux.
2. The detector's energy resolution function.
3. A `TargetParams` object for the target region.

For example, a CSV file `obs.csv` containing observations

lower bin	upper bin	$E^n d^2\Phi/dE\,d\Omega$	upper error	lower error
150.	275.0	0.0040	0.0043	0.0038
650.	900.0	0.0035	0.0043	0.003

with $n = 2$ for an instrument with energy resolution $\epsilon(E) = 0.05$ observing the target region `tp` defined above can be loaded using[1]:

```
>>> from hazma.flux_measurement import FluxMeasurement
>>> obs = FluxMeasurement("obs.dat", lambda e: 0.05, tp)
```

The attributes of the `FluxMeasurement` store all of the provide information, with the E^n prefactor removed from the flux and error bars, and the errors converted from the positions of the error bars to their sizes. These are used internally by the `Theory.binned_limit()` method, and can be accessed as follows:

```
>>> obs.e_lows, obs.e_highs
(array([150., 650.]), array([275., 900.]))
>>> obs.target
<hazma.gamma_ray_parameters.TargetParams at 0x1c1bbbafd0>
```

(continues on next page)

[1] If the CSV containing the observations uses a different power of E than $n = 2$, this can be specified using the `power` keyword argument to the initializer for `FluxMeasurement`

(continued from previous page)

```
>>> obs.fluxes
array([8.85813149e-08, 5.82726327e-09])
>>> obs.upper_errors
array([6.64359862e-09, 1.33194589e-09])
>>> obs.lower_errors
array([4.42906574e-09, 8.32466181e-10])
>>> obs.energy_res(10.)
0.05
```

2.2.2 User-Defined Models

In this subsection, we demonstrate how to implement new models in Hazma. A notebook containing all th.. code in this appendix can be downloaded from GitHub HazmaExample. The model we will consider is an effective field theory with a Dirac fermion DM particle which talks to neutral and charged pions through gauge-invariant dimension-5 operators. The Lagrangian for this model is:

$$\mathcal{L} \supset \frac{c_1}{\Lambda}\overline{\chi}\chi\pi^+\pi^- + \frac{c_2}{\Lambda}\overline{\chi}\chi\pi^0\pi^0$$

where c_1, c_2 are dimensionless Wilson coefficients and Λ is the cut-off scale of the theory. In order to implement this model in Hazma, we need to compute the annihilation cross sections and the FSR spectra. The annihilation channels for this model are simply $\bar{\chi}\chi \to \pi^0\pi^0$ and $\bar{\chi}\chi \to \pi^+\pi^-$. The computations for the cross sections are straight forward and yield:

$$\sigma(\bar{\chi}\chi \to \pi^+\pi^-) = \frac{c_1^2\sqrt{1-4\mu_\pi^2}\sqrt{1-4\mu_\chi^2}}{32\pi\Lambda^2}$$
$$\sigma(\bar{\chi}\chi \to \pi^0\pi^0) = \frac{c_2^2\sqrt{1-4\mu_{\pi^0}^2}\sqrt{1-4\mu_\chi^2}}{8\pi\Lambda^2}$$

where Q is the center of mass energy, $\mu_\chi = m_\chi/Q$, $\mu_\pi = m_{\pi^\pm}/Q$ and $\mu_{\pi^0} = m_{\pi^0}/Q$. In addition to the cross sections, we need the FSR spectrum for $\overline{\chi}\chi \to \pi^+\pi^-\gamma$. This is:

$$\frac{dN(\bar{\chi}\chi \to \pi^+\pi^-\gamma)}{dE_\gamma} = \frac{\alpha\left(2f(x) - 2\left(1-x-2\mu_\pi^2\right)\log\left(\frac{1-x-f(x)}{1-x+f(x)}\right)\right)}{\pi\sqrt{1-4\mu_\pi^2}x}$$

where

$$f(x) = \sqrt{1-x}\sqrt{1-x-4\mu_\pi^2}$$

We are now ready to set up the Hazma model. For `hazma` to work properly, we will need to define the following functions in our model:

1. `annihilation_cross_section_funcs()`: A function returning a `dict` of the annihilation cross sections functions, each of which take a center of mass energy.
2. `spectrum_funcs()`: A function returning a `dict` of functions which take photon energies and a center of mass energy and return the gamma-ray spectrum contribution from each final state.
3. `gamma_ray_lines(e_cm)`: A function returning a `dict` of the gamma-ray lines for a given center of mass energy.
4. `positron_spectrum_funcs()`: Like `spectrum_funcs()`, but for positron spectra.
5. `positron_lines(e_cm)`: A function returning a `dict` of the electron/positron lines for a center of mass energy.

We find it easiest to place all of these components is modular classes and then combine all the individual classes into a master class representing our model. Before we begin writing the classes, we will need a few helper functions and constants from `hazma`:

```
import numpy as np # NumPy is heavily used
import matplotlib.pyplot as plt # Plotting utilities
# neutral and charged pion masses
from hazma.parameters import neutral_pion_mass as mpi0
from hazma.parameters import charged_pion_mass as mpi
from hazma.parameters import qe # Electric charge
# Positron spectra for neutral and charged pions
from hazma.positron_spectra import charged_pion as pspec_charged_pion
# Deay spectra for neutral and charged pions
from hazma.decay import neutral_pion, charged_pion
# The `Theory` class which we will ultimately inherit from
from hazma.theory import Theory
```

Now, we implement a cross section class:

```
class HazmaExampleCrossSection:
    def sigma_xx_to_pipi(self, Q):
        mupi = mpi / Q
        mux = self.mx / Q

        if Q > 2 * self.mx and Q > 2 * mpi:
            sigma = (self.c1**2 * np.sqrt(1 - 4 * mupi**2) *
                     np.sqrt(1 - 4 * mux**2)**2 /
                     (32.0 * self.lam**2 * np.pi))
        else:
            sigma = 0.0

        return sigma

    def sigma_xx_to_pi0pi0(self, Q):
        mupi0 = mpi0 / Q
        mux = self.mx / Q

        if Q > 2 * self.mx and Q > 2 * mpi0:
            sigma = (self.c2**2 * np.sqrt(1 - 4 * mux**2) *
                     np.sqrt(1 - 4 * mupi0**2) /
                     (8.0 * self.lam**2 * np.pi))
        else:
            sigma = 0.0

        return sigma

    def annihilation_cross_section_funcs(self):
        return {'pi0 pi0': self.sigma_xx_to_pi0pi0,
                'pi pi': self.sigma_xx_to_pipi}
```

The key function is `annihilation_cross_sections`, which is required to be implemented by `hazma`. Next, we implement the spectrum functions which will produce the FSR and decay spectra:

```
class HazmaExampleSpectra:
    def dnde_pi0pi0(self, e_gams, e_cm):
        return 2.0 * neutral_pion(e_gams, e_cm / 2.0)

    def __dnde_xx_to_pipig(self, e_gam, Q):
```

(continues on next page)

(continued from previous page)

```
        # Unvectorized function for computing FSR spectrum
        mupi = mpi / Q
        mux = self.mx / Q
        x = 2.0 * e_gam / Q
        if 0.0 < x and x < 1. - 4. * mupi**2:
            dnde = ((qe**2 * (2 * np.sqrt(1 - x) * np.sqrt(1 - 4*mupi**2 - x) +
                          (-1 + 2 * mupi**2 + x) *
                          np.log((-1 + np.sqrt(1 - x) * np.sqrt(1 - 4*mupi**2 - x) +␣
↪x)**2/
                                 (1 + np.sqrt(1 - x)*np.sqrt(1 - 4*mupi**2 - x) -␣
↪x)**2)))/
                    (Q * 2.0 * np.sqrt(1 - 4 * mupi**2) * np.pi**2 * x))
        else:
            dnde = 0

        return dnde

    def dnde_pipi(self, e_gams, e_cm):
        return (np.vectorize(self.__dnde_xx_to_pipig)(e_gams, e_cm) +
                2. * charged_pion(e_gams, e_cm / 2.0))

    def spectrum_funcs(self):
        return {'pi0 pi0':  self.dnde_pi0pi0,
                'pi pi':  self.dnde_pipi}

    def gamma_ray_lines(self, e_cm):
        return {}
```

Note the the second __dnde_xx_to_pipig is an unvectorized helper function, which is not to be used directly. Next we implement the positron spectra:

```
class HazmaExamplePositronSpectra:
    def dnde_pos_pipi(self, e_ps, e_cm):
        return pspec_charged_pion(e_ps, e_cm / 2.)

    def positron_spectrum_funcs(self):
        return {"pi pi": self.dnde_pos_pipi}

    def positron_lines(self, e_cm):
        return {}
```

Lastly, we group all of these classes into a master class and we're done:

```
class HazmaExample(HazmaExampleCrossSection,
                   HazmaExamplePositronSpectra,
                   HazmaExampleSpectra,
                   Theory):
    # Model parameters are DM mass: mx,
    # Wilson coefficients: c1, c2 and
    # cutoff scale: lam
    def __init__(self, mx, c1, c2, lam):
        self.mx = mx
        self.c1 = c1
        self.c2 = c2
        self.lam = lam
```

(continues on next page)

(continued from previous page)

```
    @staticmethod
    def list_annihilation_final_states():
        return ['pi pi', 'pi0 pi0']
```

Now we can easily compute gamma-ray spectra, positron spectra and limit on our new model from gamma-ray telescopes. To implement our new model with $m_\chi = 200$ MeV, $c_1 = c_2 = 1$ and $\Lambda = 100$ GeV, we can use:

```
>>> model = HazmaExample(200.0, 1.0, 1.0, 100e3)
```

To compute a gamma-ray spectrum:

```
# Photon energies from 1 keV to 1 GeV
>>> egams = np.logspace(-3.0, 3.0, num=150)
# Assume the DM is moving with a velocity of 10^-3
>>> vdm = 1e-3
# Compute CM energy assuming the above velocity
>>> Q = 2.0 * model.mx * (1 + 0.5 * vdm**2)
# Compute spectra
>>> spectra = model.spectra(egams, Q)
```

Then we can plot the spectra using:

```
>>> plt.figure(dpi=100)
>>> for key, val in spectra.items():
...     plt.plot(egams, val, label=key)
>>> plt.xlabel(r'$E_{\gamma} (\mathrm{MeV})$', fontsize=16)
>>> plt.ylabel(r'$\frac{dN}{dE_{\gamma}} (\mathrm{MeV}^{-1})$', fontsize=16)
>>> plt.xscale('log')
>>> plt.yscale('log')
>>> plt.legend()
```

Additionally, we can compute limits on the thermally-averaged annihilation cross section of our model for various DM masses using

```
# Import target and background model for the E-Astrogam telescope
>>> from hazma.gamma_ray_parameters import gc_target, gc_bg_model
# Choose DM masses from half the pion mass to 250 MeV
>>> mxs = np.linspace(mpi/2., 250., num=100)
# Compute limits from E-Astrogam
>>> limits = np.zeros(len(mxs), dtype=float)
>>> for i, mx in enumerate(mxs):
...     model.mx = mx
...     limits[i] = model.unbinned_limit(target_params=gc_target,
...                                      bg_model=gc_bg_model)
```

CHAPTER 3

Models

3.1 Overview

This page contains the documentation for the models built into `hazma`.

3.1.1 Scalar mediator

Overview

The `scalar_mediator` module contains three models for which dark matter interacts with the Standard Model through a scalar mediator. For energies $\mu \gg 1$ GeV, the interaction Lagrangian can for this theory is given by:

$$\mathcal{L}_{\mathrm{Int}(S)} = -S\left(g_{S\chi} + g_{Sf}\sum_{f}\frac{y_f}{\sqrt{2}}\bar{f}f\right) + \frac{S}{\Lambda}\left(g_{SG}\frac{\alpha_{\mathrm{EM}}}{4\pi}F_{\mu\nu}F^{\mu\nu} + g_{SF}\frac{\alpha_s}{4\pi}G^{a}_{\mu\nu}G^{a\mu\nu}\right)$$

where the y_f's are the Yukawa couplings for the Standard Model fermions and $F_{\mu\nu}$ and $G^{a}_{\mu\nu}$ are the field strength tensors for the photon and gluons (we will describe the remaining parameters below.) For energies $\mu < 1$ GeV, the quarks and gluons confine into mesons and baryons. In order to describe the interactions of the scalar mediator to the mesons, we use Chiral Perturbation theory. The interaction Lagrangian becomes:

$$\begin{aligned}\mathcal{L}_{\text{Int}(S)} = &\frac{2g_{SG}}{9\Lambda}S\left[(\partial_\mu\pi^0)(\partial^\mu\pi^0) + 2(\partial_\mu\pi^+)(\partial^\mu\pi^-)\right]\\ &+\frac{4ieg_{SG}}{9\Lambda}SA^\mu\left[\pi^-(\partial_\mu\pi^+) - \pi^+(\partial_\mu\pi^-)\right]\\ &-\frac{B(m_u+m_d)}{6}\left(\frac{3g_{Sf}}{v_h} + \frac{2g_{SG}}{3\Lambda}\right)S\left[(\pi^0)^2 + 2\pi^+\pi^-\right]\\ &+\frac{B(m_u+m_d)g_{SG}}{81\Lambda}\left(\frac{2g_{SG}}{\Lambda} - \frac{9g_{Sf}}{v_h}\right)S^2\left[(\pi^0)^2 + 2\pi^+\pi^-\right]\\ &+\frac{4e^2g_{SF}}{9\Lambda}S\pi^+\pi^-A_\mu A^\mu\\ &-g_{S\chi}S\bar{\chi}\chi - g_{Sf}S\sum_{\ell=e,\mu}\frac{y_\ell}{\sqrt{2}}\bar{\ell}\ell.\end{aligned}$$

where $B \approx 2800$ MeV, $v_h = 246$ GeV and the model parameters are:

1. m_χ: dark matter mass,
2. m_S: scalar mediator mass,
3. $g_{S\chi}$: coupling of scalar mediator to dark matter,
4. g_{Sf}: coupling of scalar mediator to standard model fermions,
5. g_{SG}: effective coupling of scalar mediator to gluons,
6. g_{SF}: effective coupling of scalar mediator to photons and
7. Λ: cut-off scale for the effective interactions.

In addition to the generic scalar mediator mode, `hazma` also contains specialized models for realizations of the Higgs-portal and Heavy-quark theories. In the Higgs-portal model, we assume that the scalar mediator doesn't directly interact with the standard model particles aside from the Higgs. We assume the the scalar mixes with the Higgs and inherits all its interactions to the Standard model through the mixing. The Higgs-portal model contains the following parameters:

1. m_χ: dark matter mass,
2. m_S: scalar mediator mass,
3. $g_{S\chi}$: coupling of scalar mediator to dark matter,
4. $\sin\theta$: mixing angle between the scalar mediator and Higgs.

The generic couplings are obtained from these parameters through the following relationships:

$$g_{Sf} = \sin\theta, g_{SG} = 3\sin\theta, g_{SF} = -\frac{5}{6}\sin\theta, \Lambda = v_h.$$

The Heavy-quark model assumes that there exists a new heavy quark and that the scalar mediator only couples the the heavy quark. The parameters of this model are:

1. m_χ: dark matter mass,
2. m_S: scalar mediator mass,
3. $g_{S\chi}$: coupling of scalar mediator to dark matter,
4. g_{SQ}: coupling of scalar mediator to the heavy quark,
5. Q_Q: charge of the heavy quark,
6. m_Q: mass of the heavy quark.

The relationships between these parameters and the generic parameters are:

$$g_{SG} = g_{SQ}, g_{SF} = 2Q_Q^2 g_{SQ}, \Lambda = m_Q.$$

For details on how to uses these classes, see *Basic Usage*.

Classes

3.1.2 Vector mediator

Overview

The `vector_mediator` module contains two models for which dark matter interacts with the Standard Model through a vector mediator. For energies $\mu \gg 1$ GeV, the interaction Lagrangian can for this theory is given by:

$$\mathcal{L}_{\mathrm{Int}(V)} = V_\mu \left(g_{V\chi}\bar{\chi}\gamma^\mu\chi + \sum_f g_{Vf}\bar{f}\gamma^\mu f \right) - \frac{\epsilon}{2} V^{\mu\nu} F_{\mu\nu}.$$

where the ϵ is the kinetic mixing couplings for the Standard Model fermions and $F_{\mu\nu}$ is the field strength tensor for the photon (we will describe the remaining parameters below.) For energies $\mu < 1$ GeV, the quarks and gluons confine into mesons and baryons. In order to describe the interactions of the vector mediator to the mesons, we use Chiral Perturbation theory. The interaction Lagrangian becomes:

$$\begin{aligned}
\mathcal{L}_{\mathrm{Int}(V)} = &-i(g_{Vu} - g_{Vd})V^\mu \left(\pi^+\partial_\mu\pi^- - \pi^-\partial_\mu\pi^+\right) \\
&+ (g_{Vu} - g_{Vd})^2 V_\mu V^\mu \pi^+\pi^- \\
&+ 2e(Q_u - Q_d)(g_{Vu} - g_{Vd}) A_\mu V^\mu \pi^+\pi^- \\
&+ \frac{1}{8\pi^2 f_\pi}\epsilon^{\mu\nu\rho\sigma}(\partial_\mu\pi^0) \\
&\qquad \times \{e(2g_{Vu} + g_{Vd})\left[(\partial_\nu A_\rho)V_\sigma + (\partial_\nu V_\rho)A_\sigma\right] \\
&\qquad\qquad + 3(g_{Vu}^2 - g_{Vd}^2)(\partial_\nu V_\rho)V_\sigma\} \\
&+ V_\mu \left(g_{Ve}\bar{e}\gamma^\mu e + g_{V\mu}\bar{\mu}\gamma^\mu\mu\right)
\end{aligned}$$

where $f_\pi \approx 93$ MeV and the model parameters are:

1. m_χ: dark matter mass,
2. m_V: vector mediator mass,
3. $g_{V\chi}$: coupling of vector mediator to dark matter,
4. g_{Vq}: $(q = u, d, s)$ coupling of vector mediator to standard model quarks,
5. $g_{V\ell}$: $(\ell = e, \mu)$ coupling of vector mediator to standard model leptons,

In addition to the generic vector mediator model, `hazma` also contains specialized models for realization for a theory in which the vector mediator mixes with the Standard model photon. In the kinetic-mixing model, we assume that the vector mediator doesn't directly interact with the Standard model particles aside from the mixing with the photon. The vector then inherits its coupling to the charged Standard model fermions from the photon. The kinetic-mixing model contains the following parameters:

1. m_χ: dark matter mass,
2. m_V: scalar mediator mass,
3. $g_{V\chi}$: coupling of scalar mediator to dark matter,
4. ϵ: kinetic mixing parameter between the vector mediator and SM photon.

The generic couplings are obtained from these parameters through the following relationships:

$$g_{Vf} = \epsilon e Q_q$$

where $f = (u, d, s, e, \mu)$. For details on how to uses these classes, see *Basic Usage*.

Classes

CHAPTER 4

Gamma ray limits

4.1 Overview

`hazma` includes functionality for using existing gamma-ray data to constrain theories and for projecting the discovery reach of proposed gamma-ray detectors. For the first case, `hazma` defines a container class called `FluxMeasurement` for storing information about gamma-ray datasets, and `Theory` contains a method for using these to set limits. The second case is also handled by a method in `Theory` which takes arguments specifying various detector and target characteristics.

4.2 Limits from existing data

`Theory`.**`binned_limit`**(*measurement*, *n_sigma=2.0*)

Determines the limit on $< sigmav >$ from gamma-ray data.

We define a signal to be in conflict with the measured flux for bin i for an experiment if

$$\Phi_\chi^{(i)} > n_\sigma \sigma^{(i)} + \Phi^{(i)},$$

where $\Phi_\chi^{(i)}$ is the integrated flux due to DM annihilations for the bin, $\Phi^{(i)}$ is the measured flux in the bin, $\sigma^{(i)}$ is size of the upper error bar for the bin and $n_\sigma = 2$ is the significance. The overall limit on $\langle \sigma v \rangle$ is computed by minimizing over the limits determined for each bin.

Parameters

- **`measurement`** (`FluxMeasurement`) – Information about the flux measurement and target.
- **`n_sigma`** (*`float`*) – See the notes for this function.

Returns **<sigma v>_tot** – Largest allowed thermally averaged total cross section in cm^3 / s

Return type float

4.3 Discovery reach for upcoming detectors

`Theory.`**`unbinned_limit`**(*A_eff*, *energy_res*, *T_obs*, *target_params*, *bg_model*, *n_sigma=5.0*, *debug_msgs=False*)

Computes smallest-detectable value of <sigma v> for given target and experiment parameters.

We define a signal to be detectable if

$$N_S/\sqrt{N_B} \geq n_\sigma,$$

where N_S and N_B are the number of signal and background photons in the energy window of interest and n_σ is the significance in number of standard deviations. Note that $N_S \propto \langle \sigma v \rangle$. While the photon count statistics are properly taken to be Poissonian and using a confidence interval would be more rigorous, this procedure provides a good estimate and is simple to compute. The energy window is chosen to maximize N_S/sqrt(N_B).

Parameters

- **`A_eff`** (*`float -> float`*) – Effective area of experiment in cm^2 as a function of photon energy.
- **`energy_res`** (*`float -> float`*) – The detector's energy resolution ($\Delta E/E$) as a function of photon energy in MeV.
- **`T_obs`** (*`float`*) – Experiment's observation time in s
- **`target_params`** (`TargetParams`) – Object containing information about the observation target.
- **`bg_model`** (`BackgroundModel`) – Object representing a gamma ray background model.
- **`n_sigma`** (*`float`*) – Number of standard deviations the signal must be above the background to be considered detectable
- **`debug_msgs`** (*`bool`*) – If True, the energy window found by the optimizer will be printed.

Returns **<sigma v>** – Smallest-detectable thermally averaged total cross section in units of cm^3 / s.

Return type float

4.4 Classes, functions and constants

These data, functions and classes are relevant for setting constraints and projecting discovery reach.

`class` `hazma.flux_measurement.`**`FluxMeasurement`**(*obs_rf*, *energy_res*, *target*, *power=2*)

Container for all information about a completed gamma ray analysis.

`e_lows`

Lower edges of energy bins.

Type np.array

`e_highs`

Upper edges of energy bins.

Type np.array

`fluxes`

Flux measurements for each bin (MeV^-1 cm^-2 s^-1 sr^-1).

Type np.array

upper_errors
Size of upper error bars on flux measurements (MeV^-1 cm^-2 s^-1 sr^-1).

Type np.array

lower_errors
Size of lower error bars on flux measurements (MeV^-1 cm^-2 s^-1 sr^-1).

Type np.array

energy_res
Function returning energy resolution (Delta E / E) as a function of photon energy.

Type callable

target
Information about the target observed for this measurement.

Type *TargetParams*

__init__(*obs_rf*, *energy_res*, *target*, *power=2*)
Constructor.

Parameters

- **obs_rf** (*str*) – Name of file containing observation information. The columns of this file must be:

1. Lower bin edge (MeV)
2. Upper bin edge (MeV)
3. $E^2 d^2\Phi/dEd\Omega$ (MeV cm^-2 s^-1 sr^-1)
4. Upper error bar (MeV cm^-2 s^-1 sr^-1)
5. Lower error bar (MeV cm^-2 s^-1 sr^-1)

Note that the error bar values are their y-coordinates, not their relative distances from the central flux.

- **energy_res** (*callable*) – Energy resolution function.
- **target** (*TargetParams*) – The target of the analysis

class hazma.background_model.**BackgroundModel**(*e_range*, *dPhi_dEdOmega*)
Represents a gamma ray background model, which is required for computing projected limits for planned gamma-ray detectors.

Parameters

- **e_range** (*[float, float]*) – Minimum and maximum photon energies for which this model is valid, in MeV.
- **dPhi_dEdOmega** (*np.array*) – Background gamma ray flux (MeV^-1 sr^-1 m^-2 s^-1) as a function of photon energy (MeV). This function must be vectorized.

dPhi_dEdOmega(*es*)
Computes this background model's gamma ray flux.

Parameters **es** (*float or np.array*) – Photon energy/energies at which to compute

Returns **dPhi_dEdOmega** – Background gamma ray flux, in MeV^-1 sr^-1 m^-2 s^-1. For any energies outside of `self.e_range`, `np.nan` is returned.

Return type np.array

`hazma.gamma_ray_parameters.`**`energy_res_comptel`**(*e*)

COMPTEL energy resolution $\Delta E/E$.

Taken from ch. II, page 11.

`hazma.gamma_ray_parameters.`**`A_eff_comptel = <scipy.interpolate.interpolate.interp1d object>`**

COMPTEL effective area function

`hazma.gamma_ray_parameters.`**`comptel_diffuse = <hazma.flux_measurement.FluxMeasurement object`**

COMPTEL diffuse gamma-ray flux measurements

`hazma.gamma_ray_parameters.`**`energy_res_egret`**(*e*)

EGRET's energy resolution $\Delta E/E$.

This is the most optimistic value, taken from sec. 4.3.3.

`hazma.gamma_ray_parameters.`**`A_eff_egret = <scipy.interpolate.interpolate.interp1d object>`**

EGRET effective area function

`hazma.gamma_ray_parameters.`**`egret_diffuse = <hazma.flux_measurement.FluxMeasurement object>`**

EGRET diffuse gamma-ray flux measurements

`hazma.gamma_ray_parameters.`**`energy_res_fermi`**(*e*)

Fermi-LAT's energy resolution $\Delta E/E$.

This is the average of the most optimistic normal and 60deg off-axis values from fig. 18.

`hazma.gamma_ray_parameters.`**`A_eff_fermi = <scipy.interpolate.interpolate.interp1d object>`**

Fermi-LAT effective area function

`hazma.gamma_ray_parameters.`**`fermi_diffuse = <hazma.flux_measurement.FluxMeasurement object>`**

Fermi diffuse gamma-ray flux measurements

`hazma.gamma_ray_parameters.`**`energy_res_e_astrogam = <scipy.interpolate.interpolate.interp1d`**

e-ASTROGAM energy resolution function. From table 1 of the e-ASTROGAM whitebook.

`hazma.gamma_ray_parameters.`**`A_eff_e_astrogam = <scipy.interpolate.interpolate.interp1d obje`**

e-ASTROGAM effective area function

`class` `hazma.gamma_ray_parameters.`**`TargetParams`**(*J*, *dOmega*)

Container for information about a target region.

Currently implemented for the Draco dwarf galaxy and $10° imes 10°$ region around the galactic center, which can be imported using:

```
from hazma.gamma_ray_parameters import draco_params, gc_target
```

Parameters

- **`J`** (*`float`*) – J-factor in MeV^2 cm^-5
- **`dOmega`** (*`float`*) – Angular size in sr

`hazma.gamma_ray_parameters.`**`solid_angle`**(*l_max*, *b_min*, *b_max*)

Returns solid angle subtended for a rectangular target region centered on the galactic center.

Parameters

- **`l_max`** (*`float`*) – Maximum value of galactic longitude in deg. Note that l must lie in the interval $[-180, 180]$.
- **`b_max`** (*`b_min,`*) – Minimum and maximum values for b in deg. Note that b must lie in the interval $[-90, 90]$, with the equator at $b = 0$.

Returns **Omega** – Solid angle subtended by the region in sr.

Return type float

CHAPTER 5

CMB constraints

5.1 Overview

The `Theory` class contains functions for computing CMB limits and f_{eff} for dark matter models. Other useful constants and functions are also available.

5.2 Computing CMB limits

`Theory.`**`cmb_limit`**(*x_kd=0.0001*, *p_ann=3.5e-31*)

Computes the CMB limit on <sigma v>.

This is derived by requiring that

$$f_{\text{eff}}\langle\sigma v\rangle/m_\chi < p_{\text{ann}},$$

where f_{eff} is the efficiency with which dark matter annihilations around recombination inject energy into the plasma and p_{ann} is derived from CMB observations.

Parameters

- **`x_kd`** (*`float`*) – T_kd / m_x, where T_kd is the dark matter's kinetic decoupling temperature. This will be computed self-consistently in future versions of `hazma`.
- **`p_ann`** (*`float`*) – Constraint on energy release per DM annihilation in cm^3 s^-1 MeV^-1.

Returns **<sigma v>** – Upper bound on <sigma v>, in cm^3 s^-1.

Return type float

5.3 Functions and constants

`Theory.`**`f_eff`**(*x_kd=0.0001*)

Computes f_{eff} the efficiency with which dark matter annihilations around recombination inject energy into the thermal plasma.

`hazma.cmb.`**`p_ann_planck_temp_pol = 3.5e-31`**

Planck 2018 95% upper limit on p_ann from temperature + polarization measurements, in cm^3 s^-1 MeV^-1

`hazma.cmb.`**`p_ann_planck_temp_pol_lensing = 3.3e-31`**

Planck 2018 95% upper limit on p_ann from temperature + polarization + lensing measurements, in cm^3 s^-1 MeV^-1

`hazma.cmb.`**`p_ann_planck_temp_pol_lensing_bao = 3.2e-31`**

Planck 2018 95% upper limit on p_ann from temperature + polarization + lensing + BAO measurements, in cm^3 s^-1 MeV^-1

`hazma.cmb.`**`vx_cmb`**(*mx*, *x_kd*)

Computes the DM relative velocity at CMB using eq. 28 from this reference.

Parameters

- **`mx`** (*`float`*) – Dark matter mass in MeV.
- **`x_kd`** (*`float`*) – T_kd / m_x, where T_kd is the dark matter's kinetic decoupling temperature.

Returns **v_x** – The DM relative velocity at the time of CMB formation.

Return type float

CHAPTER 6

Particle physics quantities

6.1 Overview

In addition to providing functions for computing gamma-ray and CMB constraints, the `Theory` class has methods for computing a variety of particle physics quantities, which are documented on this page.

6.2 Methods

static `Theory.`**`list_annihilation_final_states`**`()`

Lists annihilation final states.

Subclasses must implement this method.

Returns **fss** – Possible annihilation final states.

Return type list(str)

`Theory.`**`annihilation_cross_sections`**`(`*e_cm*`)`

Computes annihilation cross sections.

Parameters **e_cm** (*float*) – Center of mass energy for the annihilation in MeV.

Returns **sigmas** – Annihilation cross section into each final state in MeV^{-2} as well as the total cross section.

Return type dict(str, float)

`Theory.`**`annihilation_cross_sections`**`(`*e_cm*`)`

Computes annihilation cross sections.

Parameters **e_cm** (*float*) – Center of mass energy for the annihilation in MeV.

Returns **sigmas** – Annihilation cross section into each final state in MeV^{-2} as well as the total cross section.

Return type dict(str, float)

`Theory.`**`annihilation_branching_fractions`**(*e_cm*)

Computes annihilation branching fractions.

Parameters **`e_cm`** (*`float`*) – Center of mass energy for the annihilation in MeV.

Returns **bfs** – Annihilation branching fractions into each final state.

Return type dict(str, float)

`Theory.`**`partial_widths`**()

Computes mediator decay widths.

Subclasses must implement this method.

Returns **widths** – Mediator partial widths in MeV as the total cross decay width.

Return type dict(str, float)

`Theory.`**`total_spectrum`**(*e_gams*, *e_cm*)

Computes total continuum gamma-ray spectrum.

Parameters

- **`e_gams`** (*`float or float numpy.array`*) – Photon energy or energies at which to compute the spectrum.
- **`e_cm`** (*`float`*) – Annihilation center of mass energy.

Returns **spec** – Array containing the total annihilation gamma-ray spectrum.

Return type float numpy.array

`Theory.`**`gamma_ray_lines`**(*e_cm*)

Gets information about annihilation into gamma-ray lines.

Subclasses must implement this method.

Parameters **`e_cm`** (*`float`*) – Annihilation center of mass energy.

Returns **lines** – For each final state containing a monochromatic photon, gives the energy of the photon and branching fraction into that final state.

Return type dict(str, dict(str, float))

`Theory.`**`spectra`**(*e_gams*, *e_cm*)

Gets the contributions to the continuum gamma-ray annihilation spectrum for each final state.

Parameters

- **`e_gams`** (*`float or float numpy.array`*) – Photon energy or energies at which to compute the spectrum.
- **`e_cm`** (*`float`*) – Center of mass energy for the annihilation in MeV.

Returns **specs** – Contribution to dN/dE_γ at the given photon energies and center-of-mass energy for each relevant final state. More specifically, this is the spectrum for annihilation into each channel rescaled by the corresponding branching fraction into that channel.

Return type dict(str, float)

`Theory.`**`spectrum_funcs`**()

Gets a function computing the continuum gamma-ray spectrum for annihilations into each relevant final state.

Subclasses must implement this method.

Returns **spec_fns** – dN/dE_γ as a function of photon energies and the annihilation center of mass energy for annihilation into each final state that produces a continuum spectrum.

Return type dict(str, (float or np.array, float) -> float)

`Theory.`**`total_conv_spectrum_fn`**(*e_gam_min*, *e_gam_max*, *e_cm*, *energy_res*, *n_pts=1000*)

Computes the total gamma-ray spectrum convolved with an energy resolution function.

Parameters

- **`e_min`** (*`float`*) – Lower bound of energy range over which to perform convolution.
- **`e_max`** (*`float`*) – Upper bound of energy range over which to perform convolution.
- **`e_cm`** (*`float`*) – Center of mass energy for DM annihilation.
- **`energy_res`** (*`float -> float`*) – The detector's energy resolution (Delta E / E) as a function of photon energy in MeV.
- **`n_pts`** (*`float`*) – Number of points to use to create resulting interpolating function. More points gives higher accuracy at the cost of computing time, but is necessary if the continuum spectrum contains very sharp features.

Returns **dnde_conv** – An interpolator giving the DM annihilation spectrum smeared by the energy resolution function. Using photon energies outside the range [e_min, e_max] will produce a `bounds_error`.

Return type InterpolatedUnivariateSpline

`Theory.`**`total_positron_spectrum`**(*e_ps*, *e_cm*)

Computes the total positron ray spectrum.

Parameters

- **`e_ps`** (*`float or float numpy.array`*) – Positron energy or energies at which to compute the spectrum.
- **`e_cm`** (*`float`*) – Annihilation center of mass energy.

Returns **spec** – Array containing the total annihilation positron spectrum.

Return type float numpy.array

`Theory.`**`positron_lines`**(*e_cm*)

Gets information about annihilation into monochromatic positrons.

Subclasses must implement this method.

Parameters **`e_cm`** (*`float`*) – Annihilation center of mass energy.

Returns **lines** – For each final state containing a monochromatic positron, gives the energy of the positron and branching fraction into that final state.

Return type dict(str, dict(str, float))

`Theory.`**`positron_spectra`**(*e_ps*, *e_cm*)

Gets the contributions to the continuum positron annihilation spectrum for each final state.

Parameters

- **`e_ps`** (*`float or float numpy.array`*) – Positron energy or energies at which to compute the spectrum.
- **`e_cm`** (*`float`*) – Center of mass energy for the annihilation in MeV.

Returns **specs** – Contribution to dN/dE_γ at the given positron energies and center-of-mass energy for each relevant final state. More specifically, this is the spectrum for annihilation into each channel rescaled by the corresponding branching fraction into that channel.

Return type dict(str, float)

`Theory.`**`positron_spectrum_funcs`**`()`

Gets a function computing the continuum positron spectrum for annihilations into each relevant final state.

Subclasses must implement this method.

Returns **spec_fns** – dN/dE_{e+} as a function of positron energies and the annihilation center of mass energy for annihilation into each final state that produces a continuum spectrum.

Return type dict(str, (float or np.array, float) -> float)

`Theory.`**`total_conv_positron_spectrum_fn`**`(`*e_p_min*, *e_p_max*, *e_cm*, *energy_res*, *n_pts=1000*`)`

Computes the total positron spectrum convolved with an energy resolution function.

Parameters

- **`e_min`** (*`float`*) – Lower bound of energy range over which to perform convolution.
- **`e_max`** (*`float`*) – Upper bound of energy range over which to perform convolution.
- **`e_cm`** (*`float`*) – Center of mass energy for DM annihilation.
- **`energy_res`** (*`float -> float`*) – The detector's energy resolution (Delta E / E) as a function of positron energy in MeV.
- **`n_pts`** (*`float`*) – Number of points to use to create resulting interpolating function. More points gives higher accuracy at the cost of computing time, but is necessary if the continuum spectrum contains very sharp features.

Returns **dnde_conv** – An interpolator giving the DM annihilation spectrum smeared by the energy resolution function. Using positron energies outside the range [e_min, e_max] will produce a `bounds_error`.

Return type InterpolatedUnivariateSpline

CHAPTER 7

Gamma ray spectra

7.1 Overview

This page discusses how to compute the gamma-ray spectrum for a particular particle physics process. Since computing gamma-ray spectra is a model-dependent process, we include in `hazma` tools for computing gamma-ray spectra from *both* FSR and the decay of final state-particles.

The `gamma_ray` module contains two functions called `gamma_ray_decay` and `gamma_ray_fsr`. The `gamma_ray_decay` function accepts a list of the final-state particles, the center-of-mass energy, the gamma-ray energies to compute the spectrum at and optionally the matrix element. Currently, the final-state particles can be π^0, $\pi^{\pm}$, $\mu^{\pm}$, $e^{\pm}$, $K^{\pm}$, K_L and K_S where K stands for kaon.

We caution that when including many final-state mesons, one needs to take care to supply the properly unitarized matrix element. The `gamma_ray_decay` functions works by first computing the energies distributions of all the final-state particles and convolving the energy distributions with the decay spectra of the final-state particles. The `gamma_ray_decay` function can be used as follows:

```
>>> from hazma.gamma_ray_decay import gamma_ray_decay
>>> import numpy as np
# specify the final-state particles
>>> particles = np.array(['muon', 'charged_kaon', 'long_kaon'])
# choose the center of mass energy
>>> cme = 5000.
# choose list of the gamma-ray energies to compute spectra at
>>> eng_gams = np.logspace(0., np.log10(cme), num=200, dtype=np.float64)
# compute the gamma-ray spectra assuming a constant matrix element
>>> spec = gamma_ray_decay(particles, cme, eng_gams)
```

The `gamma_ray_fsr` function computes the gamma-ray spectrum from $X \to Y\gamma$, i.e.:

$$\frac{dN(X \to Y\gamma)}{dE_\gamma} = \frac{1}{\sigma(X \to Y)} \frac{d\sigma(X \to Y\gamma)}{dE_\gamma}$$

where X and Y are any particles excluding the photon. This function takes in as input a list of the initial state particle masses (either 1 or 2 particles), the final state particle masses, the center-of-mass energy, a function for the tree-level matrix element (for $X \to Y$) and a function for the radiative matrix element ($X \to Y\gamma$). The functions for the matrix elements must take is a single argument which is a list of the four-momenta for the final state particles. As an example, we consider the process of two dark-matter particles annihilating into charged pions, $\bar{\chi}\chi \to \pi^+\pi^-(\gamma)$ using the model from *Advanced Usage*. In *Advanced Usage*, we gave the analytic expressions for the gamma-ray spectra. The tree-level and radiative matrix elements for this process are:

$$\begin{aligned}
|\mathcal{M}(\bar{\chi}\chi \to \pi^+\pi^-)|^2 &= \frac{c_1^2\left(s - 4m_\chi^2\right)}{2\Lambda^2} \\
|\mathcal{M}(\bar{\chi}\chi \to \pi^+\pi^-\gamma)|^2 &= \frac{2c_1^2\left(4\mu_\chi^2 - 1\right)Q^2e^2}{\Lambda^2\left(t - \mu_\pi^2 Q^2\right)^2\left(u - \mu_\pi^2 Q^2\right)^2} \\
&\quad \times \left(\left(\mu_\pi Q(t+u) - 2\mu_\pi^3 Q^3\right)^2 + s\left(t - \mu_\pi^2 Q^2\right)\left(\mu_\pi^2 Q^2 - u\right)\right)
\end{aligned}$$

where Q is the center-of-mass energy, e is the electromagnetic coupling, $\mu_{\pi,\chi} = m_{\pi,\chi}/Q$ and

$$s = (p_{\pi,1} + p_{\pi,2})^2, t = (p_{\pi,1} + k)^2, u = (p_{\pi,2} + k)^2$$

with $p_{\pi,1,2}$ are the four-momenta of the two final-state pions and k is the four-momenta of the final-state photon. Below, we create a class to implement functions for the tree and radiative matrix elements. Note that these functions take in an array of four-momenta.

```python
from hazma.field_theory_helper_functions.common_functions import \
minkowski_dot as MDot

class Msqrd(object):
    def __init__(self, mx, c1, lam):
        self.mx = mx # DM mass
        self.c1 = c1 # effective coupling of DM to charged pions
        self.lam = lam # cut off scale for effective theory

    def tree(self, momenta):
        ppi1 = momenta[0] # first charged pion four-momentum
        ppi2 = momenta[1] # second charged pion four-momentum
        Q = ppi1[0] + ppi2[0] # center-of-mass energy
        return -((self.c1**2 * (4 * self.mx**2-Q**2)) / (2 * self.lam**2))

    def radiative(self, momenta):
        ppi1 = momenta[0] # first charged pion four-momentum
        ppi2 = momenta[1] # second charged pion four-momentum
        k = momenta[2] # photon four-momentum
        Q = ppi1[0] + ppi2[0] + k[0] # center-of-mass energy

        mux = self.mx / Q
        mupi = mpi / Q

        s = MDot(ppi1 + ppi2, ppi1 + ppi2)
        t = MDot(ppi1 + k, ppi1 + k)
        u = MDot(ppi2 + k, ppi2 + k)

        return ((2*self.c1**2*(-1 + 4*mux**2)*Q**2*qe**2 *
                 (s*(-(mupi**2*Q**2) + t)*(mupi**2*Q**2 - u) +
                  (-2*mupi**3*Q**3 + mupi*Q*(t + u))**2)) /
                (self.lam**2*(-(mupi**2*Q**2) + t)**2*
                (-(mupi**2*Q**2) + u)**2))
```

Next, we can compute the gamma-ray spectrum for $\bar{\chi}\chi \to \pi^+\pi^-\gamma$ using:

```
>>> from hazma.gamma_ray import gamma_ray_fsr
# specify the parameters of the model
>>> params = {'mx': 200.0, 'c1':1.0, 'lam':1e4}
# create instance of our Msqrd class
>>> msqrds = Msqrd(**params)
# specify the initial and final state masses
>>> isp_masses = np.array([msqrds.mx, msqrds.mx])
>>> fsp_masses = np.array([mpi, mpi, 0.0])
# choose the center-of-mass energy
>>> cme = 4.0 * msqrds.mx
# compute the gamma-ray spectrum
>>> spec = gamma_ray_fsr(isp_masses, fsp_masses, cme, msqrds.tree,
                         msqrds.radiative, num_ps_pts=500000, num_bins=50)
# plot the spectrum
>>> import matplotlib.pyplot as plt
>>> plt.figure(dpi=100)
>>> plt.plot(spec[0], spec[1])
>>> plt.yscale('log')
>>> plt.xscale('log')
>>> plt.ylabel(r'$dN/dE_{\gamma} \ (\mathrm{MeV}^{-1})$', fontsize=16)
>>> plt.xlabel(r'$E_{\gamma} \ (\mathrm{MeV})$', fontsize=16)
```

7.2 Functions

CHAPTER 8

Positron spectra

8.1 Overview

In order to compute the limits on a given model from energy injections into the CMB spectra, one needs to know the gamma-ray and electron/positron spectra for the model. `hazma` contains a dedicated module, `positron_spectra`, for computing the electron/positron spectra from decays of $\pi^{\pm}$ and $\mu^{\pm}$. As in the `decay` module, the `positron_spectra` module allows users to compute the electron/positron spectra for arbitrary energies of the parent-particle. The procedure for computing the spectra for arbitrary parent-particle energies is identical to the procedure used for `decay`. In addition to the decay spectra of muons and pions, `hazma` contains a function to compute the electron/positron from a matrix element called `positron_decay`.

8.2 Functions

CHAPTER 9

Decay spectra

9.1 Overview

The `decay.py` module contains high-performance functions for computing the gamma-ray spectra from $\pi^{\pm}$, math::*pi^{0}* and $\mu^{\pm}$ decays.

The functions in this module allow the user to compute the decay spectra for arbitrary parent-particle energy. In order to obtain spectra for arbitrary parent-particle energy, we compute the decay spectra in the rest-frame of the parent-particle and perform a Lorentz boost, which amounts to doing a change-of-variables along with a "convolution" integral. To achieve higher computational performance, we perform all integrations in `c` using `cython` and build extension modules to interface with python.

9.2 Functions

CHAPTER 10

RAMBO

10.1 Overview

In hazma, the rambo module can be used to perform various tasks. We include a generic functions for generating phase-space points called generate_phase_space_point and generate_phase_space, which will compute a single or many phase-space points. The generate_phase_space function additionally allows for the user to specify a matrix element. We include a function called integrate_over_phase_space which will perform the integral:

$$\int \left(\prod_{i=1}^{N} \frac{d^3\vec{p}_i}{(2\pi)^3}\frac{1}{2E_i}\right)(2\pi)^4\delta^4\left(P - \sum_{i=1}^{N} p_i\right)|\mathcal{M}|^2$$

where P^μ is the total four-momentum, p_i^μ are the individual four-momenta for each of the N final-state particles and $\mathcal{M}$ is the matrix element. rambo.py also contains functions for computing cross-sections (decay widths) for $2 \to N$ ($1 \to N$) processes called compute_annihilation_cross_section (compute_decay_width). For example, if we would like to compute the partial decay width of $\mu \to e\nu\nu$, one could use the following. First, we declare a function for the matrix element. The function for the matrix element must take in a list of the four-momenta and return the matrix element:

```python
# Import the fermi constant
from hazma.parameters import GF
# Import a helper function for scalar products of four-vectors
from hazma.field_theory_helper_functions.common_functions import \
    minkowski_dot as MDot
# Declare the matrix element
def msqrd_mu_to_enunu(momenta):
    pe = momenta[0] # electron four-momentum
    pve = momenta[1] # electron-neutrino four-momentum
    pvmu = momenta[2] # muon-neutrino four-momentum
    pmu = sum(momenta) # muon four-momentum
    # Return matrix element
    return 64. * GF**2 * MDot(pe, pvmu) * MDot(pmu, pve)
```

Then, the partial decay width can be computed using:

```
# Import function to compute decay width
from hazma.rambo import compute_decay_width
# import masses of muon and electron
from hazma.parameters import muon_mass as mmu
from hazma.parameters import electron_mass as me
# Specify the masses of the electron and neutrinos
fsp_masses = np.array([me, 0., 0.])
# compute the partial width
partial_width = compute_decay_width(fsp_masses, mmu, num_ps_pts=50000,
                                    mat_elem_sqrd=msqrd_mu_to_enunu)
```

Using 50000 phase-space points, we are able to within 5% of the analytical result. In addition, `rambo` includes a function for performing partial integrations over all variables except the energy of one of the final-state particles called `generate_energy_histogram`. This function returns a multi-dimensional array with the first index labeling the final-state particles and zeroth component of the second index given the energies and the 1 component of the second index giving the probability that the final-state particle has the particular energy. This function can be used via:

```
from hazma.rambo import generate_energy_histogram
import numpy as np
num_ps_pts = 100000 #number of phase-space points to use
# masses of final-state particles
masses = np.array([100., 100., 0.0, 0.0])
cme = 1000. # center-of-mass energy
num_bins = 100 # number of energy bins to use
# computing energy histograms
eng_hist = generate_energy_histogram(num_ps_pts, masses, cme,
                                     num_bins=num_bins)
# plot the results
import matplotlib as plt
for i in range(len(masses)):
    # pts[i, 0] are the energies of particle i
    # pts[i, 1] are the probabilities
    plt.loglog(pts[i, 0], pts[i, 1])
```

10.2 Functions

CHAPTER 11

Indices and tables

- genindex
- search

Index