Hazma

Release 1.0

Adam Coogan and Logan Morrison

Aug 28, 2022

CONTENTS:

CHAPTER

ONE

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

TWO

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']
>>> 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
>>> 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.

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

class HiggsPortal(ScalarMediator):
    def __init__(self, mx, ms, gsxx, stheta):
        self._lam = vh
```

(continues on next page)

(continued from previous page)

```
        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>
>>> obs.fluxes
array([8.85813149e-08, 5.82726327e-09])
>>> obs.upper_errors
```

(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)

```
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)
```

(continues on next page)

(continued from previous page)

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

(continues on next page)

(continued from previous page)

```
        self.mx = mx
        self.c1 = c1
        self.c2 = c2
        self.lam = lam

    @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

THREE

SPECTRA (HAZMA.SPECTRA)

3.1 Overview

The hazma.spectra module contains functions for:

- Computing photon,positron and neutrino spectra from the decays of individual SM particles,
- Computing spectra from an N-body final state,
- Boosting spectra into new frames,
- Computing FSR spectra using Altarelli-Parisi splitting functions.

Below, we demonstrate how each of these actions can be easily performed in hazma.

3.1.1 Computing spectra (General API)

To compute the spectra for an arbitrary final state, use hazma.spectra.dnde_photon(), hazma.spectra.dnde_positron() or hazma.spectra.dnde_neutrino(). These functions takes the energies to compute the spectra, the center-of-mass energy and the final states and returns the spectrum.

```
import numpy as np
from hazma import spectra

cme = 1500.0 # 1.5 GeV
energies = np.geomspace(1e-3, 1.0, 100) * cme

# Compute the photon spectrum from a muon, a long kaon and a charged kaon
dnde = spectra.dnde_photon(energies, cme, ["mu", "kl", "k"])

plt.plot(energies, energies **2 * dnde)
plt.yscale("log")
plt.xscale("log")
plt.xlabel(r"$E_{\gamma} \ [\mathrm{MeV}]$", fontsize=16)
plt.ylabel(r"$E_{\gamma}^2\dv*{N}{E_{\gamma}} \ [\mathrm{MeV}]$", fontsize=16)
plt.tight_layout()
plt.ylim(1e-1, 2e2)
plt.xlim(np.min(energies), np.max(energies))
plt.show()
```

Under the hood, hazma computes the spectra from an N-body final state by computing the expectation values of the decay spectra from each particle w.r.t. each particles energy distribution. In the case of photon spectra, we include

both the spectra from the decays of final state particles as well as the final state radiation from charged states. The final spectrum is:

$$\frac{dN}{dE} = \sum_f \frac{dN_{\text{dec.}}}{dE} + \sum_f \frac{dN_{\text{FSR}}}{dE}$$

where the **decay** and **FSR** components are:

$$\frac{dN_{\text{dec}}}{dE} = \sum_f \int d\epsilon_f P(\epsilon_f) \frac{dN_f}{dE}(E, \epsilon_f)$$
$$\frac{dN}{dE} = \sum_{i \neq j} \frac{1}{S_i S_j} \int ds_{ij} P(s_{ij}) \left(\frac{dN_{i,\text{FSR}}}{dE}(E, s_{ij}) + \frac{dN_{j,\text{FSR}}}{dE}(E, s_{ij}) \right)$$

where $P(\epsilon_f)$ is the probability distribution of particle f having energy ϵ_f. The sum over f in the decay expression is over all final-state particles. For the FSR expression, s_{ij} is the squared invariant-mass of particles i and j. The factors S_i and S_j are symmetry factors included to avoid overcounting.

We can see what the distributions look like using the `hazma.phase_space` module. For three body final state consisting of a muon, long kaon and charged pion, we can compute the energy distributions and invariant mass distributions in a couple of ways. The first option is using Monte-Carlo integration using *Rambo*:

```
from hazma import phase_space
from hazma.parameters import standard_model_masses as sm_masses

cme = 1500.0
states = ["mu", "kl", "k"]
masses = tuple(sm_masses[s] for s in states)
dists = phase_space.Rambo(cme, masses).energy_distributions(n=1<<14, nbins=30)
```

The second option is to use the specialized `ThreeBody` class:

```
from hazma import phase_space
from hazma.parameters import standard_model_masses as sm_masses

cme = 1500.0
states = ["mu", "kl", "k"]
masses = tuple(sm_masses[s] for s in states)
dists = phase_space.ThreeBody(cme, masses).energy_distributions(nbins=30)
```

For positron and neutrino spectra, the calculation is the same except that we omit the FSR.

3.1.2 Individual Particle Spectra

`hazma` provides access to the individual spectra for all supported particles. All the photon, positron and neutrino spectra follow the naming convention `dnde_photon_*`, `dnde_positron_*` and `dnde_neutrino_*`. Each spectrum function takes in the energies of the photon/positron/neutrino and the energy of the decaying particle are returns the spectrum. As an example, let's generate all the photon spectra for the particles available in `hazma`:

```
import numpy as np
import matplotlib.pyplot as plt
from hazma import spectra

cme = 1500.0
```

(continues on next page)

(continued from previous page)

```
es = np.geomspace(1e-4, 1.0, 100) * cme

# Make a dictionary of all the available decay spectrum functions
dnde_fns = {
   "mu":    spectra.dnde_photon_muon,
   "pi":    spectra.dnde_photon_charged_pion,
   "pi0":   spectra.dnde_photon_neutral_pion,
   "k":     spectra.dnde_photon_charged_kaon,
   "kl":    spectra.dnde_photon_long_kaon,
   "ks":    spectra.dnde_photon_short_kaon,
   "eta":   spectra.dnde_photon_eta,
   "etap":  spectra.dnde_photon_eta_prime,
   "rho":   spectra.dnde_photon_charged_rho,
   "rho0":  spectra.dnde_photon_neutral_rho,
   "omega": spectra.dnde_photon_omega,
   "phi":   spectra.dnde_photon_phi,
}

# Compute the spectra. Each function takes the energy where the spectrum
# is evaluated as the 1st argument and the energy of the decaying
# particle as the 2nd
dndes = {key: fn(es, cme) for key, fn in dnde_fns.items()}

plt.figure(dpi=150)
for key, dnde in dndes.items():
   plt.plot(es, es**2 * dnde, label=key)
plt.yscale("log")
plt.xscale("log")
plt.ylim(1e-2, 1e4)
plt.xlim(np.min(es), np.max(es))
plt.ylabel(r"$E_{\gamma}^2\dv{N}{dE} \ [\mathrm{MeV}]$", fontdict=dict(size=16))
plt.xlabel(r"$E_{\gamma} \ [\mathrm{MeV}]$", fontdict=dict(size=16))
plt.legend()
plt.tight_layout()
```

You can also produce the same results as above using `dnde_photon()`. To demonstrate this, let's repeat the above with the positron and neutrino spectra:

```
import numpy as np
import matplotlib.pyplot as plt
from hazma import spectra

cme = 1500.0
es = np.geomspace(1e-3, 1.0, 100) * cme

# `dnde_photon`, `dnde_positron` and `dnde_neutrino` all have an
# `availible_final_states` attribute that returns a list of strings for all
# the available particles
e_states = spectra.dnde_positron.availible_final_states
nu_states = spectra.dnde_neutrino.availible_final_states

dndes_e = {key: spectra.dnde_positron(es, cme, key) for key in e_states}
```

(continues on next page)

(continued from previous page)

```
dndes_nu = {key: spectra.dnde_neutrino(es, cme, key) for key in nu_states}

plt.figure(dpi=150, figsize=(12,4))
axs = [plt.subplot(1,3,i+1) for i in range(3)]
for key, dnde in dndes_e.items():
    axs[0].plot(es, es**2 * dnde, label=key)

for key, dnde in dndes_nu.items():
    axs[1].plot(es, es**2 * dnde[0], label=key)
    axs[2].plot(es, es**2 * dnde[1], label=key)

titles = ["positron", "electron-neutrino", "muon-neutrino"]
for i, ax in enumerate(axs):
    ax.set_yscale("log")
    ax.set_xscale("log")
    ax.set_xlim(np.min(es), np.max(es))
    ax.set_ylim(1e-2, 1e3)
    ax.set_xlim(np.min(es), np.max(es))
    ax.set_title(titles[i])
    ax.set_ylabel(r"$E^2\dv{N}{E} \ [\mathrm{MeV}]$", fontdict=dict(size=16))
    ax.set_xlabel(r"$E \ [\mathrm{MeV}]$", fontdict=dict(size=16))
    ax.legend()

plt.tight_layout()
```

3.1.3 Including the Matrix Element

All three of the functions, `dnde_photon`, `dnde_positron` and `dnde_neutrino` allow the user to supply as squared matrix element. The matrix element is then used to correctly determine the energy distributions of the final state particles. If no matrix element is supplied, it is taken to be unity.

To demonstrate the use of a matrix element, we consider the decay of a muon into an electron and two neutrinos. We will compute the photon , the positron and the neutrino spectra. We compute the photon spectrum using only the FSR (no initial state radiation or internal bremstralung from the W.)

```
import numpy as np
import matplotlib.pyplot as plt
from hazma import spectra
from hazma.parameters import GF
from hazma.parameters import muon_mass as MMU

def msqrd(s, t):
   return 16.0 * GF**2 * t * (MMU**2 - t)

es = np.geomspace(1e-4, 1.0, 100) * MMU
final_states = ["e", "ve", "vm"]

dndes = {
   1: {
      # Photon
      r"$\gamma$ approx": spectra.dnde_photon(es, MMU, final_states, msqrd=msqrd),
```

(continues on next page)

(continued from previous page)

```
        r"$\gamma$ analytic": spectra.dnde_photon_muon(es, MMU),
    },
    2: {
        # Positron
        r"$e^{+}$ approx": spectra.dnde_positron(es, MMU, final_states, msqrd=msqrd),
        r"$e^{+}$ analytic": spectra.dnde_positron_muon(es, MMU),
    },
    3: {
        # Electron and muon neutrino
        r"$\nu_e$ approx": spectra.dnde_neutrino(es, MMU, final_states, msqrd=msqrd,␣
↪flavor="e"),
        r"$\nu_e$ analytic": spectra.dnde_neutrino_muon(es, MMU, flavor="e"),
        r"$\nu_{\mu}$ approx": spectra.dnde_neutrino(es, MMU, final_states, msqrd=msqrd,␣
↪flavor="mu"),
        r"$\nu_{\mu}$ analytic": spectra.dnde_neutrino_muon(es, MMU, flavor="mu"),
    },
}


plt.figure(dpi=150, figsize=(12, 4))
for i, d in dndes.items():
    ax = plt.subplot(1, 3, i)
    lss = ["-", "--", "-", "--"]
    for j, (key, dnde) in enumerate(d.items()):
        ax.plot(es, dnde, ls=lss[j], label=key)
    ax.set_yscale("log")
    ax.set_xscale("log")
    ax.set_xlim(1e-1, 1e2)
    ax.set_ylim(1e-5, 1)
    if i == 1:
        ax.set_ylabel(r"$\dv{N}{E} \ [\mathrm{MeV}]$", fontdict=dict(size=16))
    ax.set_xlabel(r"$E \ [\mathrm{MeV}]$", fontdict=dict(size=16))
    ax.legend()

plt.tight_layout()
```

We can see that the results match the analytic results in large portions of the energy range. For the positron and neutrino spectra, the disagreements for low energies is due to the energy binning (we a linear binning procedure.) If we set `nbins` to a larger value, we can cover more of the energy range.

3.1.4 Boosting Spectra

It's common that one is interested in a spectrum in a boosted frame. *hazma* provides facilities to boost a spectrum from one frame to another. See <table boost> for the available functions. As an example, let's take the photon spectrum from a muon decay and boost it from the muon rest frame to a new frame. There are multiple ways to do this. The first way would be to compute the spectrum in the rest frame and boost the array. This is done using:

```
import matplotlib.pyplot as plt
import numpy as np
from hazma import spectra
from hazma.parameters import muon_mass as MMU
```

(continues on next page)

(continued from previous page)

```
beta = 0.3
gamma = 1.0 / np.sqrt(1.0 - beta**2)
emu = gamma * MMU

es = np.geomspace(1e-4, 1.0, 100) * MMU
# Rest frame spectrum (emu = muon mass)
dnde_rf = spectra.dnde_photon_muon(es, MMU)
# Analytic boosted spectrum
dnde_boosted_analytic = spectra.dnde_photon_muon(es, emu)
# Approximate boosted spectrum
dnde_boosted = spectra.dnde_boost_array(dnde_rf, es, beta=beta)

plt.figure(dpi=150)
plt.plot(es, dnde_boosted, label="dnde_boost_array")
plt.plot(es, dnde_boosted_analytic, label="analytic", ls="--")
plt.yscale("log")
plt.xscale("log")
plt.ylabel(r"$\dv{N}{E} \ [\mathrm{MeV}^{-1}]$", fontdict=dict(size=16))
plt.xlabel(r"$E \ [\mathrm{MeV}]$", fontdict=dict(size=16))
plt.legend()
plt.show()
```

This method will always have issues near the minimum energy, as we have no information about the spectrum below the minimum energy. A second way of performing the calculation is to use `make_boost_function()`. This method take in the spectrum in the original frame and returns a new function which is able to compute the boost integral.

```
import matplotlib.pyplot as plt
import numpy as np
from hazma import spectra
from hazma.parameters import muon_mass as MMU

beta = 0.3
gamma = 1.0 / np.sqrt(1.0 - beta**2)
emu = gamma * MMU

es = np.geomspace(1e-4, 1.0, 100) * emu

# Rest frame function:
dnde_rf_fn = lambda e: spectra.dnde_photon_muon(e, MMU)
# New boosted spectrum function:
dnde_boosted_fn = spectra.make_boost_function(dnde_rf_fn)

# Analytic boosted spectrum
dnde_boosted_analytic = spectra.dnde_photon_muon(es, emu)
# Approximate boosted spectrum
dnde_boosted = dnde_boosted_fn(es, beta=beta)

plt.figure(dpi=150)
plt.plot(es, dnde_boosted, label="make_boost_function")
plt.plot(es, dnde_boosted_analytic, label="analytic", ls="--")
plt.yscale("log")
```

(continues on next page)

(continued from previous page)

```
plt.xscale("log")
plt.ylabel(r"$\dv{N}{E} \ [\mathrm{MeV}^{-1}]$", fontdict=dict(size=16))
plt.xlabel(r"$E \ [\mathrm{MeV}]$", fontdict=dict(size=16))
plt.legend()
plt.show()
```

The last method is to use `dnde_boost()`, which similar to `make_boost_function()` but it returns the boosted spectrum rather than a function. To use it, do the following:

```
import numpy as np
from hazma import spectra
from hazma.parameters import muon_mass as MMU

beta = 0.3
gamma = 1.0 / np.sqrt(1.0 - beta**2)
emu = gamma * MMU

es = np.geomspace(1e-4, 1.0, 100) * emu

# Analytic boosted spectrum
dnde_boosted_analytic = spectra.dnde_photon_muon(es, emu)
# Approximate boosted spectrum
dnde_boosted = spectra.dnde_boost(
    lambda e: spectra.dnde_photon_muon(e, MMU),
    es,
    beta=beta
)
```

3.1.5 Approximate FSR Using the Altarelli-Parisi Splitting Functions

In the limit that the center-of-mass energy of a process is much larger than the mass of a charged state, the final state radiation is approximately equal to the it's splitting function (with additional kinematic factors.) The splitting functions for scalar and fermionic states are:

$$P_S(x) = \frac{2(1-x)}{x},$$
$$P_F(x) = \frac{1+(1-x)^2}{x}.$$

In these expressions, $x = 2E/\sqrt{s}$, with E being the particle's energy and $\sqrt{s}$ the center-of-mass energy. Given some process $I \to A + B + \cdots + s$ or $I \to A + B + \cdots + f$ (with s and f being a scalar and fermionic state), the approximate FSR spectra are:

$$N_S E = \frac{Q_S^2 \alpha_{\mathrm{EM}}}{\sqrt{s}\pi} P_S(x) (\log(\frac{s(1-x)}{m_S^2}) - 1)$$
$$N_F E = \frac{Q_F^2 \alpha_{\mathrm{EM}}}{\sqrt{s}\pi} P_F(x) (\log(\frac{s(1-x)}{m_F^2}) - 1)$$

where $Q_{S,F}$ and $m_{S,F}$ are the charges and masses of the scalar and fermion. To compute these spectra in *hazma*, we provide the functions `dnde_photon_ap_scalar()` and `dnde_photon_ap_fermion()`. Below we demonstrate how to use these:

```
import matplotlib.pyplot as plt
import numpy as np
from hazma import spectra
from hazma.parameters import muon_mass as MMU
from hazma.parameters import charged_pion_mass as MPI

cme = 500.0 # 500 MeV
es = np.geomspace(1e-4, 1.0, 100) * cme

# These functions take s = cme^2 as second argument and mass of the state as
# the third.
dnde_fsr_mu = spectra.dnde_photon_ap_fermion(es, cme**2, MMU)
dnde_fsr_pi = spectra.dnde_photon_ap_scalar(es, cme**2, MPI)

plt.figure(dpi=150)
plt.plot(es, dnde_fsr_mu, label=r"$\mu$")
plt.plot(es, dnde_fsr_pi, label=r"$\pi$", ls="--")
plt.yscale("log")
plt.xscale("log")
plt.ylabel(r"$\dv{N}{E} \ [\mathrm{MeV}^{-1}]$", fontdict=dict(size=16))
plt.xlabel(r"$E \ [\mathrm{MeV}]$", fontdict=dict(size=16))
plt.legend()
plt.show()
```

3.2 API Reference

3.2.1 Altarelli-Parisi

Functions for compute FSR spectra using the Altarelli-Parisi splitting functions:

Function	Description
`dnde_photon_ap_fermion()`	Approximate FSR from fermion.
`dnde_photon_ap_scalar()`	Approximate FSR from scalar.

3.2.2 Boost

Function	Description
`boost_delta_function()`	Boost a delta-function.
`double_boost_delta_function()`	Perform two boosts of a delta-function.
`dnde_boost_array()`	Boost spectrum specified as an array.
`dnde_boost()`	Boost spectrum specified as function.
`make_boost_function()`	Construct a boost function.

3.2.3 Photon Spectra

hazma includes decay spectra from several unstable particles:

Function	Description
`dnde_photon()`	Photon spectrum from decay/FSR of n-body final states.
`dnde_photon_muon()`	Photon spectrum from decay of $\mu^{\pm}$.
`dnde_photon_neutral_pion()`	Photon spectrum from decay of π^0.
`dnde_photon_charged_pion()`	Photon spectrum from decay of $\pi^{\pm}$.
`dnde_photon_charged_kaon()`	Photon spectrum from decay of $K^{\pm}$.
`dnde_photon_long_kaon()`	Photon spectrum from decay of K_L.
`dnde_photon_short_kaon()`	Photon spectrum from decay of K_S.
`dnde_photon_eta()`	Photon spectrum from decay of η.
`dnde_photon_eta_prime()`	Photon spectrum from decay of η'.
`dnde_photon_charged_rho()`	Photon spectrum from decay of $\rho^{\pm}$.
`dnde_photon_neutral_rho()`	Photon spectrum from decay of ρ^0.
`dnde_photon_omega()`	Photon spectrum from decay of ω.
`dnde_photon_phi()`	Photon spectrum from decay of ϕ.

3.2.4 Positron Spectra

Function	Description
`dnde_positron()`	Positron spectrum from decay of n-body final states.
`dnde_positron_muon()`	Positron spectrum from decay of $\mu^{\pm}$.
`dnde_positron_charged_pion()`	Positron spectrum from decay of $\pi^{\pm}$.
`dnde_positron_charged_kaon()`	Positron spectrum from decay of $K^{\pm}$.
`dnde_positron_long_kaon()`	Positron spectrum from decay of K_L.
`dnde_positron_short_kaon()`	Positron spectrum from decay of K_S.
`dnde_positron_eta()`	Positron spectrum from decay of η.
`dnde_positron_eta_prime()`	Positron spectrum from decay of η'.
`dnde_positron_charged_rho()`	Positron spectrum from decay of $\rho^{\pm}$.
`dnde_positron_neutral_rho()`	Positron spectrum from decay of ρ^0.
`dnde_positron_omega()`	Positron spectrum from decay of ω.
`dnde_positron_phi()`	Positron spectrum from decay of ϕ.

3.2.5 Neutrino Spectra

Function	Description
`dnde_neutrino()`	Neutrino spectrum from decay of n-body final states.
`dnde_neutrino_muon()`	Neutrino spectrum from decay of $\mu^{\pm}$.
`dnde_neutrino_charged_pion()`	Neutrino spectrum from decay of $\pi^{\pm}$.
`dnde_neutrino_charged_kaon()`	Neutrino spectrum from decay of $K^{\pm}$.
`dnde_neutrino_long_kaon()`	Neutrino spectrum from decay of K_L.
`dnde_neutrino_short_kaon()`	Neutrino spectrum from decay of K_S.
`dnde_neutrino_eta()`	Neutrino spectrum from decay of η.
`dnde_neutrino_eta_prime()`	Neutrino spectrum from decay of η'.
`dnde_neutrino_charged_rho()`	Neutrino spectrum from decay of $\rho^{\pm}$.
`dnde_neutrino_omega()`	Neutrino spectrum from decay of ω.
`dnde_neutrino_phi()`	Neutrino spectrum from decay of ϕ.

CHAPTER

FOUR

PHASE SPACE (HAZMA.PHASE_SPACE)

4.1 Overview

The `phase_space` module contains the `hazma.phase_space.Rambo` class for working with N-body phase-space and the specialized `hazma.phase_space.ThreeBody` class for three-body phase-space. `Rambo` contains methods for generating phase-space points, integrating over N-body phase-space, computing cross-sections and computing decay widths.

4.1.1 Rambo

The `Rambo` class is instantiated by supplying the center-of-mass energy, the masses of the final-state particles and optionally a function to compute the squared matrix element. The function to compute the squared matrix element must be a vectorized unary function that return the squared matrix elements given four-momenta. The internal momenta generated by `Rambo` have a shape of *(4, nfsp, n)* where *nfsp* in the number of final-state particles and *n* is the number of points generated simultaneously. The leading dimension holds the energy, x, y and z-components of the four-momenta. If no function to compute the squared matrix element is supplied, it will be taken to be 1.

In the following snippet, we create a `Rambo` object for a process with 3 final-state particles, and a squared matrix element equal to $p_1 \cdot p_2$. Note the *Utilities (hazma.utils)* module contains a function `hazma.utils.ldot()` to compute the Lorentzian scalar product between two four-vectors.

```
from hazma import phase_space
from hazma import utils

cme = 10.0
masses = [1.0, 2.0, 3.0]
```

(continues on next page)

(continued from previous page)

```
def msqrd(momenta):
   p1 = momenta[:, 0] # Pick out the 4-momenta of particle 1
   p2 = momenta[:, 1] # Pick out the 4-momenta of particle 2
   return utils.ldot(p1, p2) # Compute p1.p2

phase_space = phase_space.Rambo(cme, masses, msqrd)
```

Integrating over phase-space

The `Rambo.integrate()` method computes the following integral:

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

This function takes in an integer specifying the number of points to use for the Monte-Carlo integration and computes the integral using the *RAMBO* algorithm. As a simple example, let's compute the N-body integral for massless particles. The analytical result is:

$$\Pi_{\text{LIPS}}^{(n)} = \frac{1}{\Gamma(n)\Gamma(n-1)} (2\pi)^{4-3n} (\frac{\pi}{2})^{n-1} {E_{\text{CM}}}^{2n-4}$$

We can verify this using the `Rambo.integrate()` method:

Listing 1: Integrating N-body phase space with massless particles

```
import math
import numpy as np
from hazma import phase_space

def analytic(n):
   fact = math.factorial(n - 2) * math.factorial(n - 1)
   return (0.5 * math.pi) ** (n - 1) * (2 * np.pi) ** (4 - 3 * n) / fact
analytic_integrals = [analytic(n) for n in range(2, 10)]

integrals = []
for n in range(2, 10):
   rambo = phase_space.Rambo(1.0, [0.0] * n)
   integral, error = rambo.integrate(n=10)
   integrals.append(integral)
np.max([abs(i1 - i2) / i2 for i1, i2 in zip(integrals, analytic_integrals)])
# Possible output: 2.2608966769988504e-15
```

Generating phase-space points

Sometimes it is useful to have access the momenta and weights of N-body phase-space. The are two methods for generating momenta and weights: `Rambo.generate()` and func:*Rambo.generator*. The func:*Rambo.generate* :py:method will return a specified number of momenta and weights. The `Rambo.generator()` method returns a python generator, allowing for iterating over batches of momenta and weights.

As described above, the momenta will have a shape of *(4, nfsp, n)* where *nfsp* is the number of final-state particles and *n* is the number of requested points. The weights will have a shape of *(n,)*. To see this explicitly, consider the following. Here we generate 10 phase-space points:

```
from hazma import phase_space

rambo = phase_space.Rambo(cme=10.0, masses=[1.0, 2.0, 3.0])
momenta, weights = rambo.generate(n=10)
print(momenta.shape)
print(weights.shape)
# (4, 3, 10)
# (10,)
```

In some cases, one may not want to generate all points at once (since it is costly memory-wise or maybe one wants to monitor convergence.) We supply the `Rambo.generator()` method for generating batches of phase-space points. For example, suppose we want to integrate over phase-space ourselves using 1M points and batches of 50,000 points at a time. To do this, we can use:

```
import numpy as np
from hazma import phase_space

rambo = phase_space.Rambo(cme=10.0, masses=[1.0, 2.0, 3.0])
n = int(1e6)
batch_size = 50_000
integrals = []
for momenta, weights in rambo.generator(n, batch_size, seed=1234):
   integrals.append(np.nanmean(weights))
np.average(integrals)
# Output: 0.0036118278252665406
```

Note: The above code is equivalent to using `hazma.phase_space.Rambo.generate()` with `n=int(1e6)` and `batch_size=50_000`. All methods accept a `batch_size` argument used can to split the computation into chunks in cases where the user has limited memory.

Computing decay widths and cross sections

The most common use of the `Rambo()` is computing cross sections or decay widths. The functions `hazma.phase_space.Rambo.cross_section()` and `hazma.phase_space.Rambo.decay_width()` can be used for these purposes. These methods just wrap `hazma.phase_space.Rambo.integrate()` and append the appropriate prefactors. As an example, let's compute the muon decay width for the process $\mu \to e\nu_e\nu_\mu$. The squared matrix element (ignoring the electron mass) is:

$$|\mathcal{M}|^2 = 16 G_F^2 t(m_\mu^2 - t)$$

where G_F is the Fermi constant and $t = (p_e + p_{\nu_\mu})^2$. The analytic result is

$$\Gamma = \frac{G_F^2 m_\mu^5}{192\pi^3} \sim 3 \times 10^{-19}$$

To compute this, we use the following:

```
from hazma import phase_space
from hazma import utils
from hazma.parameters import GF
from hazma.parameters import muon_mass as MMU
```

(continues on next page)

(continued from previous page)

```
def msqrd(momenta):
    p1 = momenta[:, 0]
    p3 = momenta[:, 2]
    t = utils.lnorm_sqr(p1 + p3)
    return 16.0 * GF**2 * t * (MMU**2 - t)


rambo = phase_space.Rambo(MMU, [0.0, 0.0, 0.0], msqrd=msqrd)
width, error = rambo.decay_width(n=50_000, seed=1234)

analytic = GF**2 * MMU**5 / (192 * np.pi**3)
actual_error = abs(width - analytic)
print(f"width = {width:.2e} +- {error:.2e}")
print(f"actual error = {actual_error:.2e} = {actual_error / analytic * 100:.2f} %")
# Output:
#  width = 3.02e-19 +- 5.99e-22
#  actual error = 9.50e-22 = 0.32 %
```

Note: The *Utilities (hazma.utils)* module contains a couple of functions useful for dealing with four-vectors, namely, `hazma.utils.ldot()` for computing scalar products and `hazma.utils.lnorm_sqr()` for computing the squared norm.

Energy and Invariant Mass Distributions

The `Rambo` class can also compute energy distributions as well as the invariant mass distributions of pairs of final state particles. To compute energies distributions, use `Rambo.energy_distributions`:

```
from hazma import phase_space
from hazma.parameters import standard_model_masses as sm_masses
import matplotlib.pyplot as plt

states = ["pi", "e", "mu", "k"]
masses = [sm_masses[s] for s in states]
cme = 3 * sum(masses)
rambo = phase_space.Rambo(cme, masses)

energy_dists = rambo.energy_distributions(n=1<<16, nbins=25)

plt.figure(dpi=150)
labels=[r"$\pi$", r"$e$", r"$\mu$", r"$K$"]
for i, dist in enumerate(energy_dists):
    plt.plot(dist.bin_centers, dist.probabilities, label=labels[i])

plt.ylabel(r"$P(\epsilon) \ [\mathrm{MeV}^{-1}]$", fontdict=dict(size=16))
plt.xlabel(r"$\epsilon \ [\mathrm{MeV}]$", fontdict=dict(size=16))
plt.tight_layout()
plt.legend()
```

We note the 'choppy' behavior of the curves. Since we are performing naive Monte-Carlo integration, we need a large

number of points to properly sample phase-space. The invariant mass distributions are generated in a similar fashion. Note that the return value of the invariant mass distributions is a dictionary with keys given by a pair of integers that specify the pair of particles the distribution corresponds to. See `hazma.phase_space.PhaseSpaceDistribution1D` for more information on the distribution objects.

4.1.2 Three Body Phase Space (`hazma.phase_space.ThreeBody`)

Attention: This class is meant for cases where the squared matrix element **only** depends on the total momentum and the momenta of the final state particles. If this isn't the case, then this class is not suitable.

As an example of a case where this class does apply, recall that the squared matrix element (summed over spins) for the decay of an unstable particle into a three-body final state only depends on the momenta of the final-state particles.

For three-body phase space where the squared matrix element only depends on the final-state momenta, the phase space integral simplifies. The result is:

$$\Phi_3 = \frac{1}{16(2\pi)^2 Q^2} \int_{s_-}^{s_+} s \int_{t_-(s)}^{t_+(s)} t |\mathcal{M}|^2$$

where Q is the center-of-mass energy, $s = (p_2 + p_3)^2$ and $t = (p_1 + p_3)^2$. The limits of integration are:

$$\begin{aligned}
s_{\text{min}} &= (m_2 + m_3)^2 \\
s_{\text{max}} &= (Q - m_1)^2 \\
t_\pm &= \frac{-s^2 + (Q^2 + m_1^2 + m_2^2 + m_3^2)s - (M^2 - m_1^2)(m_2 - m_3)^2 \pm p_1 p_2}{2s} \\
p_1 &= \lambda^{1/2}(s, m_1^2, Q^2) \\
p_2 &= \lambda^{1/2}(s, m_2^2, m_3^2)
\end{aligned}$$

Here, $\lambda(a, b, c) = a^2 + b^2 + c^2 - 2ab - 2ac - 2bc$.

Integrating over phase-space

The interface for `ThreeBody` is similar to `Rambo`. To integrate over phase space, use `ThreeBody.integrate()`. The is an important difference between `ThreeBody` and `Rambo`: the matrix element *must* be a binary function taking the variables $s = (p_2 + p_3)^2$ and $t = (p_1 + p_3)^2$.

As an example, let's consider the muon decay again. To integrate over phase space, we use:

```
import numpy as np
from hazma import phase_space
from hazma.parameters import GF
from hazma.parameters import muon_mass as MMU

def msqrd(s, t):
    return 16.0 * GF**2 * t * (MMU**2 - t)

integral, error = phase_space.ThreeBody(MMU, [0.0, 0.0, 0.0], msqrd=msqrd).integrate()
width = integral / (2.0 * MMU)
error = error / (2.0 * MMU)
```

(continues on next page)

(continued from previous page)

```
analytic = GF**2 * MMU**5 / (192 * np.pi**3)
actual_error = abs(width - analytic)
print(f"analytic = {analytic:.8e}")
print(f"width = {width:.8e} +- {error:.2e}")
print(f"actual error = {actual_error:.2e} = {actual_error / analytic * 100:.2e} %")
# Output:
# analytic = 3.00917842e-16
# width = 3.00917842e-16 +- 6.68e-30
# actual error = 4.93e-32 = 1.64e-14 %
```

Energy and Invariant Mass Distributions

The interface for computing energy and invariant-mass distributions using `ThreeBody` is nearly identical to `Rambo`. You can simply construct your `ThreeBody` instance and call either `ThreeBody.energy_distributions` or `ThreeBody.invariant_mass_distributions`. The main difference is the signature of the `msqrd` parameter. As before, it must be a binary function accepting $s = (p_2 + p_3)^2$ and $t = (p_1 + p_3)^2$.

The `hazma.form_factors` module uses `ThreeBody` for the 3-body form factors. As an example of how to generate distributions, let's use the `hazma.form_factors.vector.VectorFormFactorPiKK0` to look at the energy distributions of the final state mesons.

```
import numpy as np
import matplotlib.pyplot as plt
from hazma import phase_space
import hazma.form_factors.vector as ffv


def msqrd(s, t, q, form_factor: ffv.VectorFormFactorPiKK0, gvuu, gvdd, gvss):
    ff = form_factor.form_factor(q, s, t, gvuu=gvuu, gvdd=gvdd, gvss=gvss)
    lor = form_factor.squared_lorentz_structure(q, s, t)
    return np.abs(ff) ** 2 * lor


# Specify parameters
gvuu, gvdd, gvss = 2.0 / 3.0, -1.0 / 3.0, -1.0 / 3.0
pk0_ff = ffv.VectorFormFactorPiKK0()
q = 1.5e3  # 1.5 GeV
# Construct `ThreeBody` object
tb = phase_space.ThreeBody(
    q,
    pk0_ff.fsp_masses,
    msqrd=lambda s, t: msqrd(s, t, q, pk0_ff, gvuu, gvdd, gvss),
)

# Construct distributions
energy_dists = tb.energy_distributions(nbins=100)
inv_mass_dists = tb.invariant_mass_distributions(nbins=100)

# Make plot
plt.figure(dpi=150, figsize=(9, 3))
```

(continues on next page)

(continued from previous page)

```
ax1 = plt.subplot(1, 2, 1)
ax2 = plt.subplot(1, 2, 2)

labels = [r"$K^0$", r"$K$", r"$\pi$"]
lss = ["-", "--", "-."]
for i, dist in enumerate(energy_dists):
   ax1.plot(dist.bin_centers, dist.probabilities, label=labels[i], ls=lss[i])
ax1.set_ylabel(r"$P(\epsilon)$", fontsize=16)
ax1.set_xlabel(r"$\epsilon$", fontsize=16)

labels = {
   (0, 1): r"$(K^0,K)$",
   (0, 2): r"$(K^0,\pi)$",
   (1, 2): r"$(K,\pi)$",
}
lss = {(0, 1): "-", (0, 2): "--", (1, 2): "-."}
for key, dist in inv_mass_dists.items():
   ax2.plot(dist.bin_centers, dist.probabilities, label=labels[key], ls=lss[key])
ax2.set_ylabel(r"$P(s_{ij})$", fontsize=16)
ax2.set_xlabel(r"$s_{ij}$", fontsize=16)

ax1.legend()
ax2.legend()
plt.tight_layout()
plt.show()
```

4.2 API Reference

CHAPTER

FIVE

FORM FACTORS (HAZMA.FORM_FACTORS)

5.1 Overview

hazma has several form factors for computing matrix elements into mesonic final states. Currently, hazma has all the important vector form factors relevant for center of mass energies below the τ mass. In a future release, we may include scalar form factors or form factors for other tensor interactions.

Below, we describe how to use the form factors to compute cross-sections, widths and distributions.

5.1.1 Vector Form Factors

Each of the form factor classes listed in the *class table* follow a particular structure. They all have the following functions:

Method	Description
`form_factor(q,..., **kwargs)`	Compute the form factor (without Lorentz structure).
`integrated_form_factor(q, **kwargs)`	Compute the squared current integrated over phase space.
`width(mv, **kwargs)`	Compute the partial decay width of a massive vector.
`cross_section(q, mx, mv, gvxx, wv, **kwargs)`	Compute the annihilation cross-section of dark matter.

`q` represents the center-of-mass energy in these functions. The `...` is a place-holder for additional arguments that depend on the type of form factor.

- **Two-body final states:** For two-body final states, `q` is the only needed argument (aside from additional arguments that depend on the specific final state.)
- **Three-body final states:** For three-body final states, the squared invariant masses $s = (p_2 + p_3)^2$ and $t = (p_1 + p_3)^2$ are required, where p_1, p_2 and p_3 are the momenta of particles 1, 2, and 3.
- **N>3-body final states:** Currently, the maximum number of final states is 4. However, for N-body final states with $N > 3$, the form factors require the full momentum information of the final states. In these cases, the signature is `form_factor(q, momenta, **kwargs)`, where `momenta` is a NumPy array containing the 4-momenta of all final state particles.

The `**kwargs` contains all the additional information needed for the specific final state (e.g. couplings.) All of these functions are vectorized, so you can pass an array of values to compute the quantity for each value at once.

The form factors are given as follows. Let $J^\mu = \mathcal{H} j^\mu 0$ be the hadronic current for the given final state $\mathcal{H}$. We

decompose the current into the form

$$J^\mu_{\mathcal{H}} = \sum_{a=1}^{N} F^a_{\mathcal{H}} j^{a,\mu}_{\mathcal{H}}$$

where $F^a_{\mathcal{H}}$ is the form factor associated with current $j^{a,\mu}_{\mathcal{H}}$. For all the two- and three-body final states, there is only one current (i.e. $N = 1$.) For these cases, the hadronic currents for final states containing two pseudo-scalars, a pseudo-scalar and photon, a pseudo-scalar and vector and three pseudo-scalars are given by:

$$\begin{aligned}
J^\mu_{P_1P_2} &= -(p_1^\mu - p_2^\mu)F_{P_1P_2}(q^2), \\
q &= p_1 + p_2 \\
J^\mu_{P\gamma} &= \epsilon_{\mu\nu\alpha\beta} q^\nu \epsilon^\alpha_\gamma(p_\gamma) p^\beta_\gamma F_{P\gamma}(q^2), \\
q &= p_P + p_\gamma \\
J^\mu_{PV} &= \epsilon_{\mu\nu\alpha\beta} q^\nu \epsilon^\alpha_V(p_V) p^\beta_P F_{PV}(q^2), \\
q &= p_P + p_V \\
J^\mu_{P_1P_2P_3} &= \epsilon_{\mu\nu\alpha\beta} p_1^\nu p_2^\alpha p_3^\beta F_{P_1P_2P_3}(s,t)
\end{aligned}$$

In the two-body form factors, the momentum q is given by the sum of the final state momenta. In the three pseudo-scalar form factor, $s = (p_2 + p_3)^2$ and $t = (p_1 + p_3)^2$.

In terms of these currents, the `form_factor` functions compute the $F_{\mathcal{H}}$. The `integrated_form_factor` functions compute the following:

$$\mathcal{J}_{\mathcal{H}}(q^2) = -\frac{1}{3q^2} g_{\mu\nu} \int \Pi_{\text{LIPS}} J^\mu_{\mathcal{H}} \bar{J}^\nu_{\mathcal{H}}$$

From the integrated current, the widths and cross-sections are easily computed. The expressions are:

$$\begin{aligned}
\sigma_{\bar{\chi}\chi\to\mathcal{H}}(q^2) &= \frac{g^2_{V\chi\chi}(q^2 + 2m_\chi^2)}{\sqrt{q^2 - 4m_\chi^2}((q^2 - m_V)^2 + m_V^2\Gamma_V^2)} \frac{\sqrt{q^2}}{2} \mathcal{J}_{\mathcal{H}}(q^2) \\
\Gamma_{V\to\mathcal{H}} &= \frac{m_V}{2} \mathcal{J}_{\mathcal{H}}(m_V^2)
\end{aligned}$$

5.1.2 Examples

Partial Widths of Vector

Listing 1: $V \to \pi^+\pi^-$

```
import hazma.form_factors.vector as ffv

# Compute the width V -> pi-pi for mv = 1 GeV with couplings of vector to
# quarks gvuu = 2/3 and gvdd=-1/3
ffv.VectorFormFactorPiPi().width(mv=1e3, gvuu=2.0/3.0, gvdd=-1.0/3.0)
# output: [17.303091110833093]
```

Listing 2: $V \to \pi^+K^-K^0$

```
import hazma.form_factors.vector as ffv

# Compute the width V -> pi-k-k0 for mv = 1.3 GeV with couplings of vector to
```

(continues on next page)

(continued from previous page)

```
# quarks gvuu = 2/3, gvdd=-1/3, gvss=-1/3
ffv.VectorFormFactorPiKK0().width(mv=1.3e3, gvuu=2.0/3.0, gvdd=-1.0/3.0, gvss=-1.0/3.0)
# output: 0.0004685155427290262 [MeV]
```

Listing 3: $V \to \pi^+\pi^-\pi^+\pi^-$

```
import hazma.form_factors.vector as ffv

# Compute the width V -> pi-k-k0 for mv = 1.3 GeV with couplings of vector to
# quarks gvuu = 2/3, gvdd=-1/3, gvss=-1/3
ffv.VectorFormFactorPiPiPiPi().width(mv=1.3e3, gvuu=2.0/3.0, gvdd=-1.0/3.0)
# output: 10.799920290416575 [MeV]
```

Dark Matter Annihilation

Listing 4: $\bar{\chi}\chi \to K^+K^-$

```
import hazma.form_factors.vector as ffv

mx = 300.0  # 300 MeV
q = 1.2e3   # 1.2 GeV
mv = 600.0  # 600 MeV
wv = 1.0    # 1 MeV
gvxx, gvuu, gvdd, gvss = 1.0, 2.0 / 3.0, -1.0 / 3.0, -1.0 / 3.0
ffv.VectorFormFactorKK().cross_section(
    q=q, mx=mx, mv=mv, gvxx=gvxx, wv=wv, gvuu=gvuu, gvdd=gvss, gvss=gvss
)
# output: 5.235577288150283e-09 [MeV^-2]
```

5.2 API

Class	Description
`VectorFormFactorPiPi()`	$\pi^+\pi^-$ form factor.
`VectorFormFactorPi0Pi0()`	$\pi^0\pi^0$ form factor.
`VectorFormFactorKK()`	K^+K^- form factor.
`VectorFormFactorK0K0()`	$K^0\bar{K}^0$ form factor.
`VectorFormFactorPi0Gamma()`	$\pi^0\gamma$ form factor.
`VectorFormFactorPi0Omega()`	$\pi^0\omega$ form factor.
`VectorFormFactorPi0Phi()`	$\pi^0\phi$ form factor.
`VectorFormFactorEtaGamma()`	$\eta\gamma$ form factor.
`VectorFormFactorEtaOmega()`	$\eta\omega$ form factor.
`VectorFormFactorEtaPhi()`	$\eta\phi$ form factor.
`VectorFormFactorPi0K0K0()`	$\pi^0 K^0\bar{K}^0$ form factor.
`VectorFormFactorPi0KpKm()`	$\pi^0 K^+K^-$ form factor.
`VectorFormFactorPiKK0()`	$\pi^+K^-K^0$ or $\pi^-K^+K^0$ form factor.
`VectorFormFactorPiPiEta()`	$\pi^+\pi^-\eta$ form factor.
`VectorFormFactorPiPiEtaPrime()`	$\pi^+\pi^-\eta'$ form factor.
`VectorFormFactorPiPiOmega()`	$\pi^+\pi^-\omega$ form factor.
`VectorFormFactorPi0Pi0Omega()`	$\pi^0\pi^0\omega$ form factor.
`VectorFormFactorPiPiPi0()`	$\pi^+\pi^-\pi^0$ form factor.
`VectorFormFactorPiPiPi0Pi0()`	$\pi^+\pi^-\pi^0\pi^0$ form factor.
`VectorFormFactorPiPiPiPi()`	$\pi^+\pi^-\pi^+\pi^-$ form factor.

CHAPTER

SIX

MODELS

6.1 Overview

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

6.1.1 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:

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

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 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 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

6.1.2 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 is:

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

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 $f_\pi \approx 93\ \mathrm{MeV}$ 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

SEVEN

LIMITS

7.1 Gamma ray limits

7.1.1 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 `TheoryGammaRayLimits` contains a method for using these to set limits. The second case is also handled by a method in `TheoryGammaRayLimits` which takes arguments specifying various detector and target characteristics.

7.1.2 Limits from existing data

7.1.3 Discovery reach for upcoming detectors

7.1.4 Containers for measurements, background models and target parameters

class hazma.flux_measurement.**FluxMeasurement**(*e_lows*, *e_highs*, *fluxes*, *upper_errors*, *lower_errors*, *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 ($\mathrm{MeV}^{-1}\mathrm{cm}^{-2}\mathrm{s}^{-1}\mathrm{sr}^{-1}$).

Type np.array

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

Type np.array

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

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__(*e_lows*, *e_highs*, *fluxes*, *upper_errors*, *lower_errors*, *energy_res*, *target*, *power=2*)
Constructor.

Parameters

- **fname** (*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. $E^2 d^2\Phi/dEd\Omega$ ($\mathrm{MeV}^{-1}\mathrm{cm}^{-2}\mathrm{s}^{-1}\mathrm{sr}^{-1}$).
  4. Upper error bar ($\mathrm{MeV}^{-1}\mathrm{cm}^{-2}\mathrm{s}^{-1}\mathrm{sr}^{-1}$).
  5. Lower error bar ($\mathrm{MeV}^{-1}\mathrm{cm}^{-2}\mathrm{s}^{-1}\mathrm{sr}^{-1}$).

  Note that the error bar values are their y-coordinates, not their relative distances from the central flux.
- **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 cm^-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 cm^-2 s^-1. For any energies outside of `self.e_range`, `np.nan` is returned.

Return type np.array

class hazma.gamma_ray_parameters.**TargetParams**(*J=None*, *D=None*, *dOmega=None*, *vx=0.001*)
Container for information about a target region.

Parameters

- **J** (*float*) – (Averaged) J-factor for DM annihilation in MeV^2 cm^-5.
- **D** (*float*) – (Averaged) D-factor for DM decay in MeV cm^-2.

- **dOmega** (*float*) – Angular size in sr.
- **vx** (*float*) – Average DM velocity in target in units of c. Defaults to 1e-3, the Milky Way velocity dispersion.

7.1.5 Observation regions

hazma.gamma_ray_parameters.**comptel_diffuse_targets = {'ein': TargetParams(J=1.751e+29, D=5.541e+25, dOmega=1.433e+00, vx=1.000e-03), 'nfw': TargetParams(J=9.308e+28, D=4.866e+25, dOmega=1.433e+00, vx=1.000e-03)}**

COMPTEL diffuse targets

hazma.gamma_ray_parameters.**comptel_diffuse_targets_optimistic = {'ein': TargetParams(J=1.040e+30, D=7.098e+25, dOmega=1.433e+00, vx=1.000e-03), 'nfw': TargetParams(J=1.530e+29, D=5.571e+25, dOmega=1.433e+00, vx=1.000e-03)}**

COMPTEL diffuse targets with optimistic parameters

hazma.gamma_ray_parameters.**egret_diffuse_targets = {'ein': TargetParams(J=6.994e+27, D=1.738e+25, dOmega=6.585e+00, vx=1.000e-03), 'nfw': TargetParams(J=6.265e+27, D=1.710e+25, dOmega=6.585e+00, vx=1.000e-03)}**

EGRET diffuse targets

hazma.gamma_ray_parameters.**egret_diffuse_targets_optimistic = {'ein': TargetParams(J=9.062e+27, D=1.952e+25, dOmega=6.585e+00, vx=1.000e-03), 'nfw': TargetParams(J=7.556e+27, D=1.761e+25, dOmega=6.585e+00, vx=1.000e-03)}**

EGRET diffuse targets with optimistic parameters

hazma.gamma_ray_parameters.**fermi_diffuse_targets = {'ein': TargetParams(J=1.058e+28, D=1.832e+25, dOmega=1.082e+01, vx=1.000e-03), 'nfw': TargetParams(J=8.475e+27, D=1.782e+25, dOmega=1.082e+01, vx=1.000e-03)}**

Fermi-LAT diffuse targets

hazma.gamma_ray_parameters.**fermi_diffuse_targets_optimistic = {'ein': TargetParams(J=1.601e+28, D=2.084e+25, dOmega=1.082e+01, vx=1.000e-03), 'nfw': TargetParams(J=1.106e+28, D=1.854e+25, dOmega=1.082e+01, vx=1.000e-03)}**

Fermi-LAT diffuse targets with optimistic parameters

hazma.gamma_ray_parameters.**integral_diffuse_targets = {'ein': TargetParams(J=4.166e+29, D=8.760e+25, dOmega=5.421e-01, vx=1.000e-03), 'nfw': TargetParams(J=2.086e+29, D=7.301e+25, dOmega=5.421e-01, vx=1.000e-03)}**

INTEGRAL diffuse targets

hazma.gamma_ray_parameters.**integral_diffuse_targets_optimistic = {'ein': TargetParams(J=4.166e+29, D=8.760e+25, dOmega=5.421e-01, vx=1.000e-03), 'nfw': TargetParams(J=2.086e+29, D=7.301e+25, dOmega=5.421e-01, vx=1.000e-03)}**

INTEGRAL diffuse targets with optimistic parameters

hazma.gamma_ray_parameters.**draco_targets = {'nfw': {'1 arcmin cone': TargetParams(J=3.418e+30, D=5.949e+25, dOmega=2.660e-07, vx=1.000e-03), '5 deg cone': TargetParams(J=8.058e+26, D=1.986e+24, dOmega=2.390e-02, vx=1.000e-03)}}**

Draco dwarf

hazma.gamma_ray_parameters.**m31_targets = {'nfw': {'1 arcmin cone': TargetParams(J=7.116e+29, D=9.449e+25, dOmega=2.660e-07, vx=1.000e-03), '5 deg cone': TargetParams(J=2.639e+26, D=5.507e+24, dOmega=2.390e-02, vx=1.000e-03)}}**

Andromeda target. See Sofue 2015, https://arxiv.org/abs/1504.05368

hazma.gamma_ray_parameters.**fornax_targets = {'nfw': {'1 arcmin cone': TargetParams(J=5.316e+29, D=2.898e+26, dOmega=2.660e-07, vx=1.000e-03), '2 deg cone': TargetParams(J=2.558e+26, D=9.081e+24, dOmega=3.830e-03, vx=1.000e-03)}}**
Fornax cluster. See https://arxiv.org/abs/1009.5988.

7.1.6 Effective Areas

hazma.gamma_ray_parameters.**effective_area_comptel**(*energy*)
Compute the effective area of the COMPTEL telescope [1]_.

Parameters **energy** (*array-like*) – Energy where the effective area should be evaluated.

Returns **a_eff** – Effective area Aeff(E).

Return type array-like

References

performance." NASA. Goddard Space Flight Center, The Compton Observatory Science Workshop. 1992.

hazma.gamma_ray_parameters.**effective_area_egret**(*energy*)
Compute the effective area of the EGRET telescope [1]_.

Parameters **energy** (*array-like*) – Energy where the effective area should be evaluated.

Returns **a_eff** – Effective area Aeff(E).

Return type array-like

References

galactic continuum gamma rays: a model compatible with EGRET data and cosmic-ray measurements." The Astrophysical Journal 613.2 (2004): 962.

hazma.gamma_ray_parameters.**effective_area_fermi**(*energy*)
Compute the effective area of the Fermi telescope [1]_.

Parameters **energy** (*array-like*) – Energy where the effective area should be evaluated.

Returns **a_eff** – Effective area Aeff(E).

Return type array-like

References

-ray emission: implications for cosmic rays and the interstellar medium." The Astrophysical Journal 750.1 (2012): 3.

hazma.gamma_ray_parameters.**effective_area_adept**(*energy*)
Compute the effective area of the proposed Advanced Energetic Pair Telescope (AdEPT) [1]_.

Parameters **energy** (*array-like*) – Energy where the effective area should be evaluated.

Returns **a_eff** – Effective area Aeff(E).

Return type array-like

References

Pair Telescope (AdEPT) for medium-energy gamma-ray astronomy." Space Telescopes and Instrumentation 2010: Ultraviolet to Gamma Ray. Vol. 7732. SPIE, 2010.

hazma.gamma_ray_parameters.**effective_area_all_sky_astrogam**(*energy*)
Compute the effective area of the proposed All-Sky-ASTROGAM telescope [1]_.

Parameters **energy** (*array-like*) – Energy where the effective area should be evaluated.

Returns **a_eff** – Effective area Aeff(E).

Return type array-like

References

Multimessenger Astrophysics." 36th International Cosmic Ray Conference (ICRC2019). Vol. 36. 2019.

hazma.gamma_ray_parameters.**effective_area_gecco**(*energy*)
Compute the effective area of proposed Galactic Explorer with a Coded Aperture Mask Compton Telescope (GECCO) [1]_.

Parameters **energy** (*array-like*) – Energy where the effective area should be evaluated.

Returns **a_eff** – Effective area Aeff(E).

Return type array-like

References

Mask and Compton Telescope: The Galactic Explorer with a Coded Aperture Mask Compton Telescope (GECCO)." arXiv preprint arXiv:2112.07190 (2021).

hazma.gamma_ray_parameters.**effective_area_grams**(*energy*)
Compute the effective area of the proposed Dual MeV Gamma-Ray and Dark Matter Observator (GRAMS) [1]_.

Parameters **energy** (*array-like*) – Energy where the effective area should be evaluated.

Returns **a_eff** – Effective area Aeff(E).

Return type array-like

References

observatory-GRAMS Project." Astroparticle Physics 114 (2020): 107-114.

hazma.gamma_ray_parameters.**effective_area_mast**(*energy*)
Compute the effective area of the proposed Massive Argon Space Telescope (MAST) [1]_.

Parameters **energy** (*array-like*) – Energy where the effective area should be evaluated.

Returns **a_eff** – Effective area Aeff(E).

Return type array-like

References

(MAST): A concept of heavy time projection chamber for -ray astronomy in the 100 MeV–1 TeV energy range." Astroparticle Physics 112 (2019): 1-7.

hazma.gamma_ray_parameters.**effective_area_pangu**(*energy*)
Compute the effective area of proposed PAir-productioN Gamma-ray Unit (PANGU) [1]_.

Parameters **energy** (*array-like*) – Energy where the effective area should be evaluated.

Returns **a_eff** – Effective area Aeff(E).

Return type array-like

References

telescope." Space Telescopes and Instrumentation 2014: Ultraviolet to Gamma Ray. Vol. 9144. International Society for Optics and Photonics, 2014.

7.1.7 Energy Resolutions

hazma.gamma_ray_parameters.**energy_res_adept**(*energy*)
Energy resolution for the AdEPT telescope [1]_ [2]_.

Note that the energy dependence fro AdEPT was not specified. We thus take it to be constant.

Parameters **energy** (*array-like*) – Energy where the energy resoltuon should be evaluated.

Returns **e_res** – Energy resoluton delta_e(e) / e.

Return type array-like

References

Pair Telescope (AdEPT) for medium-energy gamma-ray astronomy." Space Telescopes and Instrumentation 2010: Ultraviolet to Gamma Ray. Vol. 7732. SPIE, 2010.

medium-energy gamma-ray polarimetry." Astroparticle physics 59 (2014): 18-28.

hazma.gamma_ray_parameters.**energy_res_amego**(*energy*)
Compute the energy resolution of the All-sky Medium Energy Gamma-ray Observatory (AMEGO) [1]_.

Parameters **energy** (*array-like*) – Energy where the energy resoltuon should be evaluated.

Returns **e_res** – Energy resoluton delta_e(e) / e.

Return type array-like

References

exploring the extreme multimessenger universe." arXiv preprint arXiv:1907.07558 (2019).

hazma.gamma_ray_parameters.**energy_res_comptel**(*energy*)
Compute the energy resolution $\Delta E/E$ of the COMPTEL [1]_.

This is the most optimistic value, taken from chapter II, page 11 of [2]_. The energy resolution at 1 MeV is 10% (FWHM).

Parameters **energy** (*array-like*) – Energy where the energy resoltuon should be evaluated.

Returns **e_res** – Energy resoluton delta_e(e) / e.

Return type array-like

References

performance." NASA. Goddard Space Flight Center, The Compton Observatory Science Workshop. 1992.

gamma-ray spectrum from 800 keV to 30 MeV. University of New Hampshire, 1998.

hazma.gamma_ray_parameters.**energy_res_all_sky_astrogam**(*energy*)
Compute the energy resolution of the proposed All-Sky-ASTROGAM telescope [1]_.

Parameters **energy** (*array-like*) – Energy where the energy resoltuon should be evaluated.

Returns **e_res** – Energy resoluton delta_e(e) / e.

Return type array-like

References

Multimessenger Astrophysics." 36th International Cosmic Ray Conference (ICRC2019). Vol. 36. 2019.

hazma.gamma_ray_parameters.**energy_res_egret**(*energy*)
Compute the energy resoluton $\Delta E/E$ of the EGRET telescope [1]_.

This is the most optimistic value, taken from section 4.3.3 of [2]_.

Parameters **energy** (*array-like*) – Energy where the energy resoltuon should be evaluated.

Returns **e_res** – Energy resoluton delta_e(e) / e.

Return type array-like

References

galactic continuum gamma rays: a model compatible with EGRET data and cosmic-ray measurements." The Astrophysical Journal 613.2 (2004): 962.

experiment telescope (EGRET) for the Compton gamma-ray observatory." The astrophysical Journal supplement series 86 (1993): 629-656.

hazma.gamma_ray_parameters.**energy_res_fermi**(*energy*)
Compute the energy resolution $\Delta E/E$ of the Fermi-LAT telescope.

This is the average of the most optimistic normal and 60deg off-axis values from Fig. (18) of [2]_.

Parameters **energy** (*array-like*) – Energy where the energy resoltuon should be evaluated.

Returns **e_res** – Energy resoluton delta_e(e) / e.

Return type array-like

References

-ray emission: implications for cosmic rays and the interstellar medium." The Astrophysical Journal 750.1 (2012): 3.

gamma-ray space telescope mission." The Astrophysical Journal 697.2 (2009): 1071.

hazma.gamma_ray_parameters.**energy_res_gecco**(*energy*)
Compute the energy resolution $\Delta E/E$ of proposed Galactic Explorer with a Coded Aperture Mask Compton Telescope (GECCO) [1]_.

Parameters **energy** (*array-like*) – Energy where the energy resoltuon should be evaluated.

Returns **e_res** – Energy resoluton delta_e(e) / e.

Return type array-like

References

Mask and Compton Telescope: The Galactic Explorer with a Coded Aperture Mask Compton Telescope (GECCO)." arXiv preprint arXiv:2112.07190 (2021).

hazma.gamma_ray_parameters.**energy_res_grams**(*energy*)
Compute the energy resolution $\Delta E/E$ of GRAMS [1]_.

Parameters **energy** (*array-like*) – Energy where the energy resoltuon should be evaluated.

Returns **e_res** – Energy resoluton delta_e(e) / e.

Return type array-like

References

observatory-GRAMS Project." Astroparticle Physics 114 (2020): 107-114.

hazma.gamma_ray_parameters.**energy_res_integral**(*energy*)
Compute the energy resolution $\Delta E/E$ of INTEGRAL.

Parameters **energy** (*array-like*) – Energy where the energy resoltuon should be evaluated.

Returns **e_res** – Energy resoluton delta_e(e) / e.

Return type array-like

References

spectrometer on INTEGRAL as an indirect probe of cosmic-ray electrons and positrons." The Astrophysical Journal 739.1 (2011): 29.

hazma.gamma_ray_parameters.**energy_res_mast**(*energy*)
Compute the energy resolution $\Delta E/E$ of the proposed Massive Argon Space Telescope (MAST) [1]_.

Parameters **energy** (*array-like*) – Energy where the energy resoltuon should be evaluated.

Returns **e_res** – Energy resoluton delta_e(e) / e.

Return type array-like

References

(MAST): A concept of heavy time projection chamber for -ray astronomy in the 100 MeV–1 TeV energy range." Astroparticle Physics 112 (2019): 1-7.

hazma.gamma_ray_parameters.**energy_res_pangu**(*energy*)
Compute the energy resolution $\Delta E/E$ of proposed PAir-productioN Gamma-ray Unit (PANGU) [1]_.

Parameters **energy** (*array-like*) – Energy where the energy resoltuon should be evaluated.

Returns **e_res** – Energy resoluton delta_e(e) / e.

Return type array-like

References

Space Telescopes and Instrumentation 2014: Ultraviolet to Gamma Ray. Vol. 9144. International Society for Optics and Photonics, 2014.

7.2 CMB constraints

7.2.1 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.

7.2.2 Computing CMB limits

7.2.3 Functions and constants

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

EIGHT

UTILITIES (`HAZMA.UTILS`)

8.1 API Reference

CHAPTER

NINE

PARAMETERS (HAZMA.PARAMETERS)

hazma's collection of physical constants.

9.1 Masses

All masses are in MeV.

hazma.parameters.**higgs_mass: float = 125100.0**
Higgs mass in MeV

hazma.parameters.**electron_mass: float = 0.5109989461**
Electron mass in MeV

hazma.parameters.**muon_mass: float = 105.6583745**
Muon mass in MeV

hazma.parameters.**tau_mass: float = 1776.86**
Tau mass in MeV

hazma.parameters.**neutral_pion_mass: float = 134.9768**
Neutral pion mass in MeV

hazma.parameters.**charged_pion_mass: float = 139.57039**
Charged pion mass in MeV

hazma.parameters.**eta_mass: float = 547.862**
Eta mass in MeV

hazma.parameters.**eta_prime_mass: float = 957.78**
Eta' mass in MeV

hazma.parameters.**charged_kaon_mass: float = 493.677**
Charged kaon mass in MeV

hazma.parameters.**neutral_kaon_mass: float = 497.611**
Neutral kaon mass in MeV

hazma.parameters.**long_kaon_mass: float = 497.611**
Long kaon mass in MeV

hazma.parameters.**short_kaon_mass: float = 497.611**
Short kaon mass in MeV

hazma.parameters.**rho_mass: float = 775.26**
Rho mass in MeV

hazma.parameters.**omega_mass: float = 782.66**
Omega mass in MeV

hazma.parameters.**phi_mass: float = 1019.461**
Phi mass in MeV

hazma.parameters.**charged_B_mass: float = 5279.29**
Charged B mass in MeV

hazma.parameters.**pion_mass_chiral_limit: float = 137.273595**
Pion mass in MeV (chiral-limit)

hazma.parameters.**kaon_mass_chiral_limit: float = 495.644**
Kaon mass in MeV (chiral-limit)

hazma.parameters.**up_quark_mass: float = 2.16**
Up-quark mass in MeV

hazma.parameters.**down_quark_mass: float = 4.67**
Down-quark mass in MeV

hazma.parameters.**strange_quark_mass: float = 95.0**
Strange-quark mass in MeV

hazma.parameters.**charm_quark_mass: float = 1270.0**
Charm-quark mass in MeV

hazma.parameters.**bottom_quark_mass: float = 4180.0**
Bottom-quark mass in MeV

hazma.parameters.**top_quark_mass: float = 172900.0**
Top-quark mass in MeV

9.2 Conversion factors

hazma.parameters.**cm_to_inv_MeV = 50677307161.56396**
Convert cm to MeV^{-1}

hazma.parameters.**sv_inv_MeV_to_cm3_per_s = 1.1673299907789853e-11**
Convert σv from MeV^{-2} to cm^3/s

hazma.parameters.**g_to_MeV = 5.609588603804452e+26**
Convert grams to MeV

hazma.parameters.**MeV_to_g = 1.7826619216278976e-27**
Convert MeV to grams

hazma.parameters.**Msun_to_g = 1.988e+33**
Solar mass to grams

hazma.parameters.**g_to_Msun = 5.030181086519115e-34**
Gram to solar mass

9.3 Physical Constants

hazma.parameters.**alpha_em: float = 0.0072971395213076475**
Electromagnetic fine structure constant.

hazma.parameters.**GF: float = 1.1663787e-11**
Fermi constant in MeV**-2

hazma.parameters.**vh: float = 246227.95**
Higg vacuum expectation value in MeV

hazma.parameters.**qe: float = 0.30281770035689687**
e - Electromagnetic coupling constant

hazma.parameters.**temp_cmb_formation: float = 2.35e-07**
T_{CMB} - CMB temperature at formation in MeV

hazma.parameters.**plank_mass: float = 1.22091e+22**
M_{pl} - Plank mass in MeV

hazma.parameters.**rho_crit: float = 0.0105375**
ρ_{crit} - Critical energy density in units of h^2 $[\mathrm{MeV}/\mathrm{cm}^3]$

hazma.parameters.**sm_entropy_density_today: float = 2891.2**
s_0 - Entropy density today $[\mathrm{cm}^{-3}]$

hazma.parameters.**omega_h2_cdm: float = 0.1198**
$\Omega_{\mathrm{CMD}}h^2$ - Energy density fraction of dark matter times h^2

hazma.parameters.**dimensionless_hubble_constant: float = 0.6774**
$h = H_0/(100\mathrm{km/s/Mpc})$ - Hubble constant scale by 100 km/s/Mpc (Plank 2015)

hazma.parameters.**sin_theta_weak_sqrd: float = 0.2229**
$\sin^2\theta_W$ - Square of the sine of the Weinberg angle

hazma.parameters.**sin_theta_weak: float = 0.47212286536451503**
$\sin\theta_W$ - Sine of the Weinberg angle

hazma.parameters.**cos_theta_weak: float = 0.8815327560561774**
$\cos\theta_W$ - Cosine of the Weinberg angle

hazma.parameters.**Vud: complex = (0.974352121697273+0j)**
V_{ud} - up-down CKM matrix element

hazma.parameters.**Vus: complex = (0.22499846626868697+0j)**
V_{us} - up-strange CKM matrix element

hazma.parameters.**Vts: complex = (-0.041088514331340434-0.0007551822273694 55j)**
V_{ts} - top-strange CKM matrix element

hazma.parameters.**Vtb: complex = (0.9991185074624703+0j)**
V_{tb} - top-bottom CKM matrix element

hazma.parameters.**Vtd: complex = (0.007916871032258304-0.0032703040945479727j)**
V_{td} - top-down CKM matrix element

hazma.parameters.**Qu: float = 0.6666666666666666**
Q_u/e - Up-type quark charge in units of electron charge.

hazma.parameters.**Qd: float = -0.3333333333333333**
Q_d/e - Down-type quark charge in units of electron charge.

hazma.parameters.**Qe: float = -1.0**
Q_e/e - Charged lepton charge in units of electron charge.

hazma.parameters.**fpi0: float = 91.924**
f_{π^0} - Neutral pion decay constant.

hazma.parameters.**fpi: float = 92.2138**
$f_{\pi^\pm}$ - Charged pion decay constant.

hazma.parameters.**fk: float = 110.379**
$f_{K^\pm}$ - Charged kaon decay constant.

hazma.parameters.**b0: float = 2759.0102319508087**
$B_0 = \bar{q}q/(3f_\pi^2)$ - ChiPT mass parameter

CHAPTER

TEN

INDICES AND TABLES

- genindex
- search

INDEX

Symbols

A

B

C

D

E

V