scdc

Release 0.0

Yonit Hochberg, Eric David Kramer, Noah Kurinsky, and Benjamin

Aug 31, 2021

CONTENTS

scdc stands for *superconductor down-conversion*. This code simulates the relaxation of quasiparticle excitations produced by an energy deposit in a superconductor, with particular attention to energy deposits from dark matter scattering. The code was developed to study directional correlations between the initial and final states, and is bundled with some simple tools to quantify that relationship.

The basic structure of the calculation is as follows: an energy deposit produces initial quasiparticles and/or phonons. The quasiparticles can relax by emitting phonons, and sufficiently energetic phonons can decay to quasiparticle pairs. This process continues until all quasiparticles are too low-energy to emit a phonon, and all phonons are too low-energy to produce a quasiparticle pair. These excitations constitute the final state, and we say that they are *ballistic*. The full set of excitations connecting initial to final states is generated as a tree of python objects, so every step of the shower can be inspected directly.

The main non-triviality is that the kinematical equations governing phonons and quasiparticles cannot be solved analytically, so the code solves them numerically.

CHAPTER

ONE

CONTENTS

1.1 Kinematics

We now review the kinematics of phonons and quasiparticles as treated by the code. Note that we work in the "clean" limit, where crystal momentum is conserved.

1.1.1 Units

In the code, we work in "material" units, setting $v_\ell = k_F = \Delta = 1$. This implies that $\hbar = 2\gamma\sqrt{z}$, where $z \equiv E_F/\Delta$.

1.1.2 Dispersion relations

The phonon dispersion relation is $E = \hbar q$. This is invertible: a phonon's energy uniquely determines its momentum. The same is not true for a quasiparticle, which has dispersion relation

$$E = \left[1 + \left(\frac{\hbar^2 k^2}{2m_e} - z\right)^2\right]^{1/2}.$$

The inverse is requires a choice of sign:

$$k = \frac{1}{\hbar}\left[2m_e\left(z \pm \sqrt{E^2 - 1}\right)\right]^{1/2}.$$

This corresponds to the fact that the energy is minimized ($E = \Delta$) at the Fermi surface, $k = k_F$, and increases in either direction. In the code, *if a quasiparticle is instantiated with only energy specified, we choose this sign randomly.*

1.1.3 Phonon decay to quasiparticles

A phonon can decay to quasiparticles if it has enough energy to produce a quasiparticle pair, i.e., $E > 2\Delta$. Such a decay is characterized by the energy ω of one of the two quasiparticles (and the azimuthal angle, which is always uniform-random). We take the differential rate in this variable from eq. (27) of Kaplan et al. (1976):

$$\frac{\mathrm{d}\Gamma}{\mathrm{d}\omega} = \frac{1}{\sqrt{\omega^2 - \Delta^2}} \frac{\omega\,(E-\omega) + \Delta^2}{\sqrt{(E-\omega)^2 - \Delta^2}}.$$

There is a complication here, however, because this gives the differential rate as a function of quasiparticle *energy*. As we just discussed, the energy does not uniquely determine the momentum of a quasiparticle, which is essential to ensure that we satisfy conservation of momentum. Since the decay rate into each allowed final-state momentum should be equivalent, we *randomly* select the sign in the inverse dispersion relation for each quasiparticle.

But this introduces yet another complication: it is possible that one or more of these sign combinations is not kinematically allowed, in that they can produce decays with $|\cos\theta| > 1$. Thus, we first select ω according to the above distribution, and then check whether each sign combination is kinematically allowed. We randomly select from the allowed combinations. These are called *candidate* final states.

1.1.4 Phonon emission by quasiparticles

1.2 Structure of the code

1.2.1 Events and overall structure of the calculation

The structure of the calculation is as follows.

A scattering event produces two quasiparticles. Each of those quasiparticles can subsequently relax by emitting phonons. If sufficiently energetic, those phonons can themselves decay to quasiparticle pairs. The process continues until none of the products have enough energy to undergo another scatter or decay.

This picture describes a tree of particles with interactions as the nodes of the tree, and we use a very similar structure to implement the calculation programmatically. Our tree consists of three types of object:

- Event
- Interaction
- Particle

In tree language, *Event* objects are nodes and `Particle` objects are edges. `Interaction` objects couple a single *Event* object to a collection of `Particle` objects.

Why not just have `Interaction` and `Particle` objects? Because a third type (*Event*) is required to specify the relationship between the two. The `Interaction` types certainly need to know about `Particle` types in order to create the final state of each interaction. But then how would a `Particle` know about the `Interaction` types for which it can be an initial state? Alternatively, how would each `Interaction` type know about all the others? At a technical level, this would require a circular import. At a conceptual level, it indicates a mutual dependency between `Particle` and `Interaction` types that deserves its own container.

So instead, the structure is as follows. The nodes in the tree are *Event* objects. `Particle` objects are the edges between *Event* objects. The *Event* objects "know" what `Interaction` types are possible for a given `Particle`, and they use such an `Interaction` object internally to produce a new final state. The *Event* can then create new *Event* nodes at which to terminate each final-state `Particle`.

Each `Interaction` can also be physically prohibited. When this happens, the "initial" state of such an `Interaction` is a leaf of the tree.

class scdc.event.**Event**(*initial_state*, *material*, **args*, ***kwargs*)

Interaction tree node.

Parameters

- **initial_state** (`list` of `Particle`) – particles in the initial state. The initial state is always listlike, even if it contains only one object.
- **material** (`Material`) – the material in which this takes place.
- **final_state** (`list` of `Particle`, optional) – particles in the final state. This is passed to the internal `Interaction` object if supplied.

initial_state

particles in the initial state. The initial state is always a list, even if it contains only one object.

Type list of Particle

final_state
particles in the final state if and only if supplied at initialization.

Type list of Particle, optional

material (obj
Material): the material in which this takes place.

interaction
the interaction which can take place given the supplied initial state.

Type Interaction

out
particle *Event* objects.

Type list of Particle

final
True if no further interaction can take place.

Type bool

act()
Run the interaction and generate a final state.

chain()
Run the interaction and run particle interactions recursively.

property leaf_events
final events in the detector.

These are the leaves of the event tree, corresponding to the final states that will be measured.

Type list of *Event*

property leaf_particles
final particles after relaxation.

Type list of Particle

1.2.2 Particle objects

This module defines the different types of (quasi)particle.

class scdc.particle.**DarkMatter**(***kwargs*)
A dark matter particle.

dispersion(*p*)
Dispersion relation for dark matter.

Be careful: there is a need for conversion to the material's own canonical units. (NOT currently implemented.)

Parameters **p** (*float*) – momentum in who-knows-what units... to fix.

Returns dark matter energy in units of Delta.

Return type float

class scdc.particle.**Particle**(***kwargs*)

Represents a single phonon or quasiparticle with a tree of children.

momentum must be specified unless *dispersion_inverse* is defined, in which case either *momentum* or *energy* can be specified.

All quantities are specified in material units.

Parameters

- **energy** (*float, optional*) – energy in units of Delta.
- **momentum** (*float, optional*) – momentum in canonical units.
- **cos_theta** (*float*) – cosine of angle wrt global z.
- **material** (Material) – material properties.
- **pid** (*object, optional*) – optional ID label. Defaults to None.
- **tolerance** (*float, optional*) – tolerance for $\cos\theta > 1$. Defaults to 1e-2.

energy
energy in units of Delta.

Type float

momentum
momentum in canonical units.

Type float, optional

cos_theta
cosine of angle wrt global z.

Type float

material
material properties.

Type Material

pid
optional ID label.

Type object, optional

tolerance
tolerance for $\cos\theta > 1$.

Type float, optional

parent
parent particle.

Type *Particle*

origin
the originating *Event* for this particle.

Type Event

dest
the destination *Event* for this particle.

Type Event

shortname
a short label for this particle type.

Type str

dispersion(*k*)
Disperion relation: wavenumber to energy.

dispersion_inverse(*E*)
Inverse dispersion relation: energy to wavenumber.

property energy
energy in units of Delta.

Type float

property momentum
momentum in material units.

Type float

class scdc.particle.**ParticleCollection**(**args*, ***kwargs*)
Represents a collection of particles.

Parameters **particles** (list of *Particle*) – particles.

particles
particles.

Type list of *Particle*

energy
energy of each particle.

Type ndarray of float

momentum
momentum of each particle.

Type ndarray of float

cos_theta
cos_theta of each particle.

Type ndarray of float

property cos_theta
array of particle *cos_theta* values.

Type ndarray

property dm
subcollection of DM particles.

Type *ParticleCollection*

property energy
array of particle energies.

Type ndarray

classmethod from_npy(*npy*, *material*)
Construct a *ParticleCollection* from the numpy representation.

Parameters

- **npy** (ndarray) – a structured array with columns as specified in the *to_npy* method.

- **material** (Material) – the material to associate with these particles. The material is not stored in the numpy format.

Returns a corresponding *ParticleCollection*.

Return type *ParticleCollection*

property momentum
array of particle energies.

Type ndarray

property nondark
subcollection of non-DM particles.

Type *ParticleCollection*

property phonons
subcollection of phonons.

Type *ParticleCollection*

property quasiparticles
subcollection of quasiparticles.

Type *ParticleCollection*

select(*test*)
Select a subset of particles which satisfy a test.

The argument *test* is a function with call signature

test(parent, particle)

which returns *True* if the leaf *particle* in the chain arising from the initial particle *parent* satisifes the test.

Parameters **test** (*function*) – the test function.

Returns

collection consisting of particles which satisfy the test.

Return type *ParticleCollection*

to_npy()
Represent this particle collection in numpy format for export.

The numpy format is a structured array with the following columns:

shortname momentum cos_theta

The numpy form REQUIRES momenta for all particles.

Returns the array form of this collection.

Return type ndarray

class scdc.particle.**Phonon**(***kwargs*)
A phonon.

dispersion(*k*)
Dispersion relation for phonons.

Parameters **k** (*float*) – wavenumber in material units.

Returns phonon energy in material units.

Return type float

dispersion_inverse(*E*)
Inverse dispersion relation for phonons.

Parameters **E** (*float*) – phonon energy in material units.

Returns phonon wavenumber in material units.

Return type float

class scdc.particle.**Quasiparticle**(**kwargs*)
A Bogoliubov quasiparticle.

dispersion(*k*)
Dispersion relation for quasiparticles.

Parameters **k** (*float*) – wavenumber in material units.

Returns quasiparticle energy in material units.

Return type float

dispersion_inverse(*E*)
Inverse dispersion relation for quasiparticles.

Warning: the dispersion relation is not actually invertible. There are two possible choices of a sign. If the sign *s* is not specified, this method selects it randomly!

Parameters

- **E** (*float*) – quasiparticle energy in material units.
- **s** (*int, optional*) – sign in the momentum solution.

Returns quasiparticle wavenumber in material units.

Return type float

property ksign
the sign as determined from energy and momentum.

Type int

1.2.3 Interaction objects

This module defines the *Interaction* class and subclasses.

This is where most of the action lives. Particles propagate from interaction to interaction, and the interaction objects determine the final states that arise at each point. The differential rates, for instance, are used here.

class scdc.interaction.**DarkMatterScatter**(*args, **kwargs)
Represents an interaction in which DM scatters and creates two QPs.

This is a container, in the sense that no DM dynamics are actually contained here. Rather, at initialization, two quasiparticles must be supplied in the final state.

Parameters

- **qp1** (Quasiparticle) – final state quasiparticle 1.
- **qp2** (Quasiparticle) – final state quasiparticle 2.

allowed()
Determine whether the process is allowed for this initial state.

Returns *True* if the process can occur. Otherwise *False*.

Return type bool

class scdc.interaction.**Interaction**(*initial_state*, *material*, **args*, ***kwargs*)

Parent class for interactions.

An interaction combines an initial state, a material, and a final state. Subclasses must provide the means to go from the initial state to the final state.

Interactions are inherently treelike and can be treated as such. The complication is that the initial and final states are *Particle* objects, not *Interaction* objects. This necessitates the introduction of a new container type, *InteractionNode*, which specifies a particle and the interaction in which it participates.

Parameters

- **initial_state** (list of Particle) – particles in the initial state. The initial state is always a list, even if it contains only one object.
- **(obj** (*material*) – *Material*): the material in which this takes place.
- **n_bins** (*int, optional*) – number of bins to use for certain discretized calculations. Defaults to 100.
- **final_state** (list of Particle, optional) – particles in the final state. If supplied, the interaction is treated as trivial and the final state is never overwritten.
- **final** (*bool, optional*) – if *True*, the interaction is treated as trivial and the final state is left unpopulated and unvalidated. Defaults to *False*.

initial_state

particles in the initial state. The initial state is always a list, even if it contains only one object.

Type list of Particle

material (obj

Material): the material in which this takes place.

final

if *True*, the interaction is treated as trivial and the final state is left unpopulated and unvalidated.

Type bool, optional

n_initial

how many particles should be in the initial state.

Type int

n_final

how many particles should be in the final state.

Type int

initial

types of the initial particles, listed in order of their appearance.

Type tuple of *type*

final

types of the final particles, listed in order of their appearance.

Type tuple of *type*

ip

first of the initial particles.

Type Particle

fixed_final_state
True if the final state was supplied at initialization. In this case, the interaction is treated as trivial and the final state is never overwritten.

Type bool

allowed()
Determine whether the process is allowed for this initial state.

Returns *True* if the process can occur. Otherwise *False*.

Return type bool

interact()
Interact, validate outcome, and set final state.

If the final state was supplied at initialization, do nothing.

Returns the final state.

Return type list of Particle

classmethod valid_initial(*initial_state*)
Check that an initial state contains the correct particle types.

Parameters **initial_state** (list of Particle) – candidate initial state to validate.

Returns

***True* if the supplied initial state has the right particle** types for this interaction. Otherwise *False*.

Return type bool

class scdc.interaction.**PhononDecayToQuasiparticles**(**args*, ***kwargs*)
The *Interaction* for a phonon to decay to two quasiparticles.

Parameters **tolerance** (*float, optional*) – a numerical tolerance for root-finding. Defaults to 1e-3.

tolerance
a numerical tolerance for root-finding.

Type float, optional

allowed()
Determine whether the process is allowed.

Phonon decay can always take place if the energy of the phonon is at least 2*Delta.

Returns *True* if phonon decay can occur. Otherwise *False*.

Return type bool

decay_angles(*Eqp1*, *Eqp2*, *s1*, *s2*)
Computes the relative angles of two QPs created in phonon decay.

Parameters

- **Eqp1** (*float*) – energy of quasiparticle 1.
- **Eqp2** (*float*) – energy of quasiparticle 2.
- **s1** (*int*) – sign appearing in the inverse dispersion relation (QP1).
- **s2** (*int*) – sign appearing in the inverse dispersion relation (QP2).

Returns relative cos(theta) of quasiparticle 1. float: relative cos(theta) of quasiparticle 2.

Return type float

qp_energy_distribution(*eps=1e-10*)
Relative probability for a quasiparticle energy in phonon decay.

This is basically the integrand of eq. (27) of Kaplan+ 1976.

Parameters **eps** (*float, optional*) – slight offset to prevent divide-by-zero errors. Defaults to 1e-10.

Returns

QP energy bin centers. ndarray of float: relative rate for each bin center,

normalized to unity.

Return type ndarray of float

class scdc.interaction.**QuasiparticlePhononEmission**(**args*, ***kwargs*)
The *Interaction* for a quasiparticle to emit a phonon.

Parameters **tolerance** (*float, optional*) – a numerical tolerance for root-finding. Defaults to 1e-3.

tolerance
a numerical tolerance for root-finding.

Type float, optional

allowed(*exact=False*)
Determine whether the process is allowed for this initial state.

Returns *True* if the process can occur. Otherwise *False*.

Return type bool

final_state_angles(*phonon_energy*, *sp*)
Cosine of angle of final state particles after QP scattering.

Both angles are computed relative to the direction of the proximal quasiparticle just before scattering, not necessarily the initial quasiparticle.

DEBUGGING NOTE: this follows section "Better" of *dispersion.nb*.

Parameters

- **phonon_energy** (*float*) – phonon energy in units of Delta.
- **qp_energy** (*float*) – Initial quasiparticle energy in units of Delta.
- **sp** (list of int) – momentum sign for final quasiparticle.

Returns cosine of the angle of the emitted quasiparticle. float: cosine of the angle of the emitted phonon.

Return type float

min_cos_deflection()
Max angular deflection, expressed as minimum of cos(theta).

For a given QP energy, there is a maximum amount of momentum it can shed via phonon emission. This produces an upper bound on the angular deflection of the QP. We express this in the form of a lower bound on the cosine of the deflection angle.

Returns min of cos(theta).

Return type float

phonon_energy_distribution()

Relative phonon emission probability by a quasiparticle.

Computation follows dGamma/dx in eq. (21) of Noah's note. We normalize directly. If the distribution cannot be normalized (all values vanish), both return values are *None*.

Returns

phonon energy bin centers. `ndarray` of `float`: relative rate for each bin center,

normalized to unity.

Return type `ndarray` of `float`

Raises **NonPhysicalException** – If the max phonon energy is not positive.

phonon_energy_region()

Min / max allowed phonon energy for emission by a quasiparticle.

The min and max energies are determined by finding the energy range in which the final-state angles are physical, i.e.

$|\cos(\theta)| < 1.$

This needs to be done for both the quasiparticle and phonon angle. Moreover, since there is a sign ambiguity in the quasiparticle momentum, this needs to be done for each sign independently to get self-consistent regions that work for both the quasiparticle angle and the phonon angle.

Thus, we take the following approach. For each sign in turn, we find the physical region for each of the two angles. The intersection of these two regions is what we want. But then we have to do this for each sign, and take the union of the resulting intersections.

In principle, this may be disjoint. For the moment, rather than significantly change the architecture of the code, I will simply throw an error if the regions are disjoint.

Parameters **qp_energy** (*float*) – Initial quasiparticle energy in units of Delta.

Returns minimum emitted phonon energy in units of Delta. float: maximum emitted phonon energy in units of Delta.

Return type float

1.2.4 Material objects

This module defines a *Material* class.

A note on units: at various places in the code, it is useful to have work with non-dimensionalized quantities. However, picking these quantities requires some care, because there are two sets of scales: those associated with behavior near and far from the gap. We are mainly interested in near-gap behavior, so we use the half-gap Delta as the energy scale. Now, excitations with energy Delta have momentum k_F, so it makes sense to take k_F as our preferred momentum scale. So far we have

Delta = k_F = 1.

Having made this choice, we automatically obtain a velocity v_m and a mass m_m as follows:

v_m = Delta/k_F = 1, m_m = k_F/v_m = k_F^2/Delta = 1.

These parameters do NOT correspond to obvious scales in the material. In particular, v_m is distinct from the sound speed c_s, and m_m is distinct from the carrier mass m_star. In other words, c_s_m != 1 and m_star_m != 1.

We still need to be able to express a unit of length in order to put hbar into these units. So far, however, time and length always appear in ratio. It seems, then, that the natural thing to do is to simply set

hbar = 1.

The natural units of time and length are then

t_m = hbar/Delta = 1, l_m = hbar/Delta*v_m = hbar*k_F/Delta^2 = 1.

It is useful to distinguish quantities in ordinary units from those in material units. To that end, we will append "_m" to the names of quantities that are defined in material units.

For uniformity, ALL quantities not in material units are assumed to be in units of an appropriate power of eV with c=hbar=kB=1.

class scdc.material.**Material**(***kwargs*)

Container class for material properties.

Parameters

- **symbol** (*str*) – chemical symbol.
- **gamma** (*float*) – gamma parameter.
- **c_s** (*float*) – speed of sound in m/s.
- **T_c** (*float*) – critical temperature in K.
- **Delta** (*float*) – half of the gap in eV.
- **E_F** (*float*) – Fermi energy in eV.
- **m_star** (*float*) – effective mass? in units of m_e.
- **beta** (*float*) – beta.

symbol

chemical symbol.

Type str

gamma

gamma parameter.

Type float

c_s

speed of sound in m/s.

Type float

T_c

critical temperature in K.

Type float

Delta

half of the gap.

Type float

E_F

Fermi energy in eV.

Type float

m_star

effective mass? in units of m_e.

Type float

z
Fermi energy in units of Delta.

Type float

beta
beta.

Type float

mcs2
m*c_s^2 in units of eV.

Type float

coherence_uvvu(*s*, *k1*, *k2*)
The coherence factor (uv' +/- vu')^2. The sign is *s*.

Note that *k1* and *k2* are assumed to be in material units (i.e. in units of k_F).

Parameters

- **s** (*int*) – sign that appears in the coherence factor (1 or -1).
- **k1** (*float*) – momentum of quasiparticle 1 in material units.
- **k2** (*float*) – momentum of quasiparticle 2 in material units.

Returns the coherence factor (uv' +/- vu')^2.

Return type float

epsilon_lindhard(*q*, *omega*)
Lindhard dielectric function.

info()
Material data in a json-serializable format.

Returns material info.

Return type dict

qpe(*k*)
Compute the energy of a quasiparticle of momentum k.

This is a convenience method that is identical to the `Quasiparticle` dispersion relation. In fact, the two might be merged later on...

Parameters **k** – QP momentum

Returns QP energy in material units

Return type float

1.2.5 Initial distribution

This module defines the distribution of the initial excitations in angles and momenta, as a function of the scattering matrix element. Unlike prior implementations using MCMC methods, here the sampling is carried by direct integration of the distribution.

class scdc.initial.distribution.integral.**InitialSampler**(**args*, ***kwargs*)
Sampler for parameter distribution of initial excitations.

Here's the general strategy:

1. **Start by computing the total rate for each of many speeds.**

- Do this by computing the total rate on a sparse grid of speeds and interpolating.
- The total rate calculation at each speed requires that we calculate the rate at many values in the (cq, rq) plane. Store these values for later.

2. Use these together with the marginal distribution of speeds in the halo to sample a set of DM speeds.
3. For each DM speed, interpolate the rates between different slices of the (cq, rq) plane, corresponding to different discrete speeds.
   - This gives an interpolated set of rates in the (cq, rq) plane at the speed of interest.
4. Interpolate these rates to sample (cq, rq).
5. **Determine two remaining kinematical variables.**
   - The azimuthal angle of the final states with respect to the momentum transfer can be sampled uniformly.
   - The angle of the incoming DM with respect to the DM wind axis has a well-determined marginal distribution at fixed DM speed.

Args:

Attributes:

Raises **NotAllowedException** – DM mass is below threshold for scattering for all available velocities.

ensemble(**args*, ***kwargs*)
Sample an ensemble of DM scatters.

Parameters **n_samples** (*int*) – number of samples to draw.

Returns an ensemble of `DarkMatterScatter` events.

Return type `Ensemble`

q_rate_grid(*r1*, *order=20*)
Compute a grid of rates at fixed (c_q, r_q).

Parameters

- **r1** (*float*) – fixed r_1 value.
- **order** (*int, optional*) – fixed order to use for Gaussian quadrature.
- **threshold** (*float, optional*) – the fraction of the integrated rate that should be preserved when doing cuts. Note that cuts are repeated `n_cuts` times. Defaults to 0.99.

Returns 1d array of unique c_q values. ndarray: 1d array of unique r_q values. ndarray: 2d array of rate values.

Return type ndarray

Raises

- **ValueError** – if all rates are zero.
- **ValueError** – if r_1 is too small for any scattering to be kinematically allowed.

q_to_p(*r1*, *rq*, *r3*, *c1*, *cq*)
Transform momentum-transfer samples to get the final-state momenta.

Note that this method also samples azimuthal angles for the quasiparticles with respect to the momentum transfer. The three input arrays must have the same length.

Parameters

- **r1** (*ndarray*) – r_1 values.
- **rq** (*ndarray*) – r_q values.
- **r3** (*ndarray*) – r_3 values.
- **c1** (*ndarray*) – c_1 values.
- **cq** (*ndarray*) – $\cos\theta_q$ values.

Returns

an array of samples of shape (len(cq), 4). These samples have the form (r_3, r_4, c_3, c_4).

Return type ndarray

sample(*n_samples*, *max_attempts=None*)

Keeps drawing samples until we get the desired number.

This method wraps `_sample` and has identical arguments and returns, except for one additional argument that specifies the max number of attempts.

Parameters

- **n_samples** (*int*) – number of samples to draw.
- **max_attempts** (*int*) – max number of calls to `_sample`. If `None`, never give up. Defaults to `None`.

Returns r_1 values. ndarray: r_2 values. ndarray: r_3 values. ndarray: r_4 values. ndarray: c_1 values. ndarray: c_2 values. ndarray: c_3 values. ndarray: c_4 values.

Return type ndarray

samples_to_ensemble(*r1*, *r2*, *r3*, *r4*, *c1*, *c2*, *c3*, *c4*)

Convert sampled final-state momenta to an `Ensemble`.

All input arrays must have the same shape.

Parameters

- **r1** (*ndarray*) – sampled r_1 values.
- **r1** – sampled r_2 values.
- **r3** (*ndarray*) – sampled r_3 values.
- **r4** (*ndarray*) – sampled r_4 values.
- **c1** (*ndarray*) – sampled c_1 values.
- **c1** – sampled c_2 values.
- **c3** (*ndarray*) – sampled c_3 values.
- **c4** (*ndarray*) – sampled c_4 values.

Returns an ensemble of `DarkMatterScatter` events.

Return type `Ensemble`

class scdc.initial.distribution.integral.**RateIntegrator**(*m1*, *matrix_element*, *material*, *response*, *vdf*, ***kwargs*)

Base class for integrating rates

Parameters

- **m1** (*float*) – mass of the incoming particle.
- **matrix_element** (*callable*) – the matrix element for the scattering process, as a function of the magnitude of the 3-momentum transfer. A `ScatteringMatrixElement` object can be supplied here.
- **material** (`Material`) – a material object.
- **response** (*callable*) – a function of q and ω characterizing material response.
- **vdf** (*callable*) – a probability distribution for the velocity as a function of zero, one, or two parameters. The number of arguments must match the length of the argument `v_args` to `likelihood`.

m1
mass of the incoming particle.

Type float

matrix_element
the matrix element for the scattering process, as a function of the magnitude of the 3-momentum transfer. A `ScatteringMatrixElement` object can be supplied here.

Type callable

material
a material object.

Type `Material`

response
a function of q and ω characterizing material response.

Type callable

vdf
a probability distribution for the velocity as a function of zero, one, or two parameters. The number of arguments must match the length of the argument `v_args` to `likelihood`.

Type callable

pdf (*r1*, *rq*, *cq*, *r3*)
Compute the differential rate summing over sign choices.

Parameters

- **r1** (*float*) – r_1.
- **rq** (*float*) – r_q.
- **cq** (*float*) – c_q.
- **r3** (*float*) – r_3.

Returns differential rate (non-normalized).

Return type float

pdf_fixed_sign (*r1*, *rq*, *cq*, *r3*, *s*)
Compute the pdf for fixed sign s.

Parameters

- **r1** (*float*) – r_1.
- **rq** (*float*) – r_q.

- **cq** (*float*) – c_q.
- **r3** (*float*) – r_3.
- **s** (*float*) – the sign s selecting between the two solutions for the second quasiparticle's momentum.

Returns differential probability.

Return type float

Raises **ValueError** – if $r_q = 0$.

q_rate(*r1*, *rq*, *cq*, *order=None*)

Integrated rate at fixed r_1 and q.

Parameters

- **r1** (*float*) – r_1.
- **rq** (*float*) – r_q.
- **cq** (*float*) – c_q.

Returns differential rate (non-normalized).

Return type float

r3_domain(*r1*, *rq*, *cq*)

Compute the minimum and maximum allowed values of r_3.

Parameters

- **r1** (*float*) – r_1.
- **rq** (*float*) – r_q.
- **cq** (*float*) – c_q.

Returns lower bound on r_3. float: upper bound on r_3.

Return type float

1.3 Analysis tools

1.3.1 Analysis functions

This module defines functions for analysis of simulated ensembles.

Many of these are 'old', in that they were written at one time for a type of analysis that has not been used since. However, they are preserved here for future applications.

scdc.analyze.**norm_asymmetry**(*angles*, *distance_function=<function p_dist.<locals>._dist>*, *n_bins=50*)

Find the asymmetry as the distance from an isotropic distribution.

For any norm, this will give zero for perfectly isotropic scattering. The maximum depends on the norm. For the default L^1 norm, if the scattering is purely directional, corresponding to a delta function, the difference will be 0.5 over the whole interval and then the delta function integrates to 1, so that'll be a maximum of 2. For the L^2 norm, the norm of the delta is not well defined.

** However, because of the default L^1 behavior, the result is divided by 2 regardless of the distance function. **

Because we're working with a histogram anyway, we don't want to use any actual quadrature. Instead, we want to use a p-norm of some kind. So the distance function here is not really a norm, but a single-point integrand thereof. For example, to use an L^2 norm, the distance function should be

lambda x: np.abs(x)**2

and the rest will be taken care of internally.

Parameters

- **angles** (ndarray) – cos(theta) values. A *ParticleCollection* object can be provided instead.
- **distance_function** (*function, optional*) – a function of one variable giving the integrand of the norm. Defaults to *p_dist(1)*.
- **n_bins** (*int, optional*) – number of bins to use for the norm. Defaults to 50.

Returns norm/2 of the distance from an isotropic distribution.

Return type float

scdc.analyze.**p_dist**(*p*)
Factory for L^p-norm integrands.

Parameters **p** (*float*) – p for the L^p norm.

Returns L^p norm integrand as a function of one argument.

Return type function

scdc.analyze.**plane_asymmetry**(*angles*, *n_bins=100*, *width=1*)
Find the asymmetry in a sliding bin of fixed width in cos(theta).

For cos(theta) = 1, this corresponds to forward-backward asymmetry.

Parameters

- **angles** (ndarray) – cos(theta) values. A *ParticleCollection* object can be provided instead.
- **width** (*float, optional*) – width of the sliding bin. Defaults to 1.

scdc.analyze.**qp_angle_pairs**(*event*)
Final-state QP angles in canonical pairs.

Here 'canonical' pairing means the following. The number of quasiparticles produced in any event must be even, so we sort them by energy and then divide into a low-energy half and a high-energy half. The lowest low-energy QP is paired with the lowest high-energy QP, the second-lowest with the second-lowest, and so on. There is nothing important about this order except that it is well-defined.

Parameters **event** (Event) – the event for which to find final-state QP pairs.

Returns a 2d array in which each row is a pair.

Return type ndarray

1.3.2 Plotting functions

This module defines plotting styles and routines. Some are out of date but are retained for possible future use.

scdc.plot.**latex_exp_format**(*x*, *mfmt='%.3f'*)
Format a number in scientific notation in LaTeX.

Parameters

- **x** (*float*) – number to format.
- **mfmt** (*str, optional*) – format string for mantissa. Defaults to '%.3f'.

Returns LaTeX string (no $'s).

Return type str

scdc.plot.**tree_plot**(*particle*, *origin=(0, 0)*, ***kwargs*)
Plot all child scattering events.

Parameters

- **fig** (Figure, optional) – matplotlib figure object.
- **ax** (AxesSubplot, optional) – matplotlib axis object.
- **origin** (tuple of float, optional) – starting coordinates for the tree.
- **dm_color** (*str, optional*) – color for DM lines.
- **phonon_color** (*str, optional*) – color for phonon lines.
- **qp_color** (*str, optional*) – color for quasiparticle lines.
- **min_linewidth** (*float, optional*) – smallest linewidth (E = Delta).
- **max_linewidth** (*float, optional*) – largest linewidth, corresponding to the energy of the initial excitation.
- **max_linewidth_energy** (*float, optional*) – energy for max linewidth.
- **final_distance** (*float, optional*) – if specified, final (ballistic) state lines will be extended to this distance from (0, 0).
- **alpha** (*float, optional*) – opacity.

1.4 Using the code

1.4.1 Contents

Batch interface

MPI capabilities

This module defines a custom MPI scatter-gather manager. It can be used independently of the remainder of the code.

This module enables parallelized down-conversion using MPI.

scdc.mpi.sim.**particle_as_dict**(*p*)
Convert a Particle object to a lightweight dict.

The dictionary form has keys 'shortname', 'momentum', and 'cos_theta'.

Parameters **p** (Particle) – particle to convert to a dict.

Returns a simple dictionary form of the particle.

Return type dict

scdc.mpi.sim.**particle_from_dict**(*d*, *material*)
Convert the output of *particle_as_dict* back to an object.

Parameters

- **d** (*dict*) – dictionary to convert to a particle.
- **material** (Material) – the material to use for this particle. Material data is not included in the dict representation.

Returns an object form of the dict.

Return type Particle

This module enables parallelized initial-QP sampling using MPI.

Command-line interface

This module defines utilities for running ensembles through the command line or batch queue.

Runs are defined by a configuration file in json format. The json file should have the following keys:

outfile: path to the output file. Extension will be appended if missing. copies: number of copies of the initial state to run. Defaults to 1. initial: initial state to use.

- **If a string, this is taken to be the path to a file. The file should** have columns of the form

  pDM cDM p1 c1 p2 c2

  where *p* and *c* are the momentum and cos(theta) for the DM, QP1, and QP2, respectively.

- **If a dict, this is taken to specify the parameters of an initial**

  state. In this case, the allowed keys are:

  - *momentum*
  - *energy*
  - *shortname*
  - *cos_theta*

  Exactly one of *energy* and *momentum* must be specified. Any of these keys may be given as a list of values. In this case, one enemble is produced for each value in the list (or for each combination of list values, if applicable).

material: material data to use (dict).

- If omitted or null, the default is *materials.Aluminum*.
- **Allowed keys are:**
  - *gamma*
  - *c_s*
  - *T_c*
  - *Delta*

- *E_F*
- *m_star*
- *beta*

- Any keys omitted are taken from the default (*materials.Aluminum*).
- **As with the *initial* key, any values specified as lists will produce** one ensemble for each value or combination of values.

params: an arbitrary dict of additional labels with primitive values.

class scdc.interface.**Configuration**(***kwargs*)

Configuration generator for multi-task downconversion runs.

Parameters

- **outfile** (*str*) – path to the output file. Extension will be appended if missing.
- **copies** (*int, optional*) – number of copies to run. Defaults to 1.
- **initial** (*object*) – initial particle specification.
- **n_initial** (*int or dict, optional*) – number of initial particles to select for downconversion. Selection is random with replacement. If `None`, use all initial particles. A two-layer dict may be provided using the mediator mass and DM mass as keys, with an 'other' entry as a fallback. Defaults to `None`.
- **material** (*object*) – material specification.
- **params** (*dict, optional*) – arbitrary parameter dict for labels. Defaults to *{}*.

outfile
original outfile specification.

Type str

copies
original copies specification.

Type int

material
original material specification.

Type object

params
original params specification.

Type dict

materials
material specifications.

Type object

ensemble_tasks
one task per ensemble.

Type `list` of *EnsembleTask*

task_by_id
a dictionary mapping ensemble task IDs to *EnsembleTask* objects.

Type dict

save()
Save an HDF5 representation of this run and the output.

class scdc.interface.**EnsembleTask**(**kwargs)
Container class for single ensembles.

Parameters

- **initial** (list of Event) – initial events.
- **material** (Material) – material object.
- **params** (*dict, optional*) – arbitrary parameter dict for labels. Defaults to *{}*.
- **task_id** (*object, optional*) – a label for this task. Defaults to *None*.

initial
initial events.

Type list of Event

material
material object.

Type Material

params
arbitrary parameter dict for labels.

Type dict

task_id
a label for this task.

Type object

result
results of running this task.

Type ndarray

class scdc.interface.**InitialConfiguration**(**kwargs)
Configuration generator for multi-task initial-particle runs.

At present, it is assumed that only the DM and mediator mass vary. The velocity is fixed to 1e-3.

save()
Save an HDF5 representation of this run and the output.

class scdc.interface.**InitialTask**(*args, **kwargs)
Container class for single initial-ensemble generation tasks.

At present, it is assumed that only the DM and mediator mass vary. The velocity is fixed to 1e-3.

scdc.interface.**expand_dict**(*d*)
Expand a dict of lists to a list of dicts.

The idea is to take a dict for which some values are iterable, and convert this to a single list of dicts, each of which has no listlike values.

Parameters **d** (*dict*) – dict to expand.

Returns expanded list of dicts.

Return type list of dict

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

S

INDEX

K

L

M

N

O

P

Q

R

S

T

V

Z