Interface functions

Martijn Wehrens

May 13, 2011

Contents

1 Introduction

This document gives an overview of functions that are usefull to end-users of the enhanced Green's Functions Reaction Dynamics (eGFRD) algorithm. In other words, it lists the "interface functions". Basically, this entails a list of function definition plus their so-called python "docstrings", obtained straight from the code. Note that functions starting with an underscore are actually C++ functions ported to python by boost.

2 Functions

2.1 Functions from `model.py`

2.1.1 Core functions:

As mentioned, all documentation in this document comes straight from the code. To obtain more information on how to structure a simple script, one could take a look into the sample directory.

The class you need to set up a particle model:

```
class ParticleModel(_gfrd.Model):
    """
    """
    def __init__(self, world_size):
        """Create a new ParticleModel.

        Arguments:
            - world_size
                the size of one side of the simulation "world". Units:
                meters.

        The simulation "world" is always assumed to be a cube with
        *periodic boundary conditions*, with 1 corner at [0, 0, 0] and
        the corner furthest away from [0, 0, 0] being at
        [world_size, world_size, world_size].

        """
```

```
def Species(name, D, radius=0, structure="world", drift=0):
    """Define a new Species (in/on a specific Region or Surface).

    Arguments:
        - name
            the name of this Species.
        - D
            the diffusion constant for this Species in/on this
            Region or Surface. Units: meters^2/second.
        - radius
            the radius for this Species in/on this Region or Surface.
            Units: meters.
        - structure
            the Region or Surface in/on which this Species can exist.
            Optional. If you do not specify a Structure the Species is
            added to the "world".
        - drift
            the drift term for this ParticleType on a
            CylindricalSurface (1D drift). Units: meters/second.
            Optional.

    If a certain Species should be able to exist in the "world" as
    well as in/on one of the previously created Regions or Surfaces,
    then two distinct Species should be created. One with and one
    without an explicit Structure argument.

    """
```

Species should be added to the model:

```
    def add_reaction_rule(self, reaction_rule):
        """Add a ReactionRule to the ParticleModel.

        Argument:
            - reaction rule
                a ReactionRule created by one of the functions
                model.create_<>_reaction_rule.

        """
```

2.1.2 Creating regions

```
_gfrd.create_cuboidal_region.__doc__ = \
"""create_cuboidal_region(id, corner, diagonal)

Create and return a new cuboidal Region.

Arguments:
    - id
        a descriptive name.
    - corner
        the point [x, y, z] of the cuboidal Region closest to
        [0, 0, 0]. Units: [meters, meters, meters]
    - diagonal
        the vector [x, y, z] from the corner closest to [0, 0, 0], to
        the corner furthest away from [0, 0, 0]. Units:
        [meters, meters, meters]

"""

_gfrd.create_cylindrical_surface.__doc__ = \
"""create_cylindrical_surface(id, corner, radius, orientation, length)

Create and return a new cylindrical Surface.

Arguments:
    - id
        a descriptive name.
    - corner
        the point [x, y, z] on the axis of the cylinder closest to
        [0, 0, 0]. Units: [meters, meters, meters]
    - radius
        the radius of the cylinder. Units: meters.
    - orientation
        the unit vector [1, 0, 0], [0, 1, 0] or [0, 0, 1] along the
        axis of the cylinder.
    - length
        the length of the cylinder. Should be equal to the world_size.
        Units: meters.

Surfaces are not allowed to touch or overlap.

"""

_gfrd.create_planar_surface.__doc__ = \
"""create_planar_surface(id, corner, unit_x, unit_y, length_x, length_y)

Create and return a new planar Surface.

Arguments:
    - id
        a descriptive name.
    - corner
        the point [x, y, z] on the plane closest to [0, 0, 0]. Units:
        [meters, meters, meters]
    - unit_x
        a unit vector [1, 0, 0], [0, 1, 0] or [0, 0, 1] along the
        plane.
    - unit_y
        a unit vector [1, 0, 0], [0, 1, 0] or [0, 0, 1] along the plane
        and perpendicular to unit_x.
    - length_x
        the length of the plane along the unit vector unit_x. Should be
        equal to the world_size. Units: meters.
    - length_y
        the length of the plane along the unit vector unit_y. Should be
        equal to the world_size. Units: meters.

Surfaces are not allowed to touch or overlap.

"""
```

As particles, regions should be added to the model.

```
    def add_structure(self, structure):
        """Add a Structure (Region or Surface) to the ParticleModel.

        Arguments:
            - structure
              a Region or Surface created with one of the functions
              model.create_<>_region or model.create_<>_surface.

        """
        assert isinstance(structure, _gfrd.Structure)
        self.structures[structure.id] = structure
        return structure
```

2.1.3 Adding reaction rules to the model

Function set_all_repulsive is called automatically, but it's docstring is instructive.

```
    def set_all_repulsive(self):
        """Set all 'other' possible ReactionRules to be repulsive.

        By default an EGFRDSimulator will assume:
            - a repulsive bimolecular reaction rule (k=0) for each
              possible combination of reactants for which no
              bimolecular reaction rule is specified.

        This method explicitly adds these ReactionRules to the
        ParticleModel.

        """

def create_unimolecular_reaction_rule(reactant, product, k):
    """Example: A -> B.

    Arguments:
        - reactant
            a Species.
        - product
```

```
            a Species.
        - k
            reaction rate. Units: per second. (Rough order of magnitude:
            1e-2 /s to 1e2 /s).

    The reactant and the product should be in/on the same
    Region or Surface.

    There is no distinction between an intrinsic and an overall reaction
    rate for a unimolecular ReactionRule.

    A unimolecular reaction rule defines a Poissonian process.

    """

def create_decay_reaction_rule(reactant, k):
    """Example: A -> 0.

    Arguments:
        - reactant
            a Species.
        - k
            reaction rate. Units: per second. (Rough order of magnitude:
            1e-2 /s to 1e2 /s).

    There is no distinction between an intrinsic and an overall reaction
    rate for a decay ReactionRule.

    A decay reaction rule defines a Poissonian process.

    """

def create_annihilation_reaction_rule(reactant1, reactant2, ka):
    """Example: A + B -> 0.

    Arguments:
        - reactant1
            a Species.
        - reactant2
            a Species.
        - ka
            intrinsic reaction rate. Units: meters^3 per second. (Rough
            order of magnitude: 1e-16 m^3/s to 1e-20 m^3/s).

    The reactants should be in/on the same Region or Surface.

    ka should be an *intrinsic* reaction rate. You can convert an
    overall reaction rate (kon) to an intrinsic reaction rate (ka) with
    the function utils.k_a(kon, kD), but only for reaction rules in 3D.
    By default an EGFRDSimulator will assume a repulsive
    bimolecular reaction rule (ka=0) for each possible combination of
    reactants for which no bimolecular reaction rule is specified.
    You can explicitly add these reaction rules to the model with the
    method model.ParticleModel.set_all_repulsive.

    """

def create_binding_reaction_rule(reactant1, reactant2, product, ka):
    """Example: A + B -> C.

    Arguments:
        - reactant1
            a Species.
        - reactant2
            a Species.
        - product
            a Species.
        - ka
            intrinsic reaction rate. Units: meters^3 per second. (Rough
            order of magnitude: 1e-16 m^3/s to 1e-20 m^3/s)

    The reactants and the product should be in/on the same
    Region or Surface.

    A binding reaction rule always has exactly one product.

    ka should be an *intrinsic* reaction rate. You can convert an
    overall reaction rate (kon) to an intrinsic reaction rate (ka) with
    the function utils.k_a(kon, kD), but only for reaction rules in 3D.

    By default an EGFRDSimulator will assume a repulsive
    bimolecular reaction rule (ka=0) for each possible combination of
    reactants for which no bimolecular reaction rule is specified.
    You can explicitly add these reaction rules to the model with the
    method model.ParticleModel.set_all_repulsive.

    """

def create_unbinding_reaction_rule(reactant, product1, product2, kd):
    """Example: A -> B + C.

    Arguments:
        - reactant
            a Species.
        - product1
            a Species.
        - product2
            a Species.
        - kd
        intrinsic reaction rate. Units: per second. (Rough order of
        magnitude: 1e-2 /s to 1e2 /s).

    The reactant and the products should be in/on the same
    Region or Surface.

    An unbinding reaction rule always has exactly two products.

    kd should be an *intrinsic* reaction rate. You can convert an
    overall reaction rate (koff) for this reaction rule to an intrinsic
    reaction rate (kd) with the function utils.k_d(koff, kon, kD) or
    utils.k_d_using_ka(koff, ka, kD).

    An unbinding reaction rule defines a Poissonian process.

    """
```

To put this reactions into effect one should also call `add_reaction_rule`, e.g.:

```
    r1 = model.create_binding_reaction_rule(A, B, C, kf)
    m.network_rules.add_reaction_rule(r1)
```

2.2 Functions from `gfrdbase.py`

2.2.1 Core functions

```
def create_world(m, matrix_size=10):
    """Create a world object.

    The world object keeps track of the positions of the particles
    and the protective domains during an eGFRD simulation.

    Arguments:
        - m
            a ParticleModel previously created with model.ParticleModel.
        - matrix_size
            the number of cells in the MatrixSpace along the x, y and z
            axis. Leave it to the default number if you don't know what
            to put here.

    The simulation cube "world" is divided into (matrix_size x matrix_size
    x matrix_size) cells. Together these cells form a MatrixSpace. The
    MatrixSpace keeps track in which cell every particle and protective
    domain is at a certain point in time. To find the neigherest
    neighbours of particle, only objects in the same cell and the 26
    (3x3x3 - 1) neighbouring cells (the simulation cube has periodic
    boundary conditions) have to be taken into account.

    The matrix_size limits the size of the protective domains. If you
    have fewer particles, you want a smaller matrix_size, such that the
    protective domains and thus the eGFRD timesteps can be larger. If
    you have more particles, you want a larger matrix_size, such that
    finding the neigherest neighbours is faster.

    Example. In samples/dimer/dimer.py a matrix_size of
    (N * 6) ** (1. / 3.) is used, where N is the average number of
    particles in the world.

    """
```

2.2.2 Handling objects

```
def get_closest_surface(world, pos, ignore):
    """Return
      - closest surface
      - distance to closest surface

    We can not use matrix_space, it would miss a surface if the
    origin of the surface would not be in the same or neighboring
    cells as pos."""

def get_closest_surface_within_radius(world, pos, radius, ignore):
    """Return:
      - surface within radius or None
      - closest surface (regardless of radius)
      - distance to closest surface"""
```

2.2.3 Adding particles

```
functions
def throw_in_particles(world, sid, n):
    """Add n particles of a certain Species to the specified world.

    Arguments:
        - sid
            a Species previously created with the function
            model.Species.
        - n
            the number of particles to add.

    Make sure to first add the Species to the model with the method
    model.ParticleModel.add_species_type.

    """

def place_particle(world, sid, position):
    """Place a particle of a certain Species at a specific position in
    the specified world.

    Arguments:
        - sid
```

```
            a Species previously created with the function
            model.Species.
        - position
            a position vector [x, y, z]. Units: [meters, meters, meters].

    Make sure to first add the Species to the model with the method
    model.ParticleModel.add_species_type.

    """
```

2.3 Functions from `egfrd.py`

2.3.1 Core functions

```
class EGFRDSimulator(ParticleSimulatorBase):
    """
    """
    def __init__(self, world, rng=myrandom.rng, network_rules=None):
        """Create a new EGFRDSimulator.

        Arguments:
            - world
                a world object created with the function
                gfrdbase.create_world.
            - rng
                a random number generator. By default myrandom.rng is
                used, which uses Mersenne Twister from the GSL library.
                You can set the seed of it with the function
                myrandom.seed.
            - network_rules
                you don't need to use this, for backward compatibility only.

        """

    def step(self):
        """Execute one eGFRD step.

        """
```

(Function belongs to EGFRDSimulator class, call with `EGFRDSimulator.get_next_time`.)

```
    def stop(self, t):
        """Synchronize all particles at time t.

        With eGFRD, particle positions are normally updated
        asynchronously. This method bursts all protective domains and
        assigns a position to each particle.

        Arguments:
            - t
                the time at which to synchronize the particles. Usually
                you will want to use the current time of the simulator:
                EGFRDSimulator.t.

        This method is called stop because it is usually called at the
        end of a simulation. It is possible to call this method at an
        earlier time. For example the Logger module does this, because
        it needs to know the positions of the particles at each log
        step.

        """
```

(Function belongs to EGFRDSimulator class, call with `EGFRDSimulator.get_next_time`.)

2.3.2 Simulator time (manipulation) functions

```
    def get_next_time(self):
    """
    Returns the time it will be when the next egfrd timestep
    is completed.
    """
```

(Function belongs to EGFRDSimulator class, call with `EGFRDSimulator.get_next_time`.)

```
    def reset(self):
    """
    This function resets the "records" of the simulator. This means
    the simulator time is reset, the step counter is reset, events
    are reset, etc.
    Can be for example usefull when users want to "stirr" the
    simulation before starting the "real experiment".
    """
```

(Function belongs to EGFRDSimulator class, call with `EGFRDSimulator.get_next_time`.)

2.3.3 Get data

```
    def print_report(self, out=None):
        """Print various statistics about the simulation.

        Arguments:
            - None

        """
```

2.3.4 Additional functions

The class `EGFRDSimulator` contains several functions which might be usefull to eGFRD users in very specific cases. Because of their specific nature, they don't contain docstrings.

```
    def get_matrix_cell_size(self):
        return self.containers[0].cell_size

    def set_user_max_shell_size(self, size):
        self.user_max_shell_size = size

    def get_user_max_shell_size(self):
        return self.user_max_shell_size

    def get_max_shell_size(self):
        return min(self.get_matrix_cell_size() * .5 / SAFETY,
                   self.user_max_shell_size)
```

2.4 Functions from `dumper.py`

2.4.1 Getting information on species/particles

```
def get_species(sim):
    """Return an iterator over the Species in the simulator.

    Arguments:
        - sim
            an EGFRDSimulator.

    """

def dump_species(sim):
    """Return a string containing the Species in the simulator.

    Arguments:
        - sim
            an EGFRDSimulator.

    """

def get_species_names(sim):
    """Return an iterator over the names of the Species in the
    simulator.

    Arguments:
        - sim
            an EGFRDSimulator.

    """

def dump_species_names(sim):
    """Return a string containing the names of the Species in the
    simulator.

    Arguments:
        - sim
```

```
            an EGFRDSimulator.

    """

def _get_species_type_by_name(sim, name):
    """ Return the type of a species with a certain name

    Arguments:
        - sim
            an EGFRDSimulator
        - name
            species name
    """

def _get_particles_by_sid(sim, sid):
    """ Return a generator (using "yield") to loop over (pid, particle).

    Arguments:
        - sim
            an EGFRDSimulator
        - sid
            ID of a species

    E.g.:

    myparticles = _get_particles_by_sid(sim, sid)

    for mypid, myparticle in myparticles:
        print str(str(mypid), str(myparticle))
    """

def get_particles(sim, identifier=None):
    """Return an iterator over the
    (particle identifier, particle)-pairs in the simulator.

    Arguments:
        - sim
            an EGFRDSimulator.
        - identifier
            a Species or the name of a Species. If none is specified,
            all (particle identifier, particle)-pairs will be returned.

    """

def dump_particles(sim, identifier=None):
    """Return a string containing the
    (particle identifier, particle)-pairs in the simulator.

    Arguments:
        - sim
            an EGFRDSimulator.
        - identifier
            a Species or the name of a Species. If none is specified,
            all (particle identifier, particle)-pairs will be returned.

    """

def _get_number_of_particles_by_sid(sim, sid):
    """
    Returns the number of particles of a certain species.

    Arguments:
        - sim
            an EGFRDSimulator.
        - sid
            ID of a species

    """

def get_number_of_particles(sim, identifier=None):
    """Return the number of particles of a certain Species in the
    simulator.

    Arguments:
        - sim
            either an EGFRDSimulator or a GillespieSimulator.
        - identifier
            a Species. Optional. If none is specified, a list of
            (Species name, number of particles)-pairs will be returned.

    """

def dump_number_of_particles(sim, identifier=None):
    """Return a string containing the number of particles of a certain
    Species in the simulator.

    Arguments:
        - sim
            either an EGFRDSimulator or a GillespieSimulator.
        - identifier
            a Species. Optional. If none is specified,
            a string of (Species name, number of particles)-pairs will
            be returned.

    """
```

2.4.2 Get information on domains

```
def get_domains(egfrdsim):
    """Return an iterator over the protective domains in the simulator.

    Arguments:
        - egfrdsim
            an EGFRDSimulator
    """

def dump_domains(egfrdsim):
    """Return an string containing the protective domains in the
    simulator.

    """
```

subsubsectionGet information on reaction rules

```
def get_reaction_rules(model_or_simulator):
    """Return three lists with all the reaction rules defined in the
    ParticleModel or EGFRDSimulator.

    The three lists are:
        - reaction rules of only one reactant.
        - reaction rules between two reactants with a reaction rate
          larger than 0.
        - repulsive reaction rules between two reactants with a
          reaction rate equal to 0.

    Arguments:
        - model_or_simulator
            a ParticleModel or EGFRDSimulator.

    """

def _dump_reaction_rule(model, reaction_rule):
    """Helper. Return ReactionRule as string.

    ReactionRule.__str__ would be good, but we are actually getting a
    ReactionRuleInfo or ReactionRuleCache object."""

def dump_reaction_rules(model_or_simulator):
    """Return a formatted string containing all the reaction rules
    defined in the ParticleModel or EGFRDSimulator.

    Arguments:
        - model_or_simulator
            a ParticleModel or EGFRDSimulator.

    """
```

2.5 Functions from `utils.py`

This file contains functions on common mathematical objects, transformations and formulas. Some of these functions are used throughout the algorithm, and some are supplied for convenience. For the user, none of these functions are *needed* to make a simulation work, but some might come in handy.

2.5.1 Mathematical comparisons

```
def feq(a, b, typical=1, tolerance=TOLERANCE):
    """Return True if a and b are equal, subject to given tolerances.
    Float comparison.

    Also see numpy.allclose().

    The (relative) tolerance must be positive and << 1.0

    Instead of specifying an absolute tolerance, you can speciy a
    typical value for a or b. The absolute tolerance is then the
    relative tolerance multipied by this typical value, and will be
    used when comparing a value to zero. By default, the typical
    value is 1."""

def fgreater(a, b, typical=1, tolerance=TOLERANCE):
    """Return True if a is greater than b, subject to given tolerances.
    Float comparison."""

def fless(a, b, typical=1, tolerance=TOLERANCE):
    """Return True if a is less than b, subject to given tolerances.
    Float comparison."""

def fgeq(a, b, typical=1, tolerance=TOLERANCE):
    """Return True if a is greater or equal than b, subject to given
    tolerances. Float comparison."""

def fleq(a, b, typical=1, tolerance=TOLERANCE):
    """Return True if a is less than or equal than b, subject to given
    tolerances. Float comparison."""
```

2.5.2 Conversions

As these functions only contain a single line of formula code, this line is also supplied.

```
def per_M_to_m3(rate):
    """Convert a reaction rate from units 'per molar per second' to
    units 'meters^3 per second'.

    """
    return rate / (1000 * N_A)

def per_microM_to_m3(rate):
    """Convert a reaction rate from units 'per micromolar per second' to
    units 'meters^3 per second'.

    """
    return per_M_to_m3(rate * 1e6)
```

```
def M_to_per_m3(molar):
    """Convert a concentration from units 'molar' to units 'per
    meters^3'.

    """
    return molar * (1000 * N_A)

def microM_to_per_m3(micromolar):
    """Convert a concentration from units 'micromolar' to units 'per
    meters^3'.

    """
    return M_to_per_m3(micromolar / 1e6)

def C2N(c, V):
    """Calculate the number of particles in a volume 'V' (dm^3)
    with a concentration 'c' (mol/dm^3).

    """
    return c * V * N_A  # round() here?
```

Conversions involving rates:

```
def k_D(Dtot, sigma):
    """Calculate the 'pseudo-'reaction rate (kD) caused by diffusion.

    kD is equal to 1 divided by the time it takes for two particles to
    meet each other by diffusion. It is needed when converting from
    an intrinsic reaction rate to an overall reaction rates or vice
    versa.

    Example:
        - A + B -> C.

    Arguments:
        - Dtot:
            the diffusion constant of particle A plus the diffusion
            constant of particle B. Units: meters^2/second.
        - sigma
            the radius of particle A plus the radius of particle B.
            Units: meters.

    This function is only available for reaction rules in 3D. No
    analytical expression for kD in 1D or 2D is currently known.

    """
    return 4.0 * numpy.pi * Dtot * sigma

def k_a(kon, kD):
    """Convert an overall reaction rate (kon) for a binding/annihilation
    reaction rule to an intrinsic reaction rate (ka).

    Example:
        - A + B -> C
            binding reaction rule
        - A + B -> 0
            annihilation reaction rule

    Arguments:
        - kon
            the overall reaction rate for the reaction rule. Units:
            meters^3/second.
        - kD
            the 'pseudo-'reaction rate caused by the diffusion of
            particles A and B. See the function k_D(). Units:
            meters^3/second.

    This function is only available for reaction rules in 3D. No
    analytical expression for kD in 1D or 2D is currently known.

    """
    if kon > kD:
        raise RuntimeError, 'kon > kD.'
    ka = 1. / ((1. / kon) - (1. / kD))
    return ka

def k_d(koff, kon, kD):
    """Convert an overall reaction rate (koff) for an unbinding reaction
    rule to an intrinsic reaction rate (kd).

    This one is a bit tricky. We consider reaction rules with only 1
    reactant. In case there is only 1 product also, no conversion in
    necessary. But when the reaction rule has 2 products, we need to
    take the reverse reaction rule into account and do the proper
    conversion.

    Example:
        - C -> A + B
            unbinding reaction rule
        - A + B -> C
            reverse reaction rule

    Arguments:
        - koff
            the overall reaction rate for the unbinding reaction rule.
            Units: meters^3/second.
        - kon
            the overall reaction rate for the reverse reaction rule.
            Units: meters^3/second.
        - kD
            the 'pseudo-'reaction rate caused by the diffusion of
            particles A and B. See the function k_D(). Units:
            meters^3/second.

    This function is only available for reaction rules in 3D. No
    analytical expression for kD in 1D or 2D is currently known.

    """
    ka = k_a(kon, kD)
    kd = k_d_using_ka(koff, ka, kD)
    return kd

def k_d_using_ka(koff, ka, kD):
    """Convert an overall reaction rate (koff) for an unbinding reaction
    rule to an intrinsic reaction rate (kd).

    Similar to the function k_d(), but expects an intrinsic rate (ka)
    instead of an overall rate (kon) for the reversed reaction rule as
    the second argument.

    This function is only available for reaction rules in 3D. No
    analytical expression for kD in 1D or 2D is currently known.

    """
    kd =  koff * (1 + float(ka) / kD)
    return kd

def k_on(ka, kD):
    """Convert an intrinsic reaction rate (ka) for a binding/annihilation
    reaction rule to an overall reaction rate (kon).

    The inverse of the function k_a().

    Rarely needed.

    This function is only available for reaction rules in 3D. No
    analytical expression for kD in 1D or 2D is currently known.

    """
    kon = 1. / ((1. / kD) + (1. / ka))  # m^3/s
    return kon

def k_off(kd, kon, kD):
    """Convert an intrinsic reaction rate (kd) for an unbinding reaction
    rule to an overall reaction rate (koff).

    The inverse of the function k_d().

    Rarely needed.
    This function is only available for reaction rules in 3D. No
    analytical expression for kD in 1D or 2D is currently known.

    """
    ka = k_a(kon, kD)
    koff = k_off_using_ka(kd, ka, kD)
    return koff

def k_off_using_ka(kd, ka, kD):
    """Convert an intrinsic reaction rate (kd) for an unbinding reaction
    rule to an overall reaction rate (koff).

    Similar to the function k_off(), but expects an intrinsic rate
    (ka) instead of an overall rate (kon) as the second argument.

    Rarely needed.

    This function is only available for reaction rules in 3D. No
    analytical expression for kD in 1D or 2D is currently known.

    """
    koff = 1. / (float(ka) / (kd * kD) + (1. / kd))
    return koff
```

2.5.3 Some convenient functoins

These functions do not contain docstrings, are sometimes self-explanatory and probably not needed that often in simulations. Therefore only definitions of functions that might be of interest to the user are listed here:

Mean arrival time:

```
def mean_arrival_time(r, D):
    return (r * r) / (6.0 * D)
```

Calculating with distances:

```
def distance_sq_array_simple(position1, positions, fsize = None):

def distance_array_simple(position1, positions, fsize = None):

distance = _gfrd.distance

distance_cyclic = _gfrd.distance_cyclic

def distance_sq_array_cyclic(position1, positions, fsize):

def distance_array_cyclic(position1, positions, fsize = 0):
```

Some vector functions:

```
def cartesian_to_spherical(c):

def spherical_to_cartesian(s):

def random_unit_vector_s():

def random_unit_vector():

def random_vector(r):

def random_vector2D(r):

def length(a):

def normalize(a, l=1):

def vector_angle(a, b):

def vector_angle_against_z_axis(b):

def crossproduct(a, b):

def crossproduct_against_z_axis(a):

def rotate_vector(v, r, alpha):
```

2.6 Notes on other files

2.6.1 `bd.py`: Brownian Dynamic Simulator

The class `BDSimulator` can be used in the same way as the EGFRDSimulator class, but performs Brownian Dynamics (BD) instead. Users who want to perform eGFRD simulations never need this. The simulator can be used for comparison of the eGFRD algorithm with Brownian Dynamics.

2.6.2 `gillespie.py`: Gillespie Simulator

Similar to the BD simulator, the class `GillespieSimulatorBase` can be used for Gillespie type simulations.

2.6.3 `legacy.py`: Old redundant functions

This module is called nowhere in the Python code. It contains an archive of outdated code.

2.6.4 `multi.py`, `pair.py`, `single.py`

These file contain the code that handle the specific events that happen in the different categories of domains.

2.6.5 `myrandom.py`

Contains a few convenient lines of code used when using random functions.

2.6.6 `make_cjy_table.py.py`, `make_sjy_table.py.py`

Generate (respectively cylindrical and spherical) bessel function tables.

2.7 Function from module `logger.py`

This module contains two loggers. One logger that logs in the hdf5 format, for obvious reasons in class `HDF5Logger`. Note that this logger requires the module h5py. The other logger gives output dictated more by the nature of the eGFRD algorithm. The loggers are not (yet) explained in very much detail here, but both function in the same way. A logger class is made, which takes input on to which file to write, the logger can be started by `start()` and steps are logged by the function `log()`.

2.7.1 HDF5 logger

```
class HDF5Logger(object):
    def __init__(self, logname, directory='data', split=False):

    def log(self, sim, time):

    def start(self, sim):
```

2.7.2 "Normal" logger

```
class Logger(object):
    def __init__(self, logname='log', directory='data', comment=''):

    def log(self, sim, time):

    def start(self, sim):
```

(Note that not all functions contained by this class are listed, just functions deemed usefull for user interface.)

3 Todo

- `sid = identifier.id` gives the sid, but what exactly is identifier?