KineticGas Implementation and Usage

Vegard Gjeldvik Jervell

November 2022

General

This document covers is an overview of the structure and internal dependencies of the `KineticGas` package. For details regarding the mathematics and theory behind the package, see: The Kinetic Gas theory of Mie Fluids (V.G. Jervell, 2022) and Revised Enskog Theory for Mie fluids: Prediction of diffusion coefficients, thermal diffusion coefficients, viscosities and thermal conductivities (V.G. Jervell and Ø. Wilhelmsen, 2023).

The KineticGas package is split into two major parts. A C++ implementation that handles almost all calculations, and a Python wrapper that handles the parameter database, matrix inversion, and acts as user-friendly interface to the C++ side. The inheritance structure of the classes in the module is summarised in Figure 0.1. The control and information flow during initialisation and computations is illustrated in Figure 0.2.

For an end-user that wants to compute transport coefficients, the relevant methods to call are found in the python-side `KineticGas` class, documented in Section 4. For example usage see the `pyExamples` directory.

For a user aiming to extend the package with a new potential model, see the C++ side `KineticGas` and `Spherical` classes.

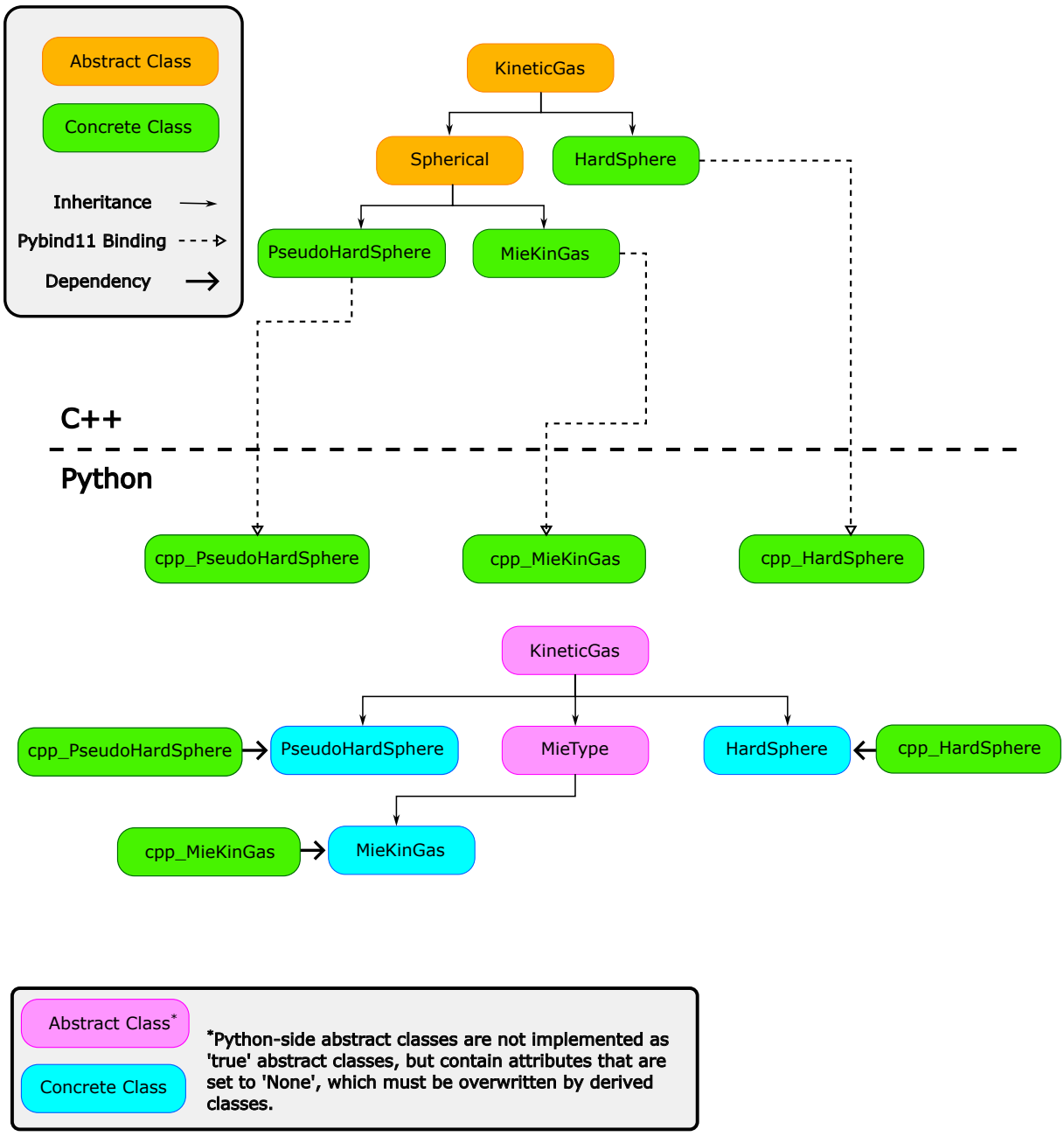

Figure 0.1: Inheritance structure and internal dependencies of the `KineticGas` package.

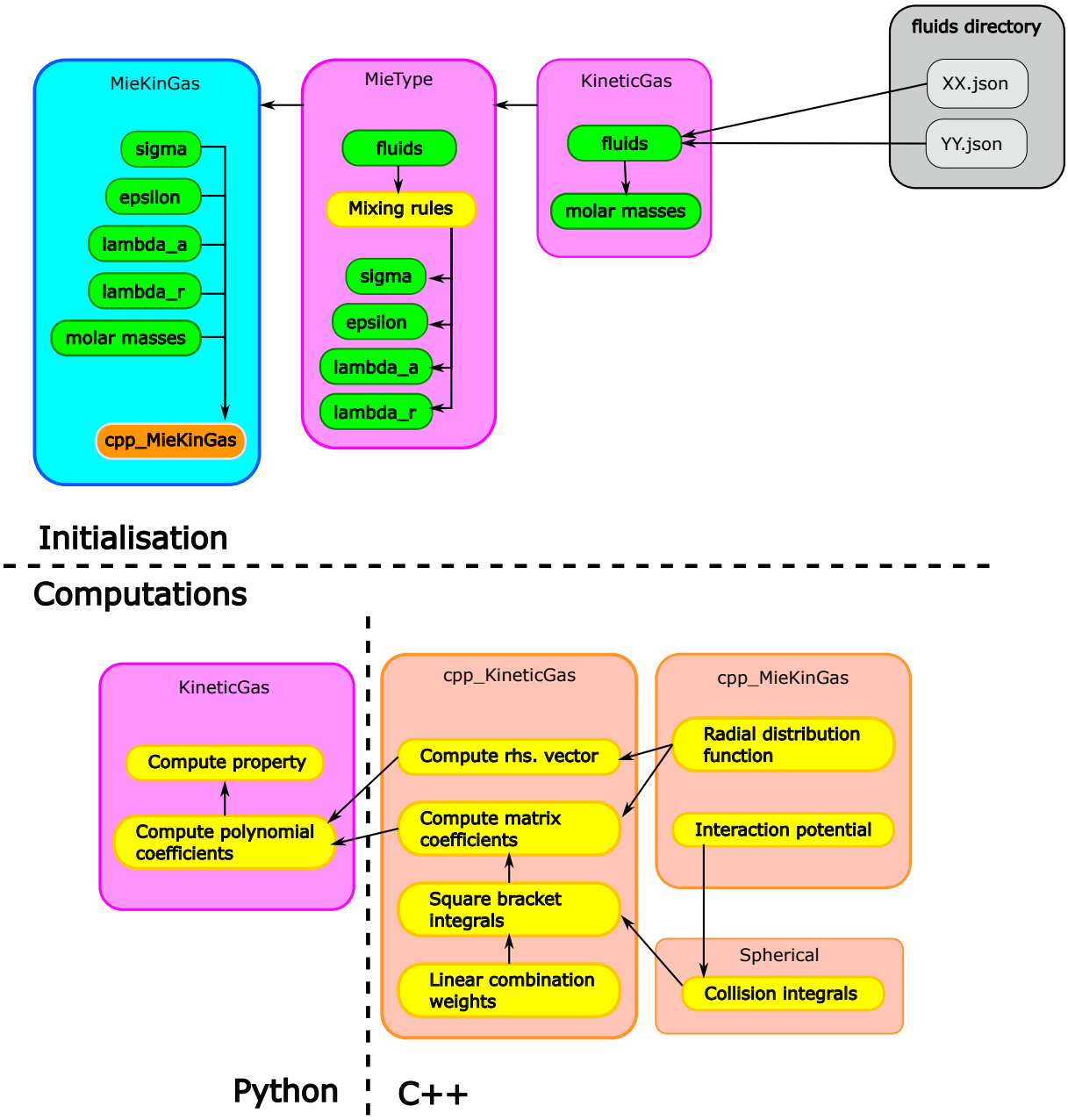

Figure 0.2: Graphical representation of control and information flow during initialisation (top) and property computations (bottom). The `MieKinGas` class is used as an example, the diagram is essentially equivalent for other classes.

Contents

Part I

C++ side

All classes exported from the C++ side to Python are exported using `pybind11`, with the prefix `cpp_`. Such that the C++ class `MieKinGas` is exported as `cpp_MieKinGas` etc.

1 KineticGas

Abstract Class - Derived classes must implement

1. `double omega()` - The collision integrals.
2. `std::vector<std::vector<double>> model_rdf()` - The radial distribution function at contact for each particle pair.
3. `std::vector<std::vector<double>> get_contact_diameters()` - Method that returns the contact diameters for each particle pair.

Contains the interface methods

1. `get_diffusion_matrix()` - Matrix corresponding to the linear system of equations that must be solved for the diffusive response functions Sonine polynomial expansion coefficients $d_{i,j}^{(r)}$.
2. `get_diffusion_vector()` - The right hand side vector of the linear system of equations that must be solved for the diffusive response functions Sonine polynomial expansion coefficients $d_{i,j}^{(r)}$.
3. `get_conductivity_matrix()` - Matrix corresponding to the linear system of equations that must be solved for the thermal response functions Sonine polynomial expansion coefficients $\ell_i^{(r)}$.
4. `get_conductivity_vector()` - The right hand side vector of the linear system of equations that must be solved for the thermal response functions Sonine polynomial expansion coefficients $\ell_i^{(r)}$.
5. `get_viscosity_matrix()` - Matrix corresponding to the linear system of equations that must be solved for the viscous response functions Sonine polynomial expansion coefficients $b_i^{(r)}$.
6. `get_viscosity_vector()` - The right hand side vector of the linear system of equations that must be solved for the viscous response functions Sonine polynomial expansion coefficients $b_i^{(r)}$.

Contains the helper functions

1. Square bracket integrals for conductivity, thermal diffusion and diffusion
   (a) `double H_ij(int p, int q, int i, int j, double T)` - $\left[S_{3/2}^{(p)}(\mathscr{U}_i^2)\boldsymbol{\mathscr{U}}_i, S_{3/2}^{(q)}(\mathscr{U}_j^2)\boldsymbol{\mathscr{U}}_j\right]_{ij}$
   (b) `double H_i(int p, int q, int i, int j, double T)` - $\left[S_{3/2}^{(p)}(\mathscr{U}_i^2)\boldsymbol{\mathscr{U}}_i, S_{3/2}^{(q)}(\mathscr{U}_i^2)\boldsymbol{\mathscr{U}}_i\right]_{ij}$
      i. Linear combination weights used to compute square bracket integrals for conductivity, thermal diffusion and diffusion
         A. `double A(p, q, r, l)` - A_{pqrl} weights
         B. `double A_prime(p, q, r, l` - A'_{pqrl} weights
         C. `double A_trippleprime(p, q, r, l)` - A'''_{pqrl} weights
2. Square bracket integrals for viscosity
   (a) `double L_ij(int p, int q, int i, int j, double T)` - $\left[S_{5/2}^{(p)}(\mathscr{U}_i^2)\boldsymbol{\mathscr{U}}_i\mathring{\boldsymbol{\mathscr{U}}}_i, S_{5/2}^{(q)}(\mathscr{U}_j^2)\boldsymbol{\mathscr{U}}_j\mathring{\boldsymbol{\mathscr{U}}}_j\right]_{ij}$

(b) `double L_i(int p, int q, int i, int j, double T)` - $\left[S^{(p)}_{5/2}(\mathscr{U}_i^2)\boldsymbol{\mathscr{U}}_i\mathring{\boldsymbol{\mathscr{U}}}_i, S^{(q)}_{5/2}(\mathscr{U}_i^2)\boldsymbol{\mathscr{U}}_i\mathring{\boldsymbol{\mathscr{U}}}_i\right]_{ij}$

i. Linear combination weights used to compute square bracket integrals for viscosity

A. `double B(p, q, r, l)` - B_{pqrl} weights

B. `double B_prime(p, q, r, l` - B'_{pqrl} weights

C. `double B_trippleprime(p, q, r, l)` - B'''_{pqrl} weights

2 HardSphere

Concrete class, inherits `KineticGas`.

Implements

1. `double omega()` - Analytical Hard sphere collision integrals.
2. `std::vector<std::vector<double>> model_rdf()` - The Carnahan-Starling radial distribution function at contact.
3. `std::vector<std::vector<double>> get_contact_diameters()` - Additive hard sphere diameters for each particle pair.

3 Spherical

Abstract class, inherits `KineticGas`

Implements

1. Numerical procedures to evaluate the collision integrals for an arbitrary spherical potential.
   (a) `double get_R(int i, int j, double T, double g, double b)` - Compute the distance of closest approach between two particles, at a given temperature (T), dimensionless relative velocity (g) and impact parameter (b).
   (b) `double chi(int i, int j, double T, double g, double b)` - Compute the deflection angle χ for a collision at a given temperature (T), dimensionless relative velocity (g) and impact parameter (b).
   (c) `double w_integrand(int i, int j, double T, double g, double b, int l, int r)` - The integrand of the dimensionless collision integral $W^{\ell r}_{ij}$.
   (d) `std::function w_integrand_export` - Function pointer to `w_integrand` that is passed to the external integration module.
2. `double omega(int i, int j, int l, int r, double T)` - The collision integrals for an arbitrary, spherical potential.
3. `OmegaDb` - An `std::map` that stores computed collision integrals, such that any collision integral is only ever computed once. Every instance of an object inheriting `Spherical` maintains its own `OmegaDb`, such that integrals are not shared between objects. Integrals are deleted when the program exits.

Derived classes must implement

1. `potential(int i, int j, double r)` - The pairwise interaction potential for particles i and j.
2. `potential_derivative_(int i, int j, double r)` - Derivative of pairwise interaction potential
3. `potential_dblderivative_rr(int i, int j, double r)` - Second derivative of pairwise interaction potential

3.1 PseudoHardSphere

Concrete class, inherits `Spherical`

The only intent of this class is to test the numerical collision integrals produced by the methods in the `Spherical` class versus the analytical solutions obtained from the `HardSphere` class, in order to check that the numerical solvers are good enough.

3.2 MieKinGas

Concrete class, inherits `Spherical` Implements the potential-related functions required by `Spherical` Also implements `get_model_rdf()` as required by `KineticGas`, For details regarding the rdf at contact see the theory docs. and Ref. [].

Part II

Python side

All classes exported from the C++ side to Python are exported using `pybind11`, with the prefix `cpp_`. Such that the C++ class `MieKinGas` is exported as `cpp_MieKinGas` etc.

4 KineticGas

"Abstract" class - Can be initialized, but methods will crash if called.

Derived classes must implement the attributes

1. `self.cpp_kingas` - An object that inherits `cpp_KineticGas`.
2. `self.eos` - An equation of state object with a member method with signature equivalent to the thermopack method `chemical_potential_tv()`.

Purpose: Implements all functions necessary to compute viscosity, conductivity, diffusion coefficients and thermal diffusion coefficients. Constructor is responsible for reading in the fluid-parameter `.json` files for a mixture, and stores them in the `self.fluids` attribute.

Constructor arguments

1. Comps (str) : Comma separated list of component ID's, corresponding to fluid-file names.
2. mole_weights (array(float), optional) : List of component molar masses, overrides values i fluid files. If an element is `None`, the fluid-file value will be used for the corresponding component.
3. N (int, optional) : Default Enskog approximation order. Defaults to 3.
4. is_idealgas (bool, optional) : Defaults to `False`. If `True`, transport properties are computed at infinite dilution. Read the theory docs for more info on what this entails. In short, the rdf at contact and the factors K_i and K_i' are unity, independent of density, temperature, etc. Additionally, the second terms of viscosity and conductivity (η'' and λ'') are set to zero. Thus, the conductivity, viscosity, thermal diffusion factors, thermal diffusion ratios and Soret coefficients become independent of density.
5. parameter_ref (str, optional) : Defaults to `'default'`. Parameter reference id for the parameter set to use from the fluid files. As of now, all components must use parameters with the same `parameter_ref`.

Note: All arguments that override fluid file parameters (`sigma`, `epsilon`, etc.) must be an array of length corresponding to the number of components. To use fluid-file values for some components and not others, set the array value to `None` for the components that are to use fluid-file parameters.

Contains the interface methods

1. `viscosity(self, T, Vm, x, N=None)` - Compute the viscosity, η, at a given temperature, molar volume, molar composition and Enskog approximation order.
2. `thermal_conductivity(self, T, Vm, x, N=None)` - Compute the thermal conductivity, λ, at a given temperature, molar volume, molar composition and Enskog approximation order.
3. `thermal_diffusion_coeff(self, T, Vm, x, N=None)` - Compute the thermal diffusion coefficients $D_{T,i}$, at a given temperature, molar volume, molar composition and Enskog approximation order.
4. `thermal_diffusion_ratio(self, T, Vm, x, N=None)` - Compute the thermal diffusion ratios $k_{T,i}$, at a given temperature, molar volume, molar composition and Enskog approximation order.
5. `thermal_diffusion_factor(self, T, Vm, x, N=None)` - Compute the thermal diffusion factors α_{ij}, at a given temperature, molar volume, molar composition and Enskog approximation order.
6. `soret_coefficient(self, T, Vm, x, N=None)` - Compute the Soret coefficients $S_{T,i}$, at a given temperature, molar volume, molar composition and Enskog approximation order.
7. `interdiffusion(self, T, Vm, x, N=None)` - Compute the interdiffusion coefficients at a given temperature, molar volume, molar composition and Enskog approximation order. Returns the self-diffusion coefficient if called for a pure fluid. Supports the optional arguments (default value indicated):
   (a) `use_binary=True` - Return the Fickean binary diffusion coefficient, defined as $J_1 = -D_{12}\nabla n_1 = D_{12}\nabla n_2$. Argument has no effect for multicomponent mixtures.
   (b) `frame_of_reference='CoN'` - The frame of reference to which the diffusion coefficients apply, can be either 'CoN' (centre of moles), 'CoM' (centre of mass), or 'solvent'. If 'solvent is used, the kwarg `solvent_idx=(int)` must also be supplied.
   (c) `solvent_idx` - (No default value) the component index of the solvent.

Contains the helper methods

1. `interdiffusion_general(self, T, Vm, x, N=None)` - Return the kinetic CoM diffusion coefficient matrix, as described in `docs/multicomponent.pdf` and Ref. [].
2. `get_Eij(self, Vm, T, x)` - Compute the factors $\frac{n_i}{k_B T}\left(\frac{\partial \mu_i}{\partial n_j}\right)$
3. `get_com_2_for_matr(self, T, Vm, x, FoR, **kwargs)` - Get the transformation matrix $\underline{\mathbf{\Psi}}^{FoR\leftarrow m}$, where FoR is either 'CoN', 'CoM' or 'solvent'. For 'CoM' returns the identity matrix, otherwise the call is dispatched to one of the methods
   (a) `get_com_2_con_matr(self, x)` - Returns $\underline{\mathbf{\Psi}}^{n\leftarrow m}$.
   (b) `get_com_2_solv_matr(self, x, **kwargs)` - Returns $\underline{\mathbf{\Psi}}^{n_i\leftarrow m}$, where i is specified by the 'solvent_idx' kwarg.
4. `get_conductivity_matrix(self, particle_density, T, mole_fracs, N=None)` - Wraps the corresponding C++ side method.
5. `get_lambda_vector(self, particle_density, T, mole_fracs, N)` - Wraps the C++ side method `get_conductivity_vector()`.
6. `compute_a_vector(self, particle_density, T, mole_fracs, N=None)` - Computes the thermal response function Sonine polynomial expansion coefficients $\ell_i^{(r)}$.
7. `compute_d_vector(self, particle_density, T, mole_fracs, N=None)` - Computes the diffusive response function Sonine polynomial expansion coefficients $d_{i,j}^{(r)}$.
8. `reshape_d_vector(self, d)` - Reshapes the vector returned by `compute_d_vector()` to an array $d_{i,j}^{(r)} =$ `d[i][r][j]`,

9. `compute_dth_vector(self, particle_density, T, mole_fracs, N=None)` - Compute the diffusive driving forces at the state of no mass fluxes, $d_i^{J=0}$.
10. `compute_b_vector(self, particle_density, T, mole_fracs, N=None)` - Compute the viscous response function Sonine polynomial expansion coefficients $b_i^{(r)}$.

5 HardSphere

Concrete class, inherits from `KineticGas`, and uses `cpp_HardSphere`.

Overrides the method `get_Eij()` to use Carnahan-Starling values for chemical potential derivative, rather than an `eos` object.

Constructor extracts Hard sphere parameters from the `self.fluids` attribute, and initialises a `cpp_HardSphere` object stored in the `self.cpp_kingas` attribute.

6 PseudoHardSphere

Concrete class, inherits from `KineticGas`, and uses `cpp_HardSphere`.

Overrides the method `get_Eij()` to use Carnahan-Starling values for chemical potential derivative, rather than an `eos` object.

Constructor extracts Hard sphere parameters from the `self.fluids` attribute, and initialises a `cpp_PseudoHardSphere` object stored in the `self.cpp_kingas` attribute.

7 MieType

Abstract class, inherits from `KineticGas`

Constructor takes the 'potential' argument which indicates whether potential parameters for a Mie potential or FH-Mie potential are to be used, then the parameters required for all "Mie-type" potentials (σ, ϵ, λ_a, λ_r) are extracted from the `self.fluids` attribute and stored in their own attributes (`self.sigma`, `self.epsilon`, etc.)

Constructor arguments

1. comps (str) : Passed to `KineticGas`
2. potential (str) : Supplied by inheriting class.
3. mole_weights (array(float), optional) : Passed to `KineticGas`
4. parameter_ref (str, optional) : Passed to `KineticGas`
5. sigma (array(float), optional) : Overrides fluid-file sigma parameters
6. eps_div_k (array(float), optional) : Overrides fluid-file eps_div_k parameters
7. la (array(float), optional) : Overrides fluid-file la parameters
8. lr (array(float), optional) : Overrides fluid-file lr parameters
9. lij (array(float), optional) : Defaults to 0. Mixing parameter for sigma.
10. kij (array(float), optional) : Defaults to 0. Mixing parameter for epsilon.

7.1 MieKinGas

Concrete class, inherits from `MieType`.

Constructor initialises a `cpp_MieKinGas` object, stored in the attribute `self.cpp_kingas`, using the parameters extracted when passing `potential='Mie'` to the `MieType` constructor.

Part III

Fluid files

The fluid files are a database of pure-fluid parameters, organised as a `.json` file for each species. The structure of the files is nested as

- ident : str
- formula : str
- cas_number : str
- name : str
- aliases : list(str)
- mol_weight : float
- Potential 1 :
  - parameter_ref 1 :
    - param 1 : float
    - param 2 : float
    - ...
  - parameter_ref 2 :
    - param 1 : float
    - param 2 : float
    - ...
- Potential 2 :
  - parameter_ref 1 :
    - param 1 : float
    - param 2 : float
    - ...
  - parameter_ref 2 :
    - param 1 : float
    - param 2 : float
    - ...
- Potential ...

where `Potential 1`, `Potential 2`, etc. identify the potential model. These can be 'Mie', 'HS", etc. the `paramter_ref` are the identifiers for each parameter set, and `param 1`, `param 2`, etc. are the identifiers for each parameter within the parameter set.