pyPRISM Documentation

Release v1.0.3

Tyler B. Martin

Jun 15, 2018

Code Manual

pyPRISM is a Python-based, open-source framework for conducting Polymer Reference Interaction Site Model (PRISM) theory calculations. This framework aims to simplify PRISM-based studies by providing a user-friendly scripting interface for setting up and numerically solving the PRISM equations.

PRISM theory describes the equilibrium spatial-correlations of liquid-like polymer systems including melts, blends, solutions, block copolymers, ionomers, liquid crystal forming polymers and nanocomposites. Using PRISM theory, one can calculate thermodynamic (e.g., second virial coefficients, Flory-Huggins χ interaction parameters, potentials of mean force) and structural (e.g., pair correlation functions, structure factors) information for these macromolecular materials. See the *Frequently Asked Questions* section for examples of systems and calculations that are available to PRISM theory.

pyPRISM provides data structures, functions, and classes that streamline PRISM calculations, allowing pyPRISM to be extended for use in other tasks such as the coarse-graining of atomistic simulation force-fields or the modeling of experimental scattering data. The goal of this framework is to reduce the barrier to correctly and appropriately using PRISM theory and to provide a platform for rapid calculations of the structure and thermodynamics of polymeric fluids and nanocomposites.

CHAPTER 1

Citations

If you use pyPRISM in your work, we ask that you please cite both of the following articles

1. Martin, T.B.; Gartner, T.E. III; Jones, R.L.; Snyder, C.R.; Jayaraman, A.; pyPRISM: A Computational Tool for Liquid State Theory Calculations of Macromolecular Materials, Macromolecules, 2018, 51 (8), p2906-2922 [link]
2. Schweizer, K.S.; Curro, J.G.; Integral Equation Theory of the Structure of Polymer Melts, Physical Review Letters, 1987, 58 (3), p246-249 doi:10.1103/PhysRevLett.58.246 [link]

CHAPTER 2

pyPRISM Example

Below is an example python script where we use pyPRISM to calculate the pair correlation functions for a nanocomposite (polymer + particle) system with attractive polymer-particle interactions. Below the script is a plot of the pair correlation functions from this calculation. See *Quickstart Guide* for a more detailed discussion of this example.

```
import pyPRISM

sys = pyPRISM.System(['particle','polymer'],kT=1.0)
sys.domain = pyPRISM.Domain(dr=0.01,length=4096)

sys.density['polymer']  = 0.75
sys.density['particle'] = 6e-6

sys.diameter['polymer']  = 1.0
sys.diameter['particle'] = 5.0

sys.omega['polymer','polymer']   = pyPRISM.omega.FreelyJointedChain(length=100,l=4.0/
↪3.0)
sys.omega['polymer','particle']  = pyPRISM.omega.InterMolecular()
sys.omega['particle','particle'] = pyPRISM.omega.SingleSite()

sys.potential['polymer','polymer']   = pyPRISM.potential.HardSphere()
sys.potential['polymer','particle']  = pyPRISM.potential.Exponential(alpha=0.5,
↪epsilon=1.0)
sys.potential['particle','particle'] = pyPRISM.potential.HardSphere()

sys.closure['polymer','polymer']   = pyPRISM.closure.PercusYevick()
sys.closure['polymer','particle']  = pyPRISM.closure.PercusYevick()
sys.closure['particle','particle'] = pyPRISM.closure.HyperNettedChain()

PRISM = sys.solve()

pcf = pyPRISM.calculate.prism.pair_correlation(PRISM)
```

CHAPTER 3

External Resources

Source Code Repository	
Question/Issue Tracker	
Interactive Binder Tutorial	
Anaconda Cloud	
Python Package Index	

CHAPTER 4

Table of Contents

4.1 API

The fundamental PRISM equation is written in Fourier space as

$$\hat{H}(k) = \hat{\Omega}(k)\hat{C}(k)\left[\hat{\Omega}(k) + \hat{H}(k)\right]$$

where $\hat{H}(k)$ is the total correlation function, $\hat{\Omega}(k)$ is the *intra*-molecular correlation function, and $\hat{C}(k)$ is the direct correlation function. At each wavenumber k, each of these variables is an $n \times n$ matrix of values, where n is the number of components or site types in the system. The goal of any PRISM calculation is to obtain the full set of partial correlation functions. Using these correlation functions, a number of structural and thermodynamic properties can be calculated.

The `pyPRISM.core` module holds the fundamental data structures that carry out the PRISM calculation.

The `pyPRISM.calculate` module provides a number of functions which use *solved* `pyPRISM.core.PRISM` objects to calculate structural and thermodynamic parameters.

The `pyPRISM.closure` module provides closure objects which are necessary for solving the PRISM equations.

The `pyPRISM.omega` module provides analytical *intra*-molecular correlation ($\hat{\omega}(k)$) functions along with methods for loading them from memory or files.

The `pyPRISM.potential` module provides pair potentials for describing the *inter*-molecular interactions in a system. Pairwise interactions are also how the chemistry of the system is described.

The `pyPRISM.trajectory` module contains classes for working with molecular simulation trajectories.

The `pyPRISM.util` module provides various global helper functions which do not fall under the above categories.

See the *Tutorial* for more information on the details of using pyPRISM and the PRISM theory formalism.

`pyPRISM.`**`test`**`()`

Run all tests using pytest.

4.1.1 pyPRISM.core package

This module provides the base machinery and data structures for carrying out PRISM calculations.

pyPRISM.core.Density module

class `pyPRISM.core.Density.`**`Density`**(*types*)

Bases: `object`

Container for pair and site densities

Mathematical Definition

$$\rho^{pair}_{\alpha,\beta} = \rho_{\alpha}\rho_{\beta}$$

$$\rho^{site}_{\alpha,\beta} = \begin{cases} \rho_{\alpha} & \text{if } i = j \\ \rho_{\alpha} + \rho_{\beta} & \text{if } i \neq j \end{cases}$$

$$\rho^{total} = \sum_{\alpha} \rho^{site}_{\alpha,\alpha}$$

Variable Definitions

ρ_{α} Number density of site α

$\rho^{pair}_{\alpha,\beta}$ Pair number density of pair α, β

$\rho^{site}_{\alpha,\beta}$ Site number density of pair α, β

ρ^{total} Total site number density

Description

This class describes the makeup of the system in terms of both total site and pair densities. The container provides a simple interface for getting and setting (via square brackets [ ]) site densities and also takes care of calculating the total site and total pair number densities. The total site and pair number densities can be accessed as MatrixArrays (*`pyPRISM.core.MatrixArray`*) attributes.

Example

```
import pyPRISM

rho = pyPRISM.Density(['A','B','C'])

rho['A'] = 0.25
rho['B'] = 0.35
rho['C'] = 0.15

rho.pair['A','B'] #pair density rho_AB = 0.25 * 0.35
rho.site['A','B'] #site density rho_AB = 0.25 + 0.35
rho.site['B','B'] #site density rho_BB = 0.35
rho.total         #total density rho   = 0.25 + 0.35 + 0.15
```

`__init__`(*types*)

Constructor

Parameters **types** (`list`) – List of types of sites

density
pyPRISM.core.ValueTable – Table of site number density values

total
float – Total number density

site
pyPRISM.core.MatrixArray – Site density for each pair.

pair
pyPRISM.core.MatrixArray – Pair site density for each pair.

check()
Are all densities set?

Raises *ValueError* if densities are not all set.

pyPRISM.core.Domain module

class pyPRISM.core.Domain.**Domain**(*length*, *dr=None*, *dk=None*)
Bases: object

Define domain and transform between Real and Fourier space

Mathematical Definition

The continuous, 1-D, radially symmetric Fourier transform is written as follows:

$$k\,\hat{f}(k) = 4\pi \int r\,f(r)\sin(k\,r)dr$$

We define the following discretizations

$$r = (i+1)\Delta r$$
$$k = (j+1)\Delta k$$
$$\Delta k = \frac{\pi}{\Delta r(N+1)}$$

to yield

$$\hat{F}_j = 4\pi\Delta r \sum_{i=0}^{N-1} F_i \sin\left(\frac{\pi}{N+1}(i+1)(j+1)\right)$$

with the following definitions:

$$\hat{F}_j = (j+1)\ \Delta k\ \hat{f}((j+1)\Delta k) = k\hat{f}(k)$$

$$F_i = (i+1)\Delta r\ f((i+1)\Delta r) = rf(r)$$

The above equations describe a Real to Real, type-II discrete sine transform (DST). To tranform to and from Fourier space we will use the type-II and type-III DST's respectively. With Scipy's interface to fftpack, the following functional coeffcients are

$$C^{DSTII} = 2\pi r\Delta r$$

$$C^{DSTIII} = \frac{k\Delta k}{4\pi^2}$$

Description

Domain describes the discretization of Real and Fourier space and also sets up the functions and coefficients for transforming data between them.

`MatrixArray_to_fourier`(*marray*)
Transform all pair-functions of a MatrixArray to Fourier space in-place

Parameters **`marray`** (*`pyPRISM.core.MatrixArray.MatrixArray`*) – MatrixArray to be transformed

Raises *ValueError*: – If the supplied MatrixArray is already in Real-space

`MatrixArray_to_real`(*marray*)
Transform all pair-functions of a MatrixArray to Real space in-place

Parameters **`marray`** (*`pyPRISM.core.MatrixArray.MatrixArray`*) – MatrixArray to be transformed

Raises ValueError: – If the supplied MatrixArray is already in Real-space

`__init__`(*length*, *dr=None*, *dk=None*)
Constructor

Parameters

- **`length`** (*`int`*) – Number of gridpoints in Real and Fourier space grid
- **`dr,dk`** (*`float`*) – Grid spacing in Real space or Fourier space. Only one can be specified as it fixes the other.

`build_grid`()
Construct the Real and Fourier Space grids and transform coefficients

`dk`
Fourier grid spacing

`dr`
Real grid spacing

`length`
Number of points in grid

`to_fourier`(*array*)
Discrete Sine Transform of a numpy array

Parameters **`array`** (*`float ndarray`*) – Real-space data to be transformed

Returns **array** – data transformed to fourier space

Return type float ndarray

Peforms a Real-to-Real Discrete Sine Transform of type II on a numpy array of non-complex values. For radial data that is symmetric in ϕ and :math‘theta‘, this is a correct transform to go from Real-space to Fourier-space.

`to_real`(*array*)
Discrete Sine Transform of a numpy array

Parameters **`array`** (*`float ndarray`*) – Fourier-space data to be transformed

Returns

- **array** (*float ndarray*) – data transformed to Real space
- *Peforms a Real-to-Real Discrete Sine Transform of type III*
- *on a numpy array of non-complex values. For radial data that is*

- **symmetric in :math:'phi' and** (math' heta', this is a correct transform)
- *to go from Real-space to Fourier-space.*

pyPRISM.core.IdentityMatrixArray module

class pyPRISM.core.IdentityMatrixArray.**IdentityMatrixArray**(*length*, *rank*, *data=None*, *space=None*, *types=None*)

Bases: *pyPRISM.core.MatrixArray.MatrixArray*

Specialization of MatrixArray which is initialized with Identity matrices

See *pyPRISM.core.MatrixArray* for details

pyPRISM.core.MatrixArray module

class pyPRISM.core.MatrixArray.**MatrixArray**(*length*, *rank*, *data=None*, *space=<Space.Real: 1>*, *types=None*)

Bases: object

A container for creating and interacting with arrays of matrices

Description

The primary data structure of MatrixArray is simply a 3D Numpy array with the first dimension accessing each individual matrix in the array and the last two dimenions corresponding to the vertical and horizontal index of each matrix element.

The terminology *pair-function* is used to refer to the set of values from all matrices in the array at a given matrix index pair. In Numpy slicing parlance:

```
pair_11 = numpy_array[:,1,1]
pair_12 = numpy_array[:,1,2]
```

Access to the MatrixArray is either by supplied types or numerical indices. If types are not supplied, captial letters starting from 'A' are used.

See the example below and the *pyPRISM Internals* section of the *Tutorial* for more information.

Example

```
mArray = MatrixArray(length=1024,rank=2,types=['polymer','solvent'])

mArray['polymer','solvent'] == mArray['solvent','polymer'] == mArray.get(0,1)
```

SpaceError = 'Attempting MatrixArray math in non-matching spaces'

__getitem__(*key*)

pair-function getter

Parameters

- **key** (*tuple of types*) – Type pair used to identify pair
- **val** (*np.ndarray*) – Values of pair-function

`__imul__`(*other*)
Scalar or elementwise multiplication

`__init__`(*length*, *rank*, *data=None*, *space=<Space.Real: 1>*, *types=None*)
Constructor

Parameters

- **`length`** (*`int`*) – Number of matrices in array. For PRISM theory, this corresponds to the number of grid points in real- and Fourier-space i.e. Domain.size.
- **`rank`** (*`int`*) – Number of rows/cols of each (square) matrix. For PRISM theory, this also equal to the number of site types.
- **`data`** (*`np.ndarray, size (length,rank,rank)`*) – Interface for specifying the MatrixArray data directly. If not given, all values in all matrices will be set to zero.
- **`space`** (`pyPRISM.core.Space.Space`) – Enumerated value tracking whether the array represents real or Fourier spaced data. As we will be transferring arrays to and from these spaces, it's important for safety that we track this.
- **`types`** (list, *optional*) – List of semantic types that are be used to reference data. These types will be output by the iterpair method as well. If not supplied, uppercase letters will be used.

`__itruediv__`(*other*)
Scalar or elementwise division

`__mul__`(*other*)
Scalar or elementwise multiplication

`__setitem__`(*key*, *val*)
pair-function setter

Parameters

- **`key`** (*`tuple of types`*) – Type pair used to identify pair
- **`val`** (*`np.ndarray`*) – Values of pair-function

Assumes all matrices are symmetric and enforces symmetry by setting the off-diagonal elements to be equal.

`__truediv__`(*other*)
Scalar or elementwise division

`dot`(*other*, *inplace=False*)
Matrix multiplication for each matrix in two MatrixArrays

Parameters

- **`other`** (*`object, MatrixArray`*) – Must be an object of MatrixArray type of the same length and dimension
- **`inplace`** (*`bool`*) – If False, a new MatrixArray is returned, otherwise just update the internal data.

`get`(*index1*, *index2*)
pair-function getter via indices

This method should be slightly more efficient than the standard __getitem__.

`getMatrix`(*matrix_index*)
Matrix getter via indices

get_copy()
Return an independent copy of this MatrixArray

invert(*inplace=False*)
Perform matrix inversion on all matrices in the MatrixArray

Parameters **inplace** (*bool*) – If False, a new MatrixArray is returned, otherwise just update the internal data.

itercurve()

iterpairs()
Iterate over the pair-function in this MatrixArray

Yields

- **(i,j)** (*2-tuple of integers*) – numerical index to the underlying data numpy array
- **(t1,t2)** (*2-tuple of string types*) – string index to the underlying data numpy array
- **pair-function** (*np.ndarray, size (self.length)*) – 1-D array representing a pair-function within the MatrixArray

setMatrix(*matrix_index*, *value*)
Matrix setter via indices

pyPRISM.core.PRISM module

class pyPRISM.core.PRISM.**PRISM**(*sys*)
Bases: `object`

Primary container for a storing a PRISM calculation

Each pyPRISM.PRISM object serves as an encapsulation of a fully specified PRISM problem including all inputs needed for the calculation and the function to be numerically minimized.

domain
pyPRISM.Domain – The Domain object fully specifies the Real- and Fourier- space solution grids.

directCorr
pyPRISM.MatrixArray – The direct correlation function for all pairs of site types

omega
pyPRISM.MatrixArray – The intra-molecular correlation function for all pairs of site types

closure
pyPRISM.core.PairTable of pyPRISM.closure.Closure – Table of closure objects used to generate the direct correlation functions (directCorr)

pairCorr
pyPRISM.MatrixArray – The *inter*-molecular pair correlation functions for all pairs of site types. Also commonly refered to as the radial distribution functions.

totalCorr
pyPRISM.MatrixArray – The *inter*-molecular total correlation function is simply the pair correlation function y-shifted by 1.0 i.e. totalCorr = pairCorr - 1.0

potential
pyPRISM.MatrixArray – Interaction potentials for all pairs of sites

`GammaIn,GammaOut`
pyPRISM.MatrixArray – Primary inputs and outputs of the PRISM cost function. Gamma is defined as “totalCorr - directCorr” (in Fourier space) and results from a change of variables used to remove divergences in the closure relations.

`OC,IOC,I,etc`
pyPRISM.MatrixArray – Various MatrixArrays used as intermediates in the PRISM functional. These arrays are pre-allocated and stored for efficiency.

`x,y`
float np.ndarray – Current inputs and outputs of the cost function

`pairDensityMatrix`
float np.ndarray – Rank by rank array of pair densities between sites. See *`pyPRISM.core.Density`*

`siteDensityMatrix`
float np.ndarray – Rank by rank array of site densities. See *`pyPRISM.core.Density`*

`cost:`
Primary cost function used to define the criteria of a “converged” PRISM solution. The numerical solver will be given this function and will attempt to find the inputs (self.x) that make the outputs (self.y) as close to zero as possible.

`cost`(*x*)
Cost function

There are likely several cost functions that could be imagined using the PRISM equations. In this case we formulate a self-consistent formulation where we expect the input of the PRISM equations to be identical to the output.

The goal of the solve method is to numerically optimize the input ($r\gamma_{in}$) so that the output ($r(\gamma_{in} - \gamma_{out})$) is minimized to zero.

`solve`(*guess=None*, *method='krylov'*, *options=None*)
Attempt to numerically solve the PRISM equations

Using the supplied inputs (in the constructor), we attempt to numerically solve the PRISM equations using the scheme laid out in *`cost()`*. If the numerical solution process is successful, the attributes of this class will contain the solved values for a given input i.e. self.totalCorr will contain the numerically optimized (solved) total correlation functions.

This function also does basic checks to ensure that the results are physical. At this point, this consists of checking to make sure that the pair correlation functions are not negative. If this isn’t true a warning is issued to the user.

Parameters

- **`guess`** (*`np.ndarray, size (rank*rank*length)`*) – The initial guess of γ to the numerical solution process. The numpy array should be of size rank x rank x length corresponding to the a full flattened MatrixArray. If not specified, an initial guess of all zeros is used.
- **`method`** (*`string`*) – Set the type of optimization scheme to use. The scipy documentation for scipy.optimize.root details the possible values for this parameter.

options: dict Dictionary of options specific to the chosen solver method. The scipy documentation for scipy.optimize.root details the possible values for this parameter.

pyPRISM.core.PairTable module

class pyPRISM.core.PairTable.**PairTable**(*types*, *name*, *symmetric=True*)

Bases: *pyPRISM.core.Table.Table*

Container for data that is keyed by pairs of types

Description

Since PRISM is a theory based in *pair*-correlation functions, it follows that many of the necessary parameters of the theory are specified between the pairs of types. This goal of this container is to make setting, getting, and checking these data easy.

Setter/getter methods have been set up to set groups of types simultaneously. This allows for the rapid construction of datasets where many of the parameters are repeated. This class also automatically assumes pair-reversibility and handles the setting of unlike pairs automatically i.e. A-B and B-A are set at the same time.

Note that, unlike the *pyPRISM.core.MatrixArray*, this container is not meant to be used for mathematics. The benefit of this is that, for each type, it can contain any arbitrary number, string, or Python object.

See the example below and the *pyPRISM Internals* section of the *Tutorial* for more information.

Example

```
import pyPRISM

PT = pyPRISM.PairTable(['A','B','C'],name='potential')

# Set the 'A-A' pair
PT['A','A']             = 'Lennard-Jones'

# Set the 'B-A', 'A-B', 'B-B', 'B-C', and 'C-B' pairs
PT['B',['A','B','C'] ] = 'Weeks-Chandler-Andersen'

# Set the 'C-A', 'A-C', 'C-C' pairs
PT['C',['A','C'] ]      = 'Exponential'

for i,t,v in PT.iterpairs():
    print('{}) {} for pair {}-{} is {}'.format(i,VT.name,t[0],t[1],v))

# The above loop prints the following:
#   (0, 0)) potential for pair A-A is Lennard-Jones
#   (0, 1)) potential for pair A-B is Weeks-Chandler-Andersen
#   (0, 2)) potential for pair A-C is Exponential
#   (1, 1)) potential for pair B-B is Weeks-Chandler-Andersen
#   (1, 2)) potential for pair B-C is Weeks-Chandler-Andersen
#   (2, 2)) potential for pair C-C is Exponential

for i,t,v in PT.iterpairs(full=True):
    print('{}) {} for pair {}-{} is {}'.format(i,VT.name,t[0],t[1],v))

# The above loop prints the following:
#   (0, 0)) potential for pair A-A is Lennard-Jones
#   (0, 1)) potential for pair A-B is Weeks-Chandler-Andersen
#   (0, 2)) potential for pair A-C is Exponential
```

(continues on next page)

(continued from previous page)

```
#   (1, 0)) potential for pair B-A is Weeks-Chandler-Andersen
#   (1, 1)) potential for pair B-B is Weeks-Chandler-Andersen
#   (1, 2)) potential for pair B-C is Weeks-Chandler-Andersen
#   (2, 0)) potential for pair C-A is Exponential
#   (2, 1)) potential for pair C-B is Weeks-Chandler-Andersen
#   (2, 2)) potential for pair C-C is Exponential
```

`__init__`(*types*, *name*, *symmetric=True*)
Constructor

Parameters

- **`types`** (*`list`*) – Lists of the types that will be used to key the PairTable. The length of this list should be equal to the rank of the PRISM problem to be solved, i.e. len(types) == number of sites in system
- **`name`** (*`string`*) – The name of the PairTable. This is simply used as a convencience for identifying the table internally.
- **`symmetric`** (*`bool`*) – If *True*, the table will automatically set both off-diagonal values during assignment e.g. PT['A','B'] = 5 will set 'A-B' and 'B-A'

`apply`(*func*, *inplace=True*)
Apply a function to all elements in the table in place

Parameters

- **`func`** (*`any object with __call__ method`*) – function to be called on all table elements
- **`inplace`** (*`bool`*) – If *True*, apply modifications to self. Otherwise, create a new PairTable.

`check`()
Is everything in the table set?

Raises ValueError if all values are not set

`exportToMatrixArray`(*space=<Space.Real: 1>*)
Convenience function for converting a table of arrays to a MatrixArray

Warning: This only works if the PairTable contains numerical data that is all of the same shape that can be cast into a np.ndarray like object.

`iterpairs`(*full=False*, *diagonal=True*)
Convenience function for looping over table pairs.

Parameters

- **`full`** (*`bool`*) – If *True*, all i,j pairs (upper and lower diagonal) will be looped over
- **`diagonal`** (*`bool`*) – If *True*, only the i==j (on-diagonal) pairs will be considered when looping

`setUnset`(*value*)
Set all values that have not been specified to a value

Parameters **`value`** – Any valid python object (number, list, array, etc) can be passed in as a value for all unset fields.

pyPRISM.core.Space module

class `pyPRISM.core.Space.`**`Space`**

Bases: `enum.Enum`

An enumeration to track which space an object is in

Description MatrixArrays can represent data in Real- or Fourier- space and they can be transformed in-place between these spaces. This class is meant to help track this state by creating a standard numerical reference for each state that can be checked. This allows classes like `pyPRISM.core.MatrixArray` to do error checking when doing math between arrays. This enumeration also defines a 'wildcard' state so that we can still do math with non-spatial data.

Example

```
import pyPRISM

A = pyPRISM.MatrixArray(length=1000,rank=3,space=pyPRISM.Space.Real)
B = pyPRISM.MatrixArray(length=1000,rank=3,space=pyPRISM.Space.Real)
C = pyPRISM.MatrixArray(length=1000,rank=3,space=pyPRISM.Space.Fourier)

A.space == B.Space # returns True
A.space == C.Space # returns False

A.dot(C) #raises exception
```

Note: The enumerated states of the Space Enum are listed below. The actual values of these states are not-important beyond being unique from one another.

`Fourier = 2`

`NonSpatial = 3`

`Real = 1`

pyPRISM.core.System module

class `pyPRISM.core.System.`**`System`**(*types*, *kT=1.0*)

Bases: `object`

Primary class used to spawn PRISM calculations

Description

The system object contains tables that fully describe a system to be simulated. This includes the domain definition, all site densities, site diameters, interaction potentials, intra-molecular correlation functions ($\hat{\omega}(k)$), and closures. This class also contains a convenience function for spawning a PRISM object.

Example

```python
import pyPRISM

sys = pyPRISM.System(['A','B'])

sys.domain = pyPRISM.Domain(dr=0.1,length=1024)

sys.density['A'] = 0.1
sys.density['B'] = 0.75

sys.diameter[sys.types] = 1.0

sys.closure[sys.types,sys.types] = pyPRISM.closure.PercusYevick()

sys.potential[sys.types,sys.types] = pyPRISM.potential.HardSphere()

sys.omega['A','A'] = pyPRISM.omega.SingleSite()
sys.omega['A','B'] = pyPRISM.omega.InterMolecular()
sys.omega['B','B'] = pyPRISM.omega.Gaussian(sigma=1.0,length=10000)

PRISM = sys.createPRISM()

PRISM.solve()
```

`__init__`(*types*, *kT=1.0*)

Parameters

- **`types`** (*`list`*) – Lists of the site types that define the system
- **`kT`** (*`float`*) – Thermal temperature where k is the Boltzmann constant and T temperature. This is typicaly specified in reduced units where $k_B = 1.0$.

`types`
list – List of site types

`rank`
int – Number of site types

`density`
`pyPRISM.core.Density` – Container for all density values

`potential`
`pyPRISM.core.PairTable` – Table of pair potentials between all site pairs in real space

`closure`
`pyPRISM.core.PairTable` – Table of closures between all site pairs

`omega`
`pyPRISM.core.PairTable` – Table of omega correlation functions in Fourier-space

`domain`
`pyPRISM.core.Domain` – Domain object which specifies the Real and Fourier space solution grid.

`kT`
float – Value of the thermal energy level. Used to vary temperature and scale the potential energy functions.

`diameter`
`pyPRISM.core.ValueTable` – Site diameters.

Warning: These diameters are currently only passed to the closures. They are not passed to potentials and it is up to the user to set sane sigma values that match these diameters.

check()

Is everything in the system specified?

Raises ValueError if all values are not set

createPRISM()

Construct a PRISM object

Note: This method calls `check()` before creating the PRISM object.

Returns **PRISM** – Fully specified PRISM object

Return type pyPRISM.core.PRISM

solve(**args*, ***kwargs*)

Construct a PRISM object and attempt a numerical solution

Note: See `solve()` for arguments to this function

Note: This method calls `check()` before creating the PRISM object.

Returns **PRISM** – **Solved** PRISM object

Return type pyPRISM.core.PRISM

pyPRISM.core.Table module

class pyPRISM.core.Table.**Table**

Bases: `object`

Baseclass used to define tables of parameters

Note: This class should not be used/instatiated directly. It is only intended to be inherited.

listify(*values*)

Helper fuction that converts any input into a list of inputs.

The purpose of this function is to help with iterating over types, and to handle the case of a single `str` type being passed.

pyPRISM.core.ValueTable module

class pyPRISM.core.ValueTable.**ValueTable**(*types*, *name*)

Bases: `pyPRISM.core.Table.Table`

Container for data that is keyed by types

Description

The goal of this class is to provide a simple inteface for setting and storing parameters that are accessed and identified by types. This is typically site properties, e.g. density, site diameter. By default the value for all types is set to *None* and therefore can be checked to see if the table has been fully specified.

Setter/getter methods have been create to set groups of types simultaneously. This allows for the rapid construction of datasets where many of the parameters are repeTated.

Note that, unlike the `pyPRISM.core.MatrixArray`, this container is not meant to be used for mathematics. The benefit of this is that, for each type, it can contain any arbitrary number, string, or Python object.

See the example below and the *pyPRISM Internals* section of the *Tutorial* for more information.

Example

```
import pyPRISM

VT = pyPRISM.ValueTable(['A','B','C','D','E'],name='density')

# set the value for type A to be 0.25
VT['A'] = 0.25

# set the value for types B & C to be 0.35
VT[ ['B','C'] ] = 0.35

# set all other values to be 0.1
VT.setUnset(0.1)

for i,t,v in VT:
    print('{}) {} for type {} is {}'.format(i,VT.name,t,v))

# The above loop prints the following:
#   0) density for type A is 0.25
#   1) density for type B is 0.35
#   2) density for type C is 0.35
#   3) density for type D is 0.1
#   4) density for type E is 0.1
```

__init__(*types*, *name*)
Constructor

Parameters

- **types** (`list`) – Lists of the types that will be used to key the ValueTable. The length of this list should be equal to the rank of the PRISM problem to be solved i.e. len(types) == number of sites in system.
- **name** (`string`) – The name of the ValueTable. Currently, this is simply used as a convencience for identifying the table internally.

__iter__()
Data iterator

This magic-method allows for ValueTables to be iterated over via *for x in y* constructs like

```
for index,type,value in ValueTable:
    print(index,type,value)
```

Yields

- **index** (*int*) – index of value
- *type* – type of value
- *value* – stored value at this type

check()

Is everything in the table set?

Raises *ValueError* if all values are not set

setUnset(*value*)

Set all values that have not been specified to a value

Parameters **value** – Any valid python object (number, list, array, etc) can be passed in as a value for all unset fields.

4.1.2 pyPRISM.calculate package

Once the PRISM equation is solved, $\hat{H}(k)$, $\hat{\Omega}(k)$, and $\hat{C}(k)$ are used to calculate various structural and thermodynamic properties. As listed below, pyPRISM provides functions that calculate these properties from solved *pyPRISM.core.PRISM* objects.

If a desired calculation is not listed, please consider filing an Issue on GitHub, implementing the calculation, and sharing with the community. See *Contributing* for more details.

pyPRISM.calculate.chi module

pyPRISM.calculate.chi.**chi**(*PRISM*, *extrapolate=True*)

Calculate the effective interaction parameter, χ

Parameters

- **PRISM** (*pyPRISM.core.PRISM*) – A **solved** PRISM object.
- **extrapolate** (bool, *optional*) – If *True*, only return the chi value extrapolated to $k = 0$ rather than returning $\chi(k)$

Returns **chi** – PairTable of all $\chi(k)$ or $\chi(k = 0)$ values

Return type pyPRISM.core.PairTable

Mathematical Definition

$$\hat{\chi}_{\alpha,\beta}(k) = \frac{0.5\rho}{R^{+0.5}\phi_{\alpha} + R^{-0.5}\phi_{\beta}}(R^{-1}\hat{C}_{\alpha,\alpha}(k) + R\hat{C}_{\beta,\beta}(k) - 2\hat{C}_{\alpha,\beta}(k))$$

$$R = v_{\alpha}/v_{\beta}$$

Variable Definitions

- $\hat{\chi}_{\alpha,\beta}(k)$ Wavenumber dependent effective interaction parameter between site types α and β
- ρ Total system density from the *pyPRISM.core.Density* instance stored in the system object (which is stored in the PRISM object)

- ϕ_α, ϕ_β Volume fraction of site types α and β.

$$\phi_\alpha = \frac{\rho_\alpha}{\rho_\alpha + \rho_\beta}$$

- v_α, v_β Volume of site type α and β

Description

$\hat{\chi}_{\alpha,\beta}(k)$ describes the overall effective interactions between site types α and β as a single number. While there are many different definitions of χ, this is an effective version that takes into account both *entropic* and *enthalpic* interactions. In this way, this χ is similar to a second virial coefficient. In terms of value, $\chi < 0$ indicates effective attraction and $\chi > 0$ effective repulsion.

As most theories do not take into account the (potentially contentious) wavenumber dependence of χ, the zero-wavenumber extrapolation is often used when reporting PRISM-based χ values. For convenience, the full wavenumber dependent curve can be requested, but only the $k = 0$ values are returned by default.

Warning: The χ calculation is only valid for multicomponent systems i.e. systems with more than one defined type. This method will throw an exception if passed a 1-component PRISM object.

Warning: This calculation is only rigorously defined in the two-component case. With that said, pyPRISM allows this method to be called for multicomponent systems in order to calculate pairwise χ values. We urge caution when using this method for multicomponent systems as it is not clear if this approach is fully rigorous.

Warning: Passing an unsolved PRISM object to this function will still produce output based on the default values of the attributes of the PRISM object.

References

Schweizer, Curro, Thermodynamics of Polymer Blends, J. Chem. Phys., 1989 91 (8) 5059, DOI: 10.1063/1.457598 [link]

Example

```
import pyPRISM

sys = pyPRISM.System(['A','B'])

# ** populate system variables **

PRISM = sys.createPRISM()

PRISM.solve()

chi = pyPRISM.calculate.chi(PRISM)
```

(continues on next page)

(continued from previous page)

```
chi_AB = chi['A','B']
chi_AA = chi['A','A'] #returns None because self-chi values are not defined
```

pyPRISM.calculate.pair_correlation module

pyPRISM.calculate.pair_correlation.**pair_correlation**(*PRISM*)

Calculate the Real-space *inter*-molecular pair correlation function

Parameters **PRISM** (*pyPRISM.core.PRISM*) – A **solved** PRISM object.

Returns **pairCorr** – The full MatrixArray of pair correlation functions.

Return type pyPRISM.core.MatrixArray

Mathematical Definition

$$g_{\alpha,\beta}(r) = h_{\alpha,\beta}(r) + 1.0$$

Variable Definitions

- $g_{\alpha,\beta}(r)$ Pair correlation function between site types α and β at a distance r
- $h_{\alpha,\beta}(r)$ Total correlation function between site types α and β at a distance r

Description

The pair correlation function describes the spatial correlations between pairs of sites in Real-space. Also known as the *radial distribution function* (rdf), the $g(r)$ function is related to the underlying spatial probability distributions of a given system. In a PRISM calculation, $g(r)$ is strictly an *inter*-molecular quantity.

After convergence of a PRISM object, the stored total correlation attribute function can simply be shifted to obtain the $g(r)$

Warning: Passing an unsolved PRISM object to this function will still produce output based on the default values of the attributes of the PRISM object.

Example

```
import pyPRISM

sys = pyPRISM.System(['A','B'])

# ** populate system variables **

PRISM = sys.createPRISM()

PRISM.solve()

rdf = pyPRISM.calculate.pair_correlation(PRISM)

rdf_AA = rdf['A','A']
rdf_AB = rdf['A','B']
rdf_BB = rdf['B','B']
```

pyPRISM.calculate.pmf module

`pyPRISM.calculate.pmf.`**`pmf`**`(`*PRISM*`)`

Calculate the potentials of mean force

Parameters **`PRISM`** (*`pyPRISM.core.PRISM`*) – A **solved** PRISM object.

Returns **pmf** – The full MatrixArray of potentials of mean force

Return type pyPRISM.core.MatrixArray

Mathematical Definition

$$w_{\alpha,\beta}(r) = -k_B T \ln(h_{\alpha,\beta}(r) + 1.0)$$

Variable Definitions

- $w_{\alpha,\beta}(r)$ Potential of mean force between site types α and β at a distance r
- $g_{\alpha,\beta}(r)$ Pair correlation function between site types α and β at a distance r
- $h_{\alpha,\beta}(r)$ Total correlation function between site types α and β at a distance r

Description

A potential of mean force (PMF) between site types α and β, $w_{\alpha,\beta}$ represents the the ensemble averaged free energy change needed to bring these two sites from infinite separation to a distance r. It can also be thought of as a potential that would be needed to reproduce the underlying $g_{\alpha,\beta}(r)$.

Warning: Passing an unsolved PRISM object to this function will still produce output based on the default values of the attributes of the PRISM object.

Example

```python
import pyPRISM

sys = pyPRISM.System(['A','B'])

# ** populate system variables **

PRISM = sys.createPRISM()

PRISM.solve()

pmf = pyPRISM.calculate.pmf(PRISM)

pmf_BB = pmf['B','B']
```

pyPRISM.calculate.second_virial module

`pyPRISM.calculate.second_virial.`**`second_virial`**`(`*PRISM*, *extrapolate=True*`)`

Calculate the second virial coefficient

Parameters

- **`PRISM`** (*`pyPRISM.core.PRISM`*) – A **solved** PRISM object.

- **extrapolate** (bool, *optional*) – If *True*, extrapolate $h_{\alpha,\beta}$ to $k = 0$ rather than reporting the value at the lowest-k. Defaults to *True*.

Returns **B2** – Pairtable of B2 values

Return type pyPRISM.core.PairTable

Mathematical Definition

$$B_2^{\alpha,\beta} = -0.5\hat{h}_{\alpha,\beta}(k=0)$$

Variable Definitions

- $B_2^{\alpha,\beta}$ Second virial coefficient between site types α and β
- $\hat{h}_{\alpha,\beta}(k)$ Fourier-space total correlation function between site types α and β at wavevector k

Description

The second virial coefficient ($B_2^{\alpha,\beta}$) is a thermodynamic descriptor related to the pairwise interactions between components in a system. In general, $B_2^{\alpha,\beta} > 0$ signifies repulsive interactions between site types α and β, and $B_2^{\alpha,\beta} < 0$ signifies attractive interactions. For example, in a polymer-solvent system, one definition of the theta condition is when $B_2^{\alpha,\beta} = 0$.

Warning: Passing an unsolved PRISM object to this function will still produce output based on the default values of the attributes of the PRISM object.

Example

```
import pyPRISM

sys = pyPRISM.System(['A','B'])

# ** populate system variables **

PRISM = sys.createPRISM()

PRISM.solve()

B2 = pyPRISM.calculate.second_virial(PRISM)

B2_BB = B2['B','B']
```

pyPRISM.calculate.solvation_potential module

pyPRISM.calculate.solvation_potential.**solvation_potential**(*PRISM*, *closure='HNC'*)

Calculate the pairwise decomposed medium-induced solvation potential

Parameters

- **PRISM** (*pyPRISM.core.PRISM*) – A **solved** PRISM object.
- **closure** (*str ('PY' or 'HNC')*) – closure used to derive the potential

Returns **psi** – MatrixArray of the *Real-space* solvation potentials

Return type pyPRISM.core.MatrixArray

Mathematical Definition

$$\text{PY: } \Delta\hat{\Psi}^{PY}(k) = -k_B T \ln(1 + \hat{C}(k)\hat{S}(k)\hat{C}(k))$$

$$\text{HNC: } \Delta\hat{\Psi}^{HNC}(k) = -k_B T \hat{C}(k)\hat{S}(k)\hat{C}(k)$$

Variable Definitions

- $\Delta\hat{\Psi}^{PY}$, $\Delta\hat{\Psi}^{HNC}$ Percus-Yevick and Hypernetted Chain derived pairwise decomposed solvation potentials, each described as a `MatrixArray`. This implies that the multiplication in the above equation is actually *matrix* multiplication and the individual solvation potentials are extracted as pair-functions of the MatrixArrays. Note that the solvation potential MatrixArrays are inverted back to Real-space for use.
- $\hat{C}(k)$ Direct correlation function `MatrixArray` at a wavenumber k
- $\hat{S}(k)$ Structure factor `MatrixArray` at a wavenumber k
- $k_B T$ Thermal temperature written as the product of the Boltzmann constant and temperature.

Description

The solvation potential ($\Delta\hat{\Psi}$) mathematically describes how a given surrounding medium perturbs the site-site pairwise interactions of a molecule.

This calculation is the foundation of the Self-Consistent PRISM formalism. See *Self-Consistent PRISM Method* for more information.

Warning: Passing an unsolved PRISM object to this function will still produce output based on the default values of the attributes of the PRISM object.

References

1. Grayce, Schweizer, Solvation potentials for macromolecules, J. Chem. Phys., 1994 100 (9) 6846 [link]
2. Schweizer, Honnell, Curro, Reference interaction site model theory of polymeric liquids: Self-consistent formulation and nonideality effects in dense solutions and melts, J. Chem. Phys., 1992 96 (4) 3211 [link]

Example

```python
import pyPRISM

sys = pyPRISM.System(['A','B'])

# ** populate system variables **

PRISM = sys.createPRISM()

PRISM.solve()

psi = pyPRISM.calculate.solvation_potential(PRISM)

psi_BB = psi['B','B']
```

pyPRISM.calculate.spinodal_condition module

`pyPRISM.calculate.spinodal_condition.`**`spinodal_condition`**(*PRISM*, *extrapolate=True*)

Calculate the spinodal condition between pairs of components

Parameters

- **`PRISM`** (*`pyPRISM.core.PRISM`*) – A **solved** PRISM object.
- **`extrapolate`** (bool, *optional*) – If *True*, only return the value extrapolated to $k = 0$ rather than reporting the value at the lowest-k. Defaults to *True*.

Returns **lambda** – The full MatrixArray of structure factors

Return type pyPRISM.core.MatrixArray

Mathematical Definition

$$\begin{aligned}
\hat{\Lambda}_{\alpha,\beta}(k) = 1 &- \rho^{site}_{\alpha,\alpha}\hat{C}_{\alpha,\alpha}(k)\hat{\omega}_{\alpha,\alpha}(k) \\
&- 2\rho^{site}_{\alpha,\beta}\hat{C}_{\alpha,\beta}(k)\hat{\omega}_{\alpha,\beta}(k) \\
&- \rho^{site}_{\beta,\beta}\hat{C}_{\beta,\beta}\hat{\omega}_{\beta,\beta}(k) \\
&+ \rho^{site}_{\alpha,\beta}\rho^{site}_{\alpha,\beta}\hat{C}_{\alpha,\beta}(k)\hat{C}_{\alpha,\beta}(k)\hat{\omega}_{\alpha,\beta}(k)\hat{\omega}_{\alpha,\beta}(k) \\
&- \rho^{site}_{\alpha,\beta}\rho^{site}_{\alpha,\beta}\hat{C}_{\alpha,\alpha}(k)\hat{C}_{\beta,\beta}(k)\hat{\omega}_{\alpha,\beta}(k)\hat{\omega}_{\alpha,\beta}(k) \\
&+ \rho^{site}_{\alpha,\alpha}\rho^{site}_{\beta,\beta}\hat{C}_{\alpha,\alpha}(k)\hat{C}_{\beta,\beta}(k)\hat{\omega}_{\alpha,\alpha}(k)\hat{\omega}_{\beta,\beta}(k) \\
&- \rho^{site}_{\alpha,\alpha}\rho^{site}_{\beta,\beta}\hat{C}_{\alpha,\beta}(k)\hat{C}_{\alpha,\beta}(k)\hat{\omega}_{\alpha,\alpha}(k)\hat{\omega}_{\beta,\beta}(k)
\end{aligned}$$

Variable Definitions

- $\hat{\omega}_{\alpha,\beta}(k)$ Intra-molecular correlation function between sites α and β at a wavenumber k
- $\hat{c}_{\alpha,\beta}(k)$ Direct correlation function between sites α and β at a wavenumber k
- $\rho^{site}_{\alpha,\beta}$ Sitewise density for sites α and β. See `pyPRISM.core.Density` for details.

Description

The spinodal condition ($\hat{\Lambda}_{\alpha,\beta}(k)$) can be used to identify liquid-liquid macrophase separation between site types α and β when $\hat{\Lambda}_{\alpha,\beta}(k \rightarrow 0) = 0$

Warning: Passing an unsolved PRISM object to this function will still produce output based on the default values of the attributes of the PRISM object.

References

1. Schweizer, Curro, Integral equation theory of the structure and thermodynamics of polymer blends, J. Chem. Phys., 1989 91 (8) 5059 [link]

Example

```
import pyPRISM

sys = pyPRISM.System(['A','B'])
```

(continues on next page)

(continued from previous page)

```
# ** populate system variables **

PRISM = sys.createPRISM()

PRISM.solve()

spin = pyPRISM.calculate.spinodal_conditon(PRISM)

spin_AB = spin['A','B']
```

pyPRISM.calculate.structure_factor module

`pyPRISM.calculate.structure_factor.`**`structure_factor`**(*PRISM*, *normalize=True*)

Calculate the structure factor from a PRISM object

Parameters

- **`PRISM`** (*`pyPRISM.core.PRISM`*) – A **solved** PRISM object.
- **`normalize`** (*`bool`*) – normalize the structure factor by the site density

Returns **structureFactor** – The full MatrixArray of structure factors

Return type pyPRISM.core.MatrixArray

Mathematical Definition

$$\hat{s}_{\alpha,\beta}(k) = \rho^{site}_{\alpha,\beta}\hat{\omega}_{\alpha,\beta}(k) + \rho^{pair}_{\alpha,\beta}\hat{h}_{\alpha,\beta}(k)$$

$$\hat{s}^{norm}_{\alpha,\beta}(k) = \hat{s}_{\alpha,\beta}(k)/\rho^{site}_{\alpha,\beta}$$

Variable Definitions

- $\hat{\omega}_{\alpha,\beta}(k)$ Intra-molecular correlation function between sites α and β at a wavenumber k
- $\hat{h}_{\alpha,\beta}(k)$ Total correlation function between sites α and β at a wavenumber k
- $\rho^{site}_{\alpha,\beta}$, $\rho^{pair}_{\alpha,\beta}$ Sitewise and pairwise densities for sites α and β. See *`pyPRISM.core.Density`* for details.

Description

The structure factor ($\hat{s}_{\alpha,\beta}(k)$) is a Fourier-space representation of the structural correlations between sites α and β. The $\hat{s}_{\alpha,\beta}(k)$ can be related to the real-space pair correlation function through a Fourier transform. In the PRISM formalism, the $\hat{s}_{\alpha,\beta}(k)$ can be calculated as the sum of the Fourier-space intra-molecular and total correlation functions, as shown above.

Warning: Passing an unsolved PRISM object to this function will still produce output based on the default values of the attributes of the PRISM object.

References

1. Chandler, D., Introduction to Modern Statistical Mechanics, Oxford U. Press, New York, 1987 [link]
2. Schweizer, Curro, Integral equation theory of the structure and thermodynamics of polymer blends, J. Chem. Phys., 1989 91 (8) 5059 [link]

Example

```
import pyPRISM

sys = pyPRISM.System(['A','B'])

# ** populate system variables **

PRISM = sys.createPRISM()

PRISM.solve()

sk = pyPRISM.calculate.structure_factor(PRISM)

sk_BB = sk['B','B']
```

4.1.3 pyPRISM.closure package

While the PRISM equation specifies the base PRISM formalism, we need additional equations called closures to numerically solve the PRISM equations for $\hat{H}(k)$ and $\hat{C}(k)$. Closures provide a mathematical relation between the direct correlation function $c(r)$, the pairwise interaction potential $u(r)$, and, often, the total correlation function $h(r)$. Since the closures include $u(r)$, it is through these closures that the chemical details of the system are specified.

pyPRISM.closure.AtomicClosure module

class `pyPRISM.closure.AtomicClosure.`**`AtomicClosure`**

Bases: `object`

Baseclass for all atomic closures

Note: Currently, this class doesn't do anything besides group all of the *atomic* closures under a single inheritance heirarchy.

pyPRISM.closure.Closure module

class `pyPRISM.closure.Closure.`**`Closure`**

Bases: `object`

Baseclass for all closures

Note: Currently, this class doesn't do anything besides group all of the closures under a single inheritance heirarchy.

pyPRISM.closure.HyperNettedChain module

class `pyPRISM.closure.HyperNettedChain.`**`HNC`**(*apply_hard_core=False*)

Bases: *`pyPRISM.closure.HyperNettedChain.HyperNettedChain`*

Alias of HyperNettedChain

class pyPRISM.closure.HyperNettedChain.**HyperNettedChain**(*apply_hard_core=False*)

Bases: *pyPRISM.closure.AtomicClosure.AtomicClosure*

HyperNettedChain closure

Mathematial Definition

$$c_{\alpha,\beta}(r) = \exp\left(\gamma_{\alpha,\beta}(r) - U_{\alpha,\beta}(r)\right) - 1.0 - \gamma_{\alpha,\beta}(r)$$

$$\gamma_{\alpha,\beta}(r) = h_{\alpha,\beta}(r) - c_{\alpha,\beta}(r)$$

Variables Definitions

- $h_{\alpha,\beta}(r)$ Total correlation function value at distance r between sites α and β.
- $c_{\alpha,\beta}(r)$ Direct correlation function value at distance r between sites α and β.
- $U_{\alpha,\beta}(r)$ Interaction potential value at distance r between sites α and β.

Description

The Hypernetted Chain Closure (HNC) is derived by expanding the direct correlation function, $c(r)$, in powers of density shift from a reference state. See Reference [1] for a full derivation and discussion of this closure.

The change of variables is necessary in order to use potentials with hard cores in the computational setting. Written in the standard form, this closure diverges with divergent potentials, which makes it impossible to numerically solve.

Compared to the PercusYevick closure, the HNC closure is a more accurate approximation of the full expression for the direct correlation function. Depsite this, it can produce inaccurate, long-range fluctuations that make it difficult to employ in phase-separating systems. The HNC closure performs well for systems where there is a disparity in site diameters and is typically used for the larger site.

References

1. Hansen, J.P.; McDonald, I.R.; Theory of Simple Liquids; Chapter 4, Section 4; 4th Edition (2013), Elsevier [link]

Example

```python
import pyPRISM

sys = pyPRISM.System(['A','B'])

sys.closure['A','A'] = pyPRISM.closure.PercusYevick()
sys.closure['A','B'] = pyPRISM.closure.PercusYevick()
sys.closure['B','B'] = pyPRISM.closure.HypernettedChain()

# ** finish populating system object **

PRISM = sys.createPRISM()

PRISM.solve()
```

`__init__`(*apply_hard_core=False*)

Contstructor

Parameters **`apply_hard_core`** (`bool`) – If *True*, the total correlation function will be assumed to be -1 inside the core ($r_{i,j} < (d_i + d_j)/2.0$) and the closure will not be applied in this region. Defaults to *True*.

`calculate`(*r*, *gamma*)

Calculate direct correlation function based on supplied γ

Parameters

- **`r`** (`np.ndarray`) – array of real-space values associated with γ
- **`gamma`** (`np.ndarray`) – array of γ values used to calculate the direct correlation function

pyPRISM.closure.MartynovSarkisov module

class `pyPRISM.closure.MartynovSarkisov.`**`MS`**(*apply_hard_core=False*)

Bases: *`pyPRISM.closure.MartynovSarkisov.MartynovSarkisov`*

Alias of MartynovSarkisov

class `pyPRISM.closure.MartynovSarkisov.`**`MartynovSarkisov`**(*apply_hard_core=False*)

Bases: *`pyPRISM.closure.AtomicClosure.AtomicClosure`*

MartynovSarkisov closure

Mathematial Definition

$$c_{\alpha,\beta}(r) = \left(\exp\left(\sqrt{\gamma_{\alpha,\beta}(r) - U_{\alpha,\beta}(r) - 0.5}\right) - 1.0\right) - 1.0 - \gamma_{\alpha,\beta}(r)$$

$$\gamma_{\alpha,\beta}(r) = h_{\alpha,\beta}(r) - c_{\alpha,\beta}(r)$$

Variables Definitions

- $h_{\alpha,\beta}(r)$ Total correlation function value at distance r between sites α and β.
- $c_{\alpha,\beta}(r)$ Direct correlation function value at distance r between sites α and β.
- $U_{\alpha,\beta}(r)$ Interaction potential value at distance r between sites α and β.

Description

The Martynov-Sarkisov (MS) closure is described as a generalization of the HyperNettedChain closure. See the references below for derivation and usage examples.

The change of variables is necessary in order to use potentials with hard cores in the computational setting. Written in the standard form, this closure diverges with divergent potentials, which makes it impossible to numerically solve.

The MS closure has been shown to be very accurate for hard-sphere spherical molecules and for high-density hard-core polymer systems.

References

1. Martynov, G.A.; Sarkisov, G.N.; Mol. Phys. 49. 1495 (1983) [link]
2. Yethiraj, A.; Schweizer, K.S.; J. Chem. Phys. 97. 1455 (1992) [link]

Example

```
import pyPRISM

sys = pyPRISM.System(['A','B'])

sys.closure['A','A'] = pyPRISM.closure.PercusYevick()
sys.closure['A','B'] = pyPRISM.closure.PercusYevick()
sys.closure['B','B'] = pyPRISM.closure.MartynovSarkisov()

# ** finish populating system object **

PRISM = sys.createPRISM()

PRISM.solve()
```

`__init__`(*apply_hard_core=False*)
Contstructor

Parameters **`apply_hard_core`** (`bool`) – If *True*, the total correlation function will be assumed to be -1 inside the core ($r_{i,j} < (d_i + d_j)/2.0$) and the closure will not be applied in this region.

`calculate`(*r*, *gamma*)
Calculate direct correlation function based on supplied γ

Parameters

- **`r`** (`np.ndarray`) – array of real-space values associated with γ
- **`gamma`** (`np.ndarray`) – array of γ values used to calculate the direct correlation function

pyPRISM.closure.MeanSphericalApproximation module

class `pyPRISM.closure.MeanSphericalApproximation.`**`MSA`**(*apply_hard_core=False*)
Bases: *`pyPRISM.closure.MeanSphericalApproximation.MeanSphericalApproximation`*

Alias of MeanSphericalApproximation

class `pyPRISM.closure.MeanSphericalApproximation.`**`MeanSphericalApproximation`**(*apply_hard_core=Fa*
Bases: *`pyPRISM.closure.AtomicClosure.AtomicClosure`*

Mean Spherical Approximation closure

Mathematial Definition

$$c_{\alpha,\beta}(r) = -U_{\alpha,\beta}(r)$$

Variables Definitions

- $c_{\alpha,\beta}(r)$ Direct correlation function value at distance r between sites α and β.
- $U_{\alpha,\beta}(r)$ Interaction potential value at distance r between sites α and β.

Description

The Mean Spherical Approximation (MSA) closure assumes an interaction potential that contains a hard-core interaction and a tail interaction. See Reference [1] for a derivation and discussion of this closure.

The MSA does a good job of describing the properties of the square-well fluid, and allows for the analytical solution of the PRISM/RISM equations for some systems. The MSA closure reduces to the PercusYevick closure if the tail is ignored.

References

1. Hansen, J.P.; McDonald, I.R.; Theory of Simple Liquids; Chapter 4, Section 4; 4th Edition (2013), Elsevier [link]

Example

```
import pyPRISM

sys = pyPRISM.System(['A','B'])

sys.closure['A','A'] = pyPRISM.closure.PercusYevick()
sys.closure['A','B'] = pyPRISM.closure.PercusYevick()
sys.closure['B','B'] = pyPRISM.closure.MeanSphericalApproximation()

# ** finish populating system object **

PRISM = sys.createPRISM()

PRISM.solve()
```

__init__(*apply_hard_core=False*)

Contstructor

Parameters **apply_hard_core** (*bool*) – If *True*, the total correlation function will be assumed to be -1 inside the core ($r_{i,j} < (d_i + d_j)/2.0$) and the closure will not be applied in this region.

calculate(*r*, *gamma*)

Calculate direct correlation function based on supplied γ

Parameters

- **r** (*np.ndarray*) – array of real-space values associated with γ
- **gamma** (*np.ndarray*) – array of γ values used to calculate the direct correlation function

pyPRISM.closure.MolecularClosure module

class pyPRISM.closure.MolecularClosure.**MolecularClosure**

Bases: object

Baseclass for all *molecular* closures

Note: Currently, this class doesn't do anything besides group all of the *molecular* closures under a single inheritance heirarchy.

pyPRISM.closure.PercusYevick module

class `pyPRISM.closure.PercusYevick.`**`PY`**(*apply_hard_core=False*)

Bases: `pyPRISM.closure.PercusYevick.PercusYevick`

Alias of PercusYevick

class `pyPRISM.closure.PercusYevick.`**`PercusYevick`**(*apply_hard_core=False*)

Bases: `pyPRISM.closure.AtomicClosure.AtomicClosure`

Percus Yevick closure evaluated in terms of a change of variables

Mathematial Definition

$$c_{\alpha,\beta}(r) = (\exp(-U_{\alpha,\beta}(r)) - 1.0)(1.0 + \gamma_{\alpha,\beta}(r))$$

$$\gamma_{\alpha,\beta}(r) = h_{\alpha,\beta}(r) - c_{\alpha,\beta}(r)$$

Variables Definitions

- $h_{\alpha,\beta}(r)$ Total correlation function value at distance r between sites α and β.
- $c_{\alpha,\beta}(r)$ Direct correlation function value at distance r between sites α and β.
- $U_{\alpha,\beta}(r)$ Interaction potential value at distance r between sites α and β.

Description

The Percus-Yevick (PY) is derived by expanding the exponential of the direct correlation function, $c_{\alpha,\beta}(r)$, in powers of density shift from a reference state. See Reference [1] for a full derivation.

The change of variables is necessary in order to use potentials with hard cores in the computational setting. Written in the standard form, this closure diverges with divergent potentials, which makes it impossible to numerically solve.

This closure has been shown to be accurate for systems with hard cores (strongly repulsive at short distances) and when the potential is short ranged.

References

1. Hansen, J.P.; McDonald, I.R.; Theory of Simple Liquids; Chapter 4, Section 4; 4th Edition (2013), Elsevier [link]

Example

```
import pyPRISM

sys = pyPRISM.System(['A','B'])
```

(continues on next page)

(continued from previous page)

```
sys.closure['A','A'] = pyPRISM.closure.PercusYevick()
sys.closure['A','B'] = pyPRISM.closure.PercusYevick()
sys.closure['B','B'] = pyPRISM.closure.HypernettedChain()

# ** finish populating system object **

PRISM = sys.createPRISM()

PRISM.solve()
```

__init__(*apply_hard_core=False*)

Contstructor

Parameters **apply_hard_core** (*bool*) – If True, the total correlation function will be assumed to be -1 inside the core ($r_{i,j} < (d_i + d_j)/2.0$) and the closure will not be applied in this region.

calculate(*r*, *gamma*)

Calculate direct correlation function based on supplied γ

Parameters

- **r** (*np.ndarray*) – array of real-space values associated with γ
- **gamma** (*np.ndarray*) – array of γ values used to calculate the direct correlation function

4.1.4 pyPRISM.omega package

In PRISM, the molecular structure of molecules is encoded into *intra*-molecular correlation functions called $\hat{\omega}(k)$. All connectivity and *intra*-molecular excluded volume is contained in these functions. During a PRISM calculation, $\hat{\omega}(k)$ are specified as input and are held fixed during the solution procedure. While this leads to a decoupling of the *intra*-molecular and *inter*-molecular correlations within a given PRISM calculation, methods such as self-consistent PRISM offer a route to mitigating this potential problem. See *Self-Consistent PRISM Method* for more details.

pyPRISM offers several analytical form factors which are listed below. If you have an analytical $\hat{\omega}(k)$ that is not listed, please consider filing an Issue on GitHub, implementing the $\hat{\omega}(k)$ and sharing with the community. See *Contributing* for more details.

Alternatively, if no analytical form exists, $\hat{\omega}(k)$ can be calculated using a simulation. The *pyPRISM.trajectory.Debyer* class implements the Debye summation method for calculating $\hat{\omega}(k)$ from simulation.

Finally, the *FromArray* and *FromFile* classes exist for loading $\hat{\omega}(k)$ calculated in memory or from another program.

pyPRISM.omega.DiscreteKoyama module

class pyPRISM.omega.DiscreteKoyama.**DiscreteKoyama**(*sigma*, *l*, *length*, *lp*)

Bases: *pyPRISM.omega.Omega.Omega*

Semi-flexible Koyama-based intra-molecular correlation function

Mathematial Definition

$$\hat{\omega}(k) = \frac{\sin(Bk)}{Bk}\exp(-A^2k^2)$$

$$A^2 = \frac{\langle r^2_{\alpha,\beta} \rangle (1 - C)}{6}$$

$$B^2 = C \langle r^2_{\alpha,\beta} \rangle$$

$$C^2 = \frac{1}{2}\left(5 - 3\frac{\langle r^4_{\alpha,\beta} \rangle}{\langle r^2_{\alpha,\beta} \rangle}\right)$$

Variable Definitions

- $\hat{\omega}(k)$ *intra*-molecular correlation function at wavenumber k
- $\langle r^2_{\alpha,\beta} \rangle$ second moment of the distance distribution between sites α and β. Please see equation (17) of the Reference [1] cited below for the mathematical representation.
- $\langle r^4_{\alpha,\beta} \rangle$ fourth moment of the distance distribution between sites α and β. Please see equations (18-24) of the reference cited below for the mathematical representation.

Description

The discrete Koyama $\hat{\omega}(k)$ was developed to represent a wormlike chain with semiflexibility. This scheme interpolates between the rigid-rod and the Gaussian chain limits to represent a chain with a given persistence length. This form for $\hat{\omega}(k)$ has been shown to match the structure of molecular dynamics simulations of Kremer-Grest style bead-spring polymer models.

References

1. Honnell, K.G., J.G. Curro, and K.S. Schweizer, LOCAL-STRUCTURE OF SEMIFLEXIBLE POLYMER MELTS. Macromolecules, 1990. 23(14): p. 3496-3505. [link]

Example

```python
import pyPRISM
import numpy as np
import matplotlib.pyplot as plt

#calculate Fourier space domain and omega values
domain = pyPRISM.domain(dr=0.1,length=1000)
omega  = pyPRISM.omega.DiscreteKoyama(sigma=1.0,l=1.0,length=100,lp=1.43)
x = domain.k
y = omega.calculate(x)

#plot the results using matplotlib
plt.plot(x,y)
plt.gca().set_xscale("log", nonposx='clip')
plt.gca().set_yscale("log", nonposy='clip')

plt.show()

#define a PRISM system and set omega(k) for type A
sys = pyPRISM.System(['A','B'],kT=1.0)
sys.domain = pyPRISM.Domain(dr=0.1,length=1024)
sys.omega['A','A']  = pyPRISM.omega.DiscreteKoyama(sigma=1.0,l=1.0,length=100,
↪lp=1.43)
```

__init__(*sigma*, *l*, *length*, *lp*)

Constructor

Parameters

- **sigma** (*float*) – contact distance between sites (i.e. site diameter)
- **l** (*float*) – bond length
- **length** (*float*) – number of monomers/sites in the chain
- **lp** (*float*) – persistence length of chain

calculate(*k*)

Return value of $\hat{\omega}$ at supplied k

Parameters **k** (*np.ndarray, float*) – array of wavenumber values to calculate ω at

cos_avg(*epsilon*)

First moment of bond angle distribution

cos_sq_avg(*epsilon*)

Second moment of bond angle distribution

density_correction(*npts=1000*)

Correction for density due to non-physical overlaps

Note: See Equation 28 in Reference [1] for more details.

Parameters **npts** (*int*) – number of points to use in numerical integral

density_correction_kernel(*r*)

Correction for density due to non-physical overlaps

Note: See Equation 28 in Reference [1] for more details.

Parameters **r** (*np.ndarray, float*) – array of real-space positions to calculate ω at

kernel_base(*n*)

Calculates the second and fourth moments of the site separate distance distributions

Note: See Equation 18 in Reference [1] for more details.

Parameters **n** (*int*) – Integer separation distance along chain.

koyama_kernel_fourier(*k, n*)

Kernel for calculating omega in Fourier-Space

Note: See Equation 16 in Reference [1] for more details.

Parameters

- **k** (*np.ndarray, float*) – array of wavenumber values to calculate ω at
- **n** (*int*) – Integer separation distance along chain.

koyama_kernel_real (*r*, *n*)

Kernel for calculating omega in Real-Space

Note: See Equation 12 in Reference [1] for more details.

Parameters

- **r** (*np.ndarray, float*) – array of real-space positions to calculate ω at
- **n** (*int*) – Integer separation distance along chain.

pyPRISM.omega.FreelyJointedChain module

class pyPRISM.omega.FreelyJointedChain.**FJC** (*length*, *l*)

Bases: *pyPRISM.omega.FreelyJointedChain.FreelyJointedChain*

Alias of FreelyJointedChain

class pyPRISM.omega.FreelyJointedChain.**FreelyJointedChain** (*length*, *l*)

Bases: *pyPRISM.omega.Omega.Omega*

Freely jointed chain intra-molecular correlation function

Mathematical Definition

$$\hat{\omega}(k) = \frac{1 - E^2 - \frac{2E}{N} + \frac{2E^{N+1}}{N}}{(1-E)^2}$$

$$E = \frac{\sin(kl)}{kl}$$

Variable Definitions

- $\hat{\omega}(k)$ *intra*-molecular correlation function at wavenumber k
- N number of repeat units in chain
- l bond-length

Description

The freely-jointed chain is an ideal polymer chain model that assumes a constant bond length l and no correlations between the directions of different bond vectors (i.e. $< cos(\theta_{ij}) >= 0$). In other words, monomer segments are assumed to have no intra-molecular excluded volume.

References

1. Schweizer, K.S.; Curro, J.G.; Integral-Equation Theory of Polymer Melts - Intramolecular Structure, Local Order, and the Correlation Hole, Macromolecules, 1988, 21 (10), pp 3070 [link]
2. Rubinstein, M; Colby, R.H; Polymer Physics. 2003. Oxford University Press.

Example

```
import pyPRISM
import numpy as np
import matplotlib.pyplot as plt

#calculate Fourier space domain and omega values
domain = pyPRISM.domain(dr=0.1,length=1000)
omega  = pyPRISM.omega.FreelyJointedChain(length=100,l=1.0)
x = domain.k
y = omega.calculate(x)

#plot using matplotlib
plt.plot(x,y)
plt.gca().set_xscale("log", nonposx='clip')
plt.gca().set_yscale("log", nonposy='clip')

plt.show()

#define a PRISM system and set omega(k) for type A
sys = pyPRISM.System(['A','B'],kT=1.0)
sys.domain = pyPRISM.Domain(dr=0.1,length=1024)
sys.omega['A','A']  = pyPRISM.omega.FreelyJointedChain(length=100,l=1.0)
```

`__init__`(*length*, *l*)
Constructor

Parameters

- **`length`** (*`float`*) – number of monomers/sites in Freely-jointed chain
- **`l`** (*`float`*) – bond length

`calculate`(*k*)
Return value of $\hat{\omega}$ at supplied k

Parameters **`k`** (*`np.ndarray`*) – array of wavenumber values to calculate ω at

pyPRISM.omega.FromArray module

class `pyPRISM.omega.FromArray.`**`FromArray`**(*omega*, *k=None*)
Bases: *`pyPRISM.omega.Omega.Omega`*

Read *intra*-molecular correlations from a list or array

This class reads the omega from a Python list or Numpy array.

`omega`
np.ndarray – Intra-molecular omega

`k`
*np.narray, *optional** – Domain of the array data. If provided, this will be checked against the Fourier-space grid specified in the *`pyPRISM.core.Domain`*. An exception will be raised is they do not match.

Example

```
import pyPRISM
import numpy as np
import matplotlib.pyplot as plt
```

(continues on next page)

(continued from previous page)

```
#set all omega(k) = 1 for type A
sys = pyPRISM.System(['A','B'],kT=1.0)
sys.domain = pyPRISM.Domain(dr=0.1,length=1024)
omega = np.ones(sys.domain.k.shape[0])
sys.omega['A','A']  = pyPRISM.omega.FromArray(omega)
x = sys.domain.k
y = sys.omega['A','A'].calculate(x)

#plot using matplotlib
plt.plot(x,y)
plt.gca().set_xscale("log", nonposx='clip')
plt.gca().set_yscale("log", nonposy='clip')

plt.show()
```

`__init__`(*omega*, *k=None*)

Constructor

Parameters

- **omega** (*`list`*, *`np.ndarray`*) – Python list or Numpy array containing values of omega as a function of wavenumber k.
- **k** (np.ndarray, *optional*) – Python list of Numpy array containing values of k. These must match the k values stored in the *`pyPRISM.core.Domain`* or an exception will be raised.

`calculate`(*k*)

Return value of $\hat{\omega}$ at supplied k

Parameters **k** (*`np.ndarray`*) – array of wavenumber values to calculate ω at

pyPRISM.omega.FromFile module

class pyPRISM.omega.FromFile.**FromFile**(*fileName*)

Bases: *`pyPRISM.omega.Omega.Omega`*

Read *intra*-molecular correlations from file

This class reads a one or two column file using Numpy's loadtxt function. A one-column file is expected to only contain intra-molecular correlation data, while a two-column file contains the k values of the data in the first column as well. If the k-values are provided, an extra check to make sure that the file data matches the same k-space grid of the domain.

fileName

str – full path + name to column file

Example

```
import pyPRISM
import numpy as np
import matplotlib.pyplot as plt

#set type A omega(k) from a file
sys = pyPRISM.System(['A','B'],kT=1.0)
sys.domain = pyPRISM.Domain(dr=0.1,length=1024)
```

(continues on next page)

(continued from previous page)

```
fileName = './test_example_filename.txt'
sys.omega['A','A']  = pyPRISM.omega.FromFile(fileName)
x = sys.domain.k
y = sys.omega['A','A'].calculate(x)

#plot using matplotlib
plt.plot(x,y)
plt.gca().set_xscale("log", nonposx='clip')
plt.gca().set_yscale("log", nonposy='clip')

plt.show()
```

__init__(*fileName*)
Constructor

Parameters **fileName** (*str*) – path to textfile containing values of omega as a function of wavenumber, k.

calculate(*k*)
Return value of $\hat{\omega}$ at supplied k

Parameters **k** (*np.ndarray*) – array of wavenumber values to calculate ω at

pyPRISM.omega.Gaussian module

class pyPRISM.omega.Gaussian.**Gaussian**(*sigma*, *length*)
Bases: *pyPRISM.omega.Omega.Omega*

Gaussian intra-molecular correlation function

Mathematical Definition

$$\hat{\omega}(k) = \frac{1 - E^2 - \frac{2E}{N} + \frac{2E^{N+1}}{N}}{(1-E)^2}$$

$$E = \exp(-k^2\sigma^2/6)$$

Variable Definitions

- $\hat{\omega}(k)$ *intra*-molecular correlation function at wavenumber k
- N number of monomers/sites in gaussian chain
- σ contact distance between sites (i.e. site diameter)

Description

The Gaussian chain is an ideal polymer chain model that assumes a random walk between successive monomer segments along the chain with no intra-molecular excluded volume.

References

1. Schweizer, K.S.; Curro, J.G.; Integral-Equation Theory of Polymer Melts - Intramolecular Structure, Local Order, and the Correlation Hole, Macromolecules, 1988, 21 (10), pp 3070 [link]
2. Rubinstein, M; Colby, R.H; Polymer Physics. 2003. Oxford University Press.

Example

```
import pyPRISM
import numpy as np
import matplotlib.pyplot as plt

#calculate Fourier space domain and omega values
domain = pyPRISM.domain(dr=0.1,length=1000)
omega  = pyPRISM.omega.Gaussian(sigma=1.0,length=100)
x = domain.k
y = omega.calculate(x)

#plot using matplotlib
plt.plot(x,y)
plt.gca().set_xscale("log", nonposx='clip')
plt.gca().set_yscale("log", nonposy='clip')

plt.show()

#Define a PRISM system and set omega(k) for type A
sys = pyPRISM.System(['A','B'],kT=1.0)
sys.domain = pyPRISM.Domain(dr=0.1,length=1024)
sys.omega['A','A']  = pyPRISM.omega.Gaussian(sigma=1.0,length=100)
```

`__init__`(*sigma*, *length*)
Constructor

Parameters

- **sigma** (*float*) – contact distance between sites (site diameter)
- **length** (*float*) – number of monomers/sites in gaussian chain

`calculate`(k)
Return value of $\hat{\omega}$ at supplied k

Parameters **k** (*np.ndarray*) – array of wavenumber values to calculate ω at

pyPRISM.omega.GaussianRing module

class pyPRISM.omega.GaussianRing.**GaussianRing**(*sigma*, *length*)
Bases: *pyPRISM.omega.Omega.Omega*

Gaussian ring polymer intra-molecular correlation function

Mathematical Definition

$$\hat{\omega}(k) = 1 + 2N^{-1}\sum_{t=1}^{N-1}(N-t)\exp(\frac{-k^2\sigma^2 t(N-t)}{6N})$$

Variable Definitions

- $\hat{\omega}(k)$ *intra*-molecular correlation function at wavenumber k
- N number of monomers/sites in gaussian ring
- σ contact distance between sites (i.e. site diameter)

Description

The Gaussian ring is an ideal model for a cyclic chain that assumes a random walk between successive monomer segments along the chain, constrained such that ends join together to form a ring with no intra-molecular excluded volume.

References

Schweizer, K.S.; Curro, J.G.; Integral-Equation Theory of Polymer Melts - Intramolecular Structure, Local Order, and the Correlation Hole, Macromolecules, 1988, 21 (10), pp 3070, doi:10.1021/ma00188a027

Example

```
import pyPRISM
import numpy as np
import matplotlib.pyplot as plt

#calculate Fourier space domain and omega values
domain = pyPRISM.domain(dr=0.1,length=1000)
omega  = pyPRISM.omega.GaussianRing(sigma=1.0,length=100)
x = domain.k
y = omega.calculate(x)

#plot it!
plt.plot(x,y)
plt.gca().set_xscale("log", nonposx='clip')
plt.gca().set_yscale("log", nonposy='clip')

plt.show()

#Define a PRISM system and set omega(k) for type A
sys = pyPRISM.System(['A','B'],kT=1.0)
sys.domain = pyPRISM.Domain(dr=0.1,length=1024)
sys.omega['A','A']  = pyPRISM.omega.GaussianRing(sigma=1.0,length=100)
```

__init__(*sigma*, *length*)
Constructor

Parameters

- **sigma** (*float*) – contact distance between sites (site diameter)
- **length** (*float*) – number of monomers/sites in gaussian ring

calculate(*k*)
Return value of $\hat{\omega}$ at supplied k

Parameters k (*np.ndarray*) – array of wavenumber values to calculate ω at

pyPRISM.omega.NoIntra module

class pyPRISM.omega.NoIntra.**NoIntra**
Bases: *pyPRISM.omega.Omega.Omega*

Inter-molecule intra-molecular correlation function

Description This is a convenience class for specifying the intra-molecular correlations between sites which are never in the same molecule. Because they have no *intra*-molecular correlation, this function returns zero at all wavenumber.

Example

```python
import pyPRISM
import numpy as np
import matplotlib.pyplot as plt

#set omega(k) for types A,B to have no intra-molecular
#correlations (sites A and B are never on the same molecule)
sys = pyPRISM.System(['A','B'],kT=1.0)
sys.domain = pyPRISM.Domain(dr=0.1,length=1024)
sys.omega['A','B']  = pyPRISM.omega.NoIntra()
x = sys.domain.k
y = sys.omega['A','B'].calculate(x)

#plot using matplotlib
plt.plot(x,y)
plt.gca().set_xscale("log", nonposx='clip')
plt.gca().set_yscale("log", nonposy='clip')

plt.show()
```

calculate(*k*)

Return value of $\hat{\omega}$ at supplied k

Parameters **k** (*np.ndarray*) – array of wavenumber values to calculate ω at

pyPRISM.omega.NonOverlappingFreelyJointedChain module

class pyPRISM.omega.NonOverlappingFreelyJointedChain.**NFJC**(*length*, *l*)

Bases: *pyPRISM.omega.NonOverlappingFreelyJointedChain.NonOverlappingFreelyJointedChain*

Alias of NonOverlappingFreelyJointedChain

class pyPRISM.omega.NonOverlappingFreelyJointedChain.**NonOverlappingFreelyJointedChain**(*length*
l)

Bases: *pyPRISM.omega.Omega.Omega*

Freely jointed chain with excluded volume intra-molecular correlation function

Warning: The numerical integrations required for the NFJC omega calculation are slow and scale poorly with chain length so this omega may take minutes or longer to calculate even for modest chain lengths e.g. N=200.

Mathematical Definition

$$\hat{\omega}(k) = \hat{\omega}_{id}(k) + \frac{2}{N}\sum_{\tau=2}^{N-1}(N-\tau)[\hat{\omega}_{\tau}(k) - (\sin(k)/k)^{\tau}]$$

$$\tau = |\alpha - \beta|$$

Variable Definitions

- $\hat{\omega}(k)$ *intra*-molecular correlation function at wavenumber k
- $\hat{\omega}_{id}(k)$ *intra*-molecular correlation function for the ideal freely-jointed chain at wavenumber k. Please see equation (15) of Reference [1] for the mathematical representation.
- $\hat{\omega}_{\tau}(k)$ Please see equations (17,18,21) of Reference [1] for the mathematical representation.
- N number of repeat units in chain
- τ number of monomers along chain separating sites α and β.

Description

The non-overlapping freely-jointed chain is an adjustment to the ideal freely jointed chain model that includes the effects of the excluded volume of monomer segments (i.e. bonds are not free to rotate over all angles). This model assumes a constant bond length l.

References

1. Schweizer, K.S.; Curro, J.G.; Integral-Equation Theory of Polymer Melts - Intramolecular Structure, Local Order, and the Correlation Hole, Macromolecules, 1988, 21 (10), pp 3070 [link]

Example

```
import pyPRISM
import numpy as np
import matplotlib.pyplot as plt

#calculate Fourier space domain and omega values
domain = pyPRISM.domain(dr=0.1,length=1000)
omega  = pyPRISM.omega.NonOverlappingFreelyJointedChain(length=100,l=1.0)
x = domain.k
y = omega.calculate(x)

#plot using matplotlib
plt.plot(x,y)
plt.gca().set_xscale("log", nonposx='clip')
plt.gca().set_yscale("log", nonposy='clip')

plt.show()

#define a PRISM system and set omega(k) for type A
sys = pyPRISM.System(['A','B'],kT=1.0)
sys.domain = pyPRISM.Domain(dr=0.1,length=1024)
sys.omega['A','A']  = pyPRISM.omega.NonOverlappingFreelyJointedChain(length=100,
↪l=1.0)
```

`__init__`(*length*, *l*)

Constructor

Parameters

- **`length`** (*`float`*) – number of monomers/sites in non-overlapping freely-jointed chain
- **`l`** (*`float`*) – bond length

`calculate`(*k*)

Return value of $\hat{\omega}$ at supplied k

Parameters **k** (*np.ndarray*) – array of wavenumber values to calculate ω at

pyPRISM.omega.Omega module

class pyPRISM.omega.Omega.**Omega**

Bases: `object`

Baseclass for all *intra*-molecular correlation functions

Note: Currently, this class doesn't do anything besides group all of the *intra*-molecular correlation functions under a single inheritance heirarchy. This will change as needs arise.

pyPRISM.omega.SingleSite module

class pyPRISM.omega.SingleSite.**SingleSite**

Bases: *pyPRISM.omega.Omega.Omega*

Single-site intra-molecular correlation function

This class is useful for dealing with single bead molecules such as solvents or large spherical particles, it sets the value of the intra-molecular correlation function to 1 for all wavenumbers.

Example

```
import pyPRISM
import numpy as np

#set omega(k) for type A to 1 (single spherical site per molecule)
sys = pyPRISM.System(['A','B'],kT=1.0)
sys.domain = pyPRISM.Domain(dr=0.1,length=1024)
sys.omega['A','A']  = pyPRISM.omega.SingleSite()
x = sys.domain.k
y = sys.omega['A','A'].calculate(x)

#plot using matplotlib
plt.plot(x,y)
plt.gca().set_xscale("log", nonposx='clip')
plt.gca().set_yscale("log", nonposy='clip')

plt.show()
```

calculate(*k*)

Return value of $\hat{\omega}$ at supplied k

Parameters **k** (*np.ndarray*) – array of wavenumber values to calculate ω at

4.1.5 pyPRISM.potential package

PRISM uses pairwise decomposed potentials to describe the interactions between site-types. pyPRISM provides the potentials listed below. If you have potential that is not listed, please consider filing an Issue on GitHub, implementing the potential, and sharing with the community. See *Contributing* for more details.

pyPRISM.potential.Exponential module

class pyPRISM.potential.Exponential.**Exponential**(*epsilon*, *alpha*, *sigma=None*, *high_value=1000000.0*)

Bases: `pyPRISM.potential.Potential.Potential`

Exponential attractive interactions

Mathematical Definition

$$U_{\alpha,\beta}(r \geq \sigma_{\alpha,\beta}) = \epsilon_{\alpha,\beta} \exp\left(-\frac{r - \sigma_{\alpha,\beta}}{\alpha}\right)$$

$$U_{\alpha,\beta}(r < \sigma_{\alpha,\beta}) = C^{high}$$

Variable Definitions

α Width of exponential attraction

$\sigma_{\alpha,\beta}$ Contact distance of interactions between sites α and β.

$\epsilon_{\alpha,\beta}$ Interaction strength between sites α and β.

C^{high} High value used to approximate an infinite potential due to overlap

Description

This potential models an exponential-like attraction between sites with a specified site size and contact distance. For example, in Reference [1] this potential is used to model the attraction between a nanoparticle and monomers of a polymer chain.

References

1. Hooper, J.B. and K.S. Schweizer, Theory of phase separation in polymer nanocomposites. Macromolecules, 2006. 39(15): p. 5133-5142. [link]

Example

```python
import pyPRISM

#Define a PRISM system and set the A-B interaction potential
sys = pyPRISM.System(['A','B'],kT=1.0)
sys.domain = pyPRISM.Domain(dr=0.1,length=1024)
sys.potential['A','B'] = pyPRISM.potential.Exponential(epsilon=1.0,sigma=8.0,
↪alpha=0.5,high_value=10**6)
```

Warning: If sigma is specified such that it does not fall on the solution grid of the `Domain` object specified in `System`, then the sigma will effectively be rounded. A warning should be emitted during the construction of a `PRISM` object if this occurs.

__init__(*epsilon*, *alpha*, *sigma=None*, *high_value=1000000.0*)

Constructor

Parameters

- **epsilon** (*float*) – Strength of attraction

- **alpha** (*float*) – Range of attraction
- **sigma** (float, *optional*) – Contact distance. If not specified, sigma will be calculated from the diameters specified in the *System* object.
- **high_value** (float, *optional*) – High value used to approximate an infinite potential due to overlap

calculate(*r*)

Calculate the value of the potential

r

float np.ndarray – Array of pair distances at which to calculate potential values

pyPRISM.potential.HardCoreLennardJones module

class pyPRISM.potential.HardCoreLennardJones.**HardCoreLennardJones**(*epsilon*, *sigma=None*, *high_value=1000000.0*)

Bases: *pyPRISM.potential.Potential.Potential*

12-6 Lennard-Jones potential with Hard Core

Warning: This potential uses a slightly different form than what is implemented for the classic LJ potential. This means that the epsilon in the LJ and HCLJ potentials will not correspond to the same interaction strengths.

Mathematical Definition

$$U_{\alpha,\beta}(r > \sigma_{\alpha,\beta}) = \epsilon_{\alpha,\beta}\left[\left(\frac{\sigma_{\alpha,\beta}}{r}\right)^{12} - 2\left(\frac{\sigma_{\alpha,\beta}}{r}\right)^{6}\right]$$

$$U_{\alpha,\beta}(r \leq \sigma_{\alpha,\beta}) = C^{high}$$

Variable Definitions

$\epsilon_{\alpha,\beta}$ Strength of interaction (attraction or repulsion) between sites α and β.

$\sigma_{\alpha,\beta}$ Length scale of interaction between sites α and β.

r Distance between sites.

C^{high} High value used to approximate an infinite potential due to overlap

Description

Unlike the classic LJ potential, the HCLJ potential has an infinitely hard core and can handle negative and positive epsilons, corresponding to attractive and repulsive interactions.

References

1. Yethiraj, A. and K.S. Schweizer, INTEGRAL-EQUATION THEORY OF POLYMER BLENDS - NUMERICAL INVESTIGATION OF MOLECULAR CLOSURE APPROXIMATIONS. Journal of Chemical Physics, 1993. 98(11): p. 9080-9093. [link]

Example

```
import pyPRISM

#Define a PRISM system and set the A-B interaction potential
sys = pyPRISM.System(['A','B'],kT=1.0)
sys.domain = pyPRISM.Domain(dr=0.1,length=1024)
sys.potential['A','B'] = pyPRISM.potential.HardCoreLennardJones(epsilon=1.0,
↪sigma=1.0,high_value=10**6)
```

Warning: If sigma is specified such that it does not fall on the solution grid of the `Domain` object specified in `System`, then the sigma will effectively be rounded. A warning should be emitted during the construction of a `PRISM` object if this occurs.

`__init__`(*epsilon*, *sigma=None*, *high_value=1000000.0*)
Constructor

Parameters

- **`epsilon`** (`float`) – Depth of attractive well
- **`sigma`** (float, *optional*) – Contact distance. If not specified, sigma will be calculated from the diameters specified in the `System` object.
- **`high_value`** (float, *optional*) – High value used to approximate an infinite potential due to overlap

`calculate`(*r*)
Calculate value of potential

`r`
float np.ndarray – Array of pair distances at which to calculate potential values

pyPRISM.potential.HardSphere module

`class` `pyPRISM.potential.HardSphere.`**`HardSphere`**(*sigma=None*, *high_value=1000000.0*)
Bases: `pyPRISM.potential.Potential.Potential`

Simple hard sphere potential

Mathematical Definition

$$U_{\alpha,\beta}(r > \sigma_{\alpha,\beta}) = 0.0$$

$$U_{\alpha,\beta}(r \leq \sigma_{\alpha,\beta}) = C^{high}$$

Variable Definitions

$\sigma_{\alpha,\beta}$ Length scale of interaction between sites α and β.

r Distance between sites.

C^{high} High value used to approximate an infinite potential due to overlap

Description

This potential models the simiple hard-sphere fluid, in which sites have hard-core repulsion and no interactions outside their contact distance.

Example

```
import pyPRISM

#Define a PRISM system and set the A-B interaction potential
sys = pyPRISM.System(['A','B'],kT=1.0)
sys.domain = pyPRISM.Domain(dr=0.1,length=1024)
sys.potential['A','B'] = pyPRISM.potential.HardSphere(sigma=1.0,high_value=10**6)
```

Warning: If sigma is specified such that it does not fall on the solution grid of the *Domain* object specified in *System*, then the sigma will effectively be rounded. A warning should be emitted during the construction of a *PRISM* object if this occurs.

__init__(*sigma=None*, *high_value=1000000.0*)

Constructor

Parameters

- **sigma** (float, *optional*) – Contact distance. If not specified, sigma will be calculated from the diameters specified in the *System* object.
- **high_value** (float, *optional*) – High value used to approximate an infinite potential due to overlap

calculate(*r*)

Calculate value of potential

r

float np.ndarray – Array of pair distances at which to calculate potential values

pyPRISM.potential.LennardJones module

class pyPRISM.potential.LennardJones.**LennardJones**(*epsilon*, *sigma=None*, *rcut=None*, *shift=False*)

Bases: *pyPRISM.potential.Potential.Potential*

12-6 Lennard-Jones potential

Mathematical Definition

$$U_{\alpha,\beta}(r) = 4\epsilon_{\alpha,\beta}\left[\left(\frac{\sigma_{\alpha,\beta}}{r}\right)^{12.0} - \left(\frac{\sigma_{\alpha,\beta}}{r}\right)^{6.0}\right]$$

$$U_{\alpha,\beta}^{shift}(r) = U_{\alpha,\beta}(r) - U_{\alpha,\beta}(r_{cut})$$

Variable Definitions

$\epsilon_{\alpha,\beta}$ Strength of attraction between sites α and β.

$\sigma_{\alpha,\beta}$ Length scale of interaction between sites α and β.

r Distance between sites.

r_{cut} Cutoff distance between sites.

Description

The classic 12-6 LJ potential. To facilitate direct comparison with molecular simulation, the simulation may be cut and shifted to zero at a specified cutoff distance by setting the rcut and shift parameters. The full (non-truncated) LJ potential is accessed using r_{cut} = *None* and $shift$ = *False*.

Example

```
import pyPRISM

#Define a PRISM system and set the A-B interaction potential
sys = pyPRISM.System(['A','B'],kT=1.0)
sys.domain = pyPRISM.Domain(dr=0.1,length=1024)
sys.potential['A','B'] = pyPRISM.potential.LennardJones(epsilon=1.0,sigma=1.0,
↪rcut=2.5,shift=True)
```

Warning: If sigma is specified such that it does not fall on the solution grid of the `Domain` object specified in `System`, then the sigma will effectively be rounded. A warning should be emitted during the construction of a `PRISM` object if this occurs.

`__init__`(*epsilon*, *sigma=None*, *rcut=None*, *shift=False*)
Constructor

Parameters

- **`epsilon`** (`float`) – Depth of attractive well
- **`sigma`** (float, *optional*) – Contact distance. If not specified, sigma will be calculated from the diameters specified in the `System` object.
- **`rcut`** (float, *optional*) – Cutoff distance for potential. Useful for comparing directly to results from simulations where cutoffs are necessary.
- **`shift`** (`bool, *optional*`) – If r_{cut} is specified, shift the potential by its value at the cutoff. If r_{cut} is not specified, this parameter is ignored.

`calculate`(*r*)
Calculate value of potential

r

float np.ndarray – Array of pair distances at which to calculate potential values

`calculate_attractive`(*r*)
Calculate the attractive tail of the Lennard Jones potential. Returns zero at $r < \sigma$

pyPRISM.potential.Potential module

class pyPRISM.potential.Potential.**Potential**
Bases: `object`

Baseclass for all intermolecular-pairwise potentials

..note:

```
Currently, this class doesn't do anything besides group all of the
potentials under a single inheritance heirarchy.
```

pyPRISM.potential.WeeksChandlerAndersen module

class pyPRISM.potential.WeeksChandlerAndersen.**WeeksChandlerAndersen**(*epsilon*, *sigma=None*)
Bases: `pyPRISM.potential.LennardJones.LennardJones`

Purely repulsive Weeks-Chandler-Andersen potential

Mathematical Definition

$$U_{\alpha,\beta}(r) = \begin{cases} 4\epsilon_{\alpha,\beta}\left[\left(\frac{\sigma_{\alpha,\beta}}{r}\right)^{12.0} - \left(\frac{\sigma_{\alpha,\beta}}{r}\right)^{6.0}\right] + \epsilon_{\alpha,\beta} & r < r_{cut} \\ 0.0 & r \geq r_{cut} \end{cases}$$

$$r_{cut} = 2^{1/6}\sigma_{\alpha,\beta}$$

Variable Definitions

$\epsilon_{\alpha,\beta}$ Strength of repulsion between sites α and β.

$\sigma_{\alpha,\beta}$ Length scale of interaction between sites α and β.

r Distance between sites.

r_{cut} Cutoff distance where the value of the potential goes to zero.

Description

The Weeks-Chandler-Andersen potential for purely repulsive interactions. This potential is equivalent to the Lennard-Jones potential cut and shifted at the minimum of the potential, which occurs at $r = 2^{1/6}\sigma$.

Example

```
import pyPRISM

#Define a PRISM system and set the A-B interaction potential
sys = pyPRISM.System(['A','B'],kT=1.0)
sys.domain = pyPRISM.Domain(dr=0.1,length=1024)
sys.potential['A','B'] = pyPRISM.potential.WeeksChandlerAndersen(epsilon=1.0,
↪sigma=1.0)
```

Warning: If sigma is specified such that it does not fall on the solution grid of the `Domain` object specified in `System`, then the sigma will effectively be rounded. A warning should be emitted during the construction of a `PRISM` object if this occurs.

`__init__`(*epsilon*, *sigma=None*)
Constructor

Parameters

- **`epsilon`** (`float`) – Repulsive strength modifier
- **`sigma`** (float, *optional*) – Contact distance. If not specified, sigma will be calculated from the diameters specified in the `System` object.

`calculate`(*r*)
Calculate value of potential

`r`

float np.ndarray – Array of pair distances at which to calculate potential values

4.1.6 pyPRISM.trajectory package

PRISM is often used in conjunction with molecular simulation techniques, such as when using Self-Consistent PRISM. This module is intended to provide classes for working with and analyzing molecular simulation trajectories.

See *Self-Consistent PRISM Method* for more information on the method.

pyPRISM.trajectory.Debyer module

class `pyPRISM.trajectory.Debyer.`**`Debyer`**(*domain*, *nthreads=None*)

Bases: `object`

Parallelized Debye method to calculate $\hat{\omega}_{\alpha,\beta}(k)$

Warning: This pyPRISM functionality is still under testing. Use with caution.

Mathematical Definition

$$\hat{\omega}_{\alpha,\beta}(k) = \delta_{\alpha,\beta} + \frac{C_{\alpha,\beta}}{N_{frame}} \sum_{f} \sum_{i,j} \frac{\sin(kr_{ij}^{f})}{kr_{ij}^{f}}$$

Variable Definitions

- $\hat{\omega}_{\alpha,\beta}(k)$ Intra-molecular correlation function in Fourier-space
- $\delta_{\alpha,\beta}$ Kronecker delta for when considering a self ($\alpha == \beta$) versus not-self ($\alpha / = \beta$) site type pair.
- N_{frame} Number of frames in simulation trajectory
- $C_{\alpha,\beta}$ Scaling coefficient. If $\alpha == \beta$, then $C_{\alpha,\beta} = 1/N_{\alpha}$ else $C_{\alpha,\beta} = 1/(N_{\alpha} + N_{\beta})$
- $r_{i,j}^{f}$ At frame f of simulation, the scalar distance between sites with index i and j.
- k Wavenumber of calculation.

Description

One of the most powerful uses of PRISM is to combine it with modern simulation techniques to calculate the intra-molecular correlation functions $\hat{\omega}(k)$. This allows PRISM to be used for systems which do not have analytical descriptions for their $\hat{\omega}(k)$ and, furthermore, allows PRISM to predic the structure of non-ideal systems.

Unfortunately, this calculation is extremely computationally intensive and requires care to calculate correctly. Here we provide a parallelized implementation of the Debye Method which can be used to calculate $\hat{\omega}(k)$ for small to medium sized simulations.

This method works by allowing the user to selectively provide two sets of coordinate trajectories. To calculate $\hat{\omega}(k)$ between site-types α and β, the user should pass one trajectory of all sites of type α and the other where all sites are of type β. To calculate a self $\hat{\omega}(k)$ where $\alpha == \beta$ (selfOmega=*True*), both trajectories should be the same. Note that selfOmega should be correctly set in either case.

The calculate method below takes six arguments: positions1, positions2, molecules1, molecules2, box, selfOmega.

The positions1 and positions2 arguments are numpy array containing multiple frames of coordinates. Each of the two arrays should only contain the coordinates of the site-type pair being considered. In other words, positions1 should have a trajectory of positions for site type α and positions2 for β when calculating $\hat{\omega}_{\alpha,\beta}(k)$. If $\alpha == \beta$, the positions1 and positions2 should be the same array.

The molecules1 and molecules2 arguments specify to which molecule each site belongs. These arrays are necessary for calculations using trajectories that contain multiple molecules. For each site in the simulation of a site-type, these lists have an integer index which specify which molecule the site belongs to. All sites which share the same molecular index in this array belong to the same molecue. For single molecule simulations, an array of zeros or ones can be specified.

See the example below for an example of this classes use.

Note: Note that the num_chunks argument **does not** set the number of threads used in the calculation. This is governed by the OMP_NUM_THREADS (at least on *nix and OSX). If you have many cores on your machine, you may actually see improved performance if you reduce the value of OMP_NUM_THREADS from its maximum. Setting this variable is also useful if you're sharing a machine or node. If running a script, you can set this variables globally:

```
$ export OMP_NUM_THREADS=4
```

or locally for a single execution

```
$ OMP_NUM_THREADS=4 python pyPRISM_script.py
```

Note: As currently written, this class assumes periodicity in all dimensions.

Example

Below we consider an arbitrary system that has at least two types of sites in it.

```
import pyPRISM
import numpy as np

# load position, type, and molecule information using a users method of
# choice. For the array sizes, F = number of frames from simulation, N =
# total number of atoms/beads/sites
positions = ...  # 3D Array of size (F,N,3)
types = ...      # 1D Array of size (N)
molecules = ...  # 1D Array of size (N)
box = ...        # 2D Array of size (F,3)

# create pyPRISM domain
domain = pyPRISM.Domain(dr = 0.25, length = 1024)

# create Debyer object
debyer = pyPRISM.trajectory.Debyer(domain=domain,nthreads=4)

# calculate omega_1_1
mask1 = (types == 1)
positions1 =  positions[:,mask1,:]
molecules1 =  molecules[mask1]
mask2 = (types == 1)
positions2 =  positions[:,mask2,:]
molecules2 =  molecules[mask2]
selfOmega = True
omega_1_1  = debyer.calculate(positions1, positions2, molecules1, molecules2, box,
↪ selfOmega)
```

(continues on next page)

(continued from previous page)

```
# calculate omega_1_2
mask1 = (types == 1)
positions1 =  positions[:,mask1,:]
molecules1 =  molecules[mask1]
mask2 = (types == 2)
positions2 =  positions[:,mask2,:]
molecules2 =  molecules[mask2]
selfOmega = False
omega_1_2  = debyer.calculate(positions1, positions2, molecules1, molecules2, box,
↪ selfOmega)
```

calculate (*self*, *positions1*, *positions2*, *molecules1*, *molecules2*, *box*, *selfOmega*)

Calculate omega from two site type trajectories

> **Warning:** The first five arguments of this method (positions1, positions2, molecules1, molecules2, box) **must** be numpy arrays (not Python lists) for this method to work.

Parameters

- **positions1** (np.ndarray, float, size (F,N_α,3)) – 3-D numpy array containing coordinate trajectory from a simulation for site type α. See above for description.
- **positions2** (np.ndarray, float, size (F,N_β,3)) – 3-D numpy array containing coordinate trajectory from a simulation for site type β. See above for description.
- **molecules1** (np.ndarray, float, size (N_α)) – 1-D numpy arrays which specify the molecular identity of a site for sites of type α. See above for a description.
- **molecules2** (np.ndarray, float, size (N_β)) – 1-D numpy arrays which specify the molecular identity of a site for sites of type β. See above for a description.
- **box** (*np.ndarray, float, size(3)*) – 1-D array of box dimensions, l_x, l_y, l_z
- **selfOmega** (*bool*) – Set to *True* if $\alpha == \beta$ and False otherwise.

Returns **omega** – Intra-molecular correlation function

Return type np.ndarray

4.1.7 pyPRISM.util package

pyPRISM.util.UnitConverter module

class pyPRISM.util.UnitConverter.**UnitConverter** (*dc=1.0*, *dc_unit='nanometer'*, *mc=14.02*, *mc_unit='gram/mole'*, *ec=2.48*, *ec_unit='kilojoule/mole'*)

Bases: `object`

Unit conversion utility

Description

pyPRISM operates in a system of reduced units commonly called 'Lennard Jones units'. In this unit system, all measures are reported relative to characteristic values: length = d_c, mass = m_c, and energy = e_c. This class is designed to make some common conversions between reduced and real units easier.

See the Theory.General notebook of the *Tutorial* for a more detailed discussion of these units and how to work with the UnitConverter utility.

Note: The methods prefixed with “to” all expect floating point values in reduced units and convert to real units.

Warning: This class uses the Pint package. While the rest of pyPRISM will function without Pint, this class will not be available without this dependency installed in the current environment.

Example

```
import pyPRISM

domain = pyPRISM.Domain(length=4096,dr=0.25)

# create unit converter utility
uc = pyPRISM.util.UnitConverter(dc=1.5,dc_unit='nm')

# convert wavenumber from LJ units to real units
# using built-in conversion
real_k = uc.toInvAngstrom(domain.k)
real_k_magnitudes = real_k.magnitude

# convert radius in real units to reduced units
# manually using pint
real_radius = 123.5 # angstrom
reduced_radius = real_radius * uc('angstrom').to('dc').magnitude
```

`__call__`(*unit_string*)
Convenience method for accessing the pint UnitRegistry

`__init__`(*dc=1.0, dc_unit='nanometer', mc=14.02, mc_unit='gram/mole', ec=2.48, ec_unit='kilojoule/mole'*)
Constructor

Parameters

- **dc,dc_unit** (`float, str`) – Magnitude and unit of characteristic distance
- **mc,mc_unit** (`float, str`) – Magnitude and unit of characteristic mass
- **ec,ec_unit** (`float, str`) – Magnitude and unit of characteristic energy

d,dc
Pint Quantity – The defined characteristic distance as a defined Pint Quantity.

m,mc
Pint Quantity – The defined characteristic mass as a defined Pint Quantity.

e,ec
Pint Quantity – The defined characteristic energy as a defined Pint Quantity.

`toCelcius`(*temperature*)
Convert thermal energy to temperature units in Celcius

Parameters **temperature** (`float or np.ndarray of floats`) – Value of thermal energy (k_BT) to be converted

Returns **temperature** – temperature in Kelvin as a Pint Quantity. Use the magnitude attribute (temperature.magnitude) to obtain the numerical value of the temperature as a floating point value.

Return type Pint Quantity

`toConcentration`(*density*)

Convert reduced number density to real concentration units

Parameters **`density`** (*`float, or np.ndarray of floats`*) – Value(s) of density to be converted

Returns **concentration** – density in mol/L as a Pint Quantity. Use the magnitude attribute (concentration.magnitude) to obtain the numerical value of the contration as a floating point value.

Return type Pint Quantity

`toInvAngstrom`(*wavenumber*)

Convert wavenumbers to real units

Parameters **`wavenumber`** (*`float, or np.ndarray of floats`*) – Value(s) of wavenumbers to be converted

Returns **wavenumber** – wavenumbers in AA^{-1} as a Pint Quantity. Use the magnitude attribute (wavenumber.magnitude) to obtain the numerical value of the wavenumber as a floating point value.

Return type Pint Quantity

`toInvNanometer`(*wavenumber*)

Convert wavenumbers to real units

Parameters **`wavenumber`** (*`float, or np.ndarray of floats`*) – Value(s) of wavenumbers to be converted

Returns **wavenumber** – wavenumbers in nm^{-1} as a Pint Quantity. Use the magnitude attribute (wavenumber.magnitude) to obtain the numerical value of the wavenumber as a floating point value.

Return type Pint Quantity

`toKelvin`(*temperature*)

Convert thermal energy to temperature units in K

Parameters **`temperature`** (*`float or np.ndarray of floats`*) – Value of thermal energy (k_BT) to be converted

Returns **temperature** – temperature in K as a Pint Quantity. Use the magnitude attribute (temperature.magnitude) to obtain the numerical value of the temperature as a floating point value.

Return type Pint Quantity

`toVolumeFraction`(*density*, *diameter*)

Convert reduced number density to volume fraction

Parameters

- **`density`** (*`float, or np.ndarray of floats`*) – Value(s) of density to be converted
- **`diameter`** (*`float`*) – diameter of the site associated with the density

Returns vol_fraction – unitless volume as a Pint Quantity. Use the magnitude attribute (vol_fraction.magnitude) to obtain the numerical value of the volume fraction as a floating point value.

Return type Pint Quantity

4.2 Installation Instructions

4.2.1 Quick Command-Line Install

Install pyPRISM with all basic dependences via conda or pip from the command line. These commands should be platform agnostic and work for Linux, macOS, and Windows *if* you have Anaconda or pip installed.

```
$ conda install -c conda-forge pyPRISM
```

or

```
$ pip install pyPRISM
```

4.2.2 Manual Command-Line Install

If the quick-install commands do not work, then you can install pyPRISM "manually". After downloading the repository from GitHub, follow the steps below.

Note: Unless specified explicitly, the commands below should work for Linux, macOS, and Windows.

Note: For windows users, please ensure you are using the Anaconda command prompt. This can be found by opening the Start menu and searching for Anaconda.

Step 1: Dependencies via Anaconda

The easiest way to get an environment set up is by using the `env/py2.yml` or `env/py3.yml` we have provided for a python2 or python3 based environment. We recommend the python3 version. If you don't already have it, install conda. Note that all of the below instructions can be executed via the anaconda-navigator GUI. To start, we'll make sure you have the latest version of conda.

```
> conda deactivate

> conda update anaconda
```

Now create the `pyPRISM_py3` environment by executing the following. Note that these commands assume your terminal is located in the base directory of the pyPRISM repository (i.e., the directory with "setup.py"):

```
> conda env create -f env/py3.yml
```

When installation is complete you must activate the environment.

```
(Windows)   > activate pyPRISM_py3
```

```
(macOS/Linux) $ source activate pyPRISM_py3
```

Later, when you are ready to exit the environment, you can type:

```
(Windows)    > deactivate
```

```
(macOS/Linux) $ source deactivate
```

If for some reason you want to remove the environment entirely, you can do so by writing:

```
> conda env remove --name pyPRISM_py3
```

Note that an environment which satisfies the above dependencies must be **active** every time you wish to use pyPRISM via script or notebook. If you open a new terminal, you will have to reactivate the conda environment before running a script or starting jupyter notebook.

See *Dependencies* for more information.

Step 2: Install pyPRISM

After the depdendencies are satisfied and/or the conda environment is created **and activated**, pyPRISM can be installed to the system by running:

```
$ cd <pyPRISM base directory>

$ python setup.py install
```

4.2.3 Non-Install

There are use-cases where it makes sense to not permanently install pyPRISM onto a workstation or computing cluster. All methods below assume that you have already satisfied the dependencies described in *Dependencies*

Method 1: Command-Line Level

You can add pyPRISM to your current terminal environment so that all scripts and notebook servers run in this environment will be able to access pyPRISM.

```
(macOS/Linux) $ export PYTHONPATH=${PYTHONPATH}:/path/to/pyPRISM/dir

(Windows) > set PATH=%PATH%;C:\path\to\pyPRISM\dir
```

Note: The path in the above examples should be to the directory **containing** pyPRISM. The specified directory should be the one containing `setup.py` in the repository you downloaded or cloned from GitHub and not the one containing `__init__.py`.

Warning: This method is entirely non-permanent and must be repeated for each new terminal or Jupyter instance that is opened.

Method 2: Script or Notebook-Level

Alternatively, you can add pyPRISM to each script or notebook at runtime by placing the following code at the top of your script or notebook.

```
>>> import sys
>>> sys.insert(0,'/path/to/pyPRISM/directory/')
```

Note: The path in the above examples should be to the directory **containing** pyPRISM. The specified directory should be the one containing `setup.py` in the repository you downloaded or cloned from GitHub and not the one containing `__init__.py`.

Warning: This method is entirely non-permanent and must be repeated for each new script that is run.

4.2.4 Dependencies

The following are the tested dependencies needed to use pyPRISM:

- Python 2.7 or 3.5
- Numpy >= 1.8.0
- Scipy

These dependencies are required for *optional* features

- Cython (simulation trajectory analyses)
- Pint (unit conversion utility)

These additional dependencies are needed to run the tutorials

- Jupyter
- matplotlib
- Bokeh
- HoloViews

These additional dependencies are needed to compile the documentation from source

- Sphinx
- sphinx-autobuild
- sphinx_rtd_theme
- nbsphinx
- Pandoc <= 1.19.2

All of these dependecies can be satisfied by creating a conda environment using the .yml files in source distribution. Note that we provide multiple environments for different use-cases (e.g., Python 2 vs. Python 3, basic user vs. developer). The environments can be created using the following command from root directory of the repository. The root directory is the directory with the file *setup.py* in it.

```
$ conda env create -f env/py3.yml
```

Alternatively, all dependecies can be installed in your current Anaconda environment using

```
$ conda install -c conda-forge numpy scipy cython pint jupyter matplotlib bokeh␣
↪holoviews
```

Alternatively, all dependencies can be installed via pip

```
$ pip install numpy scipy cython pint jupyter matplotlib bokeh holoviews
```

Alternatively, each package can be downloaded and installed manually.

4.2.5 Usage

Before `pyPRISM` can be used two primary setup tasks must occur

- All dependencies must be satistifed. See *Dependencies*.
- pyPRISM must be placed on your `PYTHONPATH` either manually or via installation

Once these tasks are satisfied, `pyPRISM` can be imported and used in scripts or notebooks. The tutorial notebooks are in the *tutorial* directory in the codebase repository.

```
$ cd <pyPRISM tutorial directory>

$ jupyter notebook
```

This should spawn a jupyter notebook tab in your web browser of choice. If the tab doesn't spawn, check the terminal for a link that can be copied and pasted. See *Tutorial* for more information on the tutorial.

4.2.6 Verifying an Install

In order to verify your installation and to help ensure that bugs haven't been introduced, it is useful to run the test suite that is packaged in pyPRISM. If everything is installed correctly, the test suite should run and successfully complete all tests. Note that you must have all dependencies satisfied (e.g. via Ananconda) along with pyPRISM before running the test suite.

```
$ cd <pyPRISM base directory>/test

$ python -m pytest --verbose
```

4.2.7 Documentation

To build the documentation you'll need to satisfy the extra dependencies described in *Dependencies*. Once these are satisfied, you can build the documentation with

```
$ cd <pyPRISM base directory>/docs

$ make clean

$ make html
```

4.2.8 Troubleshooting

1. ModuleNotFoundError or ImportError

   This means that your current distribution of python cannot find the pyPRISM package. If you run the command below in a terminal, the pyPRISM package *must* be found in one of the listed directories.

   ```
   python -c "from __future__ import print_function; import sys;print(sys.
   ↪path)"
   ```

   If pyPRISM is not listed, there are several reasons why this might have occurred:

   - You are not using the same version of python that you installed pyPRISM to. This occurs often when using anaconda because there is often a "system" python and an "anaconda" python.
   - You have not activated the conda environment to which you installed pyPRISM

   See *Non-Install* for details on how to manually add pyPRISM into your environment (assuming that the other installation methods are failing).

2. Bash Terminal vs. Windows Terminal vs. Python Terminal vs. IPython Terminal

   There are strong differences between these terminals and what you can do with them. You can identify which environment you are in by looking at the terminal itself:

   ```
   (Bash)          $
   (Windows)       >
   (Python)        >>>
   (IPython)       In [1]:
   ```

   The `Bash` and `Windows` terminals should be used for installing python packages, managing environments, and running python scripts (e.g. `$ python run.py`). The `Python` and `IPython` terminals are for interactively running and working with Python code and each line of an example can be copied and run in these terminals. In general, the `IPython` terminal is a superior tool to the standard `Python` one and offers features such as syntax highlighting and code completion.

3. Other Internal Error

   Please file a bug report on GitHub. Please see *Contributing* for instructions on how to do this.

Warning: While the developers aim to provide compatibility across operating systems and use cases, they expect installation to be a primary barrier for many users. After reading through this documentation, please file an issue on GitHub if you run into problems.

4.3 Quickstart Guide

4.3.1 Setup

Requirements

- pyPRISM
- matplotlib

See *Quick Command-Line Install* or the full install instructions to get pyPRISM and set up your environment. Note that you'll need matplotlib for this example as well. After the environment is set up, the example below can be copied into a file (e.g. test.py) and run from the command line via

```
$ python test.py
```

Alternatively, the below code can be copied into your IDE (Spyder, PyCharm) or notebook provider (Jupyter) of choice.

4.3.2 Features Used

- *pyPRISM.core.System*
- *pyPRISM.core.Domain*
- *pyPRISM.core.Density*
- *pyPRISM.core.PRISM*
- *pyPRISM.omega.FreelyJointedChain*
- pyPRISM.omega.InterMolecular
- *pyPRISM.potential.HardSphere*
- *pyPRISM.potential.Exponential*
- *pyPRISM.closure.PercusYevick*
- *pyPRISM.closure.HyperNettedChain*
- *pyPRISM.calculate.pair_correlation*

4.3.3 Annotated Example

```
import pyPRISM
import matplotlib.pyplot as plt

# The system holds all information needed to set up a PRISM problem. We
# instantiate the system by specifying the site types and thermal energy
# level (kT, coarse-grained temperature) of the system.
sys = pyPRISM.System(['particle','polymer'],kT=1.0)

# We must discretize Real and Fourier space
sys.domain = pyPRISM.Domain(dr=0.01,length=4096)

# The composition of the system is desribed via number densities
sys.density['polymer']  = 0.75
sys.density['particle'] = 6e-6

# The diameter of each site is specified (in reduced units)
sys.diameter['polymer']  = 1.0
sys.diameter['particle'] = 5.0

# The molecular structure is described via intra-molecular correlation
# functions (i.e. omegas)
sys.omega['polymer','polymer']    = pyPRISM.omega.FreelyJointedChain(length=100,l=4.0/
↪3.0)
sys.omega['polymer','particle']   = pyPRISM.omega.NoIntra()
sys.omega['particle','particle'] = pyPRISM.omega.SingleSite()

# The site-site interactions are specified via classes which are lazily
```

(continues on next page)

(continued from previous page)

```
# evaluated during the PRISM-object creation
sys.potential['polymer','polymer']   = pyPRISM.potential.HardSphere(sigma=1.0)
sys.potential['polymer','particle']  = pyPRISM.potential.Exponential(sigma=3.0,
↪alpha=0.5,epsilon=1.0)
sys.potential['particle','particle'] = pyPRISM.potential.HardSphere(sigma=5.0)

# Closure approximations are also specified via classes
sys.closure['polymer','polymer']   = pyPRISM.closure.PercusYevick()
sys.closure['polymer','particle']  = pyPRISM.closure.PercusYevick()
sys.closure['particle','particle'] = pyPRISM.closure.HyperNettedChain()

# Calling the .solve() method of the system object attempts to numerically
# solv the PRISM equation and, if successful, it returns a PRISM object
# containing all of the solved correlation functions.
PRISM = sys.solve()

# Calculate the pair-correlation functions.
rdf = pyPRISM.calculate.pair_correlation(PRISM)

# Plot the results using matplotlib
plt.plot(sys.domain.r,rdf['polymer','polymer'],color='gold',lw=1.25,ls='-')
plt.plot(sys.domain.r,rdf['polymer','particle'],color='red',lw=1.25,ls='--')
plt.plot(sys.domain.r,rdf['particle','particle'],color='blue',lw=1.25,ls=':')
plt.ylabel('pair correlation')
plt.xlabel('separation distance')
plt.show()
```

4.3.4 Discussion

The above example sets up a PRISM object, runs a PRISM calculation, and plots the real-space pair correlation functions for a system of freely-jointed polymer chains of length $N = 100$ mixed with spherical hard nanoparticles of diameter $D = 5d$ (i.e., 5 times the monomer site diameter, d).

In addition to the heterogeneity in size scales, this example also demonstrates pyPRISM's ability to handle heterogeneous interaction potentials; in this system the hard sphere potential describes pairwise interactions for all species, excepting particle-polymer interactions which are modeled via an exponential attraction.

All necessary inputs are specified (site types and system temperature, domain size and discretization, site densities and diameters, intra-molecular correlation functions, interaction potentials, and closures used for each pair of site types) and then the PRISM calculation is performed. See Reference [1] for a full discussion of this system.

4.3.5 More Examples

A detailed tutorial with examples on how to build and run PRISM calculations for a variety of systems is shown in the *Tutorial*. The tutorial includes a general introduction to Python, PRISM, and the pyPRISM package. It also contains annotated example scripts that were used to create all of the case-studies in Reference [2].

4.3.6 References

1. Hooper, J.B.; Schweizer, K.S.; Contact Aggregation, Bridging, and Steric Stabilization in Dense Polymer Particle Mixtures, Macromolecules 2005, 38, 8858-8869 [link]

2. Martin, T.B.; Gartner, T.E. III; Jones, R.L.; Snyder, C.R.; Jayaraman, A.; pyPRISM: A Computational Tool for Liquid State Theory Calculations of Macromolecular Materials, Macromolecules, 2018, 51 (8), p2906-2922 [link]

4.4 Tutorial

A companion tutorial to the documentation can be found in the source of the pyPRISM package. This tutorial can be used interactively in live Jupyter notebooks or rendered as a static document. The benefit of using a live Jupyter notebook is that users are able to edit and run real pyPRISM code while the static website provides a rapid, and setup-free way to survey the codebase. On top of teaching users how to use pyPRISM, the tutorial covers the basics of Jupyter notebooks, Python, and PRISM theory.The tutorial also goes over several case studies from the literature and illustrates how pyPRISM can be used to reproduce results from these studies.

4.4.1 Interactive Tutorial

- **Jupyter Notebooks [link]**
  - The tutorial notebooks are packaged in the main codebase repository under the *tutorial* directory. See *Usage* for more details on how to use these notebooks.
- **Binder**
  - Try out the pyPRISM tutorial without installing!

Warning: Binder is a *free* service that we are taking advantage of to give users a zero-effort chance to try pyPRISM. Users should only run the tutorial examples and not custom notebooks. Please do not abuse this resource. Once a user has decided to use pyPRISM for research or teaching, please download and install pyPRISM locally as described in the *Installation Instructions*.

4.4.2 Non-Interactive Tutorial

Welcome to the pyPRISM Tutorial!

The Polymer Reference Interaction Site Model (PRISM) theory describes the **equilibrium** spatial-correlations of liquid-like polymer systems. The goal of this tutorial is to educate users on both the basics of PRISM theory and the pyPRISM tool.

What systems can be studied with PRISM?

- Polymer melts/blends
  - olefinic and non-olefinic polymers
  - linear/branched/dendritic/sidechain polymers
- Copolymer melts/blends
- Polymer solutions

- Nanoparticle solutions
- Polymer nanocomposites
- Liquid crystals (with anistropic formalism)
- Micelle Solutions
- Polyelectrolytes
  - rod-like polymers
  - flexible polymers
- Ionomers
- Ionic liquids

What thermodynamic and structural quantities can PRISM calculate?

- Second virial coefficients, B_2
- Flory effective interaction parameters, χ^{eff}
- Potentials of mean force
- Pair correlation functions (i.e. radial distribution functions)
- Partial structure factors
- Spinodal transition temperatures
- Equations of state
- Isothermal compressibilities

What are the benefits of using PRISM over other simulation or theory methods?

- is orders of magnitude faster
  - typically takes seconds to minutes to solve equations
- **does not** have finite size effects
- **does not** need to be equilibrated
- is *mostly* free of incompressibility assumptions

For what systems is PRISM theory not applicable for?

- macrophase-separated systems
- non-isotropic phases
- systems with strong nematic ordering (without anistropic formalism)
- calculating dynamic properties (e.g., diffusion coefficients, rheological properties)

Citations

If you use pyPRISM in your work, we ask that you please cite both of the following articles

- Martin, T.B.; Gartner, T.E III; Jones, R.L.; Snyder, C.R.; Jayaraman, A.; pyPRISM: A Computational Tool for Liquid State Theory Calculations of Macromolecular Materials, Macromolecules, 2018, 51 (8), p2906-2922 doi: https://dx.doi.org/10.1021/acs.macromol.8b00011
- Schweizer, K.S.; Curro, J.G.; INTEGRAL EQUATION THEORY OF THE STRUCTURE OF POLYMER MELTS, Physical Review Letters, 1987, 58 (3) p246-249 doi: http://dx.doi.org/10.1103/PhysRevLett.58.246

Setup

Detailed instructions for installation and usage of pyPRISM can be found in the documentation on ReadTheDocs.io.

Tutorial Tracks

In order to cater to users at multiple levels and capabilities, we have created different tracks for users to follow. The image below describes our recommendations for users with different backgrounds and intentions. Of course, these are only recommendations and users should feel free to study whichever documents they wish to. Note that some of the notebooks are still under construction and we intend to continue to grow and extend this tutorial.

NB0.Introduction · NB1.PythonBasics · NB2.Theory.General · NB3.Theory.PRISM · NB4.pyPRISM.Overview · NB5.CaseStudies.PolymerMelts · NB6.CaseStudies.Nanocomposites · NB7.CaseStudies.Copolymers · NB8.pyPRISM.Internals · NB9.pyPRISM.Advanced

Legal

Disclaimer

Any identification of commercial or open-source software in these notebooks is done so purely in order to specify the methodology adequately. Such identification is not intended to imply recommendation or endorsement by the National Institute of Standards and Technology, nor is it intended to imply that the softwares identified are necessarily the best available for the purpose.

License

Python Basics

To start, we recognize that many potential users are newcomers to Python and Jupyter so we begin with a simple notebook that focuses on how to use these tools. **Please skip this notebook if you are familiar with Python and Jupyter**. Note that there are many tutorials on Python and its various libraries so we only provide a most cursory look here. Refer to the links below for more detailed tutorials.

Python Tutorials

- http://swcarpentry.github.io/python-novice-inflammation/
- http://docs.python-guide.org/en/latest/intro/learning/
- https://docs.python.org/3/tutorial/

Numpy Tutorials

- https://www.dataquest.io/blog/numpy-tutorial-python/
- https://docs.scipy.org/doc/numpy-dev/user/quickstart.html

Jupyter Tutorials

- https://www.datacamp.com/community/tutorials/tutorial-jupyter-notebook
- https://jupyter-notebook-beginner-guide.readthedocs.io/en/latest/

Matplotlib Tutorials

- https://matplotlib.org/2.0.2/users/pyplot_tutorial.html

Holoviews Tutorial

- http://holoviews.org/Tutorials/

Concepts in this Notebook

- Jupyter Notebook + Python Basics
- What is an “object” or “class” in Python?
- Vectorized Calculations in Numpy
- Plotting

Notebook Setup

To begin, please run `Kernel-> Restart & Clear Output` from the menu at the top of the notebook. It is a good idea to run this before starting any notebook so that the notebook is fresh for the user.

Now, let’s do some basic `Python` sanity checking to make sure we can import neccessary libraries. Run the cell below. Click the cell to activate it and then use the `Run` button in the toolbar above. Alternatively, the cell can be run by pressing `<Shift-Enter>`. That is, hold `Shift` and then press `Enter` while holding `Shift`. The purpose of this cell is to load in external functions and libraries.

If successful, you should see a set of logos appear below the cell. Which logos appear depend on what is inside the `hv.extension()` command at the bottom of the cell. If no logos appear and the cell throws an error, there is likely something wrong with your environment.

Troubleshooting:

- Did you activate the correct conda environment before starting the jupyter notebook?
- If not using anaconda, did you install all dependencies before starting the jupyter notebook?
- Is pyPRISM installed in your current environment on your `PYTHONPATH`?

Holoviews + Bokeh Logos:

```
In [1]: import pyPRISM
        import numpy as np
        import matplotlib.pyplot as plt
        import holoviews as hv

        hv.extension('bokeh')
```

Data type cannot be displayed: application/javascript, application/vnd.bokehjs_load.v0+json

```
<IPython.core.display.HTML object>
```

All Variables are Global in Jupyter

Variables in `Jupyter` notebooks live globally “between” cells. The first time you run the cell below, it will throw an error because the ‘b’ variable is commented out. If you run the following cell, and then rerun the cell below, you’ll see the cell runs fine. This is an important concept to remember when using Jupyter notebooks because variables defined at the bottom of the notebook can affect the cells at the top.

Tip:

When all else fails, use the `Kernel -> Restart` functionality from the top-level menu. This will clear all variables from memory.

```
In [4]: a = 1
        print('Value of a =',a)

        #b = 2
        print('Value of b =',b)
```

```
Value of a = 1
Value of b = 3
```

```
In [3]: b = 3
        print('Value of b =',b)
```

```
Value of b = 3
```

What is an "object" or "class" in Python?

pyPRISM uses an object-oriented design to hold data and carry out the PRISM calculations. While a full description of what "object-oriented" means is far outside the scope of this tutorial, we'll try to introduce the most basic features of the methodology.

A class in `Python` describes a data structure for holding functions and variables. Below we describe a simple class for defining a shape.

```
In [5]: class Square:
            def __init__(self,length,width):
                self.length = length
                self.width = width
            def area(self):
                return self.length*self.width
```

But how do we use this class? We create an object from the class. The `__init__` function actually describes how the object is initially created. We can create many objects from the same class with different input parameters.

```
In [6]: square_object1 = Square(length=10,width=2.5)
        print('square_object1 = ',square_object1)

        square_object2 = Square(length=5,width=4)
        print('square_object2 = ',square_object2)
```

```
square_object1 =  <__main__.Square object at 0x7f4cf467b2e8>
square_object2 =  <__main__.Square object at 0x7f4cf4628c18>
```

We access the variables, which are called attributes in Python, in these objects using the 'dot' operator. These are defined via the `self` variable shown in the `__init__` function above.

```
In [7]: print('square_object1 (length by width) =',square_object1.length,'by',square_object1.width)
        print('square_object2 (length by width) =',square_object2.length,'by',square_object2.width)
```

```
square_object1 (length by width) = 10 by 2.5
square_object2 (length by width) = 5 by 4
```

Functions (called methods in Python) are also accessed via the 'dot' operation, but now we have to call the function using the paren.

```
In [8]: print('square_object1 area =',square_object1.area())
        print('square_object2 area =',square_object2.area())
```

```
square_object1 area = 25.0
square_object2 area = 20
```

Using Efficient Vectorized Operations in Numpy

The NumPy library is one of the most ubiquitous libraries used in `Python`-driven data science. `Numpy` provides a number of functions and data structures that allow us to generate and manipulate data in a simple and efficient interface.

To start we generate two `Numpy arrays` that we will plot later.

```
In [9]: x = np.arange(100)
        print('x =',x)

        print('')

        y = np.random.random(100)
        print('y =',y)
```

```
x = [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47
 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71
 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95
 96 97 98 99]

y = [0.96871466 0.13760296 0.92705958 0.24109572 0.50563029 0.80134237
 0.59722568 0.21639133 0.37703911 0.43609433 0.02655977 0.71988708
 0.70907182 0.12334527 0.73284856 0.95141623 0.71099059 0.03064434
 0.76687427 0.92077048 0.84735594 0.23879096 0.27032655 0.42357319
 0.11700788 0.80653507 0.46032536 0.05262332 0.232715   0.63107869
 0.75308625 0.99781775 0.47262713 0.0691762  0.9136122  0.8119291
 0.78867143 0.74982783 0.94198547 0.20783126 0.04089624 0.88651836
 0.64336257 0.84467139 0.59211992 0.5142557  0.17448646 0.76043449
 0.40729017 0.10215748 0.15373949 0.44002134 0.60214855 0.38047772
 0.29815424 0.80502636 0.12819524 0.97573272 0.7035504  0.21003149
 0.38919393 0.64316055 0.68222711 0.4787152  0.42001444 0.94344592
 0.02804441 0.50508219 0.47422093 0.65023172 0.30454903 0.20138134
 0.38643276 0.09272204 0.93451919 0.81901792 0.96167536 0.8331298
 0.25388386 0.52126561 0.01819315 0.73659048 0.93054841 0.8445565
 0.76504531 0.83054956 0.43768718 0.6461208  0.2125739  0.98487238
 0.02612495 0.77296086 0.25881863 0.68742048 0.94622851 0.51406883
 0.48448949 0.99641145 0.12647969 0.02502326]
```

Accessing the elements of an array in `Numpy` is achieved via the square braces x[i]. Note that `Python` array indices always start at 0. The last element in the array can be accessed via the i=-1 index.

```
In [10]: print('x[0] =',x[0])
         print('y[0] =',y[0])

         print('x[10] =',x[10])
         print('y[10] =',y[10])

         print('x[-1] =',x[-1])
         print('y[-1] =',y[-1])
```

```
x[0] = 0
y[0] = 0.9687146645976468
x[10] = 10
y[10] = 0.026559771669054655
```

```
x[-1] = 99
y[-1] = 0.025023261763978955
```

One of the primary features of `Numpy` is that it allows us to use vectorized operations. These are operations that are applied to the entire array 'at once' rather than having to manually apply to each member of the array. The two approaches are compared below in which we try to increment all of the values in the y array by one. We point out that the second approach is not possible in pure `Python`.

```
In [11]: # slow iteration
         new_y1 = np.copy(y)
         for i in range(100):
             new_y1[i] = y[i] + 1.0

         #fast vectorized operation
         new_y2 = y + 1.0

         print('y1 =',new_y1)
         print('y2 =',new_y2)
```

```
y1 = [1.96871466 1.13760296 1.92705958 1.24109572 1.50563029 1.80134237
 1.59722568 1.21639133 1.37703911 1.43609433 1.02655977 1.71988708
 1.70907182 1.12334527 1.73284856 1.95141623 1.71099059 1.03064434
 1.76687427 1.92077048 1.84735594 1.23879096 1.27032655 1.42357319
 1.11700788 1.80653507 1.46032536 1.05262332 1.232715   1.63107869
 1.75308625 1.99781775 1.47262713 1.0691762  1.9136122  1.8119291
 1.78867143 1.74982783 1.94198547 1.20783126 1.04089624 1.88651836
 1.64336257 1.84467139 1.59211992 1.5142557  1.17448646 1.76043449
 1.40729017 1.10215748 1.15373949 1.44002134 1.60214855 1.38047772
 1.29815424 1.80502636 1.12819524 1.97573272 1.7035504  1.21003149
 1.38919393 1.64316055 1.68222711 1.4787152  1.42001444 1.94344592
 1.02804441 1.50508219 1.47422093 1.65023172 1.30454903 1.20138134
 1.38643276 1.09272204 1.93451919 1.81901792 1.96167536 1.8331298
 1.25388386 1.52126561 1.01819315 1.73659048 1.93054841 1.8445565
 1.76504531 1.83054956 1.43768718 1.6461208  1.2125739  1.98487238
 1.02612495 1.77296086 1.25881863 1.68742048 1.94622851 1.51406883
 1.48448949 1.99641145 1.12647969 1.02502326]
y2 = [1.96871466 1.13760296 1.92705958 1.24109572 1.50563029 1.80134237
 1.59722568 1.21639133 1.37703911 1.43609433 1.02655977 1.71988708
 1.70907182 1.12334527 1.73284856 1.95141623 1.71099059 1.03064434
 1.76687427 1.92077048 1.84735594 1.23879096 1.27032655 1.42357319
 1.11700788 1.80653507 1.46032536 1.05262332 1.232715   1.63107869
 1.75308625 1.99781775 1.47262713 1.0691762  1.9136122  1.8119291
 1.78867143 1.74982783 1.94198547 1.20783126 1.04089624 1.88651836
 1.64336257 1.84467139 1.59211992 1.5142557  1.17448646 1.76043449
 1.40729017 1.10215748 1.15373949 1.44002134 1.60214855 1.38047772
 1.29815424 1.80502636 1.12819524 1.97573272 1.7035504  1.21003149
 1.38919393 1.64316055 1.68222711 1.4787152  1.42001444 1.94344592
 1.02804441 1.50508219 1.47422093 1.65023172 1.30454903 1.20138134
 1.38643276 1.09272204 1.93451919 1.81901792 1.96167536 1.8331298
 1.25388386 1.52126561 1.01819315 1.73659048 1.93054841 1.8445565
 1.76504531 1.83054956 1.43768718 1.6461208  1.2125739  1.98487238
 1.02612495 1.77296086 1.25881863 1.68742048 1.94622851 1.51406883
 1.48448949 1.99641145 1.12647969 1.02502326]
```

Plotting

One of the most common plotting libraries in `Python` is `Matplotlib`. Below is a simple demostration of how we can plot our x and y arrays that we created above. The syntax here should be familiar to those who have experience with `Matlab`. (Note that the plot below will not appear in the static version of this tutorial.)

```
In [12]: plt.plot(x,y,'r')
         plt.xlabel('x')
         plt.ylabel('y')
         plt.show()
```

tutorial/../_build/doctrees/nbsphinx/tutorial_NB1.PythonBasics_23_0.png

A powerful wrapper library that works with `Matplotlib` and other plotting back ends is `Holoviews`. When the `hv.extension` command at the top of the notebook is invoked with 'matplotlib', the plots produced by `Holoviews` will be static. When 'bokeh' is used, the backend will be the `Bokeh` library which will allow us to create live, interactive charts in the notebook. The downside to the `Bokeh` backend is that the plots do not show up when these notebooks are viewed on Github or when the notebooks are saved as html. We'll use the the `Matplotlib` backend by default so that the notebooks can be previewed on Github, but we encourage users to switch to the `Bokeh` backend if working with the notebooks directly.

```
In [13]: %opts Curve [width=600,height=400]
         hv.Curve((x,y))

Out[13]: :Curve   [x]   (y)
```

Summary

Hopefully this notebook serves to give a brief glance into how Python, Jupyter, and some associated libraries work. As was stated at the top of the notebook, this is not intended to be comprehensive, but rather just a broad introduction for the purposes of orienting those new to Python and/or Jupyter.

NB0.Introduction · NB1.PythonBasics · NB2.Theory.General · NB3.Theory.PRISM · NB4.pyPRISM.Overview · NB5.CaseStudies.PolymerMelts · NB6.CaseStudies.Nanocomposites · NB7.CaseStudies.Copolymers · NB8.pyPRISM.Internals · NB9.pyPRISM.Advanced

General Theory Concepts

Before we start digging into the details of PRISM and our implementation, it is worth highlighting a few general theoretical topics. For those with a simulation or theory background, this notebook can probably be skipped.

Concepts

- Reduced Units

Notebook Setup

To begin, please run `Kernel-> Restart & Clear Output` from the menu at the top of the notebook. It is a good idea to run this before starting any notebook so that the notebook is fresh for the user. Next, run the cell below (via the top menu-bar or `<Shift-Enter>`. If the cell throws an import error, there is likely something wrong with your environment.

Troubleshooting:

- Did you activate the correct conda environment before starting the jupyter notebook?
- If not using anaconda, did you install all dependencies before starting the jupyter notebook?
- Is pyPRISM installed in your current environment on your `PYTHONPATH`?

```
In [1]: import pyPRISM
        import numpy as np
```

Reduced Units

pyPRISM operates in a system of reduced units commonly called ‘Lennard Jones units’. In this unit system, all measures are reported relative to characteristic values: - characteristic length = d_c - characteristic mass = m_c - characteristic energy = e_c

While pyPRISM does not need to know the characteristic values to carry out calculations, users will need to give all input parameters in terms of these values. Note that the choice of characteristic values should not affect the results if all parameters are correctly scaled. The values are mostly arbitrary, but they should be chosen such that they correspond to values taken from the real system under study. For example, for mixture of nanoparticles with diameters - $d_1 = 2.0\,nm$ - $d_2 = 10.0\,nm$

a user might want to set - $d_c = 2\,nm$

so that the diameters can be reported to pyPRISM as - $d_1^* = 1\ d_c$ - $d_2^* = 5\ d_c$.

Note that the star ,$*$, indicates a reduced variable while the subscript c indicates a characteristic unit.

The characteristic energy is often chosen to be representative of an important energy scale in the system under study. For example, one could choose the size of thermal fluctuations (k_BT) at a given temperature. To convert from reduced temperature reported in units of thermal energy, T^*, one must use the relation
- $T^* = \frac{k_BT}{e_c}$

It can be tedious for those outside of the simulation and theory fields to work in these unit systems. pyPRISM has a `UnitConverter` class that is designed to allow for easy translation between common reduced and real unit systems. The `UnitConverter` class is built on top of the `Pint` library.

In the example below we choose a set of characteristic units, and create a unit converter object.

```
In [2]: uconv = pyPRISM.util.UnitConverter( dc=1.0,   dc_unit='nanometer',
                                            mc=14.02, mc_unit='gram/mole',
                                            ec=2.48,  ec_unit='kilojoule/mole')

        print(uconv)

<UnitConverter  dc:1e-09 meter | mc:0.01402 kilogram / mole | ec:2480.0 kilogram * meter ** 2 / mole
```

Next, we can use several built in methods to convert common reduced units to real units. Note that these functions return a `Quantity` object rather than a number. To obtain the number as a floating point value, use the magnitude attribute as shown below

```
In [3]: # convert thermal energy to temperature
        Tstar = 1.25
        T = uconv.toKelvin(Tstar)

        print("Reduced Temperature (ec)")
        print(Tstar)

        print("Temperature (ec)")
        print(T)
        print(T.magnitude)

        print()

        print('T is of type',type(T))
        print('T.magnitude is of type',type(T.magnitude))
```

```
Reduced Temperature (ec)
1.25
Temperature (ec)
372.8443218289683 kelvin
372.8443218289683

T is of type <class 'pint.quantity.build_quantity_class.<locals>.Quantity'>
T.magnitude is of type <class 'float'>
```

The `UnitConverter` class also has a built in conversion for reduced wavenumber in order to make comparison to experimental scattering measurements easier.

```
In [4]: #convert reduced wavenumbers to inverse angstroms or inverse nanometers
        kstar = np.logspace(start=-2,stop=0,num=10) #reduced wavenumber values
        k = uconv.toInvAngstrom(kstar)

        print("Reduced Wavenumber (dc^-1)")
        print(kstar)
        print()

        print("Wavenumber (A^-1)")
        print(k)
```

```
Reduced Wavenumber (dc^-1)
[0.01       0.01668101 0.02782559 0.04641589 0.07742637 0.12915497
 0.21544347 0.35938137 0.59948425 1.        ]

Wavenumber (A^-1)
[0.001      0.0016681  0.00278256 0.00464159 0.00774264 0.0129155 0.02154435 0.03593814 0.05994843 0
```

Alternatively, users can work directly with the `Pint` library. In the UnitConverter class, we have defined reduced unit `Quantities` that can be used for more “manual” unit conversion.

```
In [5]: Tstar = 1.25*uconv.ec # uconv.ec is the reduced "unit"
        print('Tstar =', Tstar)

        T = Tstar/uconv.pint.boltzmann_constant/uconv.pint.avogadro_number
        print('T =',T)
        print('T =',T.to('kelvin'))
```

```
Tstar = 1.25 echar
T = 1.25 echar / avogadro_number / boltzmann_constant
T = 372.8443218289683 kelvin
```

Summary

The goal of this notebook is to demystify some general theory topics for the reader. As we are continuously developing this tutorial, users should feel free to send us suggestions for future topics to go over in this or other notebooks.

NB0.Introduction · NB1.PythonBasics · NB2.Theory.General · NB3.Theory.PRISM · NB4.pyPRISM.Overview · NB5.CaseStudies.PolymerMelts · NB6.CaseStudies.Nanocomposites · NB7.CaseStudies.Copolymers · NB8.pyPRISM.Internals · NB9.pyPRISM.Advanced

PRISM Theory

In this introductory notebook, we will cover a number of concepts from both a theoretical and technical standpoint. The primary goal is to orient you so that you have a basic understanding of Python and the Jupyter environment and the basics of PRISM theory.

Concepts

- PRISM Equation
- Total Correlation Function
- Intra-molecular Correlation Function
- Direct Correlation Function

The PRISM Equation

PRISM theory describes the spatial correlations between spherical sites which represent either an atomic species or some collection of atoms in the molecule (e.g. a monomer or statistical segment of a polymer chain). By carrying out calculations with multiple site types, one can represent homopolymers with chemically complex monomers, polymer blends, copolymers, nanocomposites, and colloidal solutions. All information about the chemistry and connectively of these systems is encoded into the pair-interactions and intra-molecular correlation functions as will be discussed below.

In general, for a material system that can be represented with n types of sites, the PRISM equation is written in Fourier space as

$$\hat{H}(k) = \hat{\Omega}(k)\hat{C}(k)[\hat{\Omega}(k) + \hat{H}(k)]$$

with $\hat{H}(k)$, $\hat{\Omega}(k)$, and $\hat{C}(k)$ as $n \times n$ matrices of correlation values at wavenumber k. Note that the overhat notation and the functional designation (k) indicate that these variables are functions in Fourier-Space. The above equation greatly benefits from the Fourier Convolution Theorem as it is a complicated integral equation in Real-space.

The meaning of each of these variables is briefly discussed below for a three component system with site-types A, B, and C.

Total Correlation Function, $\hat{H}(k)$

$$\hat{H}(k) = \begin{bmatrix} \rho^{pair}_{AA}\hat{h}_{AA}(k) & \rho^{pair}_{AB}\hat{h}_{AB}(k) & \rho^{pair}_{AC}\hat{h}_{AC}(k) \\ \rho^{pair}_{BA}\hat{h}_{BA}(k) & \rho^{pair}_{BB}\hat{h}_{BB}(k) & \rho^{pair}_{BC}\hat{h}_{BC}(k) \\ \rho^{pair}_{CA}\hat{h}_{CA}(k) & \rho^{pair}_{CB}\hat{h}_{CB}(k) & \rho^{pair}_{CC}\hat{h}_{CC}(k) \end{bmatrix}$$

in which $\rho^{pair}_{\alpha\beta} = \rho_\alpha \rho_\beta$ and ρ_α, ρ_β correspond to the site number densities of site types α and β respectively. In Real-space, $h_{\alpha\beta}(r)$ corresponds to the pair correlation function (a.k.a radial distribution function) as

$$h_{\alpha\beta}(r) = g_{\alpha\beta}(r) - 1$$

in which r is the separation distance between sites of type α and β. Note that all spatial correlations matrices are, by definition, symmetric with site-type pairs, *i.e.* $h_{\alpha\beta}(r) = h_{\beta,\alpha}(r)$.

$\hat{H}(k)$ is found as a result of a numerical PRISM calculation.

Intra-molecular Correlation Function, $\hat{\Omega}(k)$

$$\hat{\Omega}(k) = \begin{bmatrix} \rho^{site}_{AA}\hat{\omega}_{AA}(k) & \rho^{site}_{AB}\hat{\omega}_{AB}(k) & \rho^{site}_{AC}\hat{\omega}_{AC}(k) \\ \rho^{site}_{BA}\hat{\omega}_{BA}(k) & \rho^{site}_{BB}\hat{\omega}_{BB}(k) & \rho^{site}_{BC}\hat{\omega}_{BC}(k) \\ \rho^{site}_{CA}\hat{\omega}_{CA}(k) & \rho^{site}_{CB}\hat{\omega}_{CB}(k) & \rho^{site}_{CC}\hat{\omega}_{CC}(k) \end{bmatrix}$$

in which $\rho^{site}_{\alpha\beta} = \rho_\alpha + \rho_\beta$ if $\alpha \neq \beta$ and $\rho^{site}_{\alpha\beta} = \rho_\alpha$ otherwise. In brief, $\hat{\Omega}(k)$ specifies the connectivity and structure of the molecules in a system. In contrast to $\hat{H}(k)$ which only specifies the *inter*-molecular correlations between molecules, $\hat{\Omega}(k)$ only describes the correlations within molecules. Each $\hat{\omega}(k)$ is analagous to a form-factor from PRISM theory.

$\hat{\Omega}(k)$ is an input to numerical PRISM calculations.

Direct Correlation Function, $\hat{C}(k)$

$$\hat{C}(k) = \begin{bmatrix} \hat{c}_{AA}(k) & \hat{c}_{AB}(k) & \hat{c}_{AC}(k) \\ \hat{c}_{BA}(k) & \hat{c}_{BB}(k) & \hat{c}_{BC}(k) \\ \hat{c}_{CA}(k) & \hat{c}_{CB}(k) & \hat{c}_{CC}(k) \end{bmatrix}$$

$\hat{C}(k)$ describes the *inter*-molecular correlations between sites when many-molecule effects (beyond pair) are removed.

$\hat{C}(k)$ is found as a result of a numerical PRISM calculation.

Summary

The goal of this notebook is not to comprehensively cover the details of PRISM theory, but rather to give the user the broad strokes of the formalism. Hopefully, the details in this notebook will clarify some of the calculations that users will carry out later.

NB0.Introduction · NB1.PythonBasics · NB2.Theory.General · NB3.Theory.PRISM · NB4.pyPRISM.Overview · NB5.CaseStudies.PolymerMelts · NB6.CaseStudies.Nanocomposites · NB7.CaseStudies.Copolymers · NB8.pyPRISM.Internals · NB9.pyPRISM.Advanced

pyPRISM Basics

In this notebook, we'll go though the various pieces of `pyPRISM` and how we go about setting up a "problem" in the `pyPRISM` environment. We start with a classic physical system: a hard-sphere fluid.

Concepts

- `pyPRISM` basics
- posing problems in `pyPRISM`

Notebook Setup

To begin, please run `Kernel-> Restart & Clear Output` from the menu at the top of the notebook. It is a good idea to run this before starting any notebook so that the notebook is fresh for the user. Next, run the cell below (via the top menu-bar or `<Shift-Enter>`. If the cell throws an import error, there is likely something wrong with your environment.

If successful, you should see a set of logos appear below the cell. Which logos appear depend on what is inside the `hv.extension()` command at the bottom of the cell. If no logos appear and the cell throws an error, there is likely something wrong with your environment.

Troubleshooting:

- Did you activate the correct conda environment before starting the jupyter notebook?
- If not using anaconda, did you install all dependencies before starting the jupyter notebook?
- Is pyPRISM installed in your current environment on your `PYTHONPATH`?

Holoviews + Bokeh Logos:

```
In [1]: import pyPRISM
        import numpy as np
        import matplotlib.pyplot as plt
        import holoviews as hv

        hv.extension('bokeh')
```

Data type cannot be displayed: application/javascript, application/vnd.bokehjs_load.v0+json

```
<IPython.core.display.HTML object>
```

Create System

To start, we define a "System" object which will contain all of the information that will specify the structure and interactions (chemistry) of a given system. In this case, our goal is to set up the calculation for a simple, hard-sphere monomer solution. We define the system to have 1 site-type called "monomer" and we define the reduced temperature to be $k_B T = 1.0$.

```
In [2]: sys = pyPRISM.System(['monomer'],kT=1.0)
```

Define Domain

Next, we need to define the numerical solution domain. In other words, we must define the Real- and Fourier- space grid that we will solve the equations on. We do this by specifying the grid spacing in Real-space (dr) and the number of points in the grid (length). Specifying these values fixes both the Real- and Fourier- space grid, and we could have alternatively specified the Fourier-space grid spacing (dk). Note that it is best for efficiency if the length is a power of 2.

```
In [3]: sys.domain = pyPRISM.Domain(dr=0.005,length=32768)
        print('r =',sys.domain.r)
        print('k = ',sys.domain.k)
```

```
r = [5.00000e-03 1.00000e-02 1.50000e-02 ... 1.63830e+02 1.63835e+02
 1.63840e+02]
k =  [1.91747598e-02 3.83495197e-02 5.75242795e-02 ... 6.28280181e+02
 6.28299356e+02 6.28318531e+02]
```

Define Interactions

Now we'll define the interactions of the monomer solution. This effectively represents the chemistry of the system by defining how the sites attract or repel one another. For this example, we'll be examining a simple, hard-sphere fluid.

```
In [4]: sys.diameter['monomer'] = 1.0
        sys.potential['monomer','monomer'] = pyPRISM.potential.HardSphere()
```

Define Composition

The composition of a system in PRISM theory is defined via site number densities. Since this is a one component system, we only need to specify one density.

```
In [5]: sys.density['monomer'] = 0.8
        sys.density
```

```
Out[5]: <Density total:0.80>
```

Define Molecular Structure

In general, the molecular structure is specified *via* $\hat{\omega}(k)$ in PRISM calculations. These functions are essentially form-factors which describe the *intra*-molecular correlations within a molecule. It is through these functions that the molecular structure of a system is described and arbitrarily complicated $\hat{\omega}(k)$ are possible. In this case of a simple hard-sphere fluid, $\hat{\omega}(k) = 1$ for all k by definition.

```
In [6]: sys.omega['monomer','monomer'] = pyPRISM.omega.SingleSite()
```

Define Closure Approximation

In order to solve the PRISM equations numerically, an extra closure-approximation must be provided for each pair of sites. A detailed explanation of various closures is slightly beyond the scope of this introductory tutorial. A few guidelines are provided below

- The PercusYevick (PY) closure is a good starting point for most systems with divergent (hard-core, Weeks-Chandler-Andersen, Lennard-Jones) interactions.
- The HypernettedChain (HNC) closure is more accurate than PY for pairs of sites that have very different diameters. This is very useful knowledge when studying composite systems.

```
In [7]: sys.closure['monomer','monomer'] = pyPRISM.closure.PercusYevick()
```

Create PRISM Object and Solve

Finally, we take the System object and use it to spawn a PRISM object. We have created the "System" and "PRISM" classes because they have separate, specific roles. The System class *defines* the structure and interactions of a given system while the PRISM class contains the machinery for setting up, solving, and storing the results of the mathematical PRISM formalism. After solve() is called, the result of the calculation is stored in the PRISM object.

```
In [8]: PRISM = sys.createPRISM()
        PRISM.solve()
```

```
0:  |F(x)| = 2.62378; step 1; tol 0.266006
1:  |F(x)| = 1.7493; step 1; tol 0.400055
2:  |F(x)| = 0.639398; step 1; tol 0.144039
3:  |F(x)| = 0.171479; step 1; tol 0.0647323
4:  |F(x)| = 0.00617309; step 1; tol 0.00116634
5:  |F(x)| = 3.63074e-06; step 1; tol 3.11335e-07
```

```
Out[8]: fun: array([ 4.15803093e-09,  1.25704004e-08,  2.08222783e-08, ...,
                2.85459257e-11, -2.23242033e-11,  4.31762904e-12])
         message: 'A solution was found at the specified tolerance.'
             nit: 7
          status: 1
         success: True
               x: array([ 7.31923234e-02,  2.18136517e-01,  3.61390574e-01, ...,
               -2.84237873e-11,  2.22021204e-11, -4.19551144e-12])
```

Plotting the Results

To start, we'll plot the pair correlation function of this system. We calculate this value by passing the **solved** PRISM object to a calculation function.

```
In [9]: %opts Curve Scatter Area [width=500,height=400]
        %opts Scatter (size=10,alpha=0.5)

        x = sys.domain.r
        y = pyPRISM.calculate.pair_correlation(PRISM)['monomer','monomer']
        rdf1 = hv.Curve((x,y),extents=(0,0,6,4),label='BaseRDF').redim.label(x='r',y='g(r)')
        rdf1
```

```
Out[9]: :Curve   [x]   (y)
```

We could instead calculate the structure factor of this system

```
In [10]: %opts Curve Scatter Area [width=500,height=400]
         %opts Scatter (size=10,alpha=0.5)

         x = sys.domain.k
         y = pyPRISM.calculate.structure_factor(PRISM)['monomer','monomer']
         hv.Curve((x,y),extents=(0,0,20,None)).redim.label(x='k',y='S(k)')
```

```
Out[10]: :Curve   [x]   (y)
```

Or maybe the second virial coefficient

```
In [11]: B2 = pyPRISM.calculate.second_virial(PRISM)['monomer','monomer']
         print('The monomer-monomer second-virial coefficient for this system is',B2)
```

```
The monomer-monomer second-virial coefficient for this system is 0.604288942117634
```

Putting It All Together

The above steps were long and detailed, but it's important to realize that we only executed a few lines of code, shown below. Try changing the density, closures, and interactions and see how these changes affect the pair-correlation function. A few suggestions are below:

- change the closure to
  - sys.closure['monomer','monomer'] = pyPRISM.closure.HypernettedChain()
  - sys.closure['monomer','monomer'] = pyPRISM.closure.MSA(apply_hard_core=True)
- change the site diameter
  - diameter = 0.75
- change the density
  - density = 0.6

```
In [12]: %opts Curve Scatter Area [width=500,height=400]
         %opts Scatter (size=10,alpha=0.5)

         sys = pyPRISM.System(['monomer'],kT=1.0)
         sys.domain = pyPRISM.Domain(dr=0.005,length=32768)
         sys.diameter['monomer'] = 1.0
         sys.potential['monomer','monomer'] = pyPRISM.potential.HardSphere()
         sys.density['monomer'] = 0.9
         sys.omega['monomer','monomer'] = pyPRISM.omega.SingleSite()
         sys.closure['monomer','monomer'] = pyPRISM.closure.HyperNettedChain(apply_hard_core=True)
         PRISM = sys.createPRISM()
         PRISM.solve()

         x = sys.domain.r
         y = pyPRISM.calculate.pair_correlation(PRISM)['monomer','monomer']
         rdf2 = hv.Curve((x,y),extents=(0,0,6,6),label='ModifiedRDF').redim.label(x='r',y='g(r)')

         hv.Overlay([rdf1,rdf2])
```

```
0:  |F(x)| = 3.97246; step 0.23001; tol 0.561713
1:  |F(x)| = 3.06534; step 0.382596; tol 0.535897
2:  |F(x)| = 2.67269; step 0.145932; tol 0.684197
3:  |F(x)| = 2.41102; step 0.121401; tol 0.732398
4:  |F(x)| = 2.13474; step 0.21196; tol 0.705555
5:  |F(x)| = 1.7113; step 0.409184; tol 0.578369
6:  |F(x)| = 1.41496; step 0.239333; tol 0.615285
7:  |F(x)| = 1.15999; step 0.232213; tol 0.604872
8:  |F(x)| = 0.902274; step 0.332558; tol 0.544518
9:  |F(x)| = 0.646089; step 0.496285; tol 0.461477
10:  |F(x)| = 0.371822; step 1; tol 0.298076
11:  |F(x)| = 0.042139; step 1; tol 0.0115596
12:  |F(x)| = 0.000206088; step 1; tol 2.15269e-05
13:  |F(x)| = 2.27462e-08; step 1; tol 1.09636e-08
```

```
Out[12]: :Overlay
            .Curve.BaseRDF     :Curve   [x]   (y)
            .Curve.ModifiedRDF :Curve   [x]   (y)
```

Getting Extra Help

Besides going to the documentation website (see Github page) or compiling the documentation yourself, Jupyter provides a few ways to access the documentation in the notebook. Try running the cell below.

```
In [13]: pyPRISM.potential.WeeksChandlerAndersen?
```

Alternatively, if you are inside the parenthesis of a function call, typing `<Shift-Tab>` will bring up a small,floating documentation popup. Try this below:

```
In [14]: pyPRISM.potential.WeeksChandlerAndersen(epsilon=1.0,sigma=1.0)
```

```
Out[14]: <Potential: WeeksChandlerAndersen>
```

Summary

In this notebook, we stepped through the process of constructing a PRISM calculation in pyPRISM for a standard system in liquid state theory: the hard sphere fluid. We went over each step of the pyPRISM calculation in detail and then calculated several thermodynamic and structural quantities from the converged equations. In the following notebooks, we will move on to covering several real word systems and attempt to reproduce published data from scientific literature.

NB0.Introduction · NB1.PythonBasics · NB2.Theory.General · NB3.Theory.PRISM · NB4.pyPRISM.Overview · NB5.CaseStudies.PolymerMelts · NB6.CaseStudies.Nanocomposites · NB7.CaseStudies.Copolymers · NB8.pyPRISM.Internals · NB9.pyPRISM.Advanced

Case Studies: Polymer Melts

In this example, we use `pyPRISM` to study and understand two simple but important systems: A dense, linear homopolymer melt of varying density and a ring homopolymer melt of varying molecular mass. In the linear melt system, we take advantage of the fact that PRISM theory doesn't have a simulation 'box' and study very long (N=16,000) gaussian polymer chains. Note that studing chains of this length using Molecular Dynamics or Monte Carlo techniques would be difficult if not impossible. In the ring melt system, we leverage PRISM's speed to quickly span a parameter space in molecular mass that would be challenging/time consuming to replicate synthetically.

Concepts

- 1 component PRISM
- Phase space "hopping"
- Gaussian chains

Tools

- pyPRISM.calculate.structure_factor
- pyPRISM.calculate.pair_correlation

References

1. Curro, J.G.; Schweizer K.S.; Integral Equation Theory of Homopolymer Melts, Molecular Crystals and Liquid Crystals Incorporating Nonlinear Optics, 1990, 180:1, 77-89, doi: 10.1080/00268949008025790
2. Curro, J.G.; Schweizer K.S.; Theory of Polymer Melts: An Integral Equation Approach, Macromolecules, 1987, 20, 1928-1934, doi: 10.1021/ma00174a040

Notebook Setup

To begin, please run `Kernel-> Restart & Clear Output` from the menu at the top of the notebook. It is a good idea to run this before starting any notebook so that the notebook is fresh for the user. Next, run the cell below (via the top menu-bar or `<Shift-Enter>`. If the cell throws an import error, there is likely something wrong with your environment.

If successful, you should see a set of logos appear below the cell. Which logos appear depend on what is inside the `hv.extension()` command at the bottom of the cell. If no logos appear and the cell throws an error, there is likely something wrong with your environment.

Troubleshooting:

- Did you activate the correct conda environment before starting the jupyter notebook?
- If not using anaconda, did you install all dependencies before starting the jupyter notebook?
- Is pyPRISM installed in your current environment on your `PYTHONPATH`?

Holoviews + Bokeh Logos:

```
In [1]: import pyPRISM
        import numpy as np
        import holoviews as hv
        hv.extension('bokeh')
```

```
Data type cannot be displayed: application/javascript, application/vnd.bokehjs_load.v0+json
```

```
<IPython.core.display.HTML object>
```

Linear Homopolymer Melt: Comparison to Hard Sphere Fluid

First, we solve the hard-sphere fluid (HSF) system from the previous notebook. Note that while variables are “global” within a Jupyter notebook, separate notebooks do not share variables so we have to re-do this calculation here.

```
In [2]: sys = pyPRISM.System(['A'],kT=1.0)
        sys.domain = pyPRISM.Domain(dr=0.005,length=32768)
        sys.diameter['A']       = 1.0
        sys.potential['A','A'] = pyPRISM.potential.HardSphere()
        sys.density['A']        = 0.8
        sys.omega['A','A']      = pyPRISM.omega.SingleSite()
        sys.closure['A','A']    = pyPRISM.closure.PY(apply_hard_core=True)
        PRISM_HSF = sys.createPRISM()
        PRISM_HSF.solve()

        x_HSF = sys.domain.r
        y_HSF = pyPRISM.calculate.pair_correlation(PRISM_HSF)['A','A']

        print('Done!')
```

```
0:  |F(x)| = 2.62378; step 1; tol 0.266006
1:  |F(x)| = 1.7493; step 1; tol 0.400055
2:  |F(x)| = 0.639398; step 1; tol 0.144039
3:  |F(x)| = 0.171479; step 1; tol 0.0647323
4:  |F(x)| = 0.00617309; step 1; tol 0.00116634
5:  |F(x)| = 3.63074e-06; step 1; tol 3.11335e-07
Done!
```

Then, we move onto the polmer melt case. Note that the only difference is the *intra*-molecular structure function ($\hat{\omega}(k)$).

```
In [3]: sys.omega['A','A'] = pyPRISM.omega.Gaussian(length=16000,sigma=sys.diameter.sigma['A','A'])
        PRISM_Melt = sys.createPRISM()
        PRISM_Melt.solve()

        x_Melt = sys.domain.r
        y_Melt = pyPRISM.calculate.pair_correlation(PRISM_Melt)['A','A']

        print('Done!')
```

```
0:  |F(x)| = 49.5243; step 0.339595; tol 0.387728
1:  |F(x)| = 1.78198; step 1; tol 0.135299
2:  |F(x)| = 0.179788; step 1; tol 0.0091614
3:  |F(x)| = 0.00469113; step 1; tol 0.000612737
4:  |F(x)| = 1.06398e-06; step 1; tol 4.62974e-08
Done!
```

```
In [4]: %%opts Overlay [width=500,height=400,legend_position='top_right']

        HSF = hv.Curve((x_HSF,y_HSF),label='Hard-Sphere Fluid',extents=(0,0,25,None))
        Melt = hv.Curve((x_Melt,y_Melt),label='Gaussian Melt',extents=(0,0,25,None))

        hv.Overlay([HSF,Melt])
```

```
Out[4]: :Overlay
           .Curve.Hard_hyphen_minus_Sphere_Fluid :Curve   [x]   (y)
           .Curve.Gaussian_Melt                  :Curve   [x]   (y)
```

Now, let's explore the effect of $\hat{\omega}(k)$ on the pair correlation function. Insert the following $\hat{\omega}(k)$ into the cell below.

Note that the length is purposefully changed for the last two :math:'omega'. - sys.omega['A','A'] = pyPRISM.omega.GaussianRing(length=16000,sigma=diameter) - sys.omega['A','A'] = pyPRISM.omega.FreelyJointedChain(length=16000,l=diameter) - sys.omega['A','A'] = pyPRISM.omega.NFJC(length=50,l=diameter) - sys.omega['A','A'] = pyPRISM.omega.DiscreteKoyama(sigma=1.0,l=diameter,length=50,lp=4/3)

```
In [5]: %%opts Overlay [width=500,height=400,legend_position='top_right']

        # Change setting below
        # ----------------------------------
        sys.omega['A','A'] = pyPRISM.omega.GaussianRing(length=16000,sigma=sys.diameter['A','A'])
        # ----------------------------------

        PRISM_Melt = sys.createPRISM()
        PRISM_Melt.solve()
        x_Melt2 = sys.domain.r
        y_Melt2 = pyPRISM.calculate.pair_correlation(PRISM_Melt)['A','A']

        Melt2 = hv.Curve((x_Melt2,y_Melt2),label='Melt',extents=(0,0,25,None))
        hv.Overlay([HSF,Melt,Melt2])
```

```
0:  |F(x)| = 74.9546; step 0.207941; tol 0.564017
1:  |F(x)| = 66.9276; step 0.108737; tol 0.717558
2:  |F(x)| = 66.6832; step 0.00446985; tol 0.893439
3:  |F(x)| = 66.5865; step 0.00180916; tol 0.89739
4:  |F(x)| = 65.8112; step 0.0146023; tol 0.879164
5:  |F(x)| = 60.6825; step 0.0998973; tol 0.765192
6:  |F(x)| = 60.5201; step 0.0361758; tol 0.895188
7:  |F(x)| = 60.3859; step 0.0353109; tol 0.896015
8:  |F(x)| = 60.2734; step 0.0340261; tol 0.896649
9:  |F(x)| = 60.2609; step 0.00431724; tol 0.899626
10:  |F(x)| = 60.2069; step 0.00769838; tol 0.898388
11:  |F(x)| = 60.1458; step 0.0222937; tol 0.898174
12:  |F(x)| = 60.1201; step 0.0100317; tol 0.899231
13:  |F(x)| = 60.0508; step 0.0274146; tol 0.897928
14:  |F(x)| = 59.9013; step 0.03615; tol 0.895523
15:  |F(x)| = 59.8455; step 0.0269553; tol 0.898326
16:  |F(x)| = 59.7906; step 0.0278476; tol 0.898347
17:  |F(x)| = 59.6937; step 0.0311638; tol 0.897087
18:  |F(x)| = 59.6934; step 0.000105054; tol 0.899989
19:  |F(x)| = 59.5362; step 0.0359007; tol 0.895266
20:  |F(x)| = 59.5225; step 0.00873502; tol 0.899586
21:  |F(x)| = 59.4992; step 0.0149687; tol 0.899296
22:  |F(x)| = 59.4764; step 0.0148359; tol 0.899311
23:  |F(x)| = 59.4029; step 0.0288751; tol 0.897776
24:  |F(x)| = 59.3802; step 0.0151269; tol 0.899313
25:  |F(x)| = 55.5415; step 1; tol 0.787398
26:  |F(x)| = 55.4249; step 1; tol 0.896224
27:  |F(x)| = 55.4169; step 1; tol 0.899741
28:  |F(x)| = 55.4167; step 1; tol 0.899995
29:  |F(x)| = 55.4167; step 1; tol 0.9
30:  |F(x)| = 55.4167; step 1; tol 0.9
31:  |F(x)| = 6.64034; step 1; tol 0.729
32:  |F(x)| = 6.11109; step 1; tol 0.762252
33:  |F(x)| = 3.97474; step 1; tol 0.522925
34:  |F(x)| = 1.54704; step 1; tol 0.246105
35:  |F(x)| = 0.0512046; step 1; tol 0.000985955
36:  |F(x)| = 0.000268186; step 1; tol 2.46887e-05
37:  |F(x)| = 1.11475e-08; step 1; tol 1.55498e-09
```

```
Out[5]: :Overlay
           .Curve.Hard_hyphen_minus_Sphere_Fluid :Curve   [x]   (y)
           .Curve.Gaussian_Melt                  :Curve   [x]   (y)
           .Curve.Melt                           :Curve   [x]   (y)
```

Linear Homopolymer Melt: Comparison to Literature

Now that we've solved a simple, homopolymer system, let's compare to data from the literature. This will also give us a chance to demonstrate some detailed scripting that we can do with typyPRISM.

First, we'll load in the data extracted from the literature [Ref. 1] that we'll be comparing against.

```
In [6]: gr_compare = []
```

```
sk_compare = []
for density in [0.6,0.8,1.0]:
    fname = '../data/LinearMelt-Gr-rho{}.csv'.format(density)
    x,y = np.loadtxt(fname,delimiter=',').T
    gr_compare.append([density,x,y])

    fname = '../data/LinearMelt-Sk-rho{}.csv'.format(density)
    x,y = np.loadtxt(fname,delimiter=',').T
    sk_compare.append([density,x,y])
```

Next, we define a bunch of variables to define how our plots will look later in this section.

```
In [7]: %opts Curve Scatter [width=500,height=400] Layout [shared_axes=False] Scatter (size=10,alpha=
        %opts Curve Scatter [fontsize={'xlabel':14,'legend':14,'ylabel':14,'ticks':14}]
        %opts Overlay [legend_position='bottom_left']
        %opts Layout [shared_axes=False]


        colors = {} # empty dictionary
        colors[1.0] = 'blue'
        colors[0.8] = 'red'
        colors[0.6] = 'green'

        ls = {} # empty dictionary
        ls[1.0] = 'solid'
        ls[0.8] = 'dashed'
        ls[0.6] = 'dotted'

        markers = {} # empty dictionary
        markers[1.0] = 'o'
        markers[0.8] = '^'
        markers[0.6] = 'd'
```

Next, we'll use `pyPRISM` to calculate data at three densities for a linear polymer melt of of gaussian chains of length 16000, as described in Reference 1. Notice how we store the resulting "guess" from the previous density and use it as the starting guess for the next density. After solving, note how many iterations each density took.

```
In [8]: sys = pyPRISM.System(['polymer'],kT=1.0)
        sys.domain = pyPRISM.Domain(dr=0.005,length=32768)
        sys.diameter['polymer'] = 1.0
        sys.closure['polymer','polymer'] = pyPRISM.closure.PercusYevick()
        sys.potential['polymer','polymer'] = pyPRISM.potential.HardSphere()
        sys.omega['polymer','polymer'] = pyPRISM.omega.Gaussian(sigma=sys.diameter['polymer','polyme

        gr_results = []
        sk_results = []
        guess = np.zeros_like(sys.domain.r)
        for density in [0.6,0.8,1.0]:
            print('==> Solving for density {}'.format(density))
            sys.density['polymer'] = density
            PRISM = sys.createPRISM()
            result = PRISM.solve(guess)
            guess = np.copy(PRISM.x)

            y = pyPRISM.calculate.structure_factor(PRISM)['polymer','polymer']
            x = sys.domain.k
            sk_results.append([density,x,y])

            x = sys.domain.r
            y = pyPRISM.calculate.pair_correlation(PRISM)['polymer','polymer']
            gr_results.append([density,x,y])
            print('')


        print('Done!')
```

```
==> Solving for density 0.6
0:  |F(x)| = 81.9231; step 0.186349; tol 0.593894
1:  |F(x)| = 81.8849; step 0.00049023; tol 0.89916
2:  |F(x)| = 81.6474; step 0.003047; tol 0.894787
3:  |F(x)| = 81.6472; step 2.06434e-06; tol 0.899996
4:  |F(x)| = 81.6443; step 3.76541e-05; tol 0.899936
5:  |F(x)| = 81.5191; step 0.00161753; tol 0.897241
6:  |F(x)| = 52.1202; step 0.15956; tol 0.724537
7:  |F(x)| = 46.0427; step 1; tol 0.702348
8:  |F(x)| = 42.036; step 1; tol 0.750178
9:  |F(x)| = 41.0872; step 1; tol 0.859828
10:  |F(x)| = 40.8693; step 1; tol 0.89048
11:  |F(x)| = 40.8594; step 1; tol 0.899568
12:  |F(x)| = 40.8594; step 1; tol 0.899996
13:  |F(x)| = 40.8593; step 1; tol 0.899999
14:  |F(x)| = 40.8593; step 1; tol 0.9
15:  |F(x)| = 40.8593; step 1; tol 0.9
16:  |F(x)| = 48.9979; step 1; tol 0.9999
17:  |F(x)| = 40.8604; step 1; tol 0.89982
18:  |F(x)| = 40.8585; step 0.0185991; tol 0.899916
19:  |F(x)| = 48.2255; step 1; tol 0.9999
20:  |F(x)| = 39.8334; step 1; tol 0.89982
21:  |F(x)| = 39.4845; step 0.494518; tol 0.884299
22:  |F(x)| = 34.0081; step 1; tol 0.703786
23:  |F(x)| = 46.7361; step 1; tol 0.9999
24:  |F(x)| = 41.5151; step 0.431742; tol 0.89982
25:  |F(x)| = 46.705; step 1; tol 0.9999
26:  |F(x)| = 41.7892; step 0.407615; tol 0.89982
27:  |F(x)| = 37.32; step 1; tol 0.728708
28:  |F(x)| = 45.2683; step 1; tol 0.9999
29:  |F(x)| = 41.8015; step 1; tol 0.89982
30:  |F(x)| = 35.7199; step 1; tol 0.728708
31:  |F(x)| = 34.5162; step 0.419989; tol 0.840364
32:  |F(x)| = 47.4471; step 1; tol 0.9999
33:  |F(x)| = 36.6828; step 1; tol 0.89982
34:  |F(x)| = 32.6796; step 1; tol 0.728708
35:  |F(x)| = 32.3298; step 0.0507434; tol 0.880832
36:  |F(x)| = 38.8325; step 1; tol 0.9999
37:  |F(x)| = 32.9971; step 1; tol 0.89982
38:  |F(x)| = 32.4921; step 0.476009; tol 0.872663
39:  |F(x)| = 39.0104; step 1; tol 0.9999
40:  |F(x)| = 33.1882; step 1; tol 0.89982
41:  |F(x)| = 33.0107; step 0.450081; tol 0.890398
42:  |F(x)| = 36.9105; step 1; tol 0.9999
43:  |F(x)| = 32.3099; step 1; tol 0.89982
44:  |F(x)| = 30.6935; step 0.0565517; tol 0.812199
45:  |F(x)| = 27.2482; step 0.282094; tol 0.709296
46:  |F(x)| = 27.1626; step 0.00388862; tol 0.89435
47:  |F(x)| = 24.9831; step 1; tol 0.761364
48:  |F(x)| = 23.4707; step 0.28068; tol 0.794334
49:  |F(x)| = 20.5345; step 1; tol 0.688905
50:  |F(x)| = 18.9605; step 0.17719; tol 0.767312
```

```
51:  |F(x)| = 14.6473; step 1; tol 0.537103
52:  |F(x)| = 14.3981; step 0.0218796; tol 0.869634
53:  |F(x)| = 11.786; step 1; tol 0.680637
54:  |F(x)| = 10.4035; step 0.160641; tol 0.701247
55:  |F(x)| = 8.94065; step 0.300383; tol 0.664692
56:  |F(x)| = 7.92109; step 1; tol 0.706438
57:  |F(x)| = 4.49779; step 1; tol 0.449149
58:  |F(x)| = 2.77687; step 1; tol 0.343047
59:  |F(x)| = 0.622838; step 1; tol 0.105913
60:  |F(x)| = 0.0143531; step 1; tol 0.000477952
61:  |F(x)| = 2.56211e-06; step 1; tol 2.86778e-08

==> Solving for density 0.8
0:  |F(x)| = 0.706714; step 1; tol 0.000751233
1:  |F(x)| = 0.0284976; step 1; tol 0.00146343
2:  |F(x)| = 7.05967e-05; step 1; tol 5.52325e-06

==> Solving for density 1.0
0:  |F(x)| = 0.413714; step 1; tol 0.000691077
1:  |F(x)| = 0.018618; step 1; tol 0.00182267
2:  |F(x)| = 5.4341e-05; step 1; tol 7.6671e-06

Done!
```

We start by reproducing Figure 1 of Reference 1.

```
In [9]: %%opts Overlay [legend_position='bottom_right']
        from math import sqrt

        extents = (0,0,0.6,1.0)

        gr_plots = []
        for rho,x,y in gr_results:
            Rg = sqrt(16000/6.0)
            label = 'rho={} (pyPRISM)'.format(rho)
            style = {'line_dash':ls[rho],'color':colors[rho]}
            c1 = hv.Curve((x/Rg,y),label=label,extents=extents)(style=style)
            gr_plots.append(c1)

        for rho,x,y in gr_compare:
            label = 'rho={} (Ref 1)'.format(rho)
            style = {'marker':markers[rho],'color':colors[rho]}
            c1 = hv.Scatter((x,y),label=label,extents=extents)(style=style)
            gr_plots.append(c1)


        hv.Overlay(gr_plots).redim.label(x='r',y='g(r)')
```

```
Out[9]: :Overlay
           .Curve.Rho_equals_0_full_stop_6_left_parenthesis_pyPRISM_right_parenthesis :Curve   [x]
           .Curve.Rho_equals_0_full_stop_8_left_parenthesis_pyPRISM_right_parenthesis :Curve   [x]
           .Curve.Rho_equals_1_full_stop_0_left_parenthesis_pyPRISM_right_parenthesis :Curve   [x]
           .Scatter.Rho_equals_0_full_stop_6_left_parenthesis_Ref_1_right_parenthesis :Scatter   [x]
           .Scatter.Rho_equals_0_full_stop_8_left_parenthesis_Ref_1_right_parenthesis :Scatter   [x]
           .Scatter.Rho_equals_1_full_stop_0_left_parenthesis_Ref_1_right_parenthesis :Scatter   [x]
```

Next, we reproduce Figure 3:

```
In [10]: sk_plots = []
         for rho,x,y in sk_results:
             yd = y - 1.0
```

```
        label = 'rho={} (pyPRISM)'.format(rho)
        style = {'line_dash':ls[rho],'color':colors[rho]}
        c1 = hv.Curve((x,yd/yd[0]),label=label,extents=(0,0,3,None))(style=style)
        sk_plots.append(c1)

    for rho,x,y in sk_compare:
        style = {'marker':markers[rho],'color':colors[rho]}
        c1 = hv.Scatter((x,y),label='rho={} (Ref 1)'.format(rho),extents=(0.0,0,3,None))(style=s
        sk_plots.append(c1)


    hv.Overlay(sk_plots).redim.label(x='k',y='(S(k)-1)/(S(0)-1)')
```

```
Out[10]: :Overlay
            .Curve.Rho_equals_0_full_stop_6_left_parenthesis_pyPRISM_right_parenthesis :Curve   [x]
            .Curve.Rho_equals_0_full_stop_8_left_parenthesis_pyPRISM_right_parenthesis :Curve   [x]
            .Curve.Rho_equals_1_full_stop_0_left_parenthesis_pyPRISM_right_parenthesis :Curve   [x]
            .Scatter.Rho_equals_0_full_stop_6_left_parenthesis_Ref_1_right_parenthesis :Scatter   [x
            .Scatter.Rho_equals_0_full_stop_8_left_parenthesis_Ref_1_right_parenthesis :Scatter   [x
            .Scatter.Rho_equals_1_full_stop_0_left_parenthesis_Ref_1_right_parenthesis :Scatter   [x
```

Finally, we reproduce the inset of Figure 3:

```
In [11]: sk0_plots = []
         for rho,x,y in sk_results:
             yd = y - 1
             label = 'rho={} (pyPRISM)'.format(rho)
             style = {'line_dash':ls[rho],'color':colors[rho]}
             c1 = hv.Scatter((rho,yd[0]),label=label,extents=(0.4,0,1.2,11))(style=style)
             sk0_plots.append(c1)

         hv.Overlay(sk0_plots).redim.label(x='rho',y='S(0)')
```

```
Out[11]: :Overlay
            .Scatter.Rho_equals_0_full_stop_6_left_parenthesis_pyPRISM_right_parenthesis :Scatter
            .Scatter.Rho_equals_0_full_stop_8_left_parenthesis_pyPRISM_right_parenthesis :Scatter
            .Scatter.Rho_equals_1_full_stop_0_left_parenthesis_pyPRISM_right_parenthesis :Scatter
```

Ring Homopolymer Melt: Comparison to Literature

As a comparison, we'll now focus on a ring-homopolymer melt as described in [Ref. 2]. This paper is actually the first full paper in which PRISM theory was described.

Again, to start we define the aesthetics of the plots.

```
In [12]: %opts Curve Scatter [width=500,height=400] Layout [shared_axes=False] Scatter (size=10,alpha
         %opts Curve Scatter [fontsize={'xlabel':14,'xlabel':14,'ylabel':14,'ticks':12}]
         %opts Overlay [legend_position='bottom_left']
         %opts Layout [shared_axes=False]


         colors = {}
         colors[2000] = 'blue'
         colors[500] = 'red'
         colors[100] = 'green'

         ls = {}
         ls[2000] = 'solid'
         ls[500] = 'dashed'
         ls[100] = 'dotted'
         markers = {}
         markers[2000] = 'o'
         markers[500] = '^'
         markers[100] = 'd'
```

```
In [13]: gr_compare = []
         for density in [2000,500,100]:
             fname = '../data/RingMelt-Gr-N{}.csv'.format(density)
             x,y = np.loadtxt(fname,delimiter=',').T
             gr_compare.append([density,x,y])
```

Now we do the PRISM calculation…

```
In [14]: sys = pyPRISM.System(['polymer'],kT=1.0)
         sys.domain = pyPRISM.Domain(dr=0.05,length=4096)
         sys.diameter['polymer'] = 1.0
         sys.density['polymer'] = 0.9
         sys.closure['polymer','polymer'] = pyPRISM.closure.PercusYevick()
         sys.potential['polymer','polymer'] = pyPRISM.potential.HardSphere()

         gr_results = []
         guess = np.zeros_like(sys.domain.r)
         for length in [2000,500,100]:
             print('==> Solving for gaussian ring of length {}'.format(length))
             sys.omega['polymer','polymer'] = pyPRISM.omega.GaussianRing(sigma=sys.diameter['polymer
             PRISM = sys.createPRISM()
             result = PRISM.solve(guess)
             guess = np.copy(PRISM.x)

             x = sys.domain.r
             y = pyPRISM.calculate.pair_correlation(PRISM)['polymer','polymer']
             gr_results.append(['ring',length,x,y])

         print('Done!')
```

```
==> Solving for gaussian ring of length 2000
0:  |F(x)| = 13.564; step 0.116955; tol 0.701605
1:  |F(x)| = 12.8877; step 0.0499993; tol 0.812488
2:  |F(x)| = 12.6488; step 0.0187059; tol 0.866947
3:  |F(x)| = 11.7434; step 0.0724084; tol 0.775771
4:  |F(x)| = 11.7428; step 5.40487e-05; tol 0.899907
5:  |F(x)| = 11.7419; step 7.80313e-05; tol 0.899865
6:  |F(x)| = 11.7406; step 0.000120235; tol 0.899793
7:  |F(x)| = 11.7383; step 0.000203317; tol 0.899649
8:  |F(x)| = 11.7339; step 0.000393242; tol 0.899322
9:  |F(x)| = 11.7233; step 0.000937021; tol 0.898386
10:  |F(x)| = 11.6878; step 0.00316625; tol 0.894554
11:  |F(x)| = 11.4713; step 0.0193873; tol 0.866966
12:  |F(x)| = 11.4571; step 0.00132076; tol 0.897765
13:  |F(x)| = 11.4557; step 0.000126257; tol 0.899787
14:  |F(x)| = 11.453; step 0.0002504; tol 0.899577
15:  |F(x)| = 11.4463; step 0.000621288; tol 0.89895
16:  |F(x)| = 11.4221; step 0.00225248; tol 0.896198
17:  |F(x)| = 11.2524; step 0.0158588; tol 0.873447
18:  |F(x)| = 11.2523; step 6.45276e-06; tol 0.899989
19:  |F(x)| = 11.2522; step 8.00376e-06; tol 0.899987
20:  |F(x)| = 11.2521; step 1.01827e-05; tol 0.899983
21:  |F(x)| = 11.252; step 1.32855e-05; tol 0.899978
```

```
22:  |F(x)| = 11.2518; step 1.79953e-05; tol 0.89997
23:  |F(x)| = 11.2515; step 2.55292e-05; tol 0.899958
24:  |F(x)| = 11.2511; step 3.83886e-05; tol 0.899936
25:  |F(x)| = 11.2505; step 6.28229e-05; tol 0.899896
26:  |F(x)| = 11.2493; step 0.000115591; tol 0.899808
27:  |F(x)| = 11.2467; step 0.000254364; tol 0.899578
28:  |F(x)| = 11.2389; step 0.000750361; tol 0.898757
29:  |F(x)| = 11.2005; step 0.00371077; tol 0.893866
30:  |F(x)| = 10.7864; step 0.0403192; tol 0.834678
31:  |F(x)| = 10.4959; step 0.0314757; tol 0.85217
32:  |F(x)| = 10.4731; step 0.00275572; tol 0.896107
33:  |F(x)| = 10.1771; step 0.0362332; tol 0.849843
34:  |F(x)| = 10.1755; step 0.000245052; tol 0.899704
35:  |F(x)| = 10.1582; step 0.00253582; tol 0.896942
36:  |F(x)| = 9.98704; step 0.025387; tol 0.869934
37:  |F(x)| = 9.97755; step 0.0016413; tol 0.898291
38:  |F(x)| = 9.80201; step 0.0306349; tol 0.868609
39:  |F(x)| = 9.80165; step 7.641e-05; tol 0.899935
40:  |F(x)| = 9.8015; step 3.34378e-05; tol 0.899972
41:  |F(x)| = 9.80082; step 0.000147167; tol 0.899875
42:  |F(x)| = 9.79355; step 0.00157967; tol 0.898665
43:  |F(x)| = 9.69962; step 0.0206006; tol 0.882819
44:  |F(x)| = 9.67173; step 0.00710773; tol 0.894833
45:  |F(x)| = 9.66218; step 0.00256072; tol 0.898222
46:  |F(x)| = 9.64319; step 0.0051766; tol 0.896467
47:  |F(x)| = 9.58713; step 0.0158394; tol 0.889566
48:  |F(x)| = 9.54215; step 0.0142497; tol 0.891575
49:  |F(x)| = 9.49661; step 0.0159618; tol 0.89143
50:  |F(x)| = 9.48357; step 0.00511646; tol 0.897529
51:  |F(x)| = 9.48304; step 0.000214479; tol 0.8999
52:  |F(x)| = 9.47626; step 0.00275702; tol 0.898713
53:  |F(x)| = 9.44666; step 0.0122378; tol 0.894388
54:  |F(x)| = 9.41731; step 0.0131669; tol 0.894416
55:  |F(x)| = 9.38668; step 0.0149674; tol 0.894154
56:  |F(x)| = 9.37647; step 0.00548425; tol 0.898045
57:  |F(x)| = 9.35142; step 0.00696148; tol 0.895196
58:  |F(x)| = 9.34054; step 0.00456984; tol 0.897907
59:  |F(x)| = 9.33118; step 0.00387752; tol 0.898198
60:  |F(x)| = 9.33091; step 9.42877e-05; tol 0.899948
61:  |F(x)| = 9.30692; step 0.00906009; tol 0.895377
62:  |F(x)| = 9.26985; step 0.0154426; tol 0.892846
63:  |F(x)| = 9.26204; step 0.00362434; tol 0.898483
64:  |F(x)| = 9.23088; step 0.0160041; tol 0.893954
65:  |F(x)| = 9.22582; step 0.00330641; tol 0.899015
66:  |F(x)| = 9.2188; step 0.00389694; tol 0.898631
67:  |F(x)| = 9.20601; step 0.00567855; tol 0.897503
68:  |F(x)| = 9.18625; step 0.0109425; tol 0.896142
69:  |F(x)| = 9.14046; step 0.0194129; tol 0.891049
70:  |F(x)| = 9.11915; step 0.00957558; tol 0.895808
71:  |F(x)| = 9.1081; step 0.00603236; tol 0.89782
72:  |F(x)| = 9.08375; step 0.0136708; tol 0.895195
73:  |F(x)| = 9.07434; step 0.00476936; tol 0.898136
74:  |F(x)| = 9.07327; step 0.000515113; tol 0.899787
75:  |F(x)| = 9.06892; step 0.0020917; tol 0.899138
76:  |F(x)| = 9.05313; step 0.00790108; tol 0.896869
77:  |F(x)| = 9.0377; step 0.00804661; tol 0.896935
78:  |F(x)| = 9.00401; step 0.0164117; tol 0.893303
79:  |F(x)| = 8.97103; step 0.0167142; tol 0.893419
80:  |F(x)| = 8.97027; step 0.00040741; tol 0.899847
```

```
81:  |F(x)| = 8.95825; step 0.00655805; tol 0.897591
82:  |F(x)| = 8.94265; step 0.00873797; tol 0.896866
83:  |F(x)| = 8.93898; step 0.00177432; tol 0.899263
84:  |F(x)| = 8.93109; step 0.00431515; tol 0.898412
85:  |F(x)| = 8.93102; step 3.81577e-05; tol 0.899986
86:  |F(x)| = 8.92359; step 0.0043836; tol 0.898502
87:  |F(x)| = 8.89848; step 0.0127696; tol 0.894942
88:  |F(x)| = 8.89697; step 0.000852819; tol 0.899694
89:  |F(x)| = 8.87556; step 0.0122856; tol 0.895674
90:  |F(x)| = 8.85198; step 0.0132539; tol 0.895225
91:  |F(x)| = 8.84007; step 0.00638331; tol 0.89758
92:  |F(x)| = 8.82702; step 0.00722105; tol 0.897344
93:  |F(x)| = 8.82701; step 3.24097e-06; tol 0.899999
94:  |F(x)| = 7.27724; step 1; tol 0.728998
95:  |F(x)| = 1.87778; step 1; tol 0.478294
96:  |F(x)| = 1.39712; step 0.417561; tol 0.498219
97:  |F(x)| = 0.892148; step 1; tol 0.366988
98:  |F(x)| = 0.19859; step 1; tol 0.121212
99:  |F(x)| = 0.00803493; step 1; tol 0.0014733
100:  |F(x)| = 4.27395e-05; step 1; tol 2.54646e-05
101:  |F(x)| = 8.29099e-10; step 1; tol 3.38685e-10
==> Solving for gaussian ring of length 500
0:  |F(x)| = 0.0018377; step 1; tol 5.01925e-08
1:  |F(x)| = 1.16134e-06; step 1; tol 3.59432e-07
==> Solving for gaussian ring of length 100
0:  |F(x)| = 0.0146501; step 1; tol 5.13412e-06
1:  |F(x)| = 7.48487e-05; step 1; tol 2.34925e-05
2:  |F(x)| = 2.44558e-09; step 1; tol 9.60811e-10
Done!
```

Here we reproduce Figure 1 of Reference 1.

```
In [15]: %%opts Overlay [legend_position='bottom_right']
         from math import sqrt

         extents = (0,0,13,1.0)

         gr_plots = []
         for name,N,x,y in gr_results:
             label = 'N={} (PRISM)'.format(N)
             style = {'line_dash':ls[N],'color':colors[N]}
             c1 = hv.Curve((x,y),label=label,extents=extents)(style=style)
             gr_plots.append(c1)

         for N,x,y in gr_compare:
             label = 'N={} (Ref 2)'.format(N)
             style = {'marker':markers[N],'color':colors[N]}
             c1 = hv.Scatter((x,y),label=label,extents=extents)(style=style)
             gr_plots.append(c1)


         hv.Overlay(gr_plots).redim.label(x='r',y='g(r)')
```

```
Out[15]: :Overlay
            .Curve.N_equals_2000_left_parenthesis_PRISM_right_parenthesis   :Curve   [x]   (y)
            .Curve.N_equals_500_left_parenthesis_PRISM_right_parenthesis    :Curve   [x]   (y)
            .Curve.N_equals_100_left_parenthesis_PRISM_right_parenthesis    :Curve   [x]   (y)
            .Scatter.N_equals_2000_left_parenthesis_Ref_2_right_parenthesis :Scatter   [x]   (y)
            .Scatter.N_equals_500_left_parenthesis_Ref_2_right_parenthesis  :Scatter   [x]   (y)
```

```
        .Scatter.N_equals_100_left_parenthesis_Ref_2_right_parenthesis  :Scatter   [x]   (y)
```

Summary

With both the previous and this notebook finished, we have covered a basic monomer fluid system and two "real-world" polymer systems. We have verified that the pyPRISM implementation matches previous implementations by reproducing data from the literature. Finally, we have begun to touch on two more advanced topics: solving difficult to converge systems and using loops to scan parameter spaces. We will go into more detail on both of these topics in the following notebooks.

NB0.Introduction · NB1.PythonBasics · NB2.Theory.General · NB3.Theory.PRISM · NB4.pyPRISM.Overview · NB5.CaseStudies.PolymerMelts · NB6.CaseStudies.Nanocomposites · NB7.CaseStudies.Copolymers · NB8.pyPRISM.Internals · NB9.pyPRISM.Advanced

Case Studies: Nanocomposites

There have been many PRISM-based theoretical studies focused on understanding how the design of the filler and matrix material in polymer nanocomposites (PNCs) lead to controlled morphologies and particle/filler dispersion in the matrix; much of this work is summarized in several recent reviews.[1-3] Traditional simulation methodologies are challenged to equilibrate large PNC systems with long matrix chains because the relaxation times of these systems are often much longer than comparable simulations at lower densities and/or short matrix chains. Results from PRISM calculations are, by definition, equilibrium predictions so that relaxation times and equilibration are not a problem. Furthermore, the problem of finite-size effects is not present in PRISM theory.

Concepts Used

- Multicomponent PRISM
- Phase space "hopping"
- Heterogeneous interactions
- Domain interpolation
- Complex molecular structure
- Simulation derived $\hat{\omega}(k)$

Tools Used

- pyPRISM.calculate.pair_correlation
- pyPRISM.calculate.pmf

References

1. Ganesan, V. and A. Jayaraman, Theory and simulation studies of effective interactions, phase behavior and morphology in polymer nanocomposites. Soft Matter, 2014. 10(1): p. 13-38.
2. Jayaraman, A. and N. Nair, Integrating PRISM theory and Monte Carlo simulation to study polymer-functionalised particles and polymer nanocomposites. Molecular Simulation, 2012. 38(8-9): p. 751-761.

3. Hall, L.M., A. Jayaraman, and K.S. Schweizer, Molecular theories of polymer nanocomposites. Current Opinion in Solid State and Materials Science, 2010. 14(2): p. 38-48.
4. Hooper, J.B.; Schweizer, K.S.; Contact Aggregation, Bridging, and Steric Stabilization in Dense Polymer Particle Mixtures, Macromolecules 2005, 38, 8858-8869
5. Modica, K.J.; Martin, T.B.; Jayaraman, A.J.; Effect of Architecture on the Structure and Interactions of Polymer Grafted Particles: Theory and Simulation, Macromolecules, 2017, 50 (12), pp 4854-4866

Notebook Setup

To begin, please run `Kernel-> Restart & Clear Output` from the menu at the top of the notebook. It is a good idea to run this before starting any notebook so that the notebook is fresh for the user. Next, run the cell below (via the top menu-bar or `<Shift-Enter>`. If the cell throws an import error, there is likely something wrong with your environment.

If successful, you should see a set of logos appear below the cell. Which logos appear depend on what is inside the `hv.extension()` command at the bottom of the cell. If no logos appear and the cell throws an error, there is likely something wrong with your environment.

Troubleshooting:

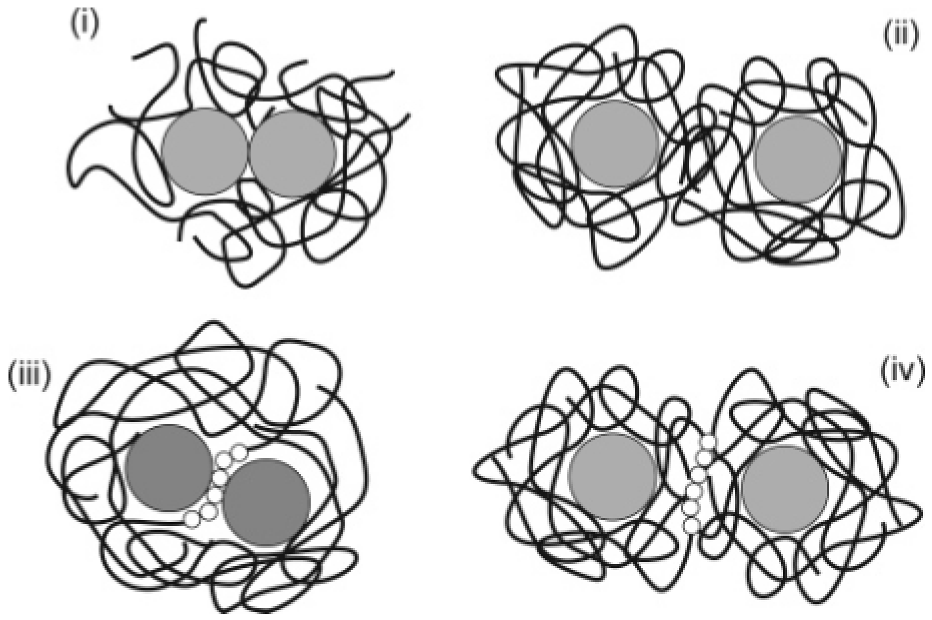
- Did you activate the correct conda environment before starting the jupyter notebook?
- If not using anaconda, did you install all dependencies before starting the jupyter notebook?
- Is pyPRISM installed in your current environment on your `PYTHONPATH`?

Holoviews + Bokeh Logos:

```
In [1]: import pyPRISM
        import numpy as np
        import holoviews as hv
        hv.extension('bokeh')


        def interpolate_guess(domain_from,domain_to,rank,guess):
            '''Helper for upscaling the intial guesses'''
            guess = guess.reshape((domain_from.length,rank,rank))
            new_guess = np.zeros((domain_to.length,rank,rank))
            for i in range(rank):
                for j in range(rank):
                    new_guess[:,i,j] = np.interp(domain_to.r,domain_from.r,guess[:,i,j])
            return new_guess.reshape((-1,))
```

Data type cannot be displayed: application/javascript, application/vnd.bokehjs_load.v0+json

```
<IPython.core.display.HTML object>
```



Bare Particle Nanocomposite

In this example, we use `pyPRISM` to study a polymer nanocomposite system in which polymer chains of length N = 100 are mixed with a bare nanoparticle.[4] The nanoparticle is of diameter D = 16d, where d is the diameter of one segment of the polymer chain (either a monomer or a coarse-grained bead). We plot the particle-particle pair correlations as a function of the range of the particle-polymer attraction.

As usual, we'll define the plot aesthetics first.

```
In [2]: %opts Curve Scatter [width=500,height=400] Layout [shared_axes=False] Scatter (size=10,alpha=
        %opts Curve Scatter [fontsize={'xlabel':14,'xlabel':14,'ylabel':14,'ticks':12}]
        %opts Overlay [legend_position='bottom_left']
        %opts Layout [shared_axes=False]


        colors = {}
        colors[1.0] = 'blue'
        colors[0.5] = 'red'
        colors[0.25] = 'green'

        ls = {}
        ls[1.0] = 'solid'
        ls[0.5] = 'dashed'
        ls[0.25] = 'dotted'

        markers = {}
        markers[1.0] = 'o'
        markers[0.5] = '^'
        markers[0.25] = 'd'
```

First, we load the reference data from Ref. 4

```
In [3]: gr_compare = []
        for density in [1.0,0.5,0.25]:
            fname = '../data/BareComposite-Gr-alpha{}.csv'.format(density)
            x,y = np.loadtxt(fname,delimiter=',').T
        gr_compare.append([density,x,y])
```

Next, we need to solve the PRISM equations, but we can't solve for the particle size we want (D=16d) directly. You can try it, but `pyPRISM` won't be able to converge the equations. Instead, we sequentially solve the PRISM equations starting for a small diameter particle and progressively increase the particle size. Each successive loop (or particle diameter) uses the solution from the last loop as a starting guess, greatly increasing the ability of `pyPRISM` to converge. This process is the so-called "phase space hopping".

```
In [4]: d = 1.0 #polymer segment diameter
        phi = 0.001 #volume fraction of nanoparticles
        eta = 0.4 #total occupied volume fraction

        sys = pyPRISM.System(['particle','polymer'],kT=1.0)
        sys.domain = pyPRISM.Domain(dr=0.1,length=1024)

        guess = np.zeros(sys.rank*sys.rank*sys.domain.length)
        for D in np.arange(1.0,16.5,0.5):
            print('==> Solving for nanoparticle diameter D=',D)

            sys.diameter['polymer'] = d
            sys.diameter['particle'] = D
            sys.density['polymer'] = (1-phi)*eta/sys.diameter.volume['polymer']
            sys.density['particle'] = phi*eta/sys.diameter.volume['particle']
            print('--> rho=',sys.density['polymer'],sys.density['particle'])

            sys.omega['polymer','polymer'] = pyPRISM.omega.FreelyJointedChain(length=100,l=4.0*d/3.0)
            sys.omega['polymer','particle'] = pyPRISM.omega.InterMolecular()
            sys.omega['particle','particle'] = pyPRISM.omega.SingleSite()


            sys.potential['polymer','polymer'] = pyPRISM.potential.HardSphere()
            sys.potential['polymer','particle'] = pyPRISM.potential.Exponential(alpha=0.5,epsilon=1.0
            sys.potential['particle','particle'] = pyPRISM.potential.HardSphere()

            sys.closure['polymer','polymer'] = pyPRISM.closure.PercusYevick()
            sys.closure['polymer','particle'] = pyPRISM.closure.PercusYevick()
            sys.closure['particle','particle'] = pyPRISM.closure.HyperNettedChain()

            PRISM = sys.createPRISM()

            result = PRISM.solve(guess)

            guess = np.copy(PRISM.x)

            print('')

        last_guess=guess
        print('Done!')
```

```
==> Solving for nanoparticle diameter D= 1.0
--> rho= 0.7631797831142566 0.0007639437268410977
0:  |F(x)| = 3.35879; step 1; tol 0.476156
1:  |F(x)| = 0.752641; step 1; tol 0.204052
2:  |F(x)| = 0.189662; step 1; tol 0.0571517
3:  |F(x)| = 0.0163282; step 1; tol 0.00667045
4:  |F(x)| = 8.51675e-05; step 1; tol 2.44859e-05
5:  |F(x)| = 5.70769e-07; step 1; tol 4.04217e-05

==> Solving for nanoparticle diameter D= 1.5
```

```
--> rho= 0.7631797831142566 0.00022635369684180674
0:  |F(x)| = 2.51597; step 1; tol 0.291082
1:  |F(x)| = 0.0149118; step 1; tol 3.1615e-05
2:  |F(x)| = 1.30481e-06; step 1; tol 6.89089e-09

==> Solving for nanoparticle diameter D= 2.0
--> rho= 0.7631797831142566 9.549296585513722e-05
```

```
/home/tbm/software/pyPRISM/dev/src/pyPRISM/core/System.py:150: UserWarning: Sigma for pair particle-p
  warnings.warn(warn_text)
/home/tbm/software/pyPRISM/dev/src/pyPRISM/core/System.py:150: UserWarning: Sigma for pair polymer-pa
  warnings.warn(warn_text)
```

```
0:  |F(x)| = 4.36967; step 1; tol 0.268366
1:  |F(x)| = 0.0170008; step 1; tol 1.36234e-05
2:  |F(x)| = 3.64061e-06; step 1; tol 4.12714e-08

==> Solving for nanoparticle diameter D= 2.5
--> rho= 0.7631797831142566 4.889239851783025e-05
0:  |F(x)| = 13.3593; step 0.449922; tol 0.456306
1:  |F(x)| = 4.12463; step 1; tol 0.187394
2:  |F(x)| = 3.46116; step 0.183782; tol 0.633747
3:  |F(x)| = 2.87865; step 0.21045; tol 0.622555
4:  |F(x)| = 2.31803; step 0.340597; tol 0.583582
5:  |F(x)| = 1.22038; step 1; tol 0.306511
```

```
/home/tbm/software/pyPRISM/dev/src/pyPRISM/core/System.py:150: UserWarning: Sigma for pair particle-p
  warnings.warn(warn_text)
/home/tbm/software/pyPRISM/dev/src/pyPRISM/core/System.py:150: UserWarning: Sigma for pair polymer-pa
  warnings.warn(warn_text)
```

```
6:  |F(x)| = 0.844008; step 0.355097; tol 0.430475
7:  |F(x)| = 0.558075; step 1; tol 0.393491
8:  |F(x)| = 0.0332253; step 1; tol 0.139352
9:  |F(x)| = 0.00112344; step 1; tol 0.00102897
10:  |F(x)| = 0.000120797; step 1; tol 0.0104053
11:  |F(x)| = 5.73715e-07; step 1; tol 2.03013e-05

==> Solving for nanoparticle diameter D= 3.0
--> rho= 0.7631797831142566 2.8294212105225843e-05
0:  |F(x)| = 33.8126; step 1; tol 0.678155
1:  |F(x)| = 28.0988; step 1; tol 0.621528
2:  |F(x)| = 8.49318; step 1; tol 0.347667
3:  |F(x)| = 7.99288; step 0.0674904; tol 0.797092
4:  |F(x)| = 7.53265; step 0.0695425; tol 0.799339
5:  |F(x)| = 6.96739; step 0.107141; tol 0.769994
6:  |F(x)| = 6.09455; step 0.289239; tol 0.688628
7:  |F(x)| = 5.23117; step 0.400351; tol 0.663066
8:  |F(x)| = 4.61057; step 0.207589; tol 0.699124
9:  |F(x)| = 3.87016; step 0.465009; tol 0.634148
10:  |F(x)| = 2.38955; step 1; tol 0.361929
11:  |F(x)| = 0.493714; step 1; tol 0.117894
12:  |F(x)| = 0.0313452; step 1; tol 0.00362772
13:  |F(x)| = 0.000587425; step 1; tol 0.000316086
14:  |F(x)| = 1.01368e-06; step 1; tol 2.68003e-06

==> Solving for nanoparticle diameter D= 3.5
--> rho= 0.7631797831142566 1.7817929489005196e-05
0:  |F(x)| = 17.5377; step 1; tol 0.121571
1:  |F(x)| = 0.0765622; step 1; tol 1.71525e-05
```

```
/home/tbm/software/pyPRISM/dev/src/pyPRISM/core/System.py:150: UserWarning: Sigma for pair particle-p
  warnings.warn(warn_text)
/home/tbm/software/pyPRISM/dev/src/pyPRISM/core/System.py:150: UserWarning: Sigma for pair polymer-pa
  warnings.warn(warn_text)

2:  |F(x)| = 0.000193902; step 1; tol 5.77267e-06
3:  |F(x)| = 2.52679e-09; step 1; tol 1.52833e-10

==> Solving for nanoparticle diameter D= 4.0
--> rho= 0.7631797831142566 1.1936620731892152e-05
0:  |F(x)| = 22.5886; step 1; tol 0.102406
1:  |F(x)| = 0.105308; step 1; tol 1.95609e-05
2:  |F(x)| = 0.000304109; step 1; tol 7.50543e-06
3:  |F(x)| = 1.11149e-08; step 1; tol 1.20224e-09

==> Solving for nanoparticle diameter D= 4.5
--> rho= 0.7631797831142566 8.38347025340025e-06
0:  |F(x)| = 84.8999; step 1; tol 0.318629
1:  |F(x)| = 0.625624; step 1; tol 4.88714e-05

/home/tbm/software/pyPRISM/dev/src/pyPRISM/core/System.py:150: UserWarning: Sigma for pair particle-p
  warnings.warn(warn_text)
/home/tbm/software/pyPRISM/dev/src/pyPRISM/core/System.py:150: UserWarning: Sigma for pair polymer-pa
  warnings.warn(warn_text)

2:  |F(x)| = 0.00991608; step 1; tol 0.000226098
3:  |F(x)| = 1.58233e-06; step 1; tol 2.29169e-08

==> Solving for nanoparticle diameter D= 5.0
--> rho= 0.7631797831142566 6.1115498147287816e-06
0:  |F(x)| = 39.3476; step 1; tol 0.0665939
1:  |F(x)| = 0.147671; step 1; tol 1.26764e-05
2:  |F(x)| = 0.0014265; step 1; tol 8.39835e-05
3:  |F(x)| = 2.326e-08; step 1; tol 2.39286e-10

==> Solving for nanoparticle diameter D= 5.5
--> rho= 0.7631797831142566 4.591697832253028e-06
0:  |F(x)| = 141.56; step 1; tol 0.207739
1:  |F(x)| = 1.19746; step 1; tol 6.43995e-05
2:  |F(x)| = 0.0378919; step 1; tol 0.000901184

/home/tbm/software/pyPRISM/dev/src/pyPRISM/core/System.py:150: UserWarning: Sigma for pair particle-p
  warnings.warn(warn_text)
/home/tbm/software/pyPRISM/dev/src/pyPRISM/core/System.py:150: UserWarning: Sigma for pair polymer-pa
  warnings.warn(warn_text)

3:  |F(x)| = 7.66943e-06; step 1; tol 3.68703e-08

==> Solving for nanoparticle diameter D= 6.0
--> rho= 0.7631797831142566 3.5367765131532304e-06
0:  |F(x)| = 62.1156; step 1; tol 0.045593
1:  |F(x)| = 0.407947; step 1; tol 3.88193e-05
2:  |F(x)| = 0.00412423; step 1; tol 9.19861e-05
3:  |F(x)| = 4.59293e-08; step 1; tol 1.11619e-10

==> Solving for nanoparticle diameter D= 6.5
--> rho= 0.7631797831142566 2.7817705119384533e-06
0:  |F(x)| = 207.772; step 1; tol 0.132233
1:  |F(x)| = 15.1124; step 1; tol 0.0047614
2:  |F(x)| = 3.13; step 1; tol 0.0386067
3:  |F(x)| = 0.0577293; step 1; tol 0.000306159
```

```
/home/tbm/software/pyPRISM/dev/src/pyPRISM/core/System.py:150: UserWarning: Sigma for pair particle-p
  warnings.warn(warn_text)
/home/tbm/software/pyPRISM/dev/src/pyPRISM/core/System.py:150: UserWarning: Sigma for pair polymer-pa
  warnings.warn(warn_text)
```

```
4:  |F(x)| = 9.18447e-05; step 1; tol 2.27802e-06
5:  |F(x)| = 3.75471e-09; step 1; tol 1.50413e-09

==> Solving for nanoparticle diameter D= 7.0
--> rho= 0.7631797831142566 2.2272411861256494e-06
0:  |F(x)| = 277.284; step 1; tol 0.120386
1:  |F(x)| = 4.18104; step 1; tol 0.000204626
2:  |F(x)| = 0.160302; step 1; tol 0.00132297
3:  |F(x)| = 1.6376e-05; step 1; tol 9.39248e-09

==> Solving for nanoparticle diameter D= 7.5
--> rho= 0.7631797831142566 1.8108295747344538e-06
0:  |F(x)| = 117.836; step 1; tol 0.0286554
1:  |F(x)| = 2.76609; step 1; tol 0.000495926
2:  |F(x)| = 0.0152838; step 1; tol 2.74772e-05
```

```
/home/tbm/software/pyPRISM/dev/src/pyPRISM/core/System.py:150: UserWarning: Sigma for pair particle-p
  warnings.warn(warn_text)
/home/tbm/software/pyPRISM/dev/src/pyPRISM/core/System.py:150: UserWarning: Sigma for pair polymer-pa
  warnings.warn(warn_text)
```

```
3:  |F(x)| = 0.000112244; step 1; tol 4.85406e-05
4:  |F(x)| = 3.3387e-08; step 1; tol 7.96289e-08

==> Solving for nanoparticle diameter D= 8.0
--> rho= 0.7631797831142566 1.492077591486519e-06
0:  |F(x)| = 397.936; step 1; tol 0.0940404
1:  |F(x)| = 10.4781; step 1; tol 0.000623994
2:  |F(x)| = 0.8405; step 1; tol 0.00579099
3:  |F(x)| = 0.00383529; step 1; tol 1.87398e-05
4:  |F(x)| = 2.39322e-05; step 1; tol 3.50438e-05

==> Solving for nanoparticle diameter D= 8.5
--> rho= 0.7631797831142566 1.2439547760490093e-06
0:  |F(x)| = 176.473; step 1; tol 0.0261535
1:  |F(x)| = 3.76473; step 1; tol 0.000409592
2:  |F(x)| = 0.0324109; step 1; tol 6.67051e-05
```

```
/home/tbm/software/pyPRISM/dev/src/pyPRISM/core/System.py:150: UserWarning: Sigma for pair particle-p
  warnings.warn(warn_text)
/home/tbm/software/pyPRISM/dev/src/pyPRISM/core/System.py:150: UserWarning: Sigma for pair polymer-pa
  warnings.warn(warn_text)
```

```
3:  |F(x)| = 1.7509e-06; step 1; tol 2.62653e-09

==> Solving for nanoparticle diameter D= 9.0
--> rho= 0.7631797831142566 1.0479337816750313e-06
0:  |F(x)| = 574.84; step 1; tol 0.0829008
1:  |F(x)| = 15.9181; step 1; tol 0.00069013
2:  |F(x)| = 1.65015; step 1; tol 0.00967176
3:  |F(x)| = 0.00655294; step 1; tol 1.41928e-05
4:  |F(x)| = 0.000175832; step 1; tol 0.000647989
5:  |F(x)| = 3.67023e-08; step 1; tol 3.92132e-08

==> Solving for nanoparticle diameter D= 9.5
--> rho= 0.7631797831142566 8.910263616749937e-07
```

```
0:  |F(x)| = 239.797; step 1; tol 0.0215953
1:  |F(x)| = 11.0745; step 1; tol 0.00191958
2:  |F(x)| = 0.0434715; step 1; tol 1.38676e-05
```

```
/home/tbm/software/pyPRISM/dev/src/pyPRISM/core/System.py:150: UserWarning: Sigma for pair particle-p
  warnings.warn(warn_text)
/home/tbm/software/pyPRISM/dev/src/pyPRISM/core/System.py:150: UserWarning: Sigma for pair polymer-pa
  warnings.warn(warn_text)
```

```
3:  |F(x)| = 4.73801e-06; step 1; tol 1.06912e-08

==> Solving for nanoparticle diameter D= 10.0
--> rho= 0.7631797831142566 7.639437268410977e-07
0:  |F(x)| = 768.395; step 1; tol 0.068102
1:  |F(x)| = 30.3432; step 1; tol 0.00140345
2:  |F(x)| = 5.95924; step 1; tol 0.0347137
3:  |F(x)| = 0.0444734; step 1; tol 5.01257e-05
4:  |F(x)| = 0.000112389; step 1; tol 5.74763e-06
5:  |F(x)| = 2.26527e-09; step 1; tol 3.65626e-10

==> Solving for nanoparticle diameter D= 10.5
--> rho= 0.7631797831142566 6.599233144075998e-07
0:  |F(x)| = 314.242; step 1; tol 0.0178885
1:  |F(x)| = 16.4152; step 1; tol 0.00245587
2:  |F(x)| = 0.0876236; step 1; tol 2.56444e-05
```

```
/home/tbm/software/pyPRISM/dev/src/pyPRISM/core/System.py:150: UserWarning: Sigma for pair particle-p
  warnings.warn(warn_text)
/home/tbm/software/pyPRISM/dev/src/pyPRISM/core/System.py:150: UserWarning: Sigma for pair polymer-pa
  warnings.warn(warn_text)
```

```
3:  |F(x)| = 3.25965e-05; step 1; tol 1.24549e-07

==> Solving for nanoparticle diameter D= 11.0
--> rho= 0.7631797831142566 5.739622290316286e-07
0:  |F(x)| = 1220.51; step 1; tol 0.0845337
1:  |F(x)| = 81.7903; step 1; tol 0.00404171
2:  |F(x)| = 32.4461; step 1; tol 0.141633
3:  |F(x)| = 0.390322; step 1; tol 0.000130245
4:  |F(x)| = 0.0027524; step 1; tol 4.47528e-05
5:  |F(x)| = 2.15594e-07; step 1; tol 5.52194e-09

==> Solving for nanoparticle diameter D= 11.5
--> rho= 0.7631797831142566 5.023054010626105e-07
0:  |F(x)| = 493.505; step 1; tol 0.0226383
1:  |F(x)| = 26.7595; step 1; tol 0.00264615
2:  |F(x)| = 0.46914; step 1; tol 0.000276626
```

```
/home/tbm/software/pyPRISM/dev/src/pyPRISM/core/System.py:150: UserWarning: Sigma for pair particle-p
  warnings.warn(warn_text)
/home/tbm/software/pyPRISM/dev/src/pyPRISM/core/System.py:150: UserWarning: Sigma for pair polymer-pa
  warnings.warn(warn_text)
```

```
3:  |F(x)| = 7.8622e-05; step 1; tol 2.5277e-08
4:  |F(x)| = 8.88924e-07; step 1; tol 0.000115049

==> Solving for nanoparticle diameter D= 12.0
--> rho= 0.7631797831142566 4.420970641441538e-07
0:  |F(x)| = 1381.16; step 1; tol 0.0562768
1:  |F(x)| = 84.4891; step 1; tol 0.0033679
2:  |F(x)| = 16.6235; step 1; tol 0.0348407
```

```
3:  |F(x)| = 0.507236; step 1; tol 0.00083795
4:  |F(x)| = 0.0100824; step 1; tol 0.000355591
5:  |F(x)| = 1.30405e-05; step 1; tol 1.50557e-06

==> Solving for nanoparticle diameter D= 12.5
--> rho= 0.7631797831142566 3.9113918814264206e-07
0:  |F(x)| = 1243.82; step 1; tol 0.0776648
1:  |F(x)| = 595.175; step 1; tol 0.20607
2:  |F(x)| = 13.4545; step 1; tol 0.000459925
```

```
/home/tbm/software/pyPRISM/dev/src/pyPRISM/core/System.py:150: UserWarning: Sigma for pair particle-p
  warnings.warn(warn_text)
/home/tbm/software/pyPRISM/dev/src/pyPRISM/core/System.py:150: UserWarning: Sigma for pair polymer-pa
  warnings.warn(warn_text)
```

```
3:  |F(x)| = 1.86278; step 1; tol 0.0172518
4:  |F(x)| = 0.0079935; step 1; tol 1.65727e-05
5:  |F(x)| = 1.46836e-05; step 1; tol 3.03692e-06

==> Solving for nanoparticle diameter D= 13.0
--> rho= 0.7631797831142566 3.4772131399230666e-07
0:  |F(x)| = 1411.41; step 1; tol 0.0319697
1:  |F(x)| = 1254.36; step 1; tol 0.710853
2:  |F(x)| = 655.034; step 1; tol 0.454781
3:  |F(x)| = 438.233; step 1; tol 0.402833
4:  |F(x)| = 91.4749; step 1; tol 0.146047
5:  |F(x)| = 18.8725; step 1; tol 0.0383089
6:  |F(x)| = 16.4094; step 1; tol 0.680402
7:  |F(x)| = 9.49042; step 1; tol 0.416652
8:  |F(x)| = 3.13953; step 1; tol 0.156239
9:  |F(x)| = 0.510996; step 1; tol 0.0238423
10:  |F(x)| = 0.372683; step 1; tol 0.478725
11:  |F(x)| = 0.0599878; step 1; tol 0.20626
12:  |F(x)| = 0.00299102; step 1; tol 0.00223747
13:  |F(x)| = 9.10151e-05; step 1; tol 0.000833355
14:  |F(x)| = 1.37508e-06; step 1; tol 0.000205434

==> Solving for nanoparticle diameter D= 13.5
--> rho= 0.7631797831142566 3.1049889827408335e-07
0:  |F(x)| = 783.486; step 1; tol 0.0173718
1:  |F(x)| = 26.3862; step 1; tol 0.00102078
2:  |F(x)| = 0.155281; step 1; tol 3.11693e-05
```

```
/home/tbm/software/pyPRISM/dev/src/pyPRISM/core/System.py:150: UserWarning: Sigma for pair particle-p
  warnings.warn(warn_text)
/home/tbm/software/pyPRISM/dev/src/pyPRISM/core/System.py:150: UserWarning: Sigma for pair polymer-pa
  warnings.warn(warn_text)
```

```
3:  |F(x)| = 1.5596e-05; step 1; tol 9.07881e-09

==> Solving for nanoparticle diameter D= 14.0
--> rho= 0.7631797831142566 2.784051482657062e-07
0:  |F(x)| = 3193.67; step 1; tol 0.0924518
1:  |F(x)| = 243.267; step 1; tol 0.0052219
2:  |F(x)| = 195.62; step 1; tol 0.581975
3:  |F(x)| = 88.044; step 1; tol 0.304825
4:  |F(x)| = 39.5622; step 1; tol 0.181721
5:  |F(x)| = 7.02791; step 1; tol 0.0284009
6:  |F(x)| = 0.293198; step 1; tol 0.00156644
7:  |F(x)| = 0.000767776; step 1; tol 6.17146e-06
8:  |F(x)| = 3.03427e-05; step 1; tol 0.00140567
```

```
==> Solving for nanoparticle diameter D= 14.5
--> rho= 0.7631797831142566 2.505863223063177e-07
0:  |F(x)| = 1216.2; step 1; tol 0.0244694
1:  |F(x)| = 64.0155; step 1; tol 0.00249347
```

```
/home/tbm/software/pyPRISM/dev/src/pyPRISM/core/PRISM.py:225: UserWarning: Pair correlations are nega
  warnings.warn(warnstr.format(val,t1,t2))
/home/tbm/software/pyPRISM/dev/src/pyPRISM/core/System.py:150: UserWarning: Sigma for pair particle-p
  warnings.warn(warn_text)
/home/tbm/software/pyPRISM/dev/src/pyPRISM/core/System.py:150: UserWarning: Sigma for pair polymer-pa
  warnings.warn(warn_text)
```

```
2:  |F(x)| = 1.09273; step 1; tol 0.000262239
3:  |F(x)| = 0.000607714; step 1; tol 2.78366e-07
4:  |F(x)| = 2.20878e-06; step 1; tol 1.18891e-05

==> Solving for nanoparticle diameter D= 15.0
--> rho= 0.7631797831142566 2.2635369684180673e-07
0:  |F(x)| = 5007.95; step 1; tol 0.132471
1:  |F(x)| = 546.895; step 1; tol 0.0107333
2:  |F(x)| = 355.407; step 0.481231; tol 0.380089
3:  |F(x)| = 137.061; step 1; tol 0.13385
4:  |F(x)| = 17.7858; step 1; tol 0.0151552
5:  |F(x)| = 0.708749; step 1; tol 0.00142916
6:  |F(x)| = 0.000705166; step 1; tol 8.90922e-07
7:  |F(x)| = 3.59783e-07; step 1; tol 2.34283e-07

==> Solving for nanoparticle diameter D= 15.5
--> rho= 0.7631797831142566 2.0514752155781216e-07
0:  |F(x)| = 1670.91; step 1; tol 0.0278465
1:  |F(x)| = 378.977; step 1; tol 0.0462983
2:  |F(x)| = 16.4635; step 1; tol 0.00169848
3:  |F(x)| = 1.77287; step 1; tol 0.0104365
```

```
/home/tbm/software/pyPRISM/dev/src/pyPRISM/core/System.py:150: UserWarning: Sigma for pair particle-p
  warnings.warn(warn_text)
/home/tbm/software/pyPRISM/dev/src/pyPRISM/core/System.py:150: UserWarning: Sigma for pair polymer-pa
  warnings.warn(warn_text)
```

```
4:  |F(x)| = 0.0164307; step 1; tol 7.73032e-05
5:  |F(x)| = 5.5953e-06; step 1; tol 1.04371e-07

==> Solving for nanoparticle diameter D= 16.0
--> rho= 0.7631797831142566 1.8650969893581487e-07
0:  |F(x)| = 4820.11; step 1; tol 0.0733899
1:  |F(x)| = 558.632; step 1; tol 0.0120887
2:  |F(x)| = 141.735; step 1; tol 0.0579356
3:  |F(x)| = 7.23757; step 1; tol 0.00234678
4:  |F(x)| = 0.216905; step 1; tol 0.000808348
5:  |F(x)| = 3.50759e-05; step 1; tol 2.35353e-08
6:  |F(x)| = 7.48092e-08; step 1; tol 4.09387e-06
```

```
Done!
```

Now that we have the D=16d solution as a guess, we can use it to solve the PRISM equations for three interaction widths. Note also that we have changed the domain characteristics (grid spacing and length of arrays), thus we need to use the interpolate_guess function (defined above) to convert the results using our prevous domain into the new domain.

```
In [5]: sys = pyPRISM.System(['particle','polymer'],kT=1.0)
```

```
    sys.domain = pyPRISM.Domain(dr=0.075,length=2048)

    sys.diameter['polymer'] = d
    sys.diameter['particle'] = D
    sys.density['polymer'] = (1-phi)*eta/sys.diameter.volume['polymer']
    sys.density['particle'] = phi*eta/sys.diameter.volume['particle']

    sys.omega['polymer','polymer'] = pyPRISM.omega.FreelyJointedChain(length=100,l=4.0*d/3.0)
    sys.omega['polymer','particle'] = pyPRISM.omega.NoIntra()
    sys.omega['particle','particle'] = pyPRISM.omega.SingleSite()

    sys.closure['polymer','polymer'] = pyPRISM.closure.PercusYevick()
    sys.closure['polymer','particle'] = pyPRISM.closure.PercusYevick()
    sys.closure['particle','particle'] = pyPRISM.closure.HyperNettedChain()

    gr_results = []
    guess = interpolate_guess(pyPRISM.Domain(dr=0.1,length=1024),sys.domain,sys.rank,last_guess)
    for alpha in [0.25,0.5,1.0]:
        print('==> Solving for alpha=',alpha)
        sys.potential['polymer','polymer'] = pyPRISM.potential.HardSphere()
        sys.potential['polymer','particle'] = pyPRISM.potential.Exponential(alpha=alpha,epsilon=1
        sys.potential['particle','particle'] = pyPRISM.potential.HardSphere()

        PRISM = sys.createPRISM()
        result = PRISM.solve(guess)

        x = sys.domain.r
        y = pyPRISM.calculate.pair_correlation(PRISM)['particle','particle']
        gr_results.append([alpha,x,y])

    print('Done!')
```

```
==> Solving for alpha= 0.25
0:  |F(x)| = 5662.39; step 1; tol 0.0366116
1:  |F(x)| = 506.876; step 1; tol 0.00721186
```

```
/home/tbm/software/pyPRISM/dev/src/pyPRISM/core/System.py:141: UserWarning: Diameter for site particl
  warnings.warn(warn_text)
/home/tbm/software/pyPRISM/dev/src/pyPRISM/core/System.py:141: UserWarning: Diameter for site polymer
  warnings.warn(warn_text)
/home/tbm/software/pyPRISM/dev/src/pyPRISM/core/System.py:150: UserWarning: Sigma for pair particle-p
  warnings.warn(warn_text)
/home/tbm/software/pyPRISM/dev/src/pyPRISM/core/System.py:150: UserWarning: Sigma for pair particle-p
  warnings.warn(warn_text)
/home/tbm/software/pyPRISM/dev/src/pyPRISM/core/System.py:150: UserWarning: Sigma for pair polymer-pa
  warnings.warn(warn_text)
/home/tbm/software/pyPRISM/dev/src/pyPRISM/core/System.py:150: UserWarning: Sigma for pair polymer-po
  warnings.warn(warn_text)
```

```
2:  |F(x)| = 484.693; step 0.0482191; tol 0.822948
3:  |F(x)| = 386.994; step 1; tol 0.60952
4:  |F(x)| = 181.28; step 1; tol 0.334363
5:  |F(x)| = 57.4303; step 1; tol 0.100619
6:  |F(x)| = 57.4266; step 6.28341e-05; tol 0.899883
7:  |F(x)| = 55.3488; step 1; tol 0.83605
8:  |F(x)| = 55.3217; step 1; tol 0.899118
9:  |F(x)| = 54.9051; step 0.0341534; tol 0.886497
10:  |F(x)| = 54.9045; step 1; tol 0.899981
11:  |F(x)| = 54.2824; step 0.0629851; tol 0.879722
```

```
12:  |F(x)| = 54.0923; step 0.00546448; tol 0.893705
13:  |F(x)| = 53.8822; step 0.00653873; tol 0.893024
14:  |F(x)| = 53.6781; step 0.0063891; tol 0.893192
15:  |F(x)| = 53.4702; step 0.00660002; tol 0.893042
16:  |F(x)| = 53.2445; step 0.00737255; tol 0.89242
17:  |F(x)| = 53.028; step 0.00714387; tol 0.892695
18:  |F(x)| = 52.7721; step 0.00869284; tol 0.891335
19:  |F(x)| = 52.4261; step 0.0123779; tol 0.888237
20:  |F(x)| = 51.9933; step 0.0181704; tol 0.885204
21:  |F(x)| = 48.7829; step 0.223636; tol 0.792286
22:  |F(x)| = 48.225; step 0.0311114; tol 0.879531
23:  |F(x)| = 46.8732; step 0.145401; tol 0.850254
24:  |F(x)| = 46.2001; step 0.0447723; tol 0.874337
25:  |F(x)| = 45.4206; step 0.0593008; tol 0.869887
26:  |F(x)| = 44.3289; step 0.101496; tol 0.857255
27:  |F(x)| = 42.7629; step 0.169827; tol 0.837535
28:  |F(x)| = 39.5406; step 0.18508; tol 0.769475
29:  |F(x)| = 39.1283; step 0.0207502; tol 0.881328
30:  |F(x)| = 37.6503; step 0.244255; tol 0.833291
31:  |F(x)| = 37.076; step 0.0283116; tol 0.872756
32:  |F(x)| = 34.9091; step 0.421424; tol 0.797871
33:  |F(x)| = 33.7516; step 0.119877; tol 0.841309
34:  |F(x)| = 32.3594; step 0.20191; tol 0.827282
35:  |F(x)| = 29.8944; step 0.216001; tol 0.768105
36:  |F(x)| = 27.571; step 0.256759; tol 0.765544
37:  |F(x)| = 27.2736; step 0.0158169; tol 0.880689
38:  |F(x)| = 26.9287; step 0.0191221; tol 0.877379
39:  |F(x)| = 26.5379; step 0.0223223; tol 0.874069
40:  |F(x)| = 26.0869; step 0.0268403; tol 0.86967
41:  |F(x)| = 25.2816; step 0.0612396; tol 0.845287
42:  |F(x)| = 23.7826; step 0.171257; tol 0.796441
43:  |F(x)| = 21.8223; step 0.280001; tol 0.757744
44:  |F(x)| = 21.2037; step 0.0651854; tol 0.849699
45:  |F(x)| = 19.4407; step 1; tol 0.756562
46:  |F(x)| = 19.0303; step 0.058792; tol 0.862404
47:  |F(x)| = 17.6025; step 0.413611; tol 0.770011
48:  |F(x)| = 17.4875; step 0.0129598; tol 0.888285
49:  |F(x)| = 15.9301; step 1; tol 0.74683
50:  |F(x)| = 15.7945; step 0.0136197; tol 0.884747
51:  |F(x)| = 15.197; step 0.239605; tol 0.833196
52:  |F(x)| = 14.2615; step 0.382576; tol 0.792602
53:  |F(x)| = 13.8855; step 0.123451; tol 0.853175
54:  |F(x)| = 13.2629; step 0.0746045; tol 0.821092
55:  |F(x)| = 13.0066; step 0.0465958; tol 0.865558
56:  |F(x)| = 11.4684; step 0.356923; tol 0.699713
57:  |F(x)| = 9.90548; step 1; tol 0.67141
58:  |F(x)| = 8.39992; step 0.286561; tol 0.647206
59:  |F(x)| = 7.99417; step 0.0713259; tol 0.815153
60:  |F(x)| = 6.71961; step 1; tol 0.635892
61:  |F(x)| = 6.56528; step 0.0403031; tol 0.859135
62:  |F(x)| = 5.78488; step 1; tol 0.698756
63:  |F(x)| = 5.6753; step 0.0449467; tol 0.866227
64:  |F(x)| = 5.38428; step 0.259107; tol 0.810063
65:  |F(x)| = 4.99153; step 0.282899; tol 0.773492
66:  |F(x)| = 4.54696; step 0.293414; tol 0.746822
67:  |F(x)| = 4.35788; step 0.144591; tol 0.826703
68:  |F(x)| = 115.882; step 1; tol 0.9999
69:  |F(x)| = 111.255; step 1; tol 0.89982
70:  |F(x)| = 5.38899; step 1; tol 0.728708
```

```
71:  |F(x)| = 4.28912; step 1; tol 0.570117
72:  |F(x)| = 4.25354; step 0.0107603; tol 0.885131
73:  |F(x)| = 3.74899; step 1; tol 0.705112
74:  |F(x)| = 3.64651; step 0.0369959; tol 0.851471
75:  |F(x)| = 3.49717; step 0.308434; tol 0.82779
76:  |F(x)| = 3.47186; step 0.0175346; tol 0.887022
77:  |F(x)| = 3.10644; step 0.452557; tol 0.720516
78:  |F(x)| = 2.70975; step 0.180118; tol 0.684819
79:  |F(x)| = 2.69391; step 1; tol 0.889504
80:  |F(x)| = 2.09303; step 1; tol 0.712095
81:  |F(x)| = 1.57302; step 1; tol 0.508347
82:  |F(x)| = 1.29932; step 1; tol 0.614057
83:  |F(x)| = 0.734189; step 1; tol 0.33936
84:  |F(x)| = 0.154052; step 1; tol 0.103648
85:  |F(x)| = 0.0124538; step 1; tol 0.00588186
86:  |F(x)| = 0.000443185; step 1; tol 0.00113974
87:  |F(x)| = 2.86368e-05; step 1; tol 0.0037577
==> Solving for alpha= 0.5
0:  |F(x)| = 6746.45; step 1; tol 0.183588
1:  |F(x)| = 899.268; step 1; tol 0.0159908
2:  |F(x)| = 883.065; step 0.0182703; tol 0.86786
3:  |F(x)| = 866.116; step 0.0196403; tol 0.865783
4:  |F(x)| = 848.79; step 0.0205053; tol 0.864353
5:  |F(x)| = 830.828; step 0.0218293; tol 0.862312
6:  |F(x)| = 811.965; step 0.0236937; tol 0.859597
7:  |F(x)| = 791.936; step 0.0261192; tol 0.856148
8:  |F(x)| = 770.518; step 0.029187; tol 0.851977
9:  |F(x)| = 747.724; step 0.0324992; tol 0.847537
10:  |F(x)| = 722.348; step 0.0386579; tol 0.839951
11:  |F(x)| = 685.311; step 0.0675702; tol 0.810074
12:  |F(x)| = 622.59; step 0.176722; tol 0.7428
13:  |F(x)| = 545.469; step 0.356716; tol 0.690842
14:  |F(x)| = 419.857; step 0.463427; tol 0.533219
15:  |F(x)| = 364.86; step 1; tol 0.679659
16:  |F(x)| = 162.208; step 1; tol 0.415743
17:  |F(x)| = 118.493; step 1; tol 0.480267
18:  |F(x)| = 43.0135; step 1; tol 0.207591
19:  |F(x)| = 42.4792; step 0.0127218; tol 0.87778
20:  |F(x)| = 34.6238; step 1; tol 0.693449
21:  |F(x)| = 18.1081; step 1; tol 0.432784
22:  |F(x)| = 18.096; step 0.00148958; tol 0.898802
23:  |F(x)| = 16.7957; step 1; tol 0.77531
24:  |F(x)| = 16.7661; step 0.00236339; tol 0.896826
25:  |F(x)| = 14.6038; step 1; tol 0.723867
26:  |F(x)| = 14.5857; step 0.00234181; tol 0.89777
27:  |F(x)| = 14.5659; step 0.00268224; tol 0.897558
28:  |F(x)| = 14.5451; step 0.00291978; tol 0.897434
29:  |F(x)| = 14.5226; step 0.0033461; tol 0.897211
30:  |F(x)| = 14.4924; step 0.00501023; tol 0.896268
31:  |F(x)| = 14.4244; step 0.0150729; tol 0.891573
32:  |F(x)| = 14.3701; step 0.0156674; tol 0.893232
33:  |F(x)| = 14.3295; step 0.00941524; tol 0.89492
34:  |F(x)| = 14.2789; step 0.0137285; tol 0.893656
35:  |F(x)| = 14.2088; step 0.0237725; tol 0.891192
36:  |F(x)| = 14.1776; step 0.00347429; tol 0.896049
37:  |F(x)| = 14.1468; step 0.00338865; tol 0.896086
38:  |F(x)| = 14.1154; step 0.00345969; tol 0.896019
39:  |F(x)| = 14.0828; step 0.00362123; tol 0.895848
40:  |F(x)| = 14.0497; step 0.0037036; tol 0.895763
```

```
41:  |F(x)| = 14.0159; step 0.0037831; tol 0.895676
42:  |F(x)| = 13.9763; step 0.0045277; tol 0.894933
43:  |F(x)| = 13.936; step 0.0046499; tol 0.894809
44:  |F(x)| = 13.8757; step 0.00739434; tol 0.892228
45:  |F(x)| = 13.6452; step 0.038678; tol 0.87035
46:  |F(x)| = 13.3834; step 0.0828193; tol 0.865797
47:  |F(x)| = 13.0434; step 0.10834; tol 0.854847
48:  |F(x)| = 12.5624; step 0.12922; tol 0.834849
49:  |F(x)| = 12.3856; step 0.0349655; tol 0.874842
50:  |F(x)| = 11.98; step 0.110287; tol 0.842021
51:  |F(x)| = 11.4691; step 0.178552; tol 0.824871
52:  |F(x)| = 11.1469; step 0.103842; tol 0.850144
53:  |F(x)| = 10.8327; step 0.113741; tol 0.849987
54:  |F(x)| = 10.5114; step 0.143656; tol 0.847405
55:  |F(x)| = 10.4234; step 0.0233655; tol 0.884981
56:  |F(x)| = 10.0591; step 0.27104; tol 0.83819
57:  |F(x)| = 9.93552; step 0.0421546; tol 0.878025
58:  |F(x)| = 9.8003; step 0.0503998; tol 0.87567
59:  |F(x)| = 9.65847; step 0.0582761; tol 0.874138
60:  |F(x)| = 9.51607; step 0.0717795; tol 0.873658
61:  |F(x)| = 9.30839; step 0.126937; tol 0.861146
62:  |F(x)| = 9.14466; step 0.088775; tol 0.868617
63:  |F(x)| = 9.07084; step 0.0175895; tol 0.885528
64:  |F(x)| = 8.88409; step 0.116758; tol 0.863323
65:  |F(x)| = 8.82598; step 0.0183666; tol 0.888265
66:  |F(x)| = 8.64064; step 0.138253; tol 0.862598
67:  |F(x)| = 8.56059; step 0.0184698; tol 0.8834
68:  |F(x)| = 8.34404; step 0.179322; tol 0.855043
69:  |F(x)| = 8.1986; step 0.0798705; tol 0.868898
70:  |F(x)| = 8.01219; step 0.129745; tol 0.85954
71:  |F(x)| = 7.77286; step 0.151839; tol 0.847035
72:  |F(x)| = 7.49663; step 0.211838; tol 0.83717
73:  |F(x)| = 7.33552; step 0.143322; tol 0.861731
74:  |F(x)| = 7.11758; step 0.202731; tol 0.847318
75:  |F(x)| = 6.92584; step 0.0841314; tol 0.852163
76:  |F(x)| = 6.66849; step 0.148919; tol 0.834358
77:  |F(x)| = 6.49641; step 0.0936889; tol 0.854148
78:  |F(x)| = 6.27982; step 0.173879; tol 0.840989
79:  |F(x)| = 6.21005; step 0.104383; tol 0.880113
80:  |F(x)| = 5.95206; step 0.311442; tol 0.826774
81:  |F(x)| = 5.84771; step 0.0658969; tol 0.86872
82:  |F(x)| = 5.52592; step 0.386533; tol 0.803673
83:  |F(x)| = 5.4901; step 0.0306027; tol 0.888372
84:  |F(x)| = 5.32078; step 0.0924585; tol 0.84534
85:  |F(x)| = 5.11388; step 1; tol 0.831368
86:  |F(x)| = 4.91527; step 0.233652; tol 0.831451
87:  |F(x)| = 3.94431; step 1; tol 0.62218
88:  |F(x)| = 2.71733; step 1; tol 0.427155
89:  |F(x)| = 1.00216; step 1; tol 0.164215
90:  |F(x)| = 0.163451; step 1; tol 0.0239411
91:  |F(x)| = 0.00369448; step 1; tol 0.000459807
92:  |F(x)| = 1.58342e-05; step 1; tol 1.65322e-05
==> Solving for alpha= 1.0
0:  |F(x)| = 8084.63; step 1; tol 0.342037
1:  |F(x)| = 2811.99; step 1; tol 0.10888
2:  |F(x)| = 2679.33; step 0.0540234; tol 0.817087
3:  |F(x)| = 2457.27; step 0.145064; tol 0.757
4:  |F(x)| = 2229.62; step 0.293304; tol 0.740964
5:  |F(x)| = 1401.84; step 1; tol 0.494125
```

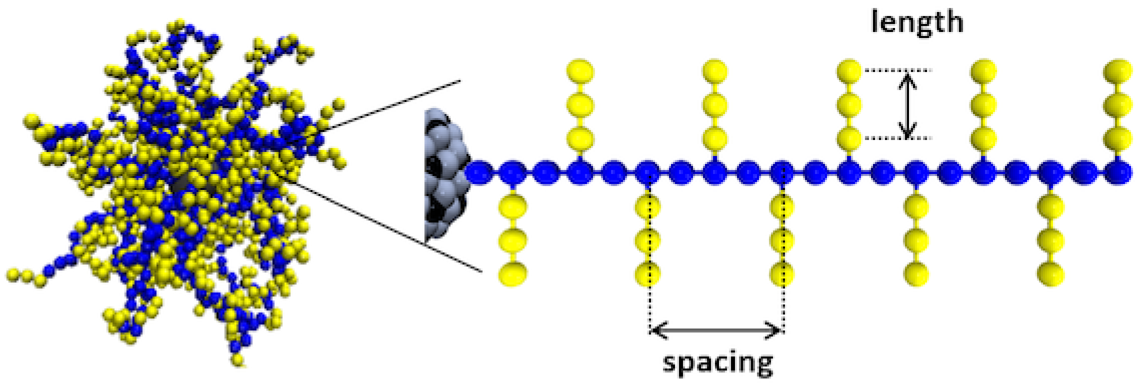
```
6:  |F(x)| = 635.165; step 1; tol 0.219744
7:  |F(x)| = 561.901; step 0.120546; tol 0.704352
8:  |F(x)| = 493.256; step 1; tol 0.693534
9:  |F(x)| = 371.347; step 1; tol 0.510102
10:  |F(x)| = 332.655; step 0.143602; tol 0.722223
11:  |F(x)| = 237.016; step 1; tol 0.469445
12:  |F(x)| = 233.74; step 0.0196123; tol 0.875297
13:  |F(x)| = 219.77; step 0.20992; tol 0.795634
14:  |F(x)| = 212.642; step 0.0406852; tol 0.842559
15:  |F(x)| = 202.983; step 0.0678949; tol 0.820095
16:  |F(x)| = 186.052; step 0.20501; tol 0.756122
17:  |F(x)| = 178.744; step 0.0933333; tol 0.830689
18:  |F(x)| = 170.871; step 0.167445; tol 0.822458
19:  |F(x)| = 145.236; step 1; tol 0.650218
20:  |F(x)| = 95.3081; step 1; tol 0.387571
21:  |F(x)| = 43.572; step 1; tol 0.188104
22:  |F(x)| = 36.1231; step 0.269139; tol 0.618582
23:  |F(x)| = 21.8998; step 1; tol 0.34438
24:  |F(x)| = 19.9895; step 0.139459; tol 0.749839
25:  |F(x)| = 19.9724; step 0.00474796; tol 0.898454
26:  |F(x)| = 17.4563; step 1; tol 0.726497
27:  |F(x)| = 14.1098; step 1; tol 0.588002
28:  |F(x)| = 7.53609; step 1; tol 0.311172
29:  |F(x)| = 1.31187; step 1; tol 0.027273
30:  |F(x)| = 0.0283652; step 1; tol 0.000420758
31:  |F(x)| = 9.136e-05; step 1; tol 9.33644e-06
32:  |F(x)| = 2.04046e-07; step 1; tol 4.48937e-06
Done!
```

Here we reproduce Figure 1 from Reference 1.

```
In [6]: %%opts Overlay [legend_position='top_right']

        extents = (16,0.0,24,4.0)

        gr_plots = []
        for alpha,x,y in gr_results:
            label = 'alpha={} (pyPRISM)'.format(alpha)
            style = {'line_dash':ls[alpha],'color':colors[alpha]}
            c1 = hv.Curve((x,y),label=label,extents=extents)(style=style)
            gr_plots.append(c1)

        for alpha,x,y in gr_compare:
            label = 'alpha={} (Ref 1)'.format(alpha)
            style = {'marker':markers[alpha],'color':colors[alpha]}
            c1 = hv.Scatter((x,y),label=label,extents=extents)(style=style)
            gr_plots.append(c1)


        hv.Overlay(gr_plots).redim.label(x='r',y='g(r)')
```

```
Out[6]: :Overlay
           .Curve.Alpha_equals_0_full_stop_25_left_parenthesis_pyPRISM_right_parenthesis :Curve   [x
           .Curve.Alpha_equals_0_full_stop_5_left_parenthesis_pyPRISM_right_parenthesis  :Curve   [x
           .Curve.Alpha_equals_1_full_stop_0_left_parenthesis_pyPRISM_right_parenthesis  :Curve   [x
           .Scatter.Alpha_equals_1_full_stop_0_left_parenthesis_Ref_1_right_parenthesis  :Scatter
           .Scatter.Alpha_equals_0_full_stop_5_left_parenthesis_Ref_1_right_parenthesis  :Scatter
           .Scatter.Alpha_equals_0_full_stop_25_left_parenthesis_Ref_1_right_parenthesis :Scatter
```

Polymer Grafted Particle Nanocomposite



In this example, we use `pyPRISM` to study a polymer nanocomposite system in which polymer chains of varying length $N_{matrix} = 10, 60$ are mixed with a nanoparticle permanently grafted with polymer chains. The grafted chains are of length $N_{graft} = 20$ and are of either linear or comb polymer architecture with side chain length $N_{sc} = 3$. All interactions in this system are modeled as athermal (hard-sphere). We plot the particle-particle potential of mean force as a function of the length of the matrix chains and the grafted chain architecture. The details of this system are described in Ref. 5.

As always, we define the aesthetics first

```
In [7]: %opts Curve Scatter [width=500,height=400] Layout [shared_axes=False] Scatter (size=10,alpha=
        %opts Curve Scatter [fontsize={'xlabel':14,'xlabel':14,'ylabel':14,'ticks':12}]
        %opts Overlay [legend_position='bottom_left']
        %opts Layout [shared_axes=False]


        colors = {}
        colors['linear',10] = 'blue'
        colors['linear',60] = 'red'
        colors['comb',10] = 'green'
        colors['comb',60] = 'magenta'

        ls = {}
        ls['linear',10] = 'solid'
        ls['linear',60] = 'dashed'
        ls['comb',10] = 'dotted'
        ls['comb',60] = 'dashdot'


        markers = {}
        markers['linear',10] = 'o'
        markers['linear',60] = '^'
        markers['comb',10] = 'd'
        markers['comb',60] = 's'
```

Next, we load the data from the literature.

```
In [8]: PMF_compare = []
        for arch in ['linear','comb']:
            for Nmatrix in [10,60]:
                fname = '../data/GraftedComposite-Gr-{}-Nmatrix{}-PP.dat'.format(arch,Nmatrix)
                x,y = np.loadtxt(fname).T
                y = -np.log(y)
                PMF_compare.append([arch,Nmatrix,x,y])
```

```
/home/tbm/software/anaconda3/4.4.0/envs/pyPRISM_py3_dev/lib/python3.5/site-packages/ipykernel_launche
```

Finally, we conduct the calculation. Unlike the bare-composite case, we can directly solve for the results but we still use loops to quickly scan a small parameter space and gather the results.

Note that this calculation is considerably more complicated than the previous if only for the fact that we have three components for the first time. This results in a significantly longer script for the calculation.

```
In [9]: import pyPRISM
        from pyPRISM.calculate.pair_correlation import pair_correlation
        from pyPRISM.calculate.pmf import pmf
        import numpy as np

        D = 5.0 #grafted nanoparticle diameter
        d = 1.0 #polymer segement diameter
        phi = 0.001 #grafted particle volume fraction
        eta = 0.35 #total occupied volume fraction
        vd = 4.0/3.0 * np.pi * (d/2.0)**(3) #volume of one polymer segement
        vD = 4.0/3.0 * np.pi * (D/2.0)**(3) #volume of one nanoparticle

        sys = pyPRISM.System(['particle','graft','matrix'],kT=1.0)
        sys.domain = pyPRISM.Domain(dr=0.1,length=1024)

        sys.diameter['particle'] = D
        sys.diameter['graft']    = d
        sys.diameter['matrix']   = d

        sys.omega['particle','matrix']  = pyPRISM.omega.InterMolecular()
        sys.omega['graft','matrix']     = pyPRISM.omega.InterMolecular()
        sys.omega['particle','particle'] = pyPRISM.omega.SingleSite()

        sys.closure[sys.types,sys.types]   = pyPRISM.closure.PercusYevick()
        sys.closure['particle','particle'] = pyPRISM.closure.HyperNettedChain()

        sys.potential[sys.types,sys.types]   = pyPRISM.potential.HardSphere()
        sys.potential['particle',sys.types]  = pyPRISM.potential.HardSphere()
        sys.potential['particle','particle'] = pyPRISM.potential.HardSphere()

        guess = np.zeros((sys.domain.length,sys.rank,sys.rank))
        PMF_results = []
        for Nmatrix in [60,10]:
            sys.omega['matrix','matrix']   = pyPRISM.omega.FromFile('../data/GraftedComposite-Omega-m
            sys.omega['graft','graft']     = pyPRISM.omega.FromFile('../data/GraftedComposite-Omega-l
            sys.omega['graft','particle']  = pyPRISM.omega.FromFile('../data/GraftedComposite-Omega-l

            numGraftBeads = 500
            vPGP = vD + numGraftBeads*vd
            sys.density['matrix']   = (1-phi)*eta/vd
            sys.density['particle'] = phi*eta/vPGP
            sys.density['graft']    = numGraftBeads * phi*eta/vPGP
            print('--> rho=',sys.density['graft'],sys.density['particle'],sys.density['matrix'])
            PRISM = sys.createPRISM()
            PRISM.solve(guess)
            guess = np.copy(PRISM.x)
            PMF_results.append(['linear',Nmatrix,sys.domain.r,pmf(PRISM)['particle','particle']])
            numGraftBeads = 1250
            vPGP = vD + numGraftBeads*vd
            sys.density['matrix']  = (1-phi)*eta/vd
            sys.density['particle'] = phi*eta/vPGP
            sys.density['graft']    = numGraftBeads * phi*eta/vPGP
            print('--> rho=',sys.density['graft'],sys.density['particle'],sys.density['matrix'])
            sys.omega['matrix','matrix']   = pyPRISM.omega.FromFile('../data/GraftedComposite-Omega-m
            sys.omega['graft','graft']     = pyPRISM.omega.FromFile('../data/GraftedComposite-Omega-g
            sys.omega['graft','particle']  = pyPRISM.omega.FromFile('../data/GraftedComposite-Omega-g
            PRISM = sys.createPRISM()
            result = PRISM.solve(guess)
            PMF_results.append(['comb',Nmatrix,sys.domain.r,pmf(PRISM)['particle','particle']])

        print('Done!')
```

```
--> rho= 0.0005347606087887683 1.0695212175775368e-06 0.6677823102249745
0:  |F(x)| = 125.67; step 0.0222735; tol 0.853672
1:  |F(x)| = 110.033; step 1; tol 0.689955
2:  |F(x)| = 98.7161; step 0.360292; tol 0.724395
3:  |F(x)| = 96.916; step 0.0346837; tol 0.867475
4:  |F(x)| = 77.7223; step 1; tol 0.677262
5:  |F(x)| = 52.3436; step 1; tol 0.412816
6:  |F(x)| = 52.2497; step 0.00201374; tol 0.896772
7:  |F(x)| = 45.2768; step 1; tol 0.723781
8:  |F(x)| = 42.7455; step 0.0847881; tol 0.80218
9:  |F(x)| = 37.6435; step 0.200722; tol 0.697978
10:  |F(x)| = 32.0038; step 1; tol 0.650526
11:  |F(x)| = 25.3211; step 0.309525; tol 0.563386
12:  |F(x)| = 21.2678; step 0.353001; tol 0.634926
13:  |F(x)| = 17.046; step 0.420115; tol 0.578152
14:  |F(x)| = 10.0361; step 1; tol 0.311978
15:  |F(x)| = 9.52045; step 0.0566309; tol 0.809896
16:  |F(x)| = 6.99251; step 1; tol 0.590338
17:  |F(x)| = 3.87908; step 1; tol 0.313649
18:  |F(x)| = 3.59957; step 0.091973; tol 0.774974
19:  |F(x)| = 3.16073; step 0.265407; tol 0.693929
20:  |F(x)| = 2.73033; step 1; tol 0.67158
21:  |F(x)| = 1.39749; step 1; tol 0.405918
22:  |F(x)| = 0.815644; step 1; tol 0.306583
23:  |F(x)| = 0.191751; step 1; tol 0.0497412
24:  |F(x)| = 0.138158; step 1; tol 0.467215
25:  |F(x)| = 0.0047945; step 1; tol 0.196461
26:  |F(x)| = 0.000179072; step 1; tol 0.00125548
27:  |F(x)| = 2.33742e-06; step 1; tol 0.000153342
--> rho= 0.0006076825099872368 4.861460079897895e-07 0.6677823102249745
0:  |F(x)| = 47.2842; step 1; tol 0.255559
1:  |F(x)| = 6.39546; step 1; tol 0.0164647
2:  |F(x)| = 0.600451; step 1; tol 0.00793331
3:  |F(x)| = 0.0275196; step 1; tol 0.00189048
4:  |F(x)| = 0.00369618; step 1; tol 0.0162354
5:  |F(x)| = 3.65937e-06; step 1; tol 8.82164e-07
--> rho= 0.0005347606087887683 1.0695212175775368e-06 0.6677823102249745
0:  |F(x)| = 24.4749; step 1; tol 0.00231931
1:  |F(x)| = 0.34516; step 1; tol 0.000178995
2:  |F(x)| = 0.202216; step 1; tol 0.308911
3:  |F(x)| = 0.0112913; step 1; tol 0.00280606
4:  |F(x)| = 5.59149e-05; step 1; tol 2.20705e-05
--> rho= 0.0006076825099872368 4.861460079897895e-07 0.6677823102249745
```

```
0:  |F(x)| = 39.6577; step 1; tol 0.0980329
1:  |F(x)| = 1.07302; step 1; tol 0.000658876
2:  |F(x)| = 0.609651; step 1; tol 0.290528
3:  |F(x)| = 0.0416829; step 1; tol 0.00420723
4:  |F(x)| = 0.0108212; step 1; tol 0.0606569
5:  |F(x)| = 0.00213452; step 1; tol 0.0350178
6:  |F(x)| = 2.32349e-05; step 1; tol 0.000106641
Done!
```

Finally, plotting the results

```
In [10]: %%opts Overlay [legend_position='top_right']
         from math import sqrt

         extents = (10,-0.2,40,0.2)
         gr_plots = []
         for arch,Nmatrix,x,y in PMF_results:
             key = (arch,Nmatrix)
             label = 'alpha={} (pyPRISM)'.format(key)
             style = {'line_dash':ls[key],'color':colors[key]}
             c1 = hv.Curve((x,y),label=label,extents=extents)(style=style)
             gr_plots.append(c1)

         step = 10
         for arch,Nmatrix,x,y in PMF_compare:
             x = x[::step]
             y = y[::step]
             key = (arch,Nmatrix)
             label = 'alpha={} (Ref 1)'.format(key)
             style = {'marker':markers[key],'color':colors[key]}
             c1 = hv.Scatter((x,y),label=label,extents=extents)(style=style)
             gr_plots.append(c1)


         hv.Overlay(gr_plots).redim.label(x='r',y='W(r)')
```

```
Out[10]: :Overlay
            .Curve.Alpha_equals_left_parenthesis_apostrophe_linear_apostrophe_comma_60_right_parenthe
            .Curve.Alpha_equals_left_parenthesis_apostrophe_comb_apostrophe_comma_60_right_parenthes
            .Curve.Alpha_equals_left_parenthesis_apostrophe_linear_apostrophe_comma_10_right_parenthe
            .Curve.Alpha_equals_left_parenthesis_apostrophe_comb_apostrophe_comma_10_right_parenthes
            .Scatter.Alpha_equals_left_parenthesis_apostrophe_linear_apostrophe_comma_10_right_parent
            .Scatter.Alpha_equals_left_parenthesis_apostrophe_linear_apostrophe_comma_60_right_parent
            .Scatter.Alpha_equals_left_parenthesis_apostrophe_comb_apostrophe_comma_10_right_parenthe
            .Scatter.Alpha_equals_left_parenthesis_apostrophe_comb_apostrophe_comma_60_right_parenthe
```

Summary

This notebook has covered two examples of nanocomposite systems which are significantly more complex than examples in previous notebooks. We also discuss the strategies we use to converge the PRISM equations for these systems, as we cannot directly solve for the answer.

NB0.Introduction · NB1.PythonBasics · NB2.Theory.General · NB3.Theory.PRISM · NB4.pyPRISM.Overview · NB5.CaseStudies.PolymerMelts · NB6.CaseStudies.Nanocomposites · NB7.CaseStudies.Copolymers · NB8.pyPRISM.Internals · NB9.pyPRISM.Advanced

Case Studies: Copolymers

There are many challenges to studying copolymer systems with PRISM with the primary challenge being that PRISM cannot describe the phase-separated state, which is often of primary interest. Furthermore, due to the vast array of chain architectures, compositions, sequences, and chemistries that can be synthetically incorporated into block copolymers, analytical $\hat{\omega}(k)$ are unlikely to be available for every system of interest. Despite this, many authors have used PRISM to study the disorded (mixed) state and to probe details about the spinodal phase transition.

Concepts

- Multicomponent PRISM
- Heterogeneous interactions
- Complex Molecular Structure
- Simulation derived $\hat{\omega}(k)$

Tools

- pyPRISM.calculate.structure_factor

References

1. Ivan Lyubimov, Daniel J. Beltran-Villegas, and Arthi Jayaraman; PRISM Theory Study of Amphiphilic Block Copolymer Solutions with Varying Copolymer Sequence and Composition, Macromolecules, 2017, 50 (18), 7419–7431 DOI: 10.1021/acs.macromol.7b01419

Notebook Setup

To begin, please run `Kernel-> Restart & Clear Output` from the menu at the top of the notebook. It is a good idea to run this before starting any notebook so that the notebook is fresh for the user. Next, run the cell below (via the top menu-bar or `<Shift-Enter>`. If the cell throws an import error, there is likely something wrong with your environment.

If successful, you should see a set of logos appear below the cell. Which logos appear depend on what is inside the `hv.extension()` command at the bottom of the cell. If no logos appear and the cell throws an error, there is likely something wrong with your environment.

Troubleshooting:

- Did you activate the correct conda environment before starting the jupyter notebook?
- If not using anaconda, did you install all dependencies before starting the jupyter notebook?
- Is pyPRISM installed in your current environment on your `PYTHONPATH`?

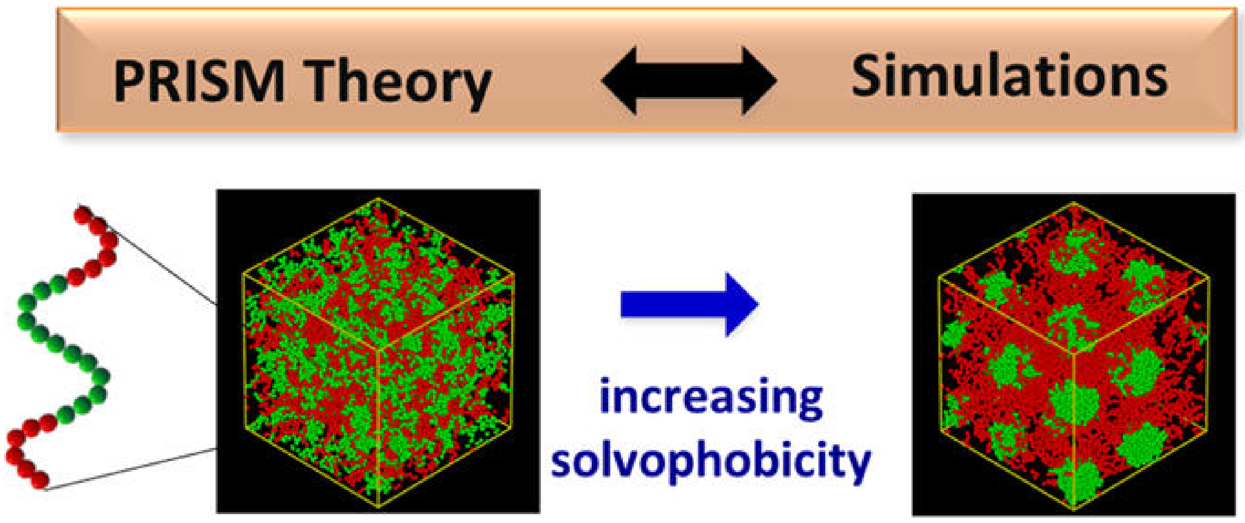
Holoviews + Bokeh Logos:

```
In [1]: import pyPRISM
        import numpy as np
        import holoviews as hv
        hv.extension('bokeh')
```

Data type cannot be displayed: application/javascript, application/vnd.bokehjs_load.v0+json

```
<IPython.core.display.HTML object>
```

Amphiphilic Block Copolymer Solution



In this example, we use `pyPRISM` to predict the structure of amphiphilic block copolymer solutions. In this case, we show the impact of block copolymer sequence on self-assembly, in which the A block is hydrophilic and the B block is hydrophobic. The block sequences shown are A-B diblock (case 1), A-B-A triblock (case 2), and B-A-B triblock (case 2i), in which the fraction of A-block (f_A) is always 0.5. The system and comparison data are taken from reference 1.

Starting with the aesthetic definitions

```
In [2]: %opts Curve Scatter [width=500,height=400] Layout [shared_axes=False] Scatter (size=10,alpha=
        %opts Curve Scatter [fontsize={'xlabel':14,'legend':14,'ylabel':14,'ticks':14}]
        %opts Overlay [legend_position='bottom_left']
        %opts Layout [shared_axes=False]


        colors = {}
        colors['1'] = 'blue'
        colors['2'] = 'red'
        colors['2i'] = 'orange'

        ls = {}
        ls['1'] = 'solid'
        ls['2'] = 'solid'
        ls['2i'] = 'solid'

        lsc = {}
        lsc['1'] = 'dotted'
        lsc['2'] = 'dotted'
        lsc['2i'] = 'dotted'
        markers = {}
        markers['1'] = 'o'
        markers['2'] = '^'
        markers['2i'] = 'd'
```

Next we'll load in the data extracted from Ref. 1 that we'll be comparing against.

```
In [3]: skAA_compare = {}
        skBB_compare = {}
        for case in ['1','2','2i']:

            fname = '../data/BCPSolution-Sk-AA-Case{}.dat'.format(case)
            dataAA = np.loadtxt(fname)
            skAA_compare[case] = dataAA[:,1]
            fname = '../data/BCPSolution-Sk-BB-Case{}.dat'.format(case)
            dataBB = np.loadtxt(fname)
            skBB_compare[case] = dataBB[:,1]

        skAA_compare['k'] = dataAA[:,0]
        skBB_compare['k'] = dataBB[:,0]
```

Next, we'll use typyPRISM to calculate structural data for the three block copolymer sequences. We chose to solve in the order of case 2, case 2i, then case 1 because case 2 more easily converged to a solution from a naive initial guess.

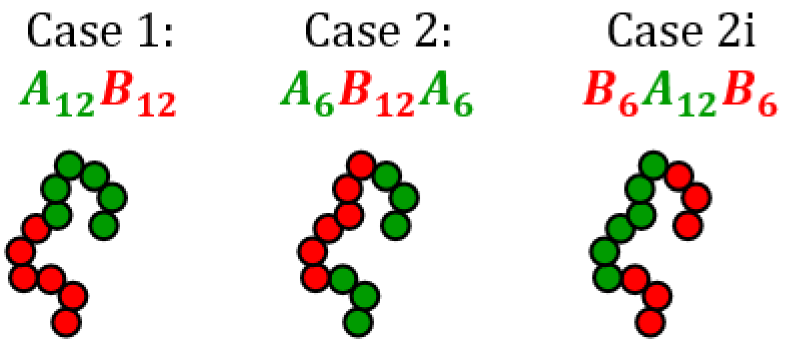
```
In [4]:
        d = 1.0                                #diameter of beads
        vd = 4.0/3.0 * np.pi * (d/2.0)**(3)   #volume of beads
        f_A = 0.5                              #volume fraction of A block
        eta = 0.1                              #total occupied volume fraction
        site_density = eta/vd
        epsilon = 1.0                          #WCA epsilon for A-A and A-B interactions
        epsilon_BB = 0.25                      #LJ epsilon for B-B interactions

        sys = pyPRISM.System(['A','B'],kT=1.0)
        sys.domain = pyPRISM.Domain(dr=0.1,length=1024)

        sys.density['A'] = f_A*site_density
        sys.density['B'] = (1.0-f_A)*site_density
        sys.diameter['A'] = d
        sys.diameter['B'] = d
        print('--> rho=',sys.density['A'],sys.density['B'])

        sys.closure['A','A'] = pyPRISM.closure.PercusYevick()
        sys.closure['A','B'] = pyPRISM.closure.PercusYevick()
        sys.closure['B','B'] = pyPRISM.closure.PercusYevick()

        sys.potential['A','A'] = pyPRISM.potential.WeeksChandlerAndersen(epsilon)
        sys.potential['A','B'] = pyPRISM.potential.WeeksChandlerAndersen(epsilon)
        sys.potential['B','B'] = pyPRISM.potential.LennardJones(epsilon_BB)

        skAA_results = {}
        skBB_results = {}
        guess = np.zeros(sys.rank*sys.rank*sys.domain.length)
        for case in ['2','2i','1']:
            print('==> Solving for case {}'.format(case))

            fname = '../data/BCPSolution-Omega-AA-Case{}.dat'.format(case)
            print("Using omega "+fname)
            sys.omega['A','A'] = pyPRISM.omega.FromFile(fname)
            fname = '../data/BCPSolution-Omega-AB-Case{}.dat'.format(case)
            print("Using omega "+fname)
            sys.omega['A','B'] = pyPRISM.omega.FromFile(fname)
            fname = '../data/BCPSolution-Omega-BB-Case{}.dat'.format(case)
            print("Using omega "+fname)
            sys.omega['B','B'] = pyPRISM.omega.FromFile(fname)

            PRISM = sys.createPRISM()
            result = PRISM.solve(guess)
            guess = np.copy(PRISM.x)

            SAA = pyPRISM.calculate.structure_factor(PRISM)['A','A']
            SBB = pyPRISM.calculate.structure_factor(PRISM)['B','B']
            skAA_results[case] = SAA
            skBB_results[case] = SBB

        skAA_results['k'] = sys.domain.k
        skBB_results['k'] = sys.domain.k

        print('Done!')
```

```
--> rho= 0.09549296585513722 0.09549296585513722
==> Solving for case 2
Using omega ../data/BCPSolution-Omega-AA-Case2.dat
Using omega ../data/BCPSolution-Omega-AB-Case2.dat
Using omega ../data/BCPSolution-Omega-BB-Case2.dat
0:  |F(x)| = 6.1997; step 1; tol 0.220434
1:  |F(x)| = 1.41862; step 1; tol 0.0471231
2:  |F(x)| = 0.0989687; step 1; tol 0.00438031
3:  |F(x)| = 0.000851413; step 1; tol 6.66082e-05
4:  |F(x)| = 5.3716e-08; step 1; tol 3.58236e-09
==> Solving for case 2i
Using omega ../data/BCPSolution-Omega-AA-Case2i.dat
Using omega ../data/BCPSolution-Omega-AB-Case2i.dat
Using omega ../data/BCPSolution-Omega-BB-Case2i.dat
0:  |F(x)| = 0.0578709; step 1; tol 0.000604868
1:  |F(x)| = 0.000226768; step 1; tol 1.38193e-05
2:  |F(x)| = 3.28947e-09; step 1; tol 1.89379e-10
==> Solving for case 1
Using omega ../data/BCPSolution-Omega-AA-Case1.dat
Using omega ../data/BCPSolution-Omega-AB-Case1.dat
Using omega ../data/BCPSolution-Omega-BB-Case1.dat
0:  |F(x)| = 0.109135; step 1; tol 0.00308876
1:  |F(x)| = 0.000506968; step 1; tol 1.9421e-05
2:  |F(x)| = 1.28353e-08; step 1; tol 5.76892e-10
Done!
```

We start by reproducing Figure 5a of Reference 1.

```
In [5]: %%opts Overlay [legend_position='top_right']

        extents = (-0.25,0,8.75,7.25)

        sk_plots = []
        for ii,case in enumerate(['1','2','2i']):
            label = 'Case{} (pyPRISM)'.format(case)
            style = {'line_dash':ls[case],'color':colors[case]}
            x = np.array(skAA_results['k'])
            y = np.array(skAA_results[case])
            c1 = hv.Curve((x,y),label=label,extents=extents)(style=style)
            sk_plots.append(c1)
            label = 'Case{} (Ref 1)'.format(case)
            style = {'marker':markers[case],'color':colors[case]}
            x = np.array(skAA_compare['k'])[::5]
            y = np.array(skAA_compare[case])[::5]
            c2 = hv.Scatter((x,y),label=label,extents=extents)(style=style)
            sk_plots.append(c2)

        hv.Overlay(sk_plots).redim.label(x='kd',y='S(k)')
```

```
Out[5]: :Overlay
           .Curve.Case1_left_parenthesis_pyPRISM_right_parenthesis  :Curve   [x]   (y)
           .Scatter.Case1_left_parenthesis_Ref_1_right_parenthesis  :Scatter   [x]   (y)
           .Curve.Case2_left_parenthesis_pyPRISM_right_parenthesis  :Curve   [x]   (y)
           .Scatter.Case2_left_parenthesis_Ref_1_right_parenthesis  :Scatter   [x]   (y)
           .Curve.Case2i_left_parenthesis_pyPRISM_right_parenthesis :Curve   [x]   (y)
           .Scatter.Case2i_left_parenthesis_Ref_1_right_parenthesis :Scatter   [x]   (y)
```

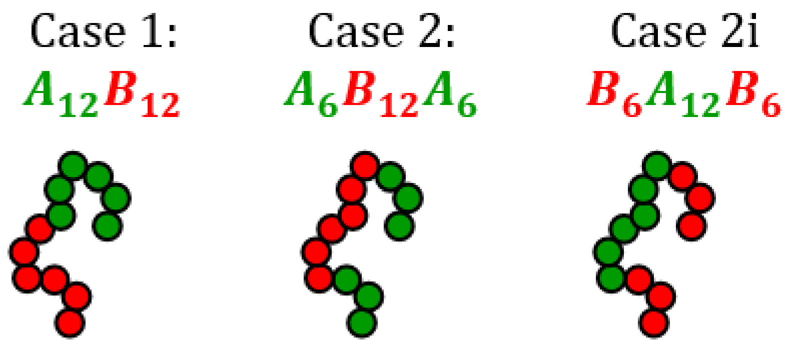


Next, we reproduce Figure 5b:

```
In [6]: %%opts Overlay [legend_position='top_right']

        extents = (-0.25,0,8.75,7.25)

        sk_plots = []
        for ii,case in enumerate(['1','2','2i']):
            label = 'Case{} (pyPRISM)'.format(case)
            style = {'line_dash':ls[case],'color':colors[case]}
            x = np.array(skBB_results['k'])
            y = np.array(skBB_results[case])
            c1 = hv.Curve((x,y),label=label,extents=extents)(style=style)
            sk_plots.append(c1)
            label = 'Case{} (Ref 1)'.format(case)
            style = {'marker':markers[case],'color':colors[case]}
            x = np.array(skBB_compare['k'])[::5]
            y = np.array(skBB_compare[case])[::5]
            c2 = hv.Scatter((x,y),label=label,extents=extents)(style=style)
            sk_plots.append(c2)

        hv.Overlay(sk_plots).redim.label(x='kd',y='S(k)')
```

```
Out[6]: :Overlay
```

```
.Curve.Case1_left_parenthesis_pyPRISM_right_parenthesis  :Curve   [x]   (y)
.Scatter.Case1_left_parenthesis_Ref_1_right_parenthesis  :Scatter   [x]   (y)
.Curve.Case2_left_parenthesis_pyPRISM_right_parenthesis  :Curve   [x]   (y)
.Scatter.Case2_left_parenthesis_Ref_1_right_parenthesis  :Scatter   [x]   (y)
.Curve.Case2i_left_parenthesis_pyPRISM_right_parenthesis :Curve   [x]   (y)
.Scatter.Case2i_left_parenthesis_Ref_1_right_parenthesis :Scatter   [x]   (y)
```



NB0.Introduction · NB1.PythonBasics · NB2.Theory.General · NB3.Theory.PRISM · NB4.pyPRISM.Overview · NB5.CaseStudies.PolymerMelts · NB6.CaseStudies.Nanocomposites · NB7.CaseStudies.Copolymers · NB8.pyPRISM.Internals · NB9.pyPRISM.Advanced

pyPRISM Internals

Here, we highlight some implementation details of the pyPRISM. This notebook is only necessary for those wishing to extend pyPRISM or better understand how the code functions. As such, this notebook will assume a higher familiarity Python and code development than the others.

Notebook Setup

To begin, please run `Kernel-> Restart & Clear Output` from the menu at the top of the notebook. It is a good idea to run this before starting any notebook so that the notebook is fresh for the user. Next, run the cell below (via the top menu-bar or `<Shift-Enter>`. If the cell throws an import error, there is likely something wrong with your environment.

If successful, you should see a set of logos appear below the cell. Which logos appear depend on what is inside the `hv.extension()` command at the bottom of the cell. If no logos appear and the cell throws an error, there is likely something wrong with your environment.

Troubleshooting:

- Did you activate the correct conda environment before starting the jupyter notebook?
- If not using anaconda, did you install all dependencies before starting the jupyter notebook?
- Is pyPRISM installed in your current environment on your `PYTHONPATH`?

Holoviews + Bokeh Logos:

```
In [1]: import numpy as np
        import pyPRISM
        import holoviews as hv
        hv.extension('bokeh')
```

Data type cannot be displayed: application/javascript, application/vnd.bokehjs_load.v0+json

```
<IPython.core.display.HTML object>
```

MatrixArray Data Structure

One of the challenges of using PRISM theory is the nature of the data itself: for every space-point in the solution domain, the user must store a $n \times n$ matrix of correlation values. The theory requires that all of these matrices be used in mathematical operations e.g., matrix multiplcation and matrix inversion. Furthermore, as these operations are being carried out on many (>1024) matrices, these operations need to be optimized. Complicating this further, in order to transfer between Fourier- and Real-space, the correlation values across matrices must be extracted as correlation curves. The schematic above describes the primary data structure that is used to carry out these tasks in pyPRISM: the `MatrixArray`.

Below, we create a 3x3 matrix array with 1024 matrices. MatrixArrays are type-aware so we also provide the site types. For this example, we will consider a polymer blend in a solvent. Finally, MatrixArrays keep track of whether they represent Real- or Fourier- space data, so we specify the "space" of the array.

```
In [2]: MA1 = pyPRISM.MatrixArray(length=1024,rank=3,types=['polymer1','polymer2','solvent'],space=py
        MA1

Out[2]: <MatrixArray rank:3 length:1024>
```

We can view the underlying `Numpy` array that stores the data and see that it's intialized to all zeros by default. Note that you **should not** access this array directly and instead should rely on the defined methods as shown below.

```
In [3]: MA1.data

Out[3]: array([[[0., 0., 0.],
                [0., 0., 0.],
                [0., 0., 0.]],

               [[0., 0., 0.],
                [0., 0., 0.],
                [0., 0., 0.]],

               [[0., 0., 0.],
                [0., 0., 0.],
                [0., 0., 0.]],

               ...,

               [[0., 0., 0.],
                [0., 0., 0.],
                [0., 0., 0.]],
               [[0., 0., 0.],
                [0., 0., 0.],
                [0., 0., 0.]],

               [[0., 0., 0.],
                [0., 0., 0.],
                [0., 0., 0.]]])
```

We can assign and extract curves from the `MatrixArray` using the square-bracket operators. We'll fill this MatrixArray with *intra*-molecular correlation functions. To do this, we'll need to employ a few other features from pyPRISM.

Note that matrix symmetry is automatically applied during assignment so assigning MA['A','B'] automatically assigns M['B','A'].

```
In [4]: domain = pyPRISM.Domain(length=1024,dr=0.25)

        MA1['polymer1','polymer1'] = pyPRISM.omega.Gaussian(length=1000,sigma=1.0).calculate(domain.k
        MA1['polymer2','polymer2'] = pyPRISM.omega.Gaussian(length=10,sigma=1.0).calculate(domain.k)
        MA1['solvent','solvent'] = pyPRISM.omega.SingleSite().calculate(domain.k)

        print('Full MatrixArray')
        print(MA1.data)
```

```
Full MatrixArray
[[[991.68567821   0.           0.        ]
  [  0.           9.99917183   0.        ]
  [  0.           0.           1.        ]]

 [[967.35716814   0.           0.        ]
  [  0.           9.99668766   0.        ]
  [  0.           0.           1.        ]]

 [[928.76831425   0.           0.        ]
  [  0.           9.9925496    0.        ]
  [  0.           0.           1.        ]]

 ...

 [[  1.           0.           0.        ]
  [  0.           1.           0.        ]
  [  0.           0.           1.        ]]

 [[  1.           0.           0.        ]
  [  0.           1.           0.        ]
  [  0.           0.           1.        ]]

 [[  1.           0.           0.        ]
  [  0.           1.           0.        ]
  [  0.           0.           1.        ]]]
```

pyPRISM also provides methods to loop over the curves so they can be operated on in sequence.

```
In [5]: for i,t,curve in MA1.itercurve():
            print('{}-{} Curve'.format(t[0],t[1]))
            print(curve,'\n')
```

```
polymer1-polymer1 Curve
[991.68567821 967.35716814 928.76831425 ...   1.           1.
   1.        ]
```

```
polymer1-polymer2 Curve
[0. 0. 0. ... 0. 0. 0.]

polymer1-solvent Curve
[0. 0. 0. ... 0. 0. 0.]

polymer2-polymer2 Curve
[9.99917183 9.99668766 9.9925496  ... 1.         1.         1.        ]

polymer2-solvent Curve
[0. 0. 0. ... 0. 0. 0.]

solvent-solvent Curve
[1. 1. 1. ... 1. 1. 1.]
```

We'll create another `MatrixArray` which will start out as an array of identity matrices before we modify the values.

```
In [6]: # This creates a matrix Array of identity matrices
        MA2 = pyPRISM.IdentityMatrixArray(length=1024,rank=3,types=['polymer1','polymer2','solvent'],

        print('Identity MatrixArray')
        print(MA2.data,'\n')

        MA2 *= 2.0 #multply all values by 2
        MA2['solvent','solvent'] = 1.2 # set the entire 'solvent-solvent' curve equal to 1.2
        MA2 += 3.0 # add three to all values

        print('Modified MatrixArray')
        print(MA2.data,'\n')
```

```
Identity MatrixArray
[[[1. 0. 0.]
  [0. 1. 0.]
  [0. 0. 1.]]

 [[1. 0. 0.]
  [0. 1. 0.]
  [0. 0. 1.]]

 [[1. 0. 0.]
  [0. 1. 0.]
  [0. 0. 1.]]

 ...

 [[1. 0. 0.]
  [0. 1. 0.]
  [0. 0. 1.]]

 [[1. 0. 0.]
  [0. 1. 0.]
  [0. 0. 1.]]

 [[1. 0. 0.]
  [0. 1. 0.]
  [0. 0. 1.]]]

Modified MatrixArray
[[[5.  3.  3. ]
```

```
  [3.  5.  3. ]
  [3.  3.  4.2]]

 [[5.  3.  3. ]
  [3.  5.  3. ]
  [3.  3.  4.2]]

 [[5.  3.  3. ]
  [3.  5.  3. ]
  [3.  3.  4.2]]

 ...

 [[5.  3.  3. ]
  [3.  5.  3. ]
  [3.  3.  4.2]]

 [[5.  3.  3. ]
  [3.  5.  3. ]
  [3.  3.  4.2]]

 [[5.  3.  3. ]
  [3.  5.  3. ]
  [3.  3.  4.2]]]
```

An important step in the PRISM solution process involves inverting the matrices at all space points. The `MatrixArray` has a reasonably fast vectorized version of this operation built in.

```
In [7]: result = MA1.invert()
        print('Inverted MatrixArray')
        print(result.data,'\n')
```

```
Inverted MatrixArray
[[[0.00100838 0.         0.        ]
  [0.         0.10000828 0.        ]
  [0.         0.         1.        ]]

 [[0.00103374 0.         0.        ]
  [0.         0.10003313 0.        ]
  [0.         0.         1.        ]]

 [[0.00107669 0.         0.        ]
  [0.         0.10007456 0.        ]
  [0.         0.         1.        ]]

 ...

 [[1.         0.         0.        ]
  [0.         1.         0.        ]
  [0.         0.         1.        ]]

 [[1.         0.         0.        ]
  [0.         1.         0.        ]
  [0.         0.         1.        ]]

 [[1.         0.         0.        ]
  [0.         1.         0.        ]
  [0.         0.         1.        ]]]
```

Finally, the `MatrixArray` provides a number of different operations between multiple arrays. Note that * represents *in-place* multiplication while the .dot() method is matrix mulitplication.

```
In [8]: result = MA1 * MA2
        print('In-place Multiplication')
        print(result.data,'\n')

        result = MA1.dot(MA2)
        print('Matrix Multiplication')
        print(result.data,'\n')
```

```
In-place Multiplication
[[[4.95842839e+03 0.00000000e+00 0.00000000e+00]
  [0.00000000e+00 4.99958591e+01 0.00000000e+00]
  [0.00000000e+00 0.00000000e+00 4.20000000e+00]]

 [[4.83678584e+03 0.00000000e+00 0.00000000e+00]
  [0.00000000e+00 4.99834383e+01 0.00000000e+00]
  [0.00000000e+00 0.00000000e+00 4.20000000e+00]]

 [[4.64384157e+03 0.00000000e+00 0.00000000e+00]
  [0.00000000e+00 4.99627480e+01 0.00000000e+00]
  [0.00000000e+00 0.00000000e+00 4.20000000e+00]]

 ...

 [[5.00000000e+00 0.00000000e+00 0.00000000e+00]
  [0.00000000e+00 5.00000000e+00 0.00000000e+00]
  [0.00000000e+00 0.00000000e+00 4.20000000e+00]]

 [[5.00000000e+00 0.00000000e+00 0.00000000e+00]
  [0.00000000e+00 5.00000000e+00 0.00000000e+00]
  [0.00000000e+00 0.00000000e+00 4.20000000e+00]]

 [[5.00000000e+00 0.00000000e+00 0.00000000e+00]
  [0.00000000e+00 5.00000000e+00 0.00000000e+00]
  [0.00000000e+00 0.00000000e+00 4.20000000e+00]]]

Matrix Multiplication
[[[4.95842839e+03 2.97505703e+03 2.97505703e+03]
  [2.99975155e+01 4.99958591e+01 2.99975155e+01]
  [3.00000000e+00 3.00000000e+00 4.20000000e+00]]

 [[4.83678584e+03 2.90207150e+03 2.90207150e+03]
  [2.99900630e+01 4.99834383e+01 2.99900630e+01]
  [3.00000000e+00 3.00000000e+00 4.20000000e+00]]

 [[4.64384157e+03 2.78630494e+03 2.78630494e+03]
  [2.99776488e+01 4.99627480e+01 2.99776488e+01]
  [3.00000000e+00 3.00000000e+00 4.20000000e+00]]

 ...

 [[5.00000000e+00 3.00000000e+00 3.00000000e+00]
  [3.00000000e+00 5.00000000e+00 3.00000000e+00]
  [3.00000000e+00 3.00000000e+00 4.20000000e+00]]

 [[5.00000000e+00 3.00000000e+00 3.00000000e+00]
  [3.00000000e+00 5.00000000e+00 3.00000000e+00]
  [3.00000000e+00 3.00000000e+00 4.20000000e+00]]
 [[5.00000000e+00 3.00000000e+00 3.00000000e+00]
  [3.00000000e+00 5.00000000e+00 3.00000000e+00]
  [3.00000000e+00 3.00000000e+00 4.20000000e+00]]]
```

The `MatrixArray` interacts with domain objects to tranform from Real- to Fourier- space and vice versa. Note that these operations occur *in-place* and this method does not return a new `MatrixArray`.

```
In [9]: print('Real-Space')
        print(MA1.data,'\n')
        domain.MatrixArray_to_fourier(MA1)
        print('Fourier-Space')
        print(MA1.data,'\n')
```

```
Real-Space
[[[991.68567821   0.           0.        ]
  [  0.           9.99917183   0.        ]
  [  0.           0.           1.        ]]

 [[967.35716814   0.           0.        ]
  [  0.           9.99668766   0.        ]
  [  0.           0.           1.        ]]

 [[928.76831425   0.           0.        ]
  [  0.           9.9925496    0.        ]
  [  0.           0.           1.        ]]

 ...

 [[  1.           0.           0.        ]
  [  0.           1.           0.        ]
  [  0.           0.           1.        ]]

 [[  1.           0.           0.        ]
  [  0.           1.           0.        ]
  [  0.           0.           1.        ]]

 [[  1.           0.           0.        ]
  [  0.           1.           0.        ]
  [  0.           0.           1.        ]]]

Fourier-Space
[[[ 2.46329939e+07  0.00000000e+00  0.00000000e+00]
  [ 0.00000000e+00  2.33022049e+07  0.00000000e+00]
  [ 0.00000000e+00  0.00000000e+00  2.13822840e+07]]

 [[-2.51675413e+06  0.00000000e+00  0.00000000e+00]
  [ 0.00000000e+00 -3.77481824e+06  0.00000000e+00]
  [ 0.00000000e+00  0.00000000e+00 -5.34036209e+06]]

 [[ 4.67049835e+06  0.00000000e+00  0.00000000e+00]
  [ 0.00000000e+00  3.51128971e+06  0.00000000e+00]
  [ 0.00000000e+00  0.00000000e+00  2.37581679e+06]]

 ...

 [[-1.63564440e+01  0.00000000e+00  0.00000000e+00]
  [ 0.00000000e+00 -3.19218659e+01  0.00000000e+00]
  [ 0.00000000e+00  0.00000000e+00 -3.20627732e+01]]
 [[ 4.77535180e+01  0.00000000e+00  0.00000000e+00]
  [ 0.00000000e+00  3.22033679e+01  0.00000000e+00]
  [ 0.00000000e+00  0.00000000e+00  3.20625988e+01]]

 [[-1.63244229e+01  0.00000000e+00  0.00000000e+00]
  [ 0.00000000e+00 -3.18593685e+01  0.00000000e+00]
  [ 0.00000000e+00  0.00000000e+00 -3.20000000e+01]]]
```

`MatrixArrays` also have safety checks build in to avoid operating across arrays of different spaces. The cell below should throw an error as MA1 is now in Fourier-space and MA2 is still in Real-space.

```
In [10]: MA1.dot(MA2)
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-10-dcdd31d919dc> in <module>()
----> 1 MA1.dot(MA2)

~/software/pyPRISM/dev/src/pyPRISM/core/MatrixArray.py in dot(self, other, inplace)
    315         '''
    316         if isinstance(other,MatrixArray):
--> 317             assert (self.space == other.space) or (Space.NonSpatial in (self.space,other.spac
    318         if inplace:
    319             self.data = np.einsum('lij,ljk->lik', self.data, other.data)

AssertionError: Attempting MatrixArray math in non-matching spaces
```

ValueTables and PairTables

While the `MatrixArray` is the workhorse of the PRISM mathematics, there are many times while setting up a calculation that a more flexible data structure is needed. In particular, it is often the case that data needs to be stored without mathematical features or the data is non-numerical in nature. pyPRISM provides two data structures towards this end: `ValueTable` and `PairTable`

To start we look at the `ValueTable`. One of the features of PRISM data and PRISM parameters is that they are almost always associated with sites. The `ValueTable` provides an efficient way to access and manipulate data via site-types. See the examples below for an overview:

```
In [11]: VT = pyPRISM.ValueTable(types=['A','B','C','D','E'],name='density')

         # set the value for type A to be 0.25
         VT['A'] = 0.25

         # set the value for types B & C to be 0.35
         VT[ ['B','C'] ] = 0.35

         # set all other values to be 0.1
         VT.setUnset(0.1)

         for i,t,v in VT:
             print('{}) {} for type {} is {}'.format(i,VT.name,t,v))
```

```
0) density for type A is 0.25
1) density for type B is 0.35
2) density for type C is 0.35
3) density for type D is 0.1
4) density for type E is 0.1
```

Alternatively, there are many data in PRISM that are identified by a pair of site types. This is what the `PairTable` was designed to handle. Note that in this example, we are storing strings rather than numbers.

```
In [12]: PT = pyPRISM.PairTable(['A','B','C'],name='potential')

         # Set the 'A-A' pair
         PT['A','A']            = 'Lennard-Jones'

         # Set the 'B-A', 'A-B', 'B-B', 'B-C', and 'C-B' pairs
         PT['B',['A','B','C']] = 'Weeks-Chandler-Andersen'

         # Set the 'C-A', 'A-C', 'C-C' pairs
         PT['C',['A','C'] ]     = 'Exponential'


         #print only independent pairs
         print('Independent Pairs')
         for i,t,v in PT.iterpairs():
             print('{}) {} for pair {}-{} is {}'.format(i,VT.name,t[0],t[1],v))

         #print all pairs
         print('\n','All Pairs')
         for i,t,v in PT.iterpairs(full=True):
             print('{}) {} for pair {}-{} is {}'.format(i,VT.name,t[0],t[1],v))
```

```
Independent Pairs
(0, 0)) density for pair A-A is Lennard-Jones
(0, 1)) density for pair A-B is Weeks-Chandler-Andersen
(0, 2)) density for pair A-C is Exponential
(1, 1)) density for pair B-B is Weeks-Chandler-Andersen
(1, 2)) density for pair B-C is Weeks-Chandler-Andersen
(2, 2)) density for pair C-C is Exponential

 All Pairs
(0, 0)) density for pair A-A is Lennard-Jones
(0, 1)) density for pair A-B is Weeks-Chandler-Andersen
(0, 2)) density for pair A-C is Exponential
(1, 0)) density for pair B-A is Weeks-Chandler-Andersen
(1, 1)) density for pair B-B is Weeks-Chandler-Andersen
(1, 2)) density for pair B-C is Weeks-Chandler-Andersen
(2, 0)) density for pair C-A is Exponential
(2, 1)) density for pair C-B is Weeks-Chandler-Andersen
(2, 2)) density for pair C-C is Exponential
```

NB0.Introduction · NB1.PythonBasics · NB2.Theory.General · NB3.Theory.PRISM · NB4.pyPRISM.Overview · NB5.CaseStudies.PolymerMelts · NB6.CaseStudies.Nanocomposites · NB7.CaseStudies.Copolymers · NB8.pyPRISM.Internals · NB9.pyPRISM.Advanced

Advanced pyPRISM

While we have attempted to cover the basic usage of pyPRISM in this tutorial, there are inevitably use cases that are more complicated. Here we try to go over a few more advanced topics of pyPRISM. As with the previous notebook, we assume a slightly higher level of programming and a familiarity with pyPRISM here.

Notebook Setup

To begin, please run `Kernel-> Restart & Clear Output` from the menu at the top of the notebook. It is a good idea to run this before starting any notebook so that the notebook is fresh for the user. Next, run the cell below (via the top menu-bar or `<Shift-Enter>`. If the cell throws an import error, there is likely something wrong with your environment.

If successful, you should see a set of logos appear below the cell. Which logos appear depend on what is inside the `hv.extension()` command at the bottom of the cell. If no logos appear and the cell throws an error, there is likely something wrong with your environment.

Troubleshooting:

- Did you activate the correct conda environment before starting the jupyter notebook?
- If not using anaconda, did you install all dependencies before starting the jupyter notebook?
- Is pyPRISM installed in your current environment on your `PYTHONPATH`?

Holoviews + Bokeh Logos:

```
In [1]: import numpy as np
        import pyPRISM
        import holoviews as hv
        hv.extension('bokeh')
```

Data type cannot be displayed: application/javascript, application/vnd.bokehjs_load.v0+json

```
<IPython.core.display.HTML object>
```

Adding a new class or feature to pyPRISM

As users seek to use pyPRISM for new studies, they will likely need a potential, ω, or closure that is not currently implemented. Here we briefly cover how to create a new class for use in pyPRISM.

As an example, we will work on a new Potential: the square well potential. The only requirements of this class are that the class inherits from Potential and that it provides a calculate method which returns the potential magnitude given an array of positions to calculate at. If these are satistified, then this class can be used in pyPRISM! Note that this contract is the same for closures and ω as well.

We create the class with three attributes which describe the width and depth of the attractive square well along with the contact distance.

```
In [2]: from pyPRISM.potential.Potential import Potential
        import numpy as np

        class SquareWell(Potential):
            def __init__(self,depth,width,sigma,high_value=1e6):
                r'''Constructor

                Attributes
                ----------
                depth: float
                depth of attractive well

            width: float
                width of attractive well. Note that this is
                the width *beyond* the contact distance (sigma)

            sigma: float
                contact distance

            high_value: float
                value of potential when overlapping
            '''
            self.sigma = sigma
            self.width = sigma + width
            self.depth = depth
            self.high_value = high_value
        def calculate(self,r):
            r'''Calculate potential values

            Attributes
            ----------
            r: float np.ndarray
                Array of pair distances at which to calculate potential values
            '''
            magnitude = np.zeros_like(r)
            magnitude[r<self.width] = -self.depth
            magnitude[r<self.sigma] = self.high_value
            return magnitude
```

Below we plot several versions of the potential with different parameters to verify our implementation.

```
In [3]: #holoviews parameters
        extents = (0,-5,6,5)
        plotParm = {'width':500,'height':400}
        styleParm = {}

        domain = pyPRISM.Domain(length=1024,dr=0.1)


        HS = pyPRISM.potential.HardSphere(sigma=1.0)
        U = HS.calculate(domain.r)

        curve = hv.Curve((domain.r,U),label='Hard Sphere',extents=extents)(style=styleParm,plot=plotP
        curves = [curve]
        for width in [1.5,3]:
            for depth in [2,4]:
                SW = SquareWell(depth=depth,width=width,sigma=1.0)
                U = SW.calculate(domain.r)

                label = 'width:{} depth:{}'.format(width,depth)
                curve = hv.Curve((domain.r,U),label=label,extents=extents)(style=styleParm,plot=plotP
                curves.append(curve)

        hv.Overlay(curves)
```

```
Out[3]: :Overlay
           .Curve.Hard_Sphere                              :Curve   [x]   (y)
           .Curve.Width_colon_1_full_stop_5_depth_colon_2 :Curve   [x]   (y)
           .Curve.Width_colon_1_full_stop_5_depth_colon_4 :Curve   [x]   (y)
           .Curve.Width_colon_3_depth_colon_2             :Curve   [x]   (y)
           .Curve.Width_colon_3_depth_colon_4             :Curve   [x]   (y)
```

Here we use the above defines potential for a simple PRISM calculation of a square well fluid and compare it to the hard sphere fluid.

```
In [4]: diameter = 1.0

        print('== Hard Sphere Calculation ==')
        sys = pyPRISM.System(['monomer'],kT=1.0)
        sys.domain = pyPRISM.Domain(dr=0.01,length=16384)
        sys.diameter['monomer'] = diameter
        sys.potential['monomer','monomer'] = pyPRISM.potential.HardSphere(sigma=diameter)
        sys.density['monomer'] = 0.9
        sys.omega['monomer','monomer'] = pyPRISM.omega.SingleSite()
        sys.closure['monomer','monomer'] = pyPRISM.closure.HyperNettedChain(apply_hard_core=True)
        PRISM = sys.createPRISM()
        PRISM.solve()
        guess = np.copy(PRISM.x)

        x = sys.domain.r
        y = pyPRISM.calculate.pair_correlation(PRISM)['monomer','monomer']
        rdf1 = hv.Curve((x,y),extents=(0,0,6,None),label='Hard Sphere').redim.label(x='r',y='g(r)')

        print('== Square Well Calculation ==')
        sys.potential['monomer','monomer'] = SquareWell(sigma=diameter,width=1.0,depth=0.3)
        PRISM = sys.createPRISM()
        PRISM.solve(guess=guess)
        x = sys.domain.r
        y = pyPRISM.calculate.pair_correlation(PRISM)['monomer','monomer']
        rdf2 = hv.Curve((x,y),extents=(0,0,6,None),label='Square Well').redim.label(x='r',y='g(r)')
```

```
== Hard Sphere Calculation ==
0:  |F(x)| = 2.84918; step 0.221669; tol 0.571933
1:  |F(x)| = 2.32298; step 0.306827; tol 0.598267
2:  |F(x)| = 1.93093; step 0.22343; tol 0.621845
3:  |F(x)| = 1.68305; step 0.17015; tol 0.683765
4:  |F(x)| = 1.45965; step 0.251084; tol 0.676929
5:  |F(x)| = 1.26112; step 0.158712; tol 0.671826
6:  |F(x)| = 1.25398; step 1; tol 0.889845
7:  |F(x)| = 1.07937; step 0.167843; tol 0.712642
8:  |F(x)| = 0.875799; step 0.343204; tol 0.592534
9:  |F(x)| = 0.864542; step 1; tol 0.877011
10:  |F(x)| = 0.811171; step 1; tol 0.792311
11:  |F(x)| = 0.784741; step 1; tol 0.842306
12:  |F(x)| = 0.782169; step 1; tol 0.894111
13:  |F(x)| = 0.781836; step 1; tol 0.899233
14:  |F(x)| = 0.781789; step 1; tol 0.899892
15:  |F(x)| = 0.781781; step 1; tol 0.899983
16:  |F(x)| = 0.595288; step 0.336799; tol 0.728972
17:  |F(x)| = 0.443791; step 0.49023; tol 0.500203
18:  |F(x)| = 0.269123; step 1; tol 0.330969
19:  |F(x)| = 0.0299694; step 1; tol 0.0111608
20:  |F(x)| = 5.33709e-05; step 1; tol 2.85427e-06
== Square Well Calculation ==
0:  |F(x)| = 34.2591; step 0.445084; tol 0.664124
1:  |F(x)| = 23.623; step 1; tol 0.427918
2:  |F(x)| = 19.4116; step 1; tol 0.607708
3:  |F(x)| = 4.31223; step 1; tol 0.332379
4:  |F(x)| = 3.53273; step 1; tol 0.60403
```

```
5:  |F(x)| = 3.53243; step 1; tol 0.89985
6:  |F(x)| = 2.86334; step 1; tol 0.728757
7:  |F(x)| = 2.84914; step 1; tol 0.891095
8:  |F(x)| = 2.84914; step 1; tol 0.9
9:  |F(x)| = 2.84914; step 1; tol 0.9
10:  |F(x)| = 2.84914; step 1; tol 0.9
11:  |F(x)| = 1.86771; step 1; tol 0.729
12:  |F(x)| = 0.380827; step 1; tol 0.478297
13:  |F(x)| = 0.113526; step 1; tol 0.205891
14:  |F(x)| = 0.00622344; step 1; tol 0.00270468
15:  |F(x)| = 1.42508e-05; step 1; tol 4.71913e-06
```

Plotting the Results

```
In [5]: hv.Overlay([rdf1,rdf2])(plot=plotParm)
```

```
Out[5]: :Overlay
           .Curve.Hard_Sphere :Curve   [x]   (y)
           .Curve.Square_Well :Curve   [x]   (y)
```

NB0.Introduction · NB1.PythonBasics · NB2.Theory.General · NB3.Theory.PRISM · NB4.pyPRISM.Overview · NB5.CaseStudies.PolymerMelts · NB6.CaseStudies.Nanocomposites · NB7.CaseStudies.Copolymers · NB8.pyPRISM.Internals · NB9.pyPRISM.Advanced

4.5 Frequently Asked Questions

4.5.1 What systems can be studied with PRISM?

- polymer melts/blends
- olefinic and non-olefinic polymers
- linear/branched/dendritic/sidechain polymers
- copolymer melts/blends
- polymer solutions
- nanoparticle solutions
- polymer nanocomposites
- liquid crystals (anisotropic formalism)
- micelle Solutions
- polyelectrolytes
- rod-like polymers
- flexible polymers
- ionomers
- ionic liquids

4.5.2 What thermodynamic and structural quantities can PRISM calculate?

- Second virial coefficients, B_2
- Flory effective interaction parameters, χ^{eff}
- Potentials of mean force
- Pair correlation functions (i.e. radial distribution functions)
- Partial structure factors
- Spinodal transition temperatures
- Equations of state
- Isothermal compressibilities

4.5.3 For what systems is PRISM theory not applicable for?

- macrophase-separated systems
- non-isotropic phases
- systems with strong nematic ordering (without anistropic formalism)
- calculating dynamic properties (e.g., diffusion coefficients, rheological properties)

4.5.4 What are the benefits of using PRISM over other simulation or theory methods?

- is orders of magnitude faster
- typically takes seconds to minutes to solve equations
- does not have finite size effects
- does not need to be equilibrated
- is mostly free of incompressibility assumptions

4.5.5 Why can't I import pyPRISM?

See *Troubleshooting*

4.5.6 Can pyPRISM be used outside of Jupyter?

Of course! pyPRISM is a Python module and can be used in any Python interface assuming that the dependencies are satisfied. The developers prefer Jupyter as a teaching environment, so the tutorial uses it. All of the code in the notebooks can be copied to a Python command line script.

4.5.7 Can pyPRISM handle anisotropic systems?

The current implementation of PRISM cannot handle anisotropic systems and will fail to converge or produce erroneous predictions if used for systems that are aligned. There is an anisotropic PRISM formalism that the developers are interested in implementing in the future.

4.5.8 How do I set up my specific system?

Please make sure you have looked at ALL case studies in the *Tutorial* and at least skimmed this documentation. We have attempted to provide a number of different examples and use-cases. If you think there is a deficiency in the documentation or tutorial, please file a bug report or submit a question via the Issues interface.

4.5.9 What do I do if the solver isn't converging?

There are a variety of reasons why the solver might seem "stuck", i.e. the function norm isn't decreasing. See *Convergence Tips*.

4.5.10 Why doesn't pyPRISM doesn't have the feature I need?

See *Contributing*

4.5.11 How to file a bug report, suggest a feature or ask a question about pyPRISM?

GitHub uses an Issue system to track communication between users and developers. The Issues tool has a flexible tagging feature which handles multiple types of posts including questions, feature requests, bug reports, and general discussion. Users should post to the issue system and the developers (or other users!) will respond as soon as they can.

4.6 Self-Consistent PRISM Method

One of the original weaknesses of PRISM theory was the lack of explicit coupling between intra- and inter-molecular correlations. This means that the global, inter-molecular structure of a system could not affect its intra-molecular structure. In the past, this limited PRISM calculations to systems where ideality assumptions could be invoked that effectively de-coupled the intra- and inter-molecular correlations. The Self-Consistent PRISM Method (SCPRISM) circumvents this limitation by combining PRISM theory and molecular simulation. Single-molecule simulations are used to calculate the intra-molecular correlation functions which are passed to a PRISM theory calculation. PRISM theory is then used to calculate an effective solvation potential (see `pyPRISM.calculate.solvation_potential()`) which is fed back to a new single-molecule simulation. This solvation potential modifies the pairwise interactions of the molecule to account for the effect of a medium (solvent or polymer matrix) which is not explicitly represented in the single-molecule simulation. This self-consistent cycle gives the intra- and inter-molecular correlations a feedback loop in which the two levels of correlations can affect one another.

pyPRISM is fully able to conduct SCPRISM studies in its current form. The developers are working on expanding the discussion of SCPRISM, adding a SCPRISM tutorial notebook, and creating helper utilities for SCPRISM calculations. These include helper functions which interface with common molecular simulation packages.

4.7 Convergence Tips

One of the most frustrating parts of working with PRISM numerically is that the numerical solver is often unable to converge to a solution. This is manifested as the cost function norm reported by the solver either not decreasing or fluctuating. The difficulty is in discerning why the solver is unable to converge. This can be related to the users choice of (perhaps unphysical) parameters or a poor choice of initial guess passed to the solver. Below are some suggestions on what to do if you cannot converge to a solution.

4.7.1 Give the solver a better initial guess

As is universally the case when using numerical methods, the better the "initial guess" the better the chance of the numerical method succeding in finding a minima. By default, pyPRISM uses an uninformed or "dumb" guess, and this is sufficient for many systems. We are currently working on developing other methods for creating improved "dumb" guesses.

As an alternative to using an uninformed guess, one can also use the solution from another PRISM calculation as the guess for their current one. This is generally done by finding a convergable system that is 'nearby' in phase space and then using the solution of that problem (PRISM.x) as the 'guess' to a new PRISM solution. This process can be iterated to achieve convergence in regions of phase space that are not directly convergable. This process is demonstrated in several of the examples in the *Tutorial*.

4.7.2 Change the closures used for some or all of your site pairs

Certain closures perform better under certain conditions, e.g., if attractive vs. repulsive interactions are used or if certain species have very asymmetric sizes or interaction strengths. Changing closures for some or all species may help. In the paper accompanying the pyPRISM tool, we discuss some suggestions for closures based on the system under study.

4.7.3 Consider your location in phase-space

Maybe the equations are not converging because you are not within the simple isotropic liquid region of your system. Try varying the interaction strength, density, composition, etc. Remember, PRISM cannot predict the structure of the phase separated state and PRISM may not predict the phase boundary where you think it should be.

4.7.4 Vary your domain spacing and domain size

There are numerical and physical reasons why the details of one's solution domain might affect the ability of the numerical solver to converge. Ensure that your domain spacing is small enough (in Real-space) to capture the smallest feature of interest. It is also important that you have enough points in your domain for the discrete sine transform to be correct. We recommend choosing a power of 2 that is (at-least) greater than or equal to length=1024. Keeping the number of points in your domain to be a power of two, as this choice improves the stability and efficiency of the Fourier transform.

4.7.5 Try a different numerical solver or solution scheme

While the default numerical solution method that pyPRISM uses is well tested and is most-often the best-choice for PRISM, there have been cases where alternative solvers converge to a solution more easily. See the Scipy docs for more information on available solvers. See the `pyPRISM.core.PRISM.PRISM.solve()` documentation for how to specify the solver.

4.8 Publications

Here we will attempt to keep an up-to-date listing of the papers which either use pyPRISM or discuss the tool itself. Please let us know if we are missing any paper by filing an Issue.

4.8.1 pyPRISM Tool Papers

1. Martin, T.B.; Gartner, T.E. III; Jones, R.L.; Snyder, C.R.; Jayaraman, A.; pyPRISM: A Computational Tool for Liquid State Theory Calculations of Macromolecular Materials, Macromolecules, 2018, 51 (8), p2906-2922 [link]
2. Martin, T.B.; Gartner, T.E. III; Jones, R.L.; Snyder, C.R.; Jayaraman, A.; Design and Implementation of pyPRISM: A Polymer Liquid-State Theory Framework, Proc. of the 17th Python in Science Conf. (Scipy 2018) (In Submission)

4.8.2 Studies Using pyPRISM

Coming Soon

4.9 Contributing

4.9.1 How to contribute to pyPRISM?

The ultimate goal of pyPRISM is to create a community of researchers who build upon each others efforts in developing PRISM-based techniques and applying these techniques to cutting edge problems. In order to make this possible, we need outside contributions to the codebase. The source code and all development for pyPRISM will be handled through the repository on GitHub. The primary route for adding a new module or feature to pyPRISM is through the common "fork + pull request" paradigm that is used across Github. See here for a description on how to carry out this process. While this method is common in the open-source community, we recognize that this may be over-complicated for some PRISM researchers. Please file an Issue or contact the developers directly if you have a contribution but don't wish to use git/GitHub to integrate it.

4.9.2 Developer Documentation

As the pyPRISM user base grows and we start having regular contributions to the codebase, we will explore putting together more detailed developer documentation including a style guide and other resources. For now, interested developers should rely on the API documentation provided here and the notebooks at the end of the *Tutorial*. Please file an Issue or contact the developers directly if you need more information to contribute or if something is missing from the documentation.

4.10 Contact Us

- **Dr. Tyler Martin, NIST,** GitHub, Webpage, Scholar
- **Mr. Thomas Gartner, University of Delaware,** GitHub, Scholar
- **Dr. Ronald Jones, NIST,** Webpage, Scholar
- **Dr. Chad Snyder, NIST,** Webpage, Scholar
- **Prof. Arthi Jayaraman, University of Delaware,** Webpage, Scholar

4.11 Legal

4.11.1 NIST Disclaimer

Any identification of commercial or open-source software in this document is done so purely in order to specify the methodology adequately. Such identification is not intended to imply recommendation or endorsement by the National Institute of Standards and Technology, nor is it intended to imply that the softwares identified are necessarily the best available for the purpose.

4.11.2 NIST License

Python Module Index

Index