Design and Implementation of pyPRISM: A Polymer Liquid-State Theory Framework

Tyler B. Martin[‡*], Thomas E. Gartner III[§], Ronald L. Jones[‡], Chad R. Snyder[‡], Arthi Jayaraman[§¶]

Abstract—In this work, we describe the code structure, implementation, and usage of a Python-based, open-source framework, pyPRISM, for conducting polymer liquid-state theory calculations. Polymer Reference Interaction Site Model (PRISM) theory describes the equilibrium spatial-correlations, thermodynamics, and structure of liquid-like polymer systems and macromolecular materials. pyPRISM provides data structures, functions, and classes that streamline predictive PRISM calculations and can be extended for other tasks such as the coarse-graining of atomistic simulation force-fields or the modeling of experimental scattering data. The goal of providing 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 polymer nanocomposites.

Index Terms—polymer, materials science, modeling, theory

Introduction

Free and open-source (FOSS) scientific software lowers the barriers to applying theoretical techniques by codifying complex approaches into usable tools that can be leveraged by non-experts. Here, we describe the implementation and structure of pyPRISM, a Python tool which implements Polymer Reference Interaction Site Model (PRISM) theory. [MGIJ+18], [dis] PRISM theory is an integral equation formalism that describes the structure and thermodynamics of polymer liquids. [SC87] Despite the successful application of PRISM theory to study a variety of complex soft-matter systems, [SC94] its use has been limited compared to other theory and simulation methods that have available open-source tools, such as Self-Consistent Field Theory (SCFT), [psc], [AQM+16] Molecular Dynamics (MD), [hoo], [GNA+15], [ALT08], [lam], [Pli95] or Monte Carlo (MC), [sim], [cas], [RM11]. Some important factors contributing to this reduced usage are the complexities associated with implementing PRISM theory and the lack of an available open-source codebase. Our previous publication, [MGIJ+18], focused primarily on the theoretical aspects of the method and presented several case studies to illustrate the utility of PRISM theory. In this work, we focus more specifically on the practical implementation and usage of PRISM theory within the pyPRISM framework. In the following sections, we will briefly discuss the basics of PRISM theory, our implementation of the theory in pyPRISM, our approach toward educating the scientific community about PRISM theory and pyPRISM, and finally our view for the future of the tool.

* *Corresponding author: tyler.martin@nist.gov*
‡ *National Institute of Standards and Technology*
§ *Chemical and Biomolecular Engineering, University of Delaware*
¶ *Materials Science and Engineering, University of Delaware*

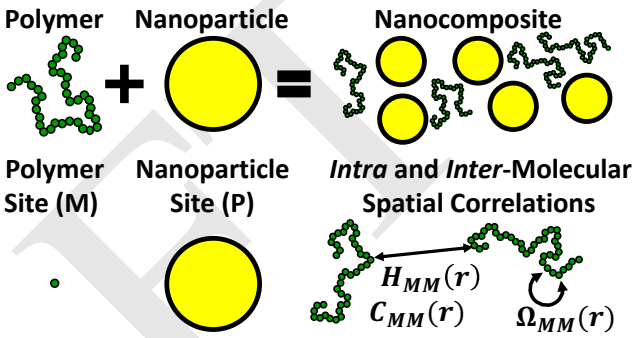

Fig. 1: *A schematic representation of the components of a coarse grained polymer nanocomposite made up of polymer chains and large spherical nanoparticles. This system is the focus of reference [HS05]. In this example, there are two site-types: a monomer site-type (M) in green and a nanoparticle site-type (P) in yellow. Also labeled are the polymer-polymer intra-molecular ($\Omega_{M,M}(r)$) and inter-molecular correlation functions ($H_{M,M}(r)$ and $C_{M,M}(r)$).*

PRISM Theory

For a detailed discussion of PRISM theory, as well as a review of key applications of the theory, we direct the reader to our previous publication. [MGIJ+18] Here, we briefly highlight the salient points of PRISM theory in order to help motivate the design of our class structure.

PRISM theory describes the *spatial correlations* in a liquid-like polymer system made up of spherical interacting "sites." The category of liquid-like polymers includes melts, blends, solutions, and nanocomposites of both homopolymers and copolymers. Within these systems, PRISM is able to handle varying chain chemistry, monomer sequence, and topology. The traditional PRISM formalism is spherically symmetric, which in general prevents the use of PRISM to study glassy, crystalline, phase-separated or otherwise non-isotropic materials. While there is a modified PRISM formalism for oriented (liquid-crystalline) materials, [OS05a], [OS05b], [PS00], [PS99] those modifications are outside the scope of the current work. Figure 1 shows a schematic of a polymer nanocomposite that could be studied with PRISM theory using a two-site model.

In general, PRISM sites represent a segment of a molecule or polymer chain, similar to the atoms or coarse-grained beads that comprise an MD or MC simulation. Unlike these simulation methods, PRISM treats all of the sites of a given type as indistinguishable and does not track the individual positions of each site in space. Instead, the structure of the system is described through average spatial correlation functions. The fundamental PRISM equation for multi-component systems is represented in Fourier-space as a matrix equation of the site-site spatial correlation functions.

$$\hat{H}(k) = \hat{\Omega}(k)\hat{C}(k)\left[\hat{\Omega}(k) + \hat{H}(k)\right] \tag{1}$$

In this expression, $\hat{H}(k)$ is the *inter*-molecular total correlation function matrix, $\hat{C}(k)$ is the *inter*-molecular direct correlation function matrix, and $\hat{\Omega}(k)$ is the *intra*-molecular correlation function matrix. $\hat{\Omega}(k)$ describes the how the monomers *within a molecule* are connected and placed, $\hat{H}(k)$ and $\hat{C}(k)$ describe how the molecules are placed in space relative to one another. The key difference between $\hat{H}(k)$ and $\hat{C}(k)$ is that the former includes many-body effects, while the latter does not. Knowledge of $\hat{H}(k)$, $\hat{C}(k)$, and $\hat{\Omega}(k)$ for a given system allows one to calculate a range of important structural and thermodynamic parameters, e.g., structure factors, radial distribution functions, second virial coefficients, Flory-Huggins χ parameters, bulk isothermal compressibilities, and spinodal decomposition temperatures.

Each of the variables in Equation 1 represents a function of wavenumber k which returns an $n \times n$ matrix, with n being the number of site-types in the calculation. Each element of a correlation function matrix (e.g., $\hat{H}_{\alpha,\beta}(k)$) represents the value of that correlation function between site types α and β at a given wavenumber k. These correlation function matrices are symmetric, therefore there are $\frac{n(n+1)}{2}$ independent site-type pairs and correlation function values in each correlation function matrix. The nanocomposite in Figure 1 is modeled using $n = 2$ site-types which yields three independent site-type pairs: polymer-polymer, polymer-particle, and particle-particle.

Equation 1, as written, has one unspecified degree of freedom for each site-type pair, therefore additional mathematical relationships must be supplied to obtain a solution. These relationships are called closures and are derived in various ways from fundamental liquid-state theory. Closures are also how the chemistry of a system is specified *via* pairwise interaction potentials $U_{\alpha,\beta}(r)$. For example, one widely-used closure is the Percus-Yevick closure shown below

$$C_{\alpha,\beta}(r) = \left(e^{-U_{\alpha,\beta}(r)} - 1.0\right)\left(1.0 + \Gamma_{\alpha,\beta}(r)\right), \tag{2}$$

where $\Gamma(r)$ is defined in real-space as

$$\Gamma_{\alpha,\beta}(r) = H_{\alpha,\beta}(r) - C_{\alpha,\beta}(r). \tag{3}$$

While the PRISM equation can be solved analytically [SC94] in select cases, we focus on a more generalizable numerical approach here. Figure 2 shows a schematic of our approach. For all site-types or site-type pairs, the user provides input values for $\hat{\Omega}_{\alpha,\beta}(k)$, site-site pair potentials $U_{\alpha,\beta}(r)$, site-type densities ρ_α, and an initial guess for all $\Gamma_{\alpha,\beta}(r)$. After the user supplies all necessary parameters and input correlation functions, pyPRISM applies a numerical optimization routine, such as a Newton-Krylov method, [new] to minimize a self-consistent cost function. The details of this cost function were discussed in our previous work. [MGIJ+18] After the cost function is minimized, the PRISM equation is considered "solved" and the resultant correlation functions can be used for calculations.

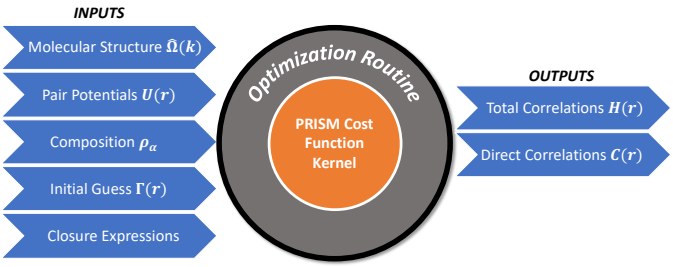

Fig. 2: *Schematic of PRISM theory numerical solution process.*

pyPRISM Overview

pyPRISM defines a scripting API (application programming interface) that allows users to conduct calculations and numerically solve the PRISM equation for a range of liquid-like polymer systems. All of the theoretical details of PRISM theory are encapsulated in classes and methods which allow users to specify parameters and input correlation functions by name e.g., `PercusYevick` for Equation 2. Furthermore, the structure of these classes was kept as simple as possible so that novice scientific programmers could easily extend pyPRISM by implementing new closures, potentials, or intra-molecular correlation functions. These code structure of pyPRISM is shown in schematically in Figure 3 and is discussed in the Implementation Section.

Providing a scripting API rather than an "input file"-based scheme gives users the ability to use the full power of Python for complex PRISM-based calculations. For example, one could use parallelized loops to fill a database with PRISM results using Python's built-in support for thread or process pools. Alternatively, pyPRISM could easily be coupled to a simulation engine by calling the engine *via* a subprocess, processing the engine output, and then feeding that output to to a pyPRISM calculation. The pyPRISM API is demonstrated in the Example pyPRISM Script section by modeling the system shown in Figure 1.

While experts in PRISM theory likely will need little guidance on how to appropriately apply pyPRISM, we also would like to make pyPRISM accessible to the widest possible audience. To this end, we have created comprehensive documentation [pyPa] and tutorial [pyPb] materials. Users can also try pyPRISM in their web-browser by visiting [pyPc]. See the Pedagogy section for more information on our philosophy in educating the scientific community about pyPRISM.

Installation

pyPRISM is a Python library that has been tested on Linux, OS X, and Windows with the CPython 2.7, 3.5 and 3.6 interpreters and only depends on Numpy [num], [WCV11] and Scipy [sci], [Oli07] for core functionality. Optionally, pyPRISM provides a unit conversion utility if the Pint [pin] library is available and a simulation trajectory analysis tool if pyPRISM is compiled with Cython [cyt]. pyPRISM is available on GitHub, [pyPd], conda-forge [pyPe] and the Python Package Index (PyPI) [pyPf] for download. It can be installed from the command line *via*

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

or alternatively

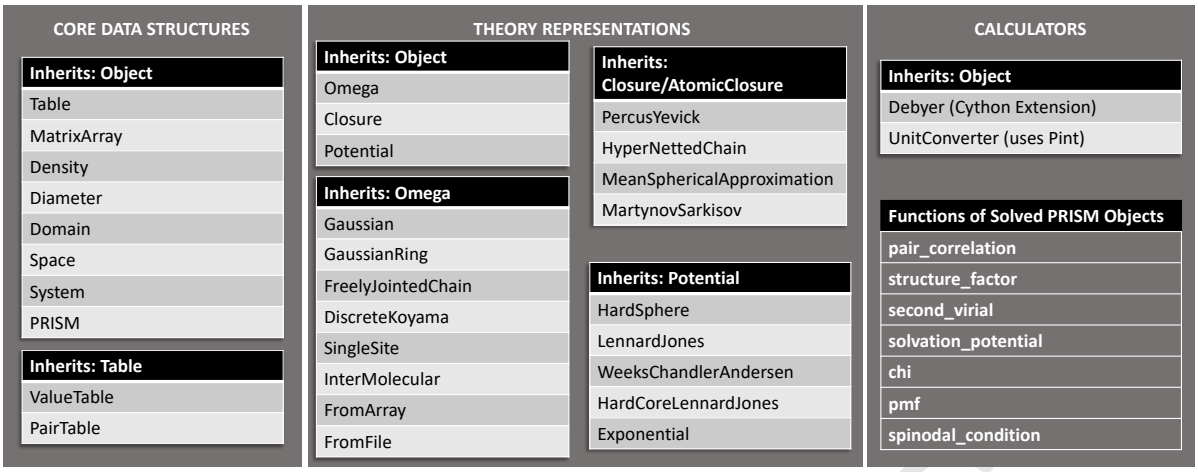

Fig. 3: *Overview of codebase and class organization. A full description of the codebase classes and methods can be found in the online documentation. [pyPa].*

```
$ pip install pyPRISM
```

Full installation instructions can be found in the documentation. [pyPa]

Implementation

Figure 3 shows an overview of the available classes and functions in pyPRISM and how they relate categorically. To begin, we consider the core data structures listed in the left column of the figure. Parameters and data in PRISM theory fall into two categories: those that define the properties of a single site-type (e.g., density, diameter) and those that define properties for a site-type pair (e.g., closure, potential, intra-molecular correlation functions). pyPRISM defines two base container classes based on this concept, both of which inherit from a parent `pyPRISM.Table` class: `pyPRISM.ValueTable` and `pyPRISM.PairTable`. These classes store numerical and non-numerical data, support complex iteration, and provide a `.check()` method that is used to ensure that all parameters are fully specified. Both `pyPRISM.Table` subclasses also support setting multiple pair-data at once, thereby making scripts easier to maintain *via* reduced visual noise and repetition. Additionally, `pyPRISM.ValueTable` automatically invokes matrix symmetry when a user sets an off-diagonal pair, assigning the α,β and β,α pairs automatically.

```
'''
Example of pyPRISM.ValueTable usage
'''

import pyPRISM

PT = pyPRISM.PairTable(types=['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'

try:
    # Raises ValueError b/c not all pairs are set
    PT.check()
except ValueError:
    print('Not all pairs are set in ValueTable!')

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

# No-op as all pairs are set
PT.check()

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

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

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

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

In some cases where additional logic or error checking is needed, we have created more specialized container classes. For example, both the site volumes and the site-site contact distances are functions of the individual site diameters. The `pyPRISM.Diameter` class contains multiple `pyPRISM.Table` objects which are dynamically updated as the user defines site-type diameters. The `pyPRISM.Density` class was created for analogous reasons so that the pair-density matrix,

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

the site-density matrix,

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

and the total site density,

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

can all be calculated dynamically as the user specifies or modifies the individual site-type densities ρ_α.

An additional specialized container is `pyPRISM.Domain`. This class specifies the discretized real- and Fourier-space grids over which the PRISM equation is solved and is instantiated by specifying the length (i.e., number of gridpoints) and grid spacing in real- or Fourier space (i.e., dr or dk). An important detail of the PRISM cost function mentioned above is that correlation functions need to be transformed to and from Fourier space during the cost function evaluation. `pyPRISM.Domain` also contains the Fast Fourier Transform (FFT) methods needed to efficiently carry out these transforms. The mathematics behind these FFT methods, which are implemented as Type II and III Discrete Sine Transforms (DST-II and DST-III), are discussed in our previous work. [MGIJ+18]

The `pyPRISM.System` class contains multiple `pyPRISM.ValueTable` and `pyPRISM.PairTable` objects in addition to the specialized container classes described above. The goal of the `pyPRISM.System` class is to be a super-container that can validate that a system is fully and correctly specified before allowing the user to attempt to solve the PRISM equation.

While `pyPRISM.System` primarily houses input property tables, `pyPRISM.PRISM` represents a fully specified PRISM calculation and contains the cost function to be numerically minimized. The correlation functions shown in Equation 1 are stored in the `pyPRISM.PRISM` object as `pyPRISM.MatrixArray` objects, which are similar to `pyPRISM.ValueTable` objects but with a focus on mathematics rather than storage. `pyPRISM.MatrixArray` objects can only contain numerical data and provide many operators and methods which simplify PRISM theory mathematics. In particular, they satisfy the need for easy access to both the matrix and pair-function representations of the correlation functions, shown schematically in Figure 4 . The former is necessary for carrying out the mathematics of the PRISM equation (Equation 1) and the latter for performing Fourier transformations of the individual pair-functions. The `pyPRISM.MatrixArray` objects also carry out a number of run-time error checks including ensuring that both `MatrixArray` objects involved in a binary operations (such as addition) are in the same space (real or Fourier). The core data structure underlying the `pyPRISM.MatrixArray` is a three-dimensional Numpy ndarray of m $n \times n$ matrices, where m is the length of the `pyPRISM.Domain`.

```
'''
Example of MatrixArray usage.
'''
## Setup ##
length = 1024       # number of gridpoints
dr = 0.1            # real-space grid spacing
rank = 2            # number of site-types
types = ['A', 'B'] # name of site-types

domain = pyPRISM.Domain(length,dr)
rho = pyPRISM.Density(types)

# Total and intra-molecular correlation functions
# dataH and dataW are size (length,rank,rank)
# numpy ndarrays that are assumed to be in memory
kwargs = dict(length=length,rank=rank,types=types)
H = pyPRISM.MatrixArray(data=dataH,**kwargs)
W = pyPRISM.MatrixArray(data=dataW,**kwargs)

## Example Calculation of Structure Factor ##
S = (W + H)/rho.site
```

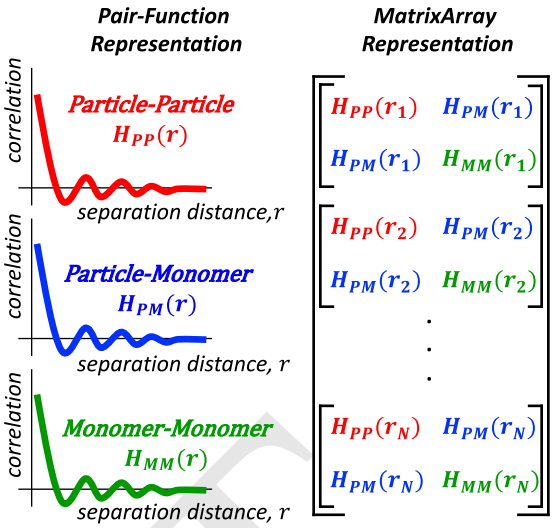

Fig. 4: *Schematic of the pair-function and MatrixArray representations of the total correlation function for the polymer nanocomposite system shown in Figure 1. The r_1, r_2, r_N variables represent specific distances in the real-space solution grid.*

```
S_AB = S['A','B'] # extract S_AB from MatrixArray

## MatrixArray by Scalar Operations ##
# All matrices in W are modified by the scalar x
x = 1 # arbitrary scalar
W+x; W-x; W*x; W/x; # elementwise ops

## MatrixArray by Matrix Operations ##
# All matrices in W are modified by the matrix rho
W+rho; W-rho; W*rho; W/rho;    # elementwise ops
W.dot(rho)                     # matrix mult.

## MatrixArray by MatrixArray Operations ##
# Operations are matrix to corresponding matrix
W+H; W-H; W*H; W/H;    # elementwise ops
W.dot(H)               # matrix mult.

##  Fourier Transformations ##
# Transform a single array versus all functions
# in a MatrixArray
W_AA = domain.to_real(W['A','A']) # one function
domain.MatrixArray_to_fourier(H)  # all functions

## Other Operations ##
W.invert()      # invert each matrix in W
W['A','B']      # set or get function for pair A-B
W.getMatrix(i) # get matrix i in MatrixArray
W.iterpairs()  # iterate over all 1-D functions
```

The `pyPRISM.PRISM` object is solved by calling the `.solve()` method which invokes a numerical algorithm to minimize the output of the `.cost()` method by varying the input $\Gamma_{\alpha,\beta}(r)$. Once a `pyPRISM.PRISM` object is numerically solved, it can be passed to a calculator that processes the optimized correlation functions and returns various structural and thermodynamic data. The current list of available calculators is shown in the rightmost column of Figure 3 and is fully described in the documentation. [pyPa]

Beyond the core data structures, pyPRISM defines classes which are meant to represent various theoretical equations or ideas. Classes which inherit from `pyPRISM.Potential`, `pyPRISM.Closure`, or `pyPRISM.Omega` represent interaction potentials, theoretical closures, or *intra*-molecular correla-

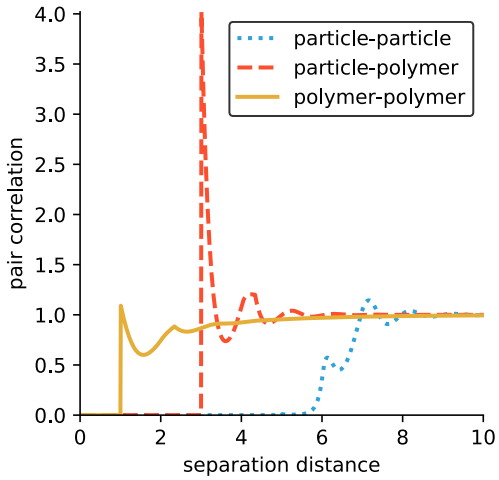

Fig. 5: *All pair-correlation functions from the pyPRISM example for the polymer nanocomposite system depicted in Figure 1.*

tion functions $\hat{\Omega}_{\alpha,\beta}(k)$, respectively. These properties must be specified for all site-type pairs before a `pyPRISM.PRISM` object can be created. To ensure that users can easily add new potentials, closures, and $\hat{\Omega}_{\alpha,\beta}(k)$ to the codebase, we have kept the programming interface contract of these classes as simple as possible: Subclasses must inherit from the proper parent class and implement a `.calculate()` method.

Example pyPRISM Script

```
'''
pyPRISM script calculating the pair correlation
functions and chi parameters of a polymer
nanocomposite.
'''

import pyPRISM

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

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

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

sys.omega['polymer','polymer']   = \
pyPRISM.omega.FreelyJointedChain(length=100,l=4/3)
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','particle']]   = \
pyPRISM.closure.PercusYevick()
sys.closure['particle','particle'] = \
pyPRISM.closure.HyperNettedChain()

PRISM = sys.solve()

pcf = pyPRISM.calculate.pair_correlation(PRISM)
pcf_11 = pcf['particle','particle']

chi = pyPRISM.calculate.chi(PRISM)
chi_12 = pcf['particle','polymer']
```

Example Discussion

The code above shows how to use pyPRISM to calculate the properties of a polymer nanocomposite made of linear polymer chains and spherical nanoparticles. This system is shown schematically in Figure 1 and is fully described in reference [HS05]. The results of this calculation are plotted in Figure 5. In this section, we will discuss the details of this example in a line by line fashion as we specify all inputs shown in Figure 2 and then solve the PRISM equation.

```
import pyPRISM

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

All pyPRISM calculations begin by first importing the pyPRISM library, and then creating a `pyPRISM.System` object. The first argument to the `pyPRISM.System` constructor is the names of the site-types for the calculation. In this case, we have two site-types which we (arbitrarily) call *polymer* and *particle*. Optionally, the constructor allows that the thermal energy level, k_BT, be specified. Next a `pyPRISM.Domain` object is created with `length=4096` grid-points and a grid spacing of `dr=0.1`.

Note that all parameters in pyPRISM are specified in a reduced unit system commonly called Lennard-Jones units. In this scheme, a characteristic length d_c, mass m_c, and energy e_c are specified. All other units are then specified in terms of these characteristic units. For example, if $d_c = 1$ *nm*, the grid spacing in the above code would be $dr = 0.1 d_c = 0.1$ *nm*. See [FB02] for more information on the Lennard-Jones reduced unit scheme.

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

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

Next, site-type diameters and number densities are specified for both site-types in units of d_c and beads per d_c^3, respectively. Qualitatively, these specifications imply that we are considering a dilute concentration of nanoparticles dissolved in a polymer matrix made up of polymer sites of significantly smaller diameter.

```
sys.omega['polymer','polymer']   = \
pyPRISM.omega.FreelyJointedChain(length=100,l=4/3)
sys.omega['polymer','particle']  = \
pyPRISM.omega.InterMolecular()
sys.omega['particle','particle'] = \
pyPRISM.omega.SingleSite()
```

The *intra*-molecular correlation function $\hat{\Omega}_{polymer,polymer}(k)$ is specified as a freely jointed chain, a well-known physical model for a polymer chain. [RC03] Since the polymer chains and particles are not connected, $\hat{\Omega}_{polymer,particle}(k)$ is specified as *inter*-molecular. The particles are modeled as spherical sites so $\hat{\Omega}_{particle,particle}(k)$ is modeled as a `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()
```

$U_{polymer,polymer}(r)$ and $U_{particle,particle}(r)$ pair potentials are specified as athermal hard sphere interactions, while the $U_{polymer,particle}(r)$ potential is an exponential attractive interaction. This configuration describes a dense melt-like polymer nanocomposite where the polymer chains are attracted to and adhere to (wet) the nanoparticle surface. The α and ε parameters in the `pyPRISM.potential.Expontential` constructor control the range and strength of the exponential attraction.

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

To demonstrate one utility of the `pyPRISM.PairTable` data structure, here we have specified both the *polymer-polymer* and *polymer-particle* closure in a single line. Both pair-data are specified to the Percus-Yevick closure, while the *particle-particle* closure is set to be the hypernetted chain closure. In this code-block and those above, note how the subclasses of `pyPRISM.Omega`, `pyPRISM.Potential` and `pyPRISM.Closure` are used to easily specify complex theoretical constructs.

```
PRISM = sys.solve()
```

When all properties are defined, the user calls the `pyPRISM.System.solve()` method which first conducts a number of sanity checks and issues any relevant exceptions or warnings if issues are found. If no issues are found, a PRISM object is created and minimization is attempted. The `.solve()` method accepts arguments which allow the user to tune the details of the minimization.

```
pcf = pyPRISM.calculate.pair_correlation(PRISM)
pcf_11 = pcf['particle','particle']

chi = pyPRISM.calculate.chi(PRISM)
chi_12 = pcf['particle','polymer']
```

Once the minimization completes, a `pyPRISM.PRISM` object is returned which contains the final solutions for $H(r)$ and $C(r)$ along with all input parameters and data. The `pyPRISM.PRISM` object is then passed through the `pyPRISM.calculate.pair_correlation` and `pyPRISM.calculate.chi` calculators. Both of these methods return `pyPRISM.ValueTables`, which can be subscripted to access the individual pair-functions. In the example, we extract the particle-particle pair correlation function, $g_{particle,particle}(r)$ and the particle-polymer $\chi_{particle,polymer}$ parameter.

While it would be feasible to study this polymer nanocomposite system *via* simulation methods such as MD or MC, the use of PRISM theory offers some distinct advantages. PRISM theory does not suffer from finite-size or equilibration effects, both of which limit simulation methods. Furthermore, a simulation of sufficient size to study the large nanoparticles and relatively long polymer chains in this example would require many hours to days of CPU or GPU time from a supercomputing resource. This is due to the computational expense of evaluating the pairwise interactions at each simulated configuration and the many millions of configurations that must be generated in order to properly equilibrate and sample such a nanocomposite. In contrast, PRISM theory can be numerically solved in seconds even on modest hardware such as a laptop computer. This is because, unlike MD or MC, solving PRISM theory does not involve generating molecular configurations, but rather is a set of integral equations which are numerically solved for the spatial correlation functions,

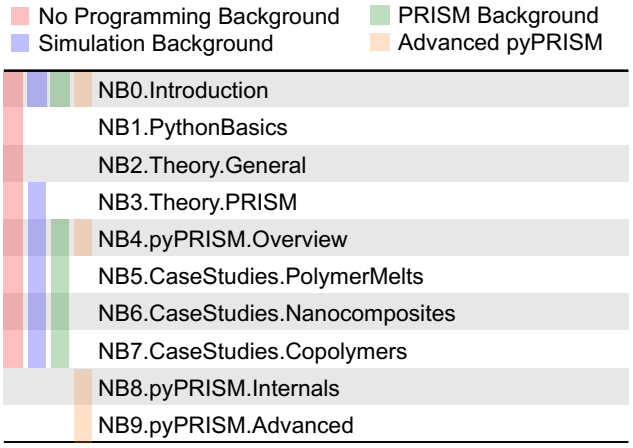

Fig. 6: *Depiction of the tutorial tracks we provide for users of different backgrounds and trainings. See the Tutorial page [pyPb] for more information.*

$H_{\alpha,\beta}(r)$ and $C_{\alpha,\beta}(r)$. This numerical solution process is briefly described above at the end of the PRISM Theory section and is described in detail in Section II.E of [MGIJ+18]. In addition to the computational performance benefits of PRISM theory over MD or MC, once the full set of pairwise spatial correlation functions is solved for, a variety of properties can quickly be screened without having to process large simulation trajectories.

PRISM theory provides a powerful alternative or complement to traditional simulation approaches, but we should note that it is not without limitation. There are restrictions on the types of systems and thermodynamic state points to which PRISM theory can be applied and the numerical closures are approximations and therefore sources of error. See Section IV.D of [MGIJ+18] for a discussion on the known limitations of PRISM theory.

Pedagogy

It is our goal to create a central platform for polymer liquid state theorists while also lowering the barriers to using PRISM theory for the greater polymer science community. Towards this effort, we have identified two primary challenges:

1) The process of understanding and numerically solving PRISM theory is complex and filled with pitfalls and opportunities for error.
2) Many of those who would benefit most from PRISM theory do not have a strong programming background.

Our strategy to address both of these challenges is a strong focus on providing pedagogical resources to users. To start, we have put significant effort into our documentation. Every page of the API documentation [pyPa] contains a written description of the theory being implemented, all necessary mathematics, descriptions of all input and output parameters, links to any relevant journal articles, and a detailed and relevant example. While including these features in our documentation is not a new idea, we are focusing on providing these resources immediately upon release and iterating based on user feedback to improve the clarity and scope of the information provided.

Moving beyond API documentation, we also have created knowledgebase materials which provide more nuanced information about using and numerically solving PRISM theory. This

knowledgebase includes everything from concise lists of systems and properties that can be studied with pyPRISM to tips and tricks for reaching convergence of the numerical solver. In reference to Challenge 2 above, we also recognize that a significant barrier for non-experts to use these tools is the installation process. Our installation documentation [pyPa] attempts to be holistic and provide detailed instructions for the several different ways that users can install pyPRISM.

We have also created a self-guided tutorial to PRISM theory and pyPRISM in the form of a series of Jupyter notebooks. [pyPb], [jup] The tutorial notebooks are designed to target a wide audience with varied programming and materials science expertise, with topics ranging from a basic introduction to Python to how to add new features to pyPRISM. The tutorial also has several case study-focused notebooks which walk users through the process of reproducing PRISM results from the literature. Figure 6 shows our recommendations for how users of different backgrounds and skill levels might move through the tutorial. In order to ensure the widest audience possible can take advantage of this tutorial, we have also set up a binder instance [pyPc], which allows users to try out pyPRISM and run the tutorial instantly in a web-browser without installing any software. This feature should also benefit users who might be hampered by Challenge 2 above.

Future Directions

While pyPRISM is a step forward in providing a central platform for polymer liquid-state theory calculations, we intend to significantly extend the tool beyond its release state. The most obvious avenue for extension will be to add new potentials, closures, and *intra*-molecular correlation functions $\left(\hat{\Omega}_{\alpha,\beta}(k)\right)$ to the codebase. As described above, we hope that a significant portion of these classes will be contributed by users. Where analytical expressions for $\hat{\Omega}_{\alpha,\beta}(k)$ do not exist, they can also be calculated from simulation trajectories. While we do provide a Cython-enhanced tool to do the calculation, we also plan to add features to more easily couple pyPRISM to common MD and MC simulation packages. [hoo], [lam], [sim], [cas] These linkages would also make it easier for users to carry out the Self-Consistent PRISM (SCPRISM) method. [MGIJ+18]

PRISM theory also has advanced applications that are not possible in the current pyPRISM workflow. One example is the use of PRISM theory to translate a detailed atomistic simulation model to a less detailed, less computationally expensive coarse-grained model in a methodology called Integral Equation Coarse Graining (IECG). [DG17b], [DG17a], [MCLG12], [YSNG04] We plan to provide utilities in the pyPRISM codebase that aid in carrying out this method. PRISM theory can also be used to model or fit neutron and X-ray scattering data. In particular, PRISM theory can be used to take existing scattering models for single particles or polymer chains and model the effects of intermolecular interactions. This approach would greatly extend the applicability of existing scattering models, which on their own are only valid in the infinitely dilute concentration limit, but could be combined with pyPRISM to model higher concentrations.

Summary

pyPRISM is an open-source tool with the goal of facilitating the usage of PRISM theory, a polymer liquid-state theory. Compared to more widely-used simulation methods such as MD and MC, PRISM theory is significantly more computationally efficient, does not need to be equilibrated, and does not suffer from finite size effects. pyPRISM lowers the barriers to using PRISM theory by providing a simple scripting interface for setting up and numerically solving the theory. Furthermore, in order to ensure users correctly and appropriately use pyPRISM, we have created extensive pedagogical materials in the form of API documentation, knowledgebase materials, and Jupyter-notebook powered tutorials.

Acknowledgements

TBM is supported by a National Research Council (NRC) fellowship at the National Institute of Standards and Technology (NIST). In addition, this work has been supported by the members of the NIST nSoft consortium (nist.gov/nsoft). TEG and AJ thank National Science Foundation Division of Materials Research Condensed Matter and Materials Theory (NSF DMR-CMMT) grant number 1609543 for financial support. This research was supported in part through the use of Information Technologies (IT) resources at the University of Delaware, specifically the high-performance computing resources of the Farber supercomputing cluster. This work used the Extreme Science and Engineering Discovery Environment (XSEDE) Stampede cluster at the University of Texas through allocation MCB100140 (AJ), which is supported by National Science Foundation grant number ACI-1548562.

References

[ALT08] Joshua A. Anderson, Chris D. Lorenz, and A. Travesset. General purpose molecular dynamics simulations fully implemented on graphics processing units. *Journal of Computational Physics*, 227(10):5342 – 5359, 2008. doi:10.1016/j.jcp.2008.01.047.

[AQM+16] Akash Arora, Jian Qin, David C. Morse, Kris T. Delaney, Glenn H. Fredrickson, Frank S. Bates, and Kevin D. Dorfman. Broadly accessible self-consistent field theory for block polymer materials discovery. *Macromolecules*, 49(13):4675–4690, 2016. doi:10.1021/acs.macromol.6b00107.

[cas] URL: https://www3.nd.edu/~ed/research/cassandra.html.

[cyt] URL: http://cython.org.

[DG17a] M. Dinpajooh and M. G. Guenza. Thermodynamic consistency in the structure-based integral equation coarse-grained method. *Polymer*, 117:282–286, 2017. doi:https://doi.org/10.1016/j.polymer.2017.04.025.

[DG17b] Mohammadhasan Dinpajooh and Marina G. Guenza. On the density dependence of the integral equation coarse-graining effective potential. *The Journal of Physical Chemistry B*, 2017. doi:10.1021/acs.jpcb.7b10494.

[dis] Any identification of commerical or open-source software in this paper 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.

[FB02] Daan Frenkel and Smit Berend. *Monte Carlo Simulations: A Basic Monte Carlo Algorithm*, book section 3, pages 40–42. Computational Science Series. Academic Press, San Diego, California, 2 edition, 2002.

[GNA+15] Jens Glaser, Trung Dac Nguyen, Joshua A. Anderson, Pak Lui, Filippo Spiga, Jaime A. Millan, David C. Morse, and Sharon C. Glotzer. Strong scaling of general-purpose molecular dynamics simulations on gpus. *Computer Physics Communications*, 192:97 – 107, 2015. doi:10.1016/j.cpc.2015.02.028.

[hoo] URL: http://glotzerlab.engin.umich.edu/hoomd-blue/index.html.

[HS05] Justin B. Hooper and Kenneth S. Schweizer. Contact aggregation, bridging, and steric stabilization in dense polymer-particle mixtures. *Macromolecules*, 38(21):8858–8869, 2005. doi:10.1021/ma051318k.

[jup] URL: https://jupyter.org.

[lam] URL: http://lammps.sandia.gov/.

[MCLG12] J. McCarty, A. J. Clark, I. Y. Lyubimov, and M. G. Guenza. Thermodynamic consistency between analytic integral equation theory and coarse-grained molecular dynamics simulations of homopolymer melts. *Macromolecules*, 45(20):8482–8493, 2012. URL: <GotoISI>://WOS:000310101600039http://pubs.acs.org/doi/pdfplus/10.1021/ma301502w, doi:10.1021/ma301502w.

[MGIJ+18] T. B. Martin, T. E. Gartner III, R. L. Jones, C. R. Snyder, and A. Jayaraman. pyprism: A computational tool for liquid-state theory calculations of macromolecular materials. *Macromolecules*, 51(8):2906–2922, 2018. doi:10.1021/acs.macromol.8b00011.

[new] URL: https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.newton_krylov.html.

[num] URL: http://numpy.org/.

[Oli07] T. E. Oliphant. Python for scientific computing. *Computing in Science & Engineering*, 9(3):10–20, 2007. URL: http://ieeexplore.ieee.org/document/4160250/, doi:10.1109/MCSE.2007.58.

[OS05a] F. T. Oyerokun and K. S. Schweizer. Theory of glassy dynamics in conformationally anisotropic polymer systems. *Journal of Chemical Physics*, 123(22), 2005. URL: <GotoISI>://WOS:000234120800047http://aip.scitation.org/doi/pdf/10.1063/1.2135776, doi:10.1063/1.2135776.

[OS05b] F. T. Oyerokun and K. S. Schweizer. Thermodynamics, orientational order and elasticity of strained liquid crystalline melts and elastomers. *Journal of Physical Chemistry B*, 109(14):6595–6603, 2005. URL: <GotoISI>://WOS:000228231200018http://pubs.acs.org/doi/pdfplus/10.1021/jp045646i, doi:10.1021/jp045646i.

[pin] URL: http://pint.readthedocs.io/.

[Pli95] S. Plimpton. False parallel algorithms for short-range molecular-dynamics. *Journal of Computational Physics*, 117(1):1–19, 1995. doi:10.1006/jcph.1995.1039.

[PS99] G. T. Pickett and K. S. Schweizer. Liquid-state theory of anisotropic flexible polymer fluids. *Journal of Chemical Physics*, 110(14):6597–6600, 1999. URL: <GotoISI>://WOS:000079541700002http://aip.scitation.org/doi/pdf/10.1063/1.478566, doi:10.1063/1.478566.

[PS00] G. T. Pickett and K. S. Schweizer. Liquid crystallinity in flexible and rigid rod polymers. *Journal of Chemical Physics*, 112(10):4881–4892, 2000. URL: <GotoISI>://WOS:000085563800049http://aip.scitation.org/doi/pdf/10.1063/1.481039, doi:10.1063/1.481039.

[psc] URL: http://pscf.cems.umn.edu/.

[pyPa] URL: http://pyprism.readthedocs.io/.

[pyPb] URL: http://pyprism.readthedocs.io/en/latest/tutorial/tutorial.html.

[pyPc] URL: https://mybinder.org/v2/gh/usnistgov/pyprism/master?filepath=tutorial.

[pyPd] URL: https://github.com/usnistgov/pyPRISM.

[pyPe] URL: https://anaconda.org/conda-forge/pyprism.

[pyPf] URL: https://pypi.org/project/pyPRISM/.

[RC03] M. Rubinstein and R.H. Colby. *Polymer Physics*. OUP Oxford, 2003.

[RM11] Neeraj Rai and Edward J. Maginn. Vapor–liquid coexistence and critical behavior of ionic liquids via molecular simulations. *The Journal of Physical Chemistry Letters*, 2(12):1439–1443, 2011. doi:10.1021/jz200526z.

[SC87] K. S. Schweizer and J. G. Curro. Integral-equation theory of the structure of polymer melts. *Physical Review Letters*, 58(3):246–249, 1987. doi:10.1103/PhysRevLett.58.246.

[SC94] K. S. Schweizer and J. G. Curro. *PRISM Theory of the Structure, Thermodynamics, and Phase-Transitions of Polymer Liquids and Alloys*, volume 116 of *Advances in Polymer Science*, pages 319–377. 1994. doi:10.1007/BFb0080203.

[sci] URL: http://scipy.org/.

[sim] URL: http://dmorse.github.io/simpatico/index.html.

[WCV11] S. van der Walt, S. C. Colbert, and G. Varoquaux. The numpy array: A structure for efficient numerical computation. *Computing in Science & Engineering*, 13(2):22–30, 2011. URL: http://ieeexplore.ieee.org/document/5725236/, doi:10.1109/MCSE.2011.37.

[YSNG04] G. Yatsenko, E. J. Sambriski, M. A. Nemirovskaya, and M. Guenza. Analytical soft-core potentials for macromolecular fluids and mixtures. *Physical Review Letters*, 93(25), 2004. doi:10.1103/PhysRevLett.93.257803.