

Theses and Dissertations

6-1-2009

Theory and simulation of metal-organic materials and biomolecules

Jonathan L. Belof

University of South Florida

Scholar Commons Citation

Belof, Jonathan L., "Theory and simulation of metal-organic materials and biomolecules" (2009). *Theses and Dissertations*. Paper 1851.
<http://scholarcommons.usf.edu/etd/1851>

This Dissertation is brought to you for free and open access by Scholar Commons. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of Scholar Commons. For more information, please contact scholarcommons@usf.edu.

Theory and Simulation of Metal-Organic Materials and Biomolecules

by

Jonathan L. Belof

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
Department of Chemistry
College of Arts and Sciences
University of South Florida

Major Professor: Brian Space, Ph.D.
Randy Larsen, Ph.D.
H. Lee Woodcock, Ph.D.
Preston Moore, Ph.D.

Date of Approval:
November 12, 2009

Keywords: nanomaterials, metal-organic frameworks, condensed matter, statistical mechanics, computer simulation, Monte Carlo, molecular dynamics

© Copyright 2009, Jonathan L. Belof

Dedication

In loving memory of Joan Elaine Belof (1939-2002)

Acknowledgments

I owe everything to the love of my life, my dearest wife Maiky. Without your unwaivering support, none of this would have been possible. Your love, understanding, encouragement and patience are behind every printed word – I love you with all my heart.

And also to my sweet daughter Isabella, the sparkling gem of my life.

To my father Lewis Belof, who has helped me in so many ways to sustain my graduate work throughout these years, I'm so thankful for your support. This work honors you – it is as much yours as it is mine.

And to my family: Stephen and Suzanne Belof, Dawn and Fred Ganter, Beth and Dan Jasko, Josh and Elizabeth Belof, for their love and support.

I am grateful to my Ph.D. committee of Randy Larsen, Lee Woodcock, Preston Moore (USP) and Chair Venkat Bhethanabotla for their guidance and insight throughout the years.

My fellow labmates and friends Chris Cioce, Tony Green, Abe Stern, Katherine Forrest and Ashley Mullen – your comaraderie will be greatly missed.

To my fellow graduate student and postdoctoral collaborators Audrey Mokdad, Will Lowe and Neil McIntye, and especially to Mohamed Alkordi whom I consider to be one of the most talented chemists I've ever known.

A special thanks to David Rabson, from whom I've learned to appreciate the broader aspects of physics, far beyond my current discipline.

I'm grateful to my undergraduate mentor Bob Potter for his kindness and encouragement. The path toward graduate school began with the many sage discussions we had in his office, which have greatly influenced my perspective toward science.

Last but certainly not least, I am infinitely grateful to Brian Space, my research advisor, my mentor and my friend. You have always gone the extra mile for me, both personally and professionally. You've always been generous and supportive of me, and I will never forget the kindness that you've shown me. I've learned so much from you over the years, and I've greatly enjoyed our scientific work together. I could not have asked for a better mentor, you've provided me with direction rooted in wisdom and at the same time nurtured my desire to intellectually wander. You are a true friend, and have been there for me at the times I've most needed your help, in the darkest of moments when it truly counts. I admire you and look up to you, and feel fortunate to have been mentored by such a gifted scientist and honorable person – thank you.

Note to Reader

Note to Reader: The original of this document contains color that is necessary for understanding the data. The original dissertation is on file with the USF library in Tampa, Florida.

Table of Contents

List of Tables	vi
List of Figures	vii
Abstract	x
Chapter 1 Introduction	1
Chapter 2 On the Mechanism of Hydrogen Storage in a Metal-Organic Framework (MOF) Material	6
2.1 Introduction	7
2.2 Models and Methods	11
2.2.1 Molecular Simulation Parameters	11
2.2.2 Polarizability Model	16
2.2.3 NVT Monte Carlo	22
2.2.4 Hybrid Monte Carlo	24
2.3 Results and Discussion	26
2.4 Conclusions	33

Chapter 3	An Accurate and Transferable Intermolecular Diatomic Hydrogen Potential for Condensed Phase Simulation	35
3.1	Introduction	35
3.2	Methods	37
3.2.1	Born-Oppenheimer Surface	37
3.2.2	Many-body Polarization	39
3.2.3	Potential Energy Function	41
3.3	Model Validation	44
3.3.1	Second Virial Coefficient	44
3.3.2	Equation of State	46
3.4	Conclusions	47
Chapter 4	A Predictive Model of Gas Sorption for Metal-Organic Materials	50
4.1	Introduction	50
4.2	Methods	52
4.2.1	Hydrogen Potential	52
4.2.2	MOF-5 Potential	53
4.2.3	Analysis of Potential Parameters	55
4.2.4	Grand Canonical Monte Carlo	57
4.3	Results and Discussion	60
4.3.1	Hydrogen Isotherms	60
4.3.2	Isosteric Heat of Adsorption of Hydrogen	60
4.3.3	Isothermal Compressibility of Hydrogen	63
4.4	Conclusions	65
Chapter 5	Photophysical Studies of the Trans to Cis Isomerization of the Push-Pull Molecule: 1-(Pyridin-4-yl)-2-(N-methylpyrrol-2-yl)ethene (mepepy)	67

5.1	Introduction	68
5.2	Methods	70
5.3	Results and Discussion	71
Chapter 6	Insight into the Assembly Mechanism of Metal-Organic Materials	75
6.1	Introduction	75
6.2	Methods	78
6.3	Results and Discussion	79
Chapter 7	Molecular Squares: Confined Space with Specific Geometry for Hydrogen Uptake	86
7.1	Introduction	86
7.2	Methods and Results	87
Chapter 8	Rapidly Convergent Iterative Techniques Toward the Solution of Many-body Molecular Polarization Field Equations	91
8.1	Introduction	91
8.2	Iterative Methods for Many-body Polarization	96
8.3	Results	99
Chapter 9	Calculation of Rotational Spectra for Sorbed Hydrogen in Metal-Organic Materials	103
9.1	Introduction	103
9.2	Methods and Results	105
Chapter 10	Microcanonical Effective Partition Function	109
10.1	Introduction	109
10.2	Microcanonical Derivation	112
10.3	Anharmonic Oscillator and Numerical Evaluation	113

Chapter 11	A Student-Friendly Derivation of the Partition Function for Generalized Ensembles	116
11.1	Introduction	117
11.2	The Gibbs Entropy and the Microcanonical Ensemble	118
11.3	A Simplified Derivation of the Canonical Partition Function . .	121
11.4	Grand Canonical Partition Function	124
11.5	Isothermal-Isobaric Partition Function	126
11.6	Connection with Generalized Ensemble Theory	128
11.7	Conclusions	131
Chapter 12	Volume Determination of Globular Proteins by Molecular Dynamics	133
12.1	Introduction	133
12.2	Methods	134
12.3	Results and Discussion	135
12.3.1	Horse-heart myoglobin	135
12.3.2	Aspartate aminotransferase	136
12.4	Conclusions	139
Chapter 13	Massively Parallel Monte Carlo (MPMC)	140
References		142
Appendices		158
Appendix A.	Virial Equation for the Pressure	159
Appendix B.	Frenkel Volume Derivative	160
Appendix C.	Widom Potential Distribution Theorem	162
Appendix D.	Fugacity	164
Appendix E.	Isosteric Heat of Adsorption (Q_{st})	167

Appendix F. autocorr.c	170
Appendix G. get_virial_coefficients.c	172
Appendix H. virial_iso.c	174
Appendix I. structure_factor.c	176
Appendix J. rng.c	179
Appendix K. Q_{st} R Code	184
Appendix L. MPMC	186

About the Author

End Page

List of Tables

Table 2.1	Partial charges obtained for <i>soc</i> -MOF Fragment A	14
Table 2.2	Partial charges obtained for <i>soc</i> -MOF Fragment B	14
Table 3.1	H_2 polarizability tensor <i>via</i> TDHF	39
Table 3.2	Polarizable hydrogen potential parameters	42
Table 3.3	Non-polarizable hydrogen potential parameters	44
Table 4.1	MOF-5 potential parameters	54
Table 4.2	MOF-5 partial charge comparison for different basis sets . . .	56
Table 4.3	Charge analysis for three MOF-5 fragments	57
Table 4.4	Complete MOF-5 potential parameters	57
Table 5.1	MEPEPY geometry	71
Table 5.2	MEPEPY ground-state energy and dipole moments	72
Table 8.1	SCF Results as a 2-step iteration scheme	102

List of Figures

Figure 2.1	<i>soc</i> -MOF hydrogen population isosurface	9
Figure 2.2	Molecular gas-phase <i>soc</i> -MOF fragments used to determine the partial charges	13
Figure 2.3	In-hydrogen $g(r)$	27
Figure 2.4	Azo-hydrogen $g(r)$	28
Figure 2.5	Bi-modal dipole distribution for hydrogen in <i>soc</i> -MOF	29
Figure 2.6	Induced dipole isosurfaces	30
Figure 3.1	MP4 Energy Surface for H ₂ dimer	38
Figure 3.2	Potential energy surface obtained through simulated annealing of the parameter space	43
Figure 3.3	Comparison of isotropic hydrogen potential vs. Silvera-Goldman	45
Figure 3.4	Hydrogen second virial coefficients	46
Figure 3.5	Bulk hydrogen 77 K isotherm	48
Figure 3.6	Bulk hydrogen 298 K isotherm	48
Figure 4.1	MOF-5 molecular fragment used in the calculation of partial charges	54
Figure 4.2	MOF-5 Fragment 1	55
Figure 4.3	MOF-5 Fragment 2	56
Figure 4.4	MOF-5 Fragment 3	56

Figure 4.5	MOF-5 hydrogen 77 K low-pressure isotherm	61
Figure 4.6	MOF-5 hydrogen 77 K high-pressure isotherm (weight percent)	61
Figure 4.7	MOF-5 hydrogen 77 K high-pressure isotherm (excess weight percent)	62
Figure 4.8	Comparison of MOF-5 hydrogen excess isotherm vs. experimental data	62
Figure 4.9	Isosteric heat of adsorption for hydrogen in MOF-5	64
Figure 4.10	Compressibility of hydrogen in MOF-5	65
Figure 5.1	trans isomer of MEPEPY	71
Figure 5.2	MEPEPY isomer geometries	72
Figure 5.3	MEPEPY ground-state energy surface	74
Figure 6.1	Cobalt nanocube	77
Figure 6.2	Ligand used in nanocube synthesis	77
Figure 6.3	Time-series NMR spectra of reaction forming nanocube	79
Figure 6.4	Simulated potential inorganic complexes	80
Figure 6.5	T1 relaxation distances as computed using DFT	81
Figure 6.6	Comparison of experimental and theoretical T1 relaxation spectra	81
Figure 6.7	Metal-Organic Square	82
Figure 6.8	Depiction of nanocube with counterions	83
Figure 6.9	Proposed nanocube assembly mechanism	84
Figure 7.1	Molecular Squares: ME193	87
Figure 7.2	Molecular Squares: Dispersive energy surface and the CBS limit	89
Figure 8.1	Energy convergence for various iterative solvers	99

Figure 8.2	Energy convergence using PK correction for various iterative solvers	100
Figure 8.3	Dipole RRMS for various iterative solvers	101
Figure 9.1	Rotational tunnel splitting for a hindered rotor	106
Figure 9.2	Rotational tunnel splitting for H ₂ in MOF-5	107
Figure 10.1	Classical vs. quantum entropy for a proton in an anharmonic well	115
Figure 11.1	Gibbs construction for the canonical ensemble	124
Figure 11.2	Gibbs construction for the grand canonical ensemble	127
Figure 11.3	Gibbs construction for the isothermal-isobaric ensemble	129
Figure 12.1	Horse-heart myoglobin	134
Figure 12.2	Volume of protein and water system during equilibration	137
Figure 12.3	Sterics of AspAT residues	138

Theory and Simulation of Metal-Organic Materials and Biomolecules

Jonathan L. Belof

ABSTRACT

The emerging field of nanomaterials has raised a number of fascinating scientific questions that remain unanswered. Molecular theory and computer simulation are key tools to unlocking future discoveries in materials science, and various computational techniques and results toward this goal are elucidated here. High-performance computing methods (utilizing the latest supercomputers and codes) have been developed to explore and predict the chemistry and physical properties of systems as diverse as Metal-Organic Frameworks, discrete nanocubes, photoswitch molecules, porphyrins and several interesting enzymes. In addition, highlights of fundamental statistical physics, such as the Feynman-Hibbs effective partition function and generalized ensemble theory, are expounded and upon from the perspective of both research and pedagogy.

Chapter 1

Introduction

The emerging field of nanomaterials has raised a number of fascinating scientific questions that remain unanswered. Molecular theory and computer simulation are key tools to unlocking future discoveries in materials science, and various computational techniques and results toward this goal are elucidated here. High-performance computing methods (utilizing the latest supercomputers and codes) have been developed to explore and predict the chemistry and physical properties of systems as diverse as Metal-Organic Frameworks, discrete nanocubes, photoswitch molecules, porphyrins and several interesting enzymes. In addition, highlights of fundamental statistical physics, such as the Feynman-Hibbs effective partition function and generalized ensemble theory, are expounded and upon from the perspective of both research and pedagogy.

Of importance toward engineering new Metal-Organic Materials (MOMs) for hydrogen storage, Chapter 2 deals with computer simulations of hydrogen in a novel, highly charged MOF. These simulations revealed, for the first time, the existence of a dipolar fluid in the nanopore and the role of electrostatic induction in retaining attractive interactions deep within the pore. The principles of enhancing hydrogen uptake gleamed from this study will hopefully lead to the design of new materials capable of meeting the DOE goals. [1]

Chapter 3 highlights the development of a new hydrogen potential that is ac-

curate, transferable and (most importantly) suitable for condensed phase simulation. The potential was developed from first principles and includes anisotropic interactions as well as many-body polarization. The functional form of the potential is amenable to extant simulation codes, and has been demonstrated to yield accurate properties of hydrogen (second virial, equation of state) well into the liquid regime.

In Chapter 4 the newly developed hydrogen potential from Chapter 3 has been validated in an interfacial environment of a strongly sorbing surface. The system investigated is the well characterized MOF-5 [2] and Monte Carlo simulations were performed yielding the hydrogen uptake across a wide range of temperature and pressure, and found to be in quantitative agreement with experiment. The isosteric heat of adsorption, a thermodynamic measure of how strongly hydrogen binds to the MOF surface, has also been calculated and found to be in remarkable agreement. Analysis of the isothermal compressibility of hydrogen revealed that the hydrogen in MOF-5 is indeed liquid-like, with the H₂ taking on it's bulk liquid compressibility value when the excess uptake saturates. This potential validation study showed that when careful attention is payed to the development of potential energy functions, the methodology is accurate enough to be used as a predictive tool for studying hydrogen storage.

The theoretical and experimental research of a non-linear optical material (MEPEPY) is described in Chapter 5. This molecule is of special interest due to the fact that, when found in it's cis and trans isomers, it possesses both strong and weak field ligand characteristics – for this reason, MEPEPY (when coordinated to a high-spin metal such as iron or cobalt) may have applications as a molecular photoswitch. The ground-state energy surface for both isomers of MEPEPY was calculated using Density Functional Theory, and it was found that there is a substantial barrier to ground state interconversion between two distinct trans states. In addition, the energy difference between the isomeric forms revealed that there is the loss of a hydrogen

bond in solution. Finally, electrostatic analysis revealed that the difference in dipole between the two isomers was negligible, and thus electrorestriction would be minimal for PAC experiments.

Chapter 6 describes an interesting collaborative theoretical/experimental study on the stability and assembly of a novel Metal-Organic Cube (MOC). Electronic structures calculations were undertaken to answer the question of whether a nanocube was present in solution. Theoretical calculations, when combined with NMR and MALDI-TOF experiments, are shown to be a powerful tool for understanding non-crystalline discrete Metal-Organic Polyhedra.

It has been reported that any material capable of meeting the DOE 2012 criteria for hydrogen storage must reversibly sorb H₂ with a binding enthalpy of at least 20 kJ/mol. [3] In Chapter 7 theoretical calculations have been performed to address the following question: “what is the maximum binding enthalpy attainable through purely organic moieties such as benzene rings?”. Furthermore, the optimal size of the nanosquare giving rise to this maximal value of 14 kJ/mol has been determined and a new Metal-Organic Framework, ME193, has been synthesized to have a high isosteric heat of adsorption based upon this work.

The many-body polarization methods used in the aforementioned calculations result in an additional order of magnitude computational cost beyond the classical Monte Carlo simulations. [4] While this cost if not as high as performing *ab initio* MD, the additional demand requires HPC resources that are substantial. In Chapter 8, new methods of solving the self-consistent field equations are presented and shown to decrease this cost greatly. Iterative techniques using a novel ranking scheme as applied to Gauss-Seidel iterations, along with a preconditioning transformation, are shown to yield accurate polarization energies (suitable for Monte Carlo) even as a 2-step iteration scheme. It is expected that this technique will replace the more conventional Simultaneous Over-Relaxation method (i.e. linear mixing) and become

the dominant means of including electrostatic polarization in both materials and biological simulation.

In Chapter 9, a methodology and codebase is presented that allows the rotational energy levels of hydrogen to be calculated in the vicinity of a surface. The rotational energy levels are found through direct diagonalization of the hamiltonian in a spherical harmonic basis, for each H₂ subject to the potential energy surface of the material. Tunnel splitting effects are demonstrated, as well allowing for the inclusion of *ortho/para* nuclear spin states to be included in GCMC sorption studies. This code has been implemented into MPMC and may be used to calculate rotational Raman or Inelastic Neutron Scattering (INS) spectra.

As seen in Chapters 3 and 4, inclusion of quantum nuclear dispersion is critical to attaining quantitative accuracy with experiment under conditions of 77 K. [2, 5] While a common method of including these effects is through path integral Monte Carlo, yet another straight-forward approach is through the Feynman-Hibbs effective potential formalism. [6] However, both PIMC and the FH effective approach have only been derived for the canonical ensemble. In Chapter 10, we derive an effective partition function for the microcanonical ensemble, suitable for use with Microcanonical Monte Carlo [7, 8] or Molecular Dynamics methods.

Chapter 11 deals with the development of an alternative approach to deriving the partition function of statistical mechanics. This approach is unique in that it makes direct connection with the fundamental notions of entropy and thermodynamics, and offers the appeal that the partition function for any ensemble may be derived easily and in formulaic fashion. In addition, connections are made to generalized ensemble theory and its use in both pedagogy and research is discussed.

Molecular dynamics studies of myoglobin and aspartate aminotransferase are described in Chapter 12 and shown to be an effective and accurate tool for determining the molar volume of proteins in solution.

Finally, the Massively Parallel Monte Carlo codebase (MPMC), developed by the author in order to perform many of the calculations in this work, is described in Chapter 13.

Chapter 2

On the Mechanism of Hydrogen Storage in a Metal-Organic Framework (MOF) Material

Monte Carlo simulations were performed modeling hydrogen sorption in a recently synthesized metal-organic framework material (MOF) that exhibits large molecular hydrogen uptake capacity. The MOF is remarkable because at 78 K and 1.0 atmosphere it sorbs hydrogen at a density near that of liquid hydrogen (at 20 K and 1.0 atmosphere) when considering H₂ density in the pores. Unlike most other MOF's that have been investigated for hydrogen storage, it has a highly ionic framework and many relatively small channels. The simulations demonstrate that it is both of these physical characteristics that lead to relatively strong hydrogen interactions in the MOF and ultimately large hydrogen uptake. Microscopically, hydrogen interacts with the MOF *via* three principle attractive potential energy contributions: Van der Waals, charge-quadrupole and induction. Previous simulations of hydrogen storage in MOF's and other materials have not focused on the role of polarization effects, but they are demonstrated here to be the dominant contribution to hydrogen physisorption. Indeed, polarization interactions in the MOF lead to two distinct populations of dipolar hydrogen that are identified from the simulations that should be experimentally discernible using, *e.g.* Raman spectroscopy. Since polarization interactions are significantly enhanced by the presence of a charged framework with narrow pores,

MOF's are excellent hydrogen storage candidates.

2.1 Introduction

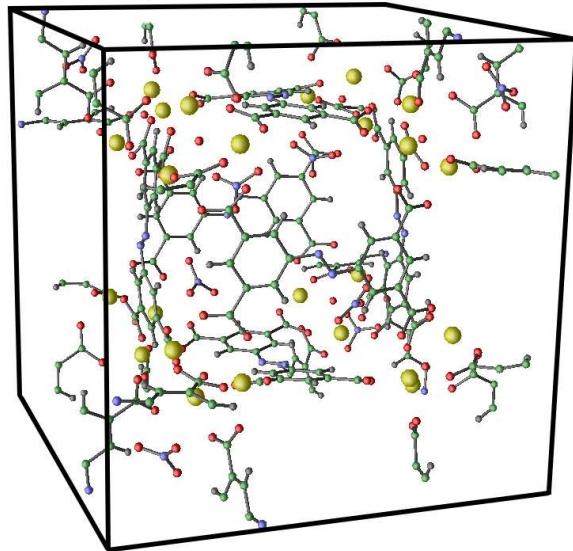
A major obstacle in achieving a hydrogen-based fuel economy is the ability to store and transport molecular hydrogen safely - at reasonable temperatures and pressures. For example, at one atmosphere, hydrogen does not liquefy until 20 K [9] because of its relatively weak intermolecular interactions, making the transport of neat hydrogen difficult. Thus, finding materials capable of storing large amounts of diatomic hydrogen is a promising avenue. The challenge is that hydrogen typically interacts weakly with its environment. However, hydrogen molecules can interact strongly with some materials, by undergoing chemisorption or dissociation, but such materials are typically inadequate for hydrogen storage because it is difficult to release the stored gas when it is needed. [10,11] On the other hand, materials that physisorb molecular hydrogen offer the promise of storing it under moderate conditions and the ability to release the hydrogen facilely. Such a material would require optimizing the attractive intermolecular interactions between the hydrogen and the condensed phase environment and, at the same time, enhancing H₂-H₂ interactions, thus leading to a favorable sorption enthalpy. The experimental [12–15] and theoretical [16–18] study of this unique problem has become an area of intense research in recent years, with a diverse range of prospective candidate materials being studied.

Metal-organic framework materials (MOF's) are a class of materials that have already shown promise for hydrogen storage. [19, 20] MOF's represent a novel class of solid crystalline materials that are built with rigid organic ligands linked to metal-containing clusters (also known as secondary building units or SBUs) [21] They can be constructed to have large surface areas, are relatively light weight and can be assembled from molecular building blocks with desired chemical functionality. Recently, a MOF was reported [19] (referred to here as *soc*-MOF) that was synthesized using

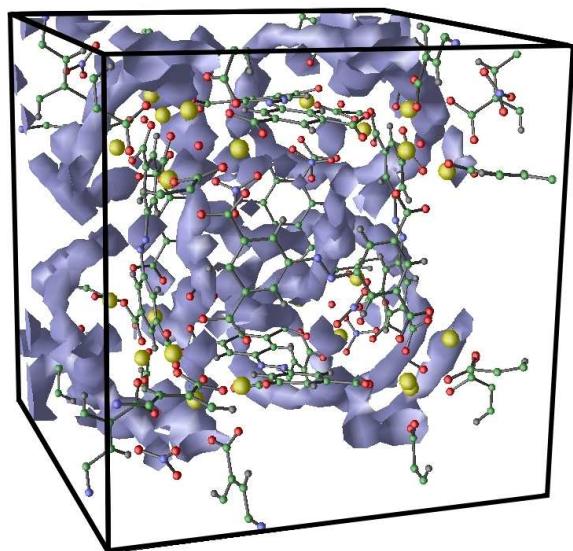
a novel indium trimer building block that resulted in a nanoporous material with an ionic framework, narrow channels (around 1 nm in diameter) and nanometer scale cages and capsules. The MOF has a rare *soc* topology [22] (*e.g.* not found in zeolites) characterized by its square-octahedral connectivity net. [23] The molecular formula, $[In_3O(C_{16}N_2O_8H_6)_{1.5}](NO_3)$, includes ionically-bound nitrate anions proximal to the indium trimer. The cationic indium of the SBU possesses an open-coordination site which, along with the bound nitrate anion, contributes toward giving the framework a highly ionic character. The unit-cell has been visualized with the cages and capsules located in the center, where each capsule contains tetrahedrally-positioned nitrate anions. The MOF possesses an estimated 57% extra-framework volume, a large Langmuir surface area of 1417 m^2 and $0.50\text{ cm}^3\text{ g}^{-1}$ pore volume. [19]

Figure 2.1 shows snapshots of *soc*-MOF alone and with a hydrogen density isosurface calculated from simulation (described below). The hydrogen is resident in the extra-framework volume and Figure 2.1 serves to visually highlight the pore topology within *soc*-MOF. Hydrogen uptake studies on *soc*-MOF show a large storage capacity with reversible sorption. For example, at 78 K and 1.0 atmosphere, the density of H_2 in the pores is approximately 0.05 g cm^{-3} while liquid hydrogen at its boiling point of 20 K has a density of 0.07 g cm^{-3} . [19] The experiments are conducted at liquid nitrogen temperature as a step toward finding superior hydrogen storage materials that will ultimately operate at room temperature. [1] This pore density of H_2 represents a compression factor, compared to the ideal gas volume under the same conditions, of approximately 100. Hydrogen sorption isotherms were measured at 78 K on *soc*-MOF and showed that the pores were filled at relatively lower pressure. [19] The MOF approached saturation at 1.0 atmosphere, indicative of the near liquid density of the sorbed hydrogen.

Thus, to investigate the physical basis of the large hydrogen uptake, canonical Monte Carlo simulations were performed on hydrogen in *soc*-MOF (at the experimen-



(a) *soc*-MOF empty



(b) *soc*-MOF populated

Figure 2.1: This illustration shows the *soc*-MOF both empty and populated *via* simulation of 113 H₂ with a polarizable model. The hydrogen density is represented by a 90% populated isosurface (generated by a custom module written for Data Explorer [24–26]) and has been rendered with a clipping plane for clarity. It should be noted that the indium (yellow) trimers are not depicted as being bonded, strictly in order to enhance the visibility of the structure.

tally observed hydrogen density at 78 K and 1.0 atmosphere). Since *soc*-MOF has a highly charged lattice with narrow pores, simulations were performed with and without explicit many-body polarization [27–30] contributions, in contrast to most extant molecular simulations. Molecular hydrogen’s attractive interactions in a heterogeneous condensed phase matrix (*e.g.* MOF’s, carbon nanostructures and zeolites) are dominated by three intermolecular contributions: Van der Waals, charge-quadrupole, and induction. As an estimate of the relative importance of quadrupole (that are typically accounted for) and polarization (usually neglected) terms, consider a polarizable site with a point quadrupole, both of the magnitude appropriate for molecular hydrogen placed 0.30 nm distant from a double-charged cation (representing, *e.g.* indium in a charge state similar to that in *soc*-MOF). Given the most favorable geometry for the quadrupolar interactions, the polarization energy is a factor of 4 larger and is always attractive. Clearly, polarization is not negligible and needs to be included in some fashion. Below, we demonstrate that including many-body polarization explicitly has a dramatic effect on the physisorption of hydrogen to *soc*-MOF, even compared to explicitly including induction as a one-body interaction.

Further, these observations suggest that polarization needs to be included in modeling hydrogen sorption in a variety of materials, essentially because the quadrupole interactions are relatively weak. Note, *ab initio* molecular dynamics (MD) simulations of molecular hydrogen in another MOF (MOF-5) have been previously performed [31] and implicitly include a reasonably accurate representation of polarization interactions. Unfortunately, the high cost of performing *ab initio* MD simulations limits such finite temperature investigations to very short times, although they are quite effective at finding minimum energy configurations.

2.2 Models and Methods

2.2.1 Molecular Simulation Parameters

A variety of theoretical methodologies have been used to study hydrogen sorption in nanostructured materials. [32–35] Recent studies include MD, grand-canonical Monte Carlo simulations to permit calculation of sorption isotherms, electronic structure studies to investigate binding mechanisms/affinities and semi-classical simulations that differentiate the interaction of *ortho* and *para* hydrogen with a material (relevant at very low temperatures where neutron diffraction studies are performed to characterize underlying hydrogen interaction sites). [16,31–33,35–57]

Here, classical Monte Carlo simulation methods were chosen to study *soc*-MOF in order to perform equilibrium finite temperature simulations and to be able to explicitly study the role of induction in hydrogen sorption. Note, a recent study modeling hydrogen sorption in MOF materials [16] has shown that, under the relatively low temperature conditions studied here, a quantum statistical mechanical description of the hydrogen structure is required to obtain quantitative accuracy. The earlier work included nuclear quantum dispersion effects *via* the computationally intensive path integral Monte Carlo simulations (PIMC), but did not include electronic induction. The results showed a decreased sorption of 10-15% relative to the purely classical simulations. However, that study also showed that the uncertainties in the Born-Oppenheimer potential surface were the dominant source of error in the simulations. Given that it is computationally prohibitive to perform PIMC simulations including polarization, the present study will focus on the role of electronic polarization within a classical model. Given the prior work, our estimates of hydrogen sorption strength may be a slight overestimate and the present work will focus on the critical role that polarization plays in shaping the Born-Oppenheimer surface. Including quantum dispersion effects will be the subject of future investigations.

Critical to any classical simulation based on empirical potentials is the careful selection of force field parameters. A minimal, yet effective, force field needs to include electrostatic, repulsive and van der Waals-type interactions; [58, 59] accurately describing the total potential energy surface is essential and the relevant parameters are not especially well-characterized for MOF's. Thus, striving for simplicity in this initial study, the need for framework intramolecular interactions was avoided by holding the scaffold rigid during simulation. Phonons are not thought to play an important role in hydrogen sorption, especially not at the temperatures considered here. [34] Lennard-Jones parameters, representing repulsive and van der Waals interactions between hydrogen atoms and framework were taken from the Universal Force Field; [60] this set of parameters was used in earlier MOF studies. [16, 35, 44, 57] The UFF interactions are parameterized for energetics between like atoms and all other interactions are accounted for in a standard way using (the approximate) Lorentz-Berthelot mixing rules. [58, 60]

When neglecting polarizability, electrostatic interactions in atomistic simulations stem from point (partial) charges assigned to the coordinate corresponding to the nuclear center of each atom. Since the true electrostatic potential energy surface of *soc*-MOF is unknown and *ab initio* calculations on the *soc*-MOF unit cell are computationally prohibitive, point charges were determined from electronic structure calculations on several model compounds that mimic the chemical environment of the MOF atoms. [35] The GAMESS *ab initio* simulation package was used to perform the Hartree-Fock quantum mechanical calculations. [61]

The structure of *soc*-MOF is characterized by corner-sharing octahedral indium trimers joined by bent 1,3-benzenedicarboxylate organic linkers. Three representative fragments are shown in Figure 2.2. They all produced similar partial charges to within 10% on average, and the electrostatic parameters used were derived from the largest of the candidates - the results are presented in Table 2.1.

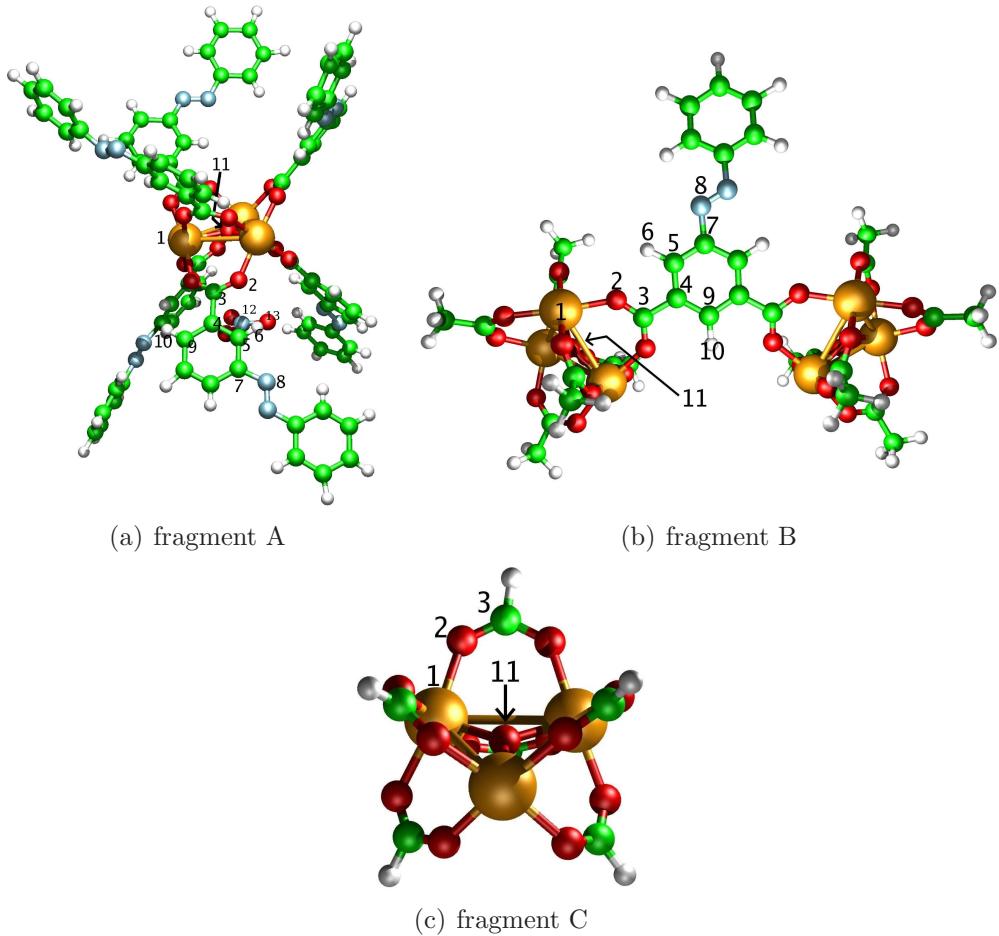


Figure 2.2: Illustrations depicting the fragments used in the *ab initio* calculation of the partial charges. Structure A is an SBU with six bound linkers, essentially one complete corner of a pore. Structure B, two SBUs coupled by one linker, can be regarded as the edge of a pore. Structure C is essentially the bare SBU used in the design of *soc*-MOF. Note that for clarity, the nitrate anions are not depicted in fragments B and C of the figure (but they were included in all calculations).

Atom	Label	Charge (e^-)
In	1	2.0697
O	2	-0.7588
C	3	0.9108
C	4	-0.1086
C	5	-0.2252
H	6	0.1366
C	7	0.3785
N	8	-0.2243
C	9	-0.1327
H	10	0.2167
O	11	-1.3978
N (nitrate)	12	0.6934
O (nitrate)	13	-0.4652

Table 2.1: Partial charges taken from the fragment 2.2(a) used in simulation of *soc*-MOF

Atom	Label	Charge (e^-)
In	1	2.228
O	2	-0.8216
C	3	0.9652
C	4	-0.1286
C	5	-0.2622
H	6	0.1443
C	7	0.4631
N	8	-0.2111
C	9	-0.2150
H	10	0.1854
O	11	-1.5424
N (nitrate)	12	1.029
O (nitrate)	13	-0.6424

Table 2.2: Partial charges for the fragment in Figure 2.2(b) for comparison

Study of the lattice shown in Figure 2.1 reveals the repetition of certain structures in a variety of geometries within the unit cell (for a detailed discussion of the structure see the literature). [19] The 448 atom unit cell may be produced from crystallographic symmetry operations from only twenty atoms. [19,22] Although this type of symmetry cannot be taken advantage of by quantum simulation packages, this repetition was used as a basis for deciding on representative chemical fragments. The largest fragments were chosen to include at least one complete metal center and an azobenzene linker; the smallest is the lone SBU. Adding hydrogen atoms where appropriate was required for chemical termination of the fragment boundaries. Measurements from the crystal structure indicate that the environments of the azobenzene linkers are essentially chemically equivalent in that their interface with the metal centers differs very slightly. Defining the azobenzene linkers as all chemically equivalent allows the entire unit cell to be defined in terms of only thirteen chemically different atoms. Using the electrostatic potential surface from the *ab initio* calculations, atomic point charges were fit using a standard algorithm. [61,62]

Since *soc*-MOF and our model fragments contain many-electron metal atoms (indium), the inner electrons require treatment *via* relativistic methods. Here we use semi-relativistic pseudopotentials, and two were compared, namely SBKJC and LANL2 [63–65] that include a different number of explicit electrons for indium (36 and 12, respectively) but gave similar results. The light atoms were treated at the 6-31G* level that produces over-polarized charges appropriate for condensed phase simulations (to account, in an effective way, for the effect of self-induction of the unpopulated lattice). [66] As a further test of the *ab initio* calculations, relativistic electronic structure calculations were performed on the smallest fragment (without need for pseudopotentials) using a third-order Douglas-Kroll transformation and the corresponding DK3 basis set. [67] The resulting charges agreed within 7.0% of those used in this study; the charges used herein are tabulated in Table 2.1. Note, the

condensed phase polarization of the neat MOF is included implicitly, while the polarization interactions in our simulations between the hydrogen and the MOF will be treated explicitly.

Partial charges for hydrogen were chosen to reproduce the quadrupole moment of the molecule. [40] The hydrogen is also treated as rigid and its high frequency vibration is not expected to contribute to sorption. [40] The electrostatic model is a three-point model with a charge located at the center of mass. The hydrogen atoms are separated by 0.741 Å and interact (between distinct molecules) *via* a polarizable many-body potential, along with the Coulombic and UFF-defined Lennard-Jones intermolecular pair potentials.

2.2.2 Polarizability Model

Molecular polarization was explicitly included in the Monte Carlo simulations by use of the Thole-Applequist model. [27–29] This model treats the system in terms of site (atomic) point dipoles that interact *via* many-body polarization equations. Once atomic point polarizabilities are fit to a training set of molecules the model has been shown to accurately reproduce molecular/system dipoles in a transferable (*i.e.* system-independent) manner. [27, 29] This model of explicit polarization has been successfully applied in numerous areas where inclusion of polarizable effects is paramount, such as vibrational spectroscopy, [30, 68, 69] liquid dynamics, [70–73] and biomolecules. [74, 75]

Consider a static electric field applied to a molecule. The induced dipole on this molecule will be equal to

$$\vec{\mu}_{mol} = \alpha_{mol} \vec{E}^{stat} \tag{2.1}$$

where α_{mol} is the 3×3 molecular polarizability tensor unique to that molecule. We

now consider the molecular dipole as being a sum of atomic point dipoles, one for each atom of the molecule. If we label each atomic point dipole vector $\vec{\mu}_i$ then we have

$$\begin{aligned}\vec{\mu}_i &= \alpha_i \vec{E}_i^{stat} \\ \vec{\mu}_{mol} &= \sum_i^N \vec{\mu}_i\end{aligned}\tag{2.2}$$

where α_i is the 3×3 site polarizability tensor and \vec{E}_i^{stat} is the electrostatic field at the site. In the Thole-Applequist model the system is treated as a collection of N dipoles along with a dipole field tensor $T_{ij}^{\alpha\beta}$ which contains the complete set of induced dipole-dipole interactions. This dipole field tensor, when contracted with the system dipoles, yields the (many-body) induced-dipole contribution to the electric field - this contribution is denoted here as \vec{E}^{ind} . Since the dipole field tensor (by construction) contains the entire induction contribution, we can assign a *scalar* point polarizability, α_i° , to each site rather than a polarizability tensor:

$$\alpha_i \vec{E}_i^{stat} = \alpha_i^\circ \left(\vec{E}_i^{stat} + \vec{E}_i^{ind} \right) \tag{2.3}$$

$$= \alpha_i^\circ \left(\vec{E}_i^{stat} - T_{ij}^{\alpha\beta} \vec{\mu}_j \right) \tag{2.4}$$

This equivalence can be demonstrated by reproducing the site polarizability tensors *via*

$$A\vec{\mu} = \vec{E}^{stat} \tag{2.5}$$

$$\vec{\mu} = B\vec{E}^{stat} \tag{2.6}$$

where $\vec{\mu}$ and \vec{E}^{stat} are supervectors formed by stacking the system dipole/field vectors:

$$\vec{\mu} = \begin{pmatrix} \vec{\mu}_1 \\ \vec{\mu}_2 \\ \vec{\mu}_3 \\ \vdots \\ \vdots \\ \vdots \\ \vec{\mu}_N \end{pmatrix}$$

$$\vec{E}^{stat} = \begin{pmatrix} \vec{E}_1^{stat} \\ \vec{E}_2^{stat} \\ \vec{E}_3^{stat} \\ \vdots \\ \vdots \\ \vdots \\ \vec{E}_N^{stat} \end{pmatrix}$$

and the matrices A and B are defined by

$$A = \left[(\alpha^\circ)^{-1} + T_{ij}^{\alpha\beta} \right] \quad (2.7)$$

$$B = A^{-1}$$

Just as A is a supermatrix composed of 3×3 block elements T_{ij} , B can be decomposed as [29]

$$B = \begin{pmatrix} B_{11} & B_{12} & \bullet & B_{1N} \\ B_{21} & B_{22} & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet \\ B_{N1} & \bullet & \bullet & B_{NN} \end{pmatrix}$$

where each block element B_{ij} is a 3×3 matrix. These block elements B_{ij} are the site polarizability tensors and thus characterize the site's response to an electric field. For example, the matrix B_η formed by summing the η^{th} row of B

$$B_\eta = (B_{\eta 1} + B_{\eta 2} + \dots + B_{\eta N})$$

determines the dipole response to a field for site η as a function of all N sites since (making use of Equation 2.6)

$$\begin{aligned}\vec{\mu}_\eta &= B_{\eta 1} \vec{E}_1^{stat} + B_{\eta 2} \vec{E}_2^{stat} + \dots + B_{\eta N} \vec{E}_N^{stat} \\ &= \vec{\mu}_\eta(1) + \vec{\mu}_\eta(2) + \dots + \vec{\mu}_\eta(N)\end{aligned}$$

whereby each μ term represents a contribution toward η 's dipole. Therefore, summing all of the ij blocks over the tensor components $\alpha\beta$ for an appropriate set of sites yields the molecular polarizability tensor: [29]

$$\alpha_{\alpha\beta}^{mol} = \sum_{i,j} (B_{ij})_{\alpha\beta} \quad (2.8)$$

The Applequist dipole field tensor [27] can be derived from first principles as

$$\begin{aligned}T_{ij}^{\alpha\beta} &= \nabla_\alpha \nabla_\beta \frac{1}{r_{ij}} \\ &= \frac{\delta_{\alpha\beta}}{r_{ij}^3} - \frac{3x^\alpha x^\beta}{r_{ij}^5}\end{aligned} \quad (2.9)$$

The most direct way to calculate the system dipoles is through Equation 2.6. However, since inversion of the $3N \times 3N$ matrix A is computationally efficient for only the smallest systems the dipoles must be solved for by an iterative method. The iterative method employed here makes an initial guess of $\vec{\mu}_i = \alpha_i^\circ \cdot \vec{E}_i^{stat}$ and then iteratively solves Equation 2.4 until convergence is achieved.

The Thole model introduces the additional consideration of treating each dipole as interacting with a well-behaved charge distribution $\rho(u)$ (in contrast to the Applequist model where the dipole field tensor is derived by considering a dipole interacting with a point charge, giving rise to Equation 2.9), which results in a mod-

ified form of the dipole field tensor. The net result of this modification is that the charge which induces each dipole is “smeared” at short range, and it is this additional structure that Thole added to the model which imparts transferability. One such exponential distribution [28] found to accurately and transferably [29] reproduce molecular dipoles for an associated series of dependent polarizabilities is

$$\begin{aligned}\rho(u_{ij}) &= \frac{\lambda^3}{8\pi} e^{-\lambda u_{ij}} \\ u_{ij} &= x_{ij} (\alpha_i^\circ \alpha_j^\circ)^{-\frac{1}{6}}\end{aligned}\tag{2.10}$$

where the free parameter λ has the effect of damping the dipole interactions near the regions of discontinuity that would otherwise exist in the Applequist model. The coordinate scaling of $x_{ij} \rightarrow u_{ij}$ is done for convenience in order to allow simple functional forms such as Equation 2.10 to be used for damping. Taking the exponential charge distribution into account, the modified dipole field tensor becomes [29]

$$\begin{aligned}\hat{T}_{ij}^{\alpha\beta} &= \frac{\delta_{\alpha\beta}}{r_{ij}^3} \left[1 - \left(\frac{\lambda^2 r_{ij}^2}{2} + \lambda r_{ij} + 1 \right) e^{-\lambda r_{ij}} \right] \\ &\quad - \frac{3x^\alpha x^\beta}{r_{ij}^5} \left[1 - \left(\frac{\lambda^3 r_{ij}^3}{6} + \frac{\lambda^2 r_{ij}^2}{2} + \lambda r_{ij} + 1 \right) e^{-\lambda r_{ij}} \right]\end{aligned}\tag{2.11}$$

The many-body potential energy due to the interaction of the induced dipoles (referred to as the polarization energy) is described by [28]

$$\begin{aligned}U_{pol} &= \sum_i \frac{1}{2} \vec{\mu}_i \cdot A \vec{\mu}_i - \vec{\mu}_i \cdot \vec{E}_i^{stat} \\ &= \sum_i \frac{1}{2} \vec{\mu}_i \cdot \vec{E}_i^{stat} - \vec{\mu}_i \cdot \vec{E}_i^{stat} \\ &= -\frac{1}{2} \sum_i \vec{\mu}_i \cdot \vec{E}_i^{stat}\end{aligned}\tag{2.12}$$

where it should be pointed out that \vec{E}_i^{stat} is not the total electric field, but rather only the static electric field due to the presence of the partial charges present in the system.

Calculating the polarization energy for the system amounts to self-consistently solving the dipole field equation for each atomic dipole vector $\vec{\mu}_i$ through an iterative process until a sufficient degree of precision is achieved. Thus, to make the calculations practical, efficient methods of solving the field equations were required. Typically, a simultaneous over-relaxation scheme (*i.e.* linear solution mixing) is used to improve the convergence rate for the iterative method of solution. [71] However, recently a number of multigrid techniques [76] have been applied to the Thole model for a one-dimensional system [77] and a similar Gauss-Seidel smoothing technique was applied here and found to reduce the number of iterations required for convergence by two-fold over linear mixing. The applied Gauss-Seidel numerical iteration method for a slowly converging process consists of updating the current dipole vector set for the k^{th} iteration step as the new dipole vectors become available:

$$\begin{aligned}\vec{\mu}_i^k &= \alpha_i^\circ \left(\vec{E}_i^{stat} - \hat{T}_{ij}^{\alpha\beta} \vec{\mu}_j^{k-1+\zeta} \right) \\ \zeta &= \begin{cases} 0, & \text{if } i < j \\ 1, & \text{if } i > j \end{cases}\end{aligned}\quad (2.13)$$

Dipoles were calculated to a precision of 10^{-4} Debye, and were subject to a 11.2 Å (half the unit cell length) spherical cutoff. Since the atomic point charges of the MOF were calculated by *ab initio* to implicitly include polarization, MOF-MOF self-polarization was disallowed by excluding MOF-MOF electric field interactions and only the induced field interactions between the H₂ and MOF atoms were calculated. All system dipoles were allowed to interact through the exponentially damped dipole field tensor (given by Equation 2.11) subject to the constraint of the spherical cutoff.

The polarizability tensor of diatomic hydrogen with an equilibrium bond dis-

tance of 0.741 Å was calculated by the restricted Hartree-Fock method with a correlation-consistent double-zeta basis set [78] (augcc-pVDZ) using GAMESS. [61] The atomic Thole polarizabilities for molecular hydrogen were then determined by fitting α^{mol} to the HF polarizability tensor form while at the same time yielding one-third of the trace equal to the experimentally measured [79] H₂ polarizability of 0.787 Å³; the values that best satisfied both criteria were found to be 0.2658 Å³ for H and 0.5865 Å³ for the center of mass site.

The SBU of *soc*-MOF contains indium which, in complex, has a partial charge of about In²⁺ as fit to the calculated electrostatic potential surface of the gas-phase fragment (see Table 2.1). While the polarizability of closed shell indium is known, [80] the polarizability of indium in the 2+ state has not been parameterized previously for the Thole model, nor has it been experimentally elucidated. In order to ascertain the polarizability, *ab initio* simulations were performed using finite-field calculations on In⁰,In¹⁺,In²⁺ and In³⁺. To assure that the results obtained from the finite-field calculations were reasonable the data was compared to polarizabilities calculated with an analytic Hessian for the In¹⁺ and In³⁺ states. Thus an estimate of 2.0 Å³ for In²⁺ was determined for these results; future research will be directed at applying fully-relativistic field equations to the SBU and fitting this parameter within the Thole model. The remainder of the MOF atoms were given the exponential polarizabilities and associated damping parameter as calculated by Duijnen et al. [29]

2.2.3 NVT Monte Carlo

Monte Carlo [81] simulations were performed on the H₂-MOF system at 78 K and with the experimentally determined hydrogen density of 113 molecules per unit cell, [19] with periodic boundary conditions applied. The total potential for the system is described by:

$$U = U_{elect} + U_{pol} + U_{LJ} \quad (2.14)$$

where U_{elect} is the electrostatic potential energy calculated from the Ewald field, U_{pol} is the polarization energy calculated from Equation 2.12 and U_{LJ} is the Lennard-Jones potential.

Monte Carlo moves were made by selecting an H₂ molecule at random, and performing a random rigid-body translation and rotation of the molecule. The MC move was then accepted or rejected according to the Metropolis function:

$$\min(1, e^{-\beta\Delta U}) \quad (2.15)$$

Using simple Monte Carlo moves is not especially desirable given the need to entirely reevaluate the many-body polarization energy after each small move. In an attempt to make global moves that would more efficiently explore phase space, several hybrid Monte Carlo schemes [82, 83] were implemented with forces that were computed from the non-polarizable potential. Unfortunately, the potential energy surfaces are sufficiently dissimilar and the approach failed; this result, however, was also an indication of the essential role of polarization interactions in this system.

The Monte Carlo algorithm and many-body polarization code were implemented within a package originally developed by the Klein group at the Center for Molecular Modeling at the University of Pennsylvania. [84–87] A cellular automata-based (rule 30) pseudo-random number generator was implemented for it's superior random number quality. [88, 89] The magnitude of the MC moves were adjusted to yield a 25% acceptance rate in order to minimize the correlation time. Autocorrelation of the polarization energy gave a correlation time of $\tau = 25,000$ Monte Carlo

steps. After a system equilibration time of 500,000 steps, atomic configurations and system dipoles were then sampled at intervals of 2τ for the collection of uncorrelated states of H₂ in the MOF. A total of 17.5 million MC steps were calculated on the massively parallel supercomputer LoneStar (Teragrid/University of Texas at Austin) to sample 350 uncorrelated configurations from the canonical ensemble.

Non-polarizable MC runs, calculating only the electrostatic and LJ terms in Equation 2.14, were also enacted for comparison. The magnitude of trial movement was able to be greatly increased and the potential energy correlation time was $\tau = 2,500$ MC steps - an order of magnitude less than that of the polarized system. Clearly, inclusion of the many-body polarizable potential greatly decreases the efficiency of the MC technique (when making simple one-body trial moves) *in addition to* the computational overhead introduced by the Thole model; the calculation of the system dipoles consumes approximately 95 % of the total CPU time. However, the computational cost is still far below that of performing *ab initio* dynamics.

The radial distribution function $g(r)$ was calculated between the hydrogen and various sites on the MOF and error analysis was performed to ensure their convergence. Isosurface analysis was performed over both the hydrogen population density and the hydrogen dipole magnitude.

2.2.4 Hybrid Monte Carlo

While MD simulations with many-body polarizable forces can yield insight into the behavior of complex systems, they are also reputed for their instability and complexity of implementation; development of stable and efficient polarizable dynamics is currently an active area of research in computational chemistry and physics. [70–73] The polarizable forces are troublesome and expensive to calculate; the polarization potential energy is (relatively) less problematic and is easier to compute *via* the dipole field equations.

Our initial approach was to perform a hybrid Monte Carlo [82] (HMC) calculation whereby the potential energy, especially including the polarization energy, was used in the Metropolis acceptance function but the system configuration was globally propagated by MD using *a non-polarizable force field*. In this way, we would circumvent the need to compute polarizable forces and yet properly sample the configuration space that would reflect polarization effects - all the while, getting the benefit of short correlation times and efficient sampling of phase space due to the global moves being made (as opposed to the long correlation times that are typical of many-body potentials in the standard Monte Carlo scheme).

Unfortunately, the above HMC approach proved unworkable for a number of reasons. While the HMC method has been demonstrated to work well for Lennard-Jones fluids [83] and fermionic particles, [82] in a system with many degrees of freedom we found a difficulty in the standard momentum resampling scheme that is not, to our knowledge, specifically addressed in the literature. It was our experience that completely resampling the momenta hindered the molecular rotations of the system, even in the non-polarizable scenario, and the structure for bulk molecular fluids that we tested (such as hydrogen and water) did not accurately correspond to the known structure, as was evident by the comparison of radial distribution functions from MD and Monte Carlo. However, for a Lennard-Jones system (*i.e.* possessing only translational degrees of freedom) the complete momenta resampling scheme yields the correct structure. It has been clarified that a complete resampling of momenta from a Boltzmann distribution is not required, [90] and in fact we implemented an alternative momentum update recently given in the literature [91] whereby the momentum vectors can be less drastically perturbed and then a Metropolis evaluation

of the kinetic energy is performed:

$$\begin{aligned} \{\vec{p}'\} &= \{\vec{p}\} + \alpha \{\vec{\pi}\} \\ \min \left\{ 1, \exp \left[-\beta \left(\frac{p'^2}{2m} - \frac{p^2}{2m} \right) \right] \right\} \end{aligned}$$

where $\{\vec{p}\}$ is the set of momentum vectors in the system, $\{\vec{\pi}\}$ is a set of vectors randomly sampled from a Boltzmann distribution and α is a free parameter used to tune the acceptance ratio. This update scheme restored the correct rotational dynamics and structure for the bulk systems studied with HMC. While the prime motivation for the initial development of alternative momentum resampling schemes has been to dispensably improve the efficiency, it is our view that less drastic momentum resampling is requisite in simulating systems with rotational degrees of freedom. However, a second (unfortunately insurmountable) difficulty with utilizing HMC to study *soc*-MOF was in using a polarizable MC potential with non-polarizable dynamics: in situations where the polarization effects are significant (e.g. *soc*-MOF at low temperature), the polarizable and non-polarizable potential energy surfaces are simply discrepant enough to prevent the adequate sampling of phase space (this again stresses the importance of polarization interactions for H₂ in *soc*-MOF). Thus, the computationally demanding standard Monte Carlo scheme was implemented.

2.3 Results and Discussion

To demonstrate the effect of polarization on sorption in *soc*-MOF, Figure 2.3 shows the radial distribution function between the sorbed hydrogen and indium ions both when induction is included and neglected. The polarization interactions strongly influence the structure of the sorbed hydrogen in the region of the metal ions. The figure also shows the distribution function when both charge-quadrupole and induction effects are neglected - in this case hydrogen is interacting as a Lennard-Jones

species with the MOF framework. Thus, the charge-quadrupole interactions are also making an important contribution to the sorption structure. While the effect is less dramatic, the radial distribution function shown in Figure 2.4 between the hydrogen and azobenzene nitrogen shows that the polarization effect of the azobenzene is also significant.

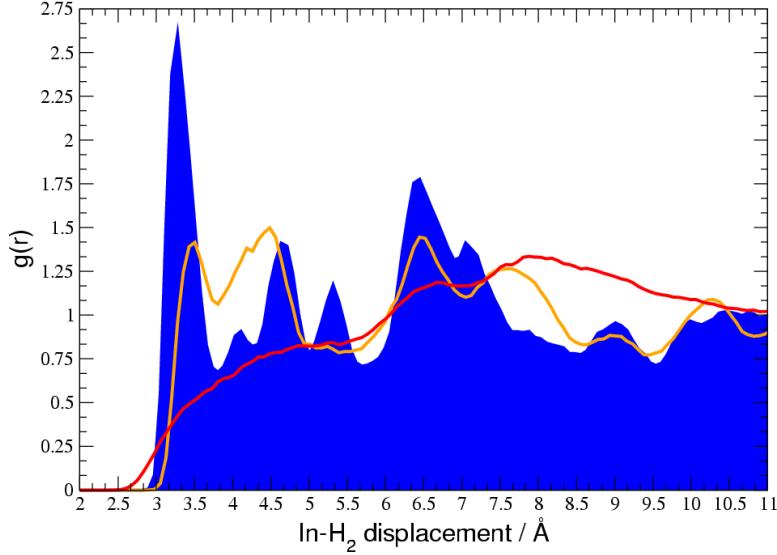


Figure 2.3: Radial distribution functions between the center of mass for the hydrogen to the indium under experimental conditions for three different potentials: Thole-Applequist many-body polarizable potential (blue), non-polarizable potential (orange) and Lennard-Jones only (red).

Further insight is gained by examining the distribution of H_2 induced dipoles that are produced by the field from the charges on the MOF (previous simulations in this study show that the contribution toward MOF polarization from the quadrupolar hydrogen-hydrogen interactions are negligible - yet, for completeness, they were also included here). Figure 2.5 plots the distribution of induced dipoles that is approximately a bi-modal Gaussian distribution with a dominant low dipolar (78% of the population with a mean dipole 0.18 Debye) and high dipolar species (22% of the population with a mean dipole 0.33 Debye). Analysis of molecular dipole magnitude isosurfaces reveals that the low dipolar population corresponds to spatial regions lo-

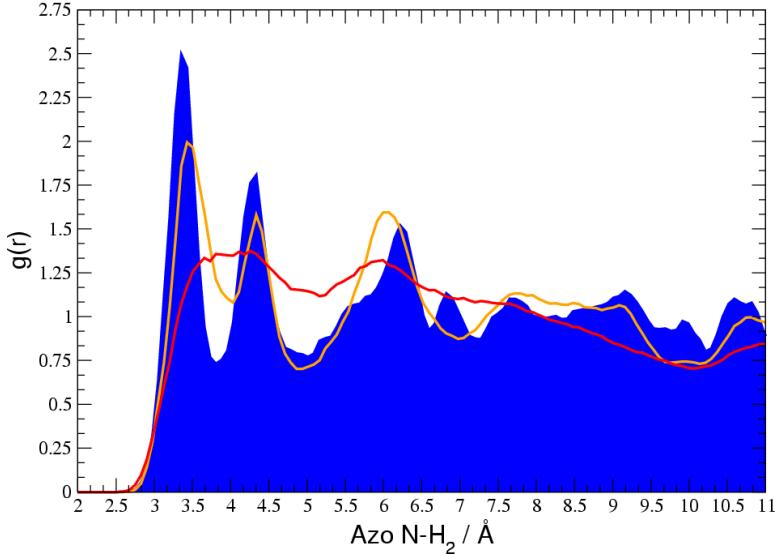


Figure 2.4: Radial distribution functions between the center of mass for the hydrogen to the azo nitrogen under experimental conditions for three different potentials: Thole-Applequist many-body polarizable potential (blue), non-polarizable potential (orange) and Lennard-Jones only (red).

calized in the vicinity of the open coordination sites of the SBU's (shown in Figure 2.6), while the high dipolar population is localized in the middle of the window formed by the azobenzene linkers. The reason for the high dipole distribution being associated with the azobenzene linkers seems to be due to the fact that the location of the window is geometrically proximal to all of the dominantly charged structures, namely the indium complex, nitrate anion and azo linkage. Enhancement of the first neighbor peak of the radial distribution function to the nitrogen of the azobenzene, shown in Figure 2.4, would seem to support this. The low dipolar population is associated with the electric field of the positively charged and highly polarizable indium ions of the SBU. The open coordination site of the indium also permits hydrogen to strongly associate with the metal ions as shown in Figure 2.1.

To put the magnitude of the dipoles in context, if we consider the hydrogen in the MOF a polar diatomic liquid with a dipole characteristic of the dominant species of 0.18 Debye, this magnitude is comparable to the permanent dipoles of NO

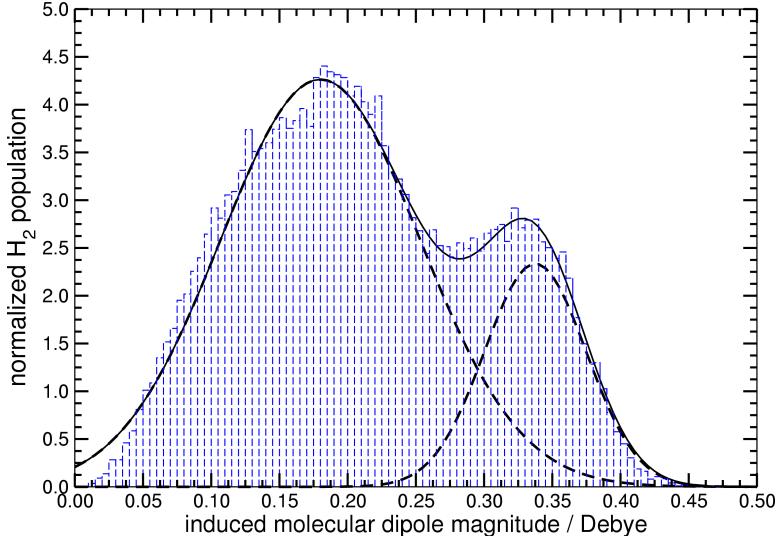
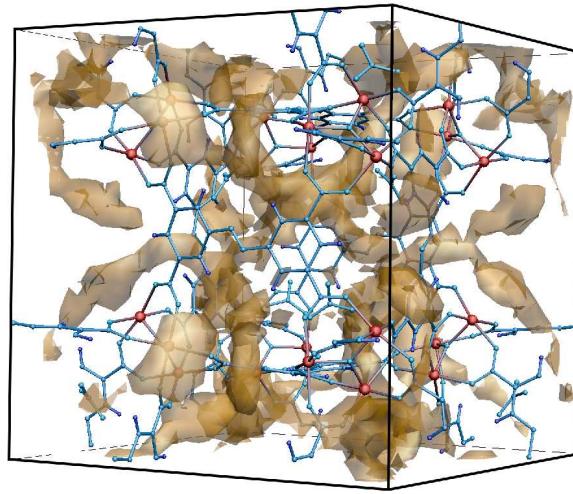


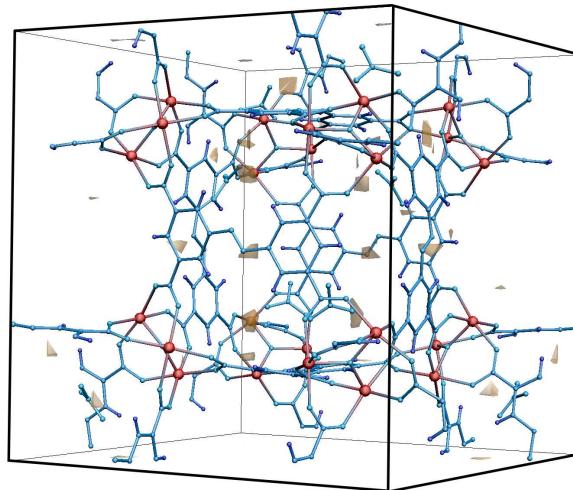
Figure 2.5: Bi-modal H_2 molecular dipole magnitude distribution for *soc*-MOF at 78 K under the experimental density. The figure presents a fit of the dipole distributions to two decomposed Gaussian distributions.

(0.16 Debye) or CO (0.11 Debye) that have ambient boiling points of 121 K and 82 K respectively. Thus, while neat hydrogen at 78 K has a negligible induced dipole (and is essentially an ideal gas with $\text{PV}/\text{NkT} = 1.00$), [9] the hydrogen in the MOF experiences mutual, many-body, dipolar attractions. Thus, it is reasonable that it is nearly condensed in the MOF pores at the conditions considered here and in the experiment at 78 K.

To assess the importance of polarization effects in a MOF with a less polar framework, we used the same force field parameters and charge fitting protocols to develop a potential energy surface for the much studied IRMOF-1 (a.k.a. MOF-5). [20] Simulations including many-body polarization for IRMOF-1 [20] reveal no significant change in radial distribution functions for the hydrogen to the metal-centered SBU when compared to simulations neglecting induction. MOF-5, which sorbs substantially less hydrogen than the new material studied here under like conditions, also does not possess an open coordination site on its zinc SBU and thus produces a more weakly polarizing field in comparison with *soc*-MOF. Most importantly, MOF-5 does



(a) 0.18 D isosurface



(b) 0.33 D isosurface

Figure 2.6: Isosurfaces showing the low-dipolar and high-dipolar hydrogen species corresponding to the respective peaks of the bi-modal hydrogen dipole distribution. Note the location of the indium open coordination sites (red) and the nitrate anions (lower corners of the box most visible) with respect to the high-dipolar isosurface.

not possess narrow channels or a highly polar framework, but rather has an open topology with larger void spaces. This comparison strongly suggests that MOF's, like MOF-5, with large pores are not the best target materials for super hydrogen storage. *The key result of this study is that hydrogen needs to interact sufficiently strongly with a MOF to produce a dipolar fluid with a characteristically higher condensation temperature.*

Recently, hydrogen storage in a Mn-containing MOF was studied [92] in which a similar sorption isotherm as that of *soc*-MOF was measured; this MOF also possesses relatively narrow channels and a polar framework. This cubic topology MOF contains an Mn^{2+} open coordination site on the SBU; it is not unreasonable then to assume that the high sorption capacity of hydrogen in both that material and *soc*-MOF are correlated with the structural motif of the SBU. The dipole isosurfaces generated by this work would suggest that the open coordination sites serve to polarize the hydrogen under the experimental conditions.

Using the polarizable potential, the integral enthalpy of adsorption for the hydrogen in *soc*-MOF was calculated by:

$$\Delta H^a = E(MOF + H_2) - [E(MOF) + E(H_2)]$$

Note that the PV terms are negligible in this case. To be clear, the energy for the MOF with N hydrogen molecules present is calculated and then subtracted from the combined energy for the MOF alone and the neat gas of N hydrogen molecules; the energy is then divided by the number of moles corresponding to the N molecules. The integral enthalpy was calculated at both the saturated state (113 hydrogen molecules per unit cell) and the zero-loading limit (a single hydrogen molecule per unit cell).

It is important to note that this differs from the experimentally measured

isosteric heat of adsorption, q_{st} in that we are calculating the integrated energy per mole of adsorbent. The numbers are expected to be very similar, however, for the following reasons: the isosteric heat is given by

$$q_{st} = kT^2 \frac{\partial \ln P}{\partial T}$$

while the differential enthalpy is equal to

$$\Delta h^a = kT^2 \frac{\partial \ln f}{\partial T}$$

where $f = \phi P$ is the fugacity and ϕ is the fugacity coefficient. [16,93,94] The condition for these quantities to be equal is that the temperature dependence of the fugacity coefficient is small, namely: $\frac{1}{\phi} \frac{\partial \phi}{\partial T} \ll \frac{1}{P} \frac{\partial P}{\partial T}$, a requirement that is clearly met under the sorptions conditions considered here where the hydrogen structure is not changing dramatically with temperature (note that even for an ideal gas, the $\frac{\partial \ln P}{\partial T}$ term has a strong $\frac{1}{T}$ dependence). To complete the argument, note (considering *soc*-MOF) there is a weak dependence on loading in the isosteric heat measurements; [19] and in the limit that $q_{st} \approx \Delta h^a$ is constant it is equal to the the integral adsorption heat, making a comparison reasonable. [93,94]

ΔH_{sat}^a was found to be -10.645 ± 0.188 kJ mol $^{-1}$ while the unsaturated state yielded ΔH_{unsat}^a of -14.354 ± 1.685 kJ mol $^{-1}$; the relatively constant ΔH^a values indicate the retention of strong attractive interactions even at higher loadings, which is also supported by the experimentally measured isosteric heats. [19] Polarization contributed 23% of the total potential energy at saturation and 27% when unsaturated. These enthalpies appear large compared to the experimental measurements that give

a relatively constant isosteric heat of sorption of about 6.5 kJ mol⁻¹. [19] Thus, as a control, the integral heats of sorption in IRMOF-1 at the low density limit were calculated to compare with existing experimental [20] and theoretical [16] values. The result was $\Delta H_{unsat}^a = -5.998 \pm 0.298$ kJ mol⁻¹, in good agreement with extant values. Including polarization gave a value of $\Delta H_{unsat}^a = -6.343 \pm 0.277$ kJ mol⁻¹, consistent with the result that including polarization in simulating saturated IRMOF-1 did not significantly change the hydrogen structure within the MOF. As was pointed out in a previous work [16] it is often difficult to say why there is a discrepancy between the experiment and theory in one case *vs.* another [16] but all of the evidence does clearly support that polarization is an critical factor influencing sorption in *soc*-MOF.

2.4 Conclusions

The results of this study suggest desirable MOF design characteristics for hydrogen, and gas storage in general - MOF's are promising candidates for gas storage/sequestration. For example, one would expect CO₂ sorption in *soc*-MOF to be quite strong given the significantly higher molecular quadrupole and polarizabilty. This study suggests that MOF's should have relatively small pores and interconnected pores with high surface area to create strong MOF-H₂ interactions and, thus, indirectly H₂-H₂ attractions. To promote these interactions, the MOF also needs to be locally polar with large charge separations on its surface sufficiently far apart to allow hydrogen molecules to be sensitive to the dipolar interface. Further, while the surface area needs to be large, the open spatial network should not be so expansive that hydrogen molecules farthest from the MOF surface do not possess significant induced dipoles and charge-quadrupole forces. For example, if a MOF were to possess a large surface area due to sizable pores, hydrogen toward the center of the void will be similar to neat hydrogen with characteristically weak intermolecular interactions and a correspondingly lower condensation temperature. There is, however, a trade-off

between having larger void volumes that produce a lower molecular weight material (but do not promote strong sorptive forces) versus having a highly nanostructured pore system (with correspondingly more material per unit volume, but strong scaffold-H₂ interactions).

These initial studies have provided significant insights into the nature of hydrogen interactions in nanoporous, polar MOF materials. The results presented also suggest several future avenues of inquiry. Foremost, we will proceed to calculate sorption isotherms for our model using a Widom insertion method [95–97] that can be compared with experiment. This will also serve to further calibrate the potential energy surface of our molecular mechanics model by, for example, giving us the system pressure at the experimentally observed hydrogen loading. We can then proceed to mutate the MOF in an experimentally plausible fashion in an attempt to increase its hydrogen storage capacity. Lastly, this study suggests that including polarization in modeling other extant and future MOF’s can give reliable physical insight into the mechanism of gas storage.

Chapter 3

An Accurate and Transferable Intermolecular Diatomic Hydrogen Potential for Condensed Phase Simulation

An anisotropic many-body H₂ potential energy function has been developed for use in heterogeneous systems. The intermolecular potential has been derived from first principles and expressed in a form that is readily transferred to exogenous systems, *e.g.* in modeling H₂ sorption in solid-state materials. Explicit many-body polarization effects, known to be important in simulating hydrogen at high density, are incorporated. The analytic form of the potential energy function is suitable for methods of statistical physics, such as Monte Carlo or Molecular Dynamics simulation. The model has been validated on dense supercritical hydrogen and demonstrated to reproduce the experimental data with high accuracy.

3.1 Introduction

The development of intermolecular H₂ potential energy functions has a long history. [98] The topic has generated renewed interest in recent years due to the increased theoretical study of hydrogen storage materials. [1] In this work, we present our own revisit of the issue to include transferability, necessary for the practical simulation of hydrogen in heterogeneous materials, as well as anisotropic many-body effects which have been shown to be important in modeling hydrogen at high density.

For example, in modeling hydrogen sorption in Metal-Organic Framework materials (MOFs), a transferable potential is required that includes the ability to capture anisotropic many-body effects in a consistent, effective manner. [99, 100]

To date, most hydrogen sorption studies of materials have focused on isotropic H₂ potentials [101–104] in particular that of Buch since it is easily transferable and accurately reproduces the bulk thermodynamic properties of hydrogen; in addition, it has been shown to calculate the correct uptake of hydrogen in weakly-interacting materials. However, in heterogeneous environments where electrostatic quadrupole and induced dipole effects are non-negligible such an approach cannot reproduce the correct behavior: the isotropic potential includes these effects in a mean-field fashion, appropriate only for systems in which dispersion interactions dominate.

Previous anisotropic potentials [105–111] have focused on neat H₂, with the most notable being the highly accurate potential of Diep and Johnson; however, it is not clear how such specialized forms can be systematically mixed with chemically different environments without extensive reparameterization. [112] While the anisotropic potential of Darkrim and Levesque is both transferable and includes the quadrupole term, it has been shown to overestimate the attractive part of the potential and neglects induced many-body effects. [99]

Herein we apply high-level electronic structure calculations to the H₂-H₂ dimer and then fit an anisotropic functional form to the *ab initio* surface; electron correlation methods have been utilized as they have been shown to be crucial in accurately describing the relatively weak H₂-H₂ interaction. [105, 106] This functional form contains quadrupole as well as electronic repulsion/dispersion terms that are transferable using the Lorentz-Berthelot mixing rules.

Several existing potentials [104, 109, 113] include non-additive three-body dipole terms *via* Axilrod-Teller-Muto [114] dispersion, which are known to influence the structure of hydrogen at high density. In this work we apply a many-body polar-

ization term to hydrogen and demonstrate improvement in the equation of state at high density. Further, explicit many-body polarization interactions have been demonstrated to be essential in modeling H₂ sorption in polar MOFs [115] and the present polarization model is also transferable to such systems.

3.2 Methods

3.2.1 Born-Oppenheimer Surface

Construction of the Born-Oppenheimer surface for the H₂ dimer proceeds from optimizing the RHF [116–120] wavefunction parametrically as a function of the nuclear coordinates. The hydrogen molecule was approximated as a rigid rotor with a bondlength of 0.742 Å, corresponding to the value determined by rotational spectra. [121, 122] The surface was taken over the domain of center-of-mass separation for the rotors from 2.0 to 10.0 Å in 0.1 Å increments. A subspace of the complete surface was taken along four unique relative orientations; the orientations were chosen as being energetically and geometrically distinct (shown pictorially in Figure 3.1) and the four computed surfaces produce an average in agreement with experiment. [104] All electronic structure calculations were performed using the quantum chemistry code PC GAMESS. [61, 123]

Since the H₂-H₂ dimer interaction is dominated by electron correlation, [105, 106] the RHF energy calculations included Møller-Plesset perturbation terms [124, 125] to fourth-order, [126] including triplet states; [127] prior work in this area has sufficiently demonstrated that, for the hydrogen dimer, MP4 energies are in agreement with the CCSD(T) level of theory. [105] The set of basis functions employed in the solution of the wavefunction were those of Dunning [128] (aug-cc-pVTZ/QZ) with the energy eigenvalues then extrapolated to the complete basis set limit. [129] The effect of basis set superposition error was corrected by the counterpoise method. [130]

The computed energy surfaces are depicted in Figure 3.1 along with the isotropically averaged surface.

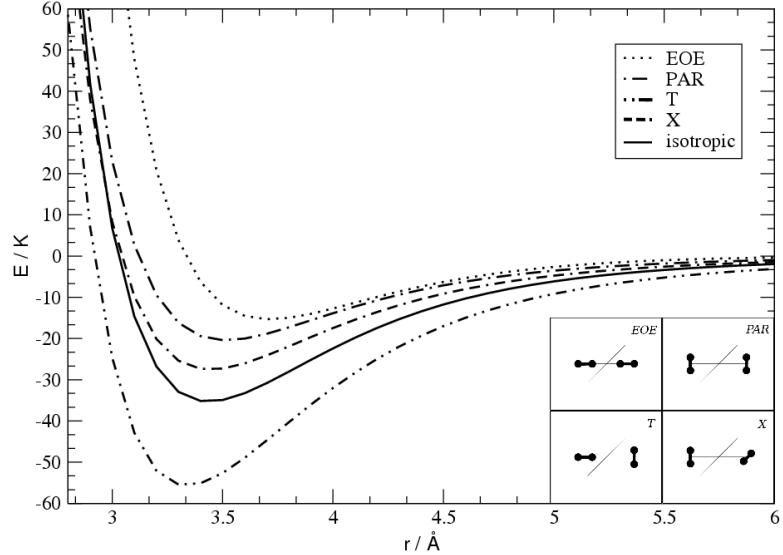


Figure 3.1: *Ab initio* energy curves of four distinct relative H_2 orientations (as a function of the rotor center-of-mass separation) along the Born-Oppenheimer surface. The orientations chosen were end-on-end (EOE), parallel (PAR), T-configuration (T) and X-configuration (X).

The electric quadrupole value for H_2 was calculated from the electronic wavefunction (MP4-SDTQ/aug-cc-pVQZ) and found to be $0.664 \text{ D} \cdot \text{\AA}$, a value comparable to other investigations. [121, 131] The molecular (static) polarizability tensor of the monomer was determined *via* TDHF [132] since it has been shown to accurately reproduce non-linear optical properties at frequencies away from resonance; the resultant components are listed in Table 3.1. The components of the TDHF polarizability tensor were then rescaled such that the isotropic polarizability matched the experimental value [133] of 0.787 \AA^3 subject to the constraint that the ratio of the XX to ZZ TDHF components remain invariant. Prior work has shown that, for H_2 , the non-adiabatic contributions to both the quadrupole and polarizability are negligible. [131, 134]

component	TDHF / Å ³	rescaled / Å ³
XX	0.6831	0.6945
YY	0.6831	0.6945
ZZ	0.9561	0.9720
isotropic	0.7741	0.7870

Table 3.1: H₂ molecular polarizability tensor (molecular bond axis aligned along the Z axis) calculated *via* time-dependent Hartree-Fock along with it's rescaled components. The components were rescaled such that the isotropic polarizability, $\frac{1}{3}Tr\{\alpha\}$, matched experiment subject to the constraint that the ratio of TDHF values, XX/ZZ , be preserved.

3.2.2 Many-body Polarization

Here we briefly review the Thole-Applequist polarization model - details can be found elsewhere. [135–137] Briefly, the atomic point dipole, $\vec{\mu}$, of the i^{th} atom is:

$$\vec{\mu}_i = \alpha_i \vec{E}_i^{\text{stat}} \quad (3.1)$$

where α_i is an atomic site polarizability tensor and \vec{E}_i^{stat} is the electrostatic field vector. We can decompose the dipole into separate *static* and *induced* contributions as follows:

$$\begin{aligned} \vec{\mu}_i &= \alpha_i^\circ \left(\vec{E}_i^{\text{stat}} + \vec{E}_i^{\text{ind}} \right) \\ &= \alpha_i^\circ \left(\vec{E}_i^{\text{stat}} - T_{ij} \vec{\mu}_j \right) \end{aligned} \quad (3.2)$$

where α_i° is a *scalar* point polarizability and T_{ij} is the dipole field tensor which, when contracted with the dipole $\vec{\mu}_j$, represents the electrostatic contribution of the j^{th} dipole toward inducing the i^{th} dipole.

The dipole field tensor is constructed based upon the positions and scalar point polarizabilities of the system, and can be derived from classical electrostatic principles as:

$$T_{ij}^{\alpha\beta} = \frac{\delta_{\alpha\beta}}{r_{ij}^3} - \frac{3x^\alpha x^\beta}{r_{ij}^5}$$

with an additional aspect of the model being that the second dipole may be taken to be a model distribution, with the intent that the discontinuity at short range will be avoided; here we employ the common exponential charge distribution [137] and associated damping parameter of 2.1304.

Equation 3.2 is a self-consistent field equation with respect to the dipoles, and hence must be solved iteratively. However, it is possible to recast the field equations in matrix form as:

$$A\vec{\mu} = \vec{E}^{stat} \quad (3.3)$$

where the matrix A is constructed from block matrices according to:

$$A = [(\alpha^\circ)^{-1} + T_{ij}] \quad (3.4)$$

The advantage to Equation 3.4 is that the solution is then exact (to numerical precision), without the additional complication of iteration and convergence. Upon inversion of the A matrix, the molecular polarizability tensor can then be easily determined by:

$$\alpha_{\alpha\beta}^{mol} = \sum_{i,j} (A_{ij}^{-1})_{\alpha\beta} \quad (3.5)$$

The polarizable aspect of the hydrogen model was developed as follows. With the molecule aligned along the Z axis, the polarizability tensor was calculated with the Thole model (i.e. Equation 3.5) and compared with the *ab initio* TDHF tensor; the scalar point polarizabilities being varied until the two tensors agree. The scalar polarizabilities, α° , were assigned at the same nuclear coordinates as the partial charges, Q , used in producing the quadrupole. The matrix in Equation 3.4 was constructed, inverted and then summed according to Equation 3.5 yielding a trial polarizability tensor. The resulting trial tensor was then compared to the rescaled tensor in Table 3.1 and the α° 's adjusted until the difference was minimized. The site polarizabilities were completely converged (*i.e.* the molecular polarizability tensor determined *via* Equation 3.5 matched the rescaled tensor in Table 3.1 to within all significant digits) and are listed in Table 3.2.

3.2.3 Potential Energy Function

The interest of this work is the development of a potential energy function for use in Monte Carlo or Molecular Dynamics simulation of hydrogen interacting with heterogeneous systems. In this context, the following functional form was chosen for the potential energy function:

$$U = U_{rd} + U_{es} + U_{pol} \quad (3.6)$$

where U_{rd} is the energy of electronic repulsion/dispersion, U_{es} is the electrostatic energy and U_{pol} is the many-body polarization energy given by:

$$U_{pol} = -\frac{1}{2} \sum_i^N \vec{\mu}_i \cdot \vec{E}_i^{stat} \quad (3.7)$$

where the site dipoles are found by either matrix inversion of equation 3.3 or iterative solution of the dipole field equations given by expression 3.2. For the purpose of fitting the potential energy function, we employed the matrix inversion method for maximal accuracy in the many-body potential since the computations are fast for a two-rotor system (however, in application toward condensed-phase systems the iterative method is used almost exclusively).

All of the free parameters of these functions have well-established ways of being “mixed”, thereby imparting transferability. It is the main contribution of this work that these functions have been fit to the first principles data presented.

The electrostatic energy follows from the quadrupole-quadrupole interactions between the hydrogen monomers. Given the quadrupole value of $0.664 \text{ D}\cdot\text{\AA}$ calculated from first principles and a bond length of 0.742 \AA , this corresponds to partial charges of $Q = +0.3732 e$ and the atomic sites and $-2Q$ at the center-of-mass.

site	R / Å	Q / e	$\alpha^\circ / \text{\AA}^3$	ϵ / K	$\sigma / \text{\AA}$
H2GP	0.000	-0.7464	0.69380	12.76532	3.15528
H2NP	0.363	0.0000	0.00000	2.16726	2.37031
H2EP	0.371	0.3732	0.00044	0.00000	0.00000

Table 3.2: Parameter fits for the intermolecular hydrogen potential, including many-body polarization terms. H2GP corresponds to the center-of-mass site, H2EP coincides with the true atomic locations (which, when combined with H2GP, provides the quadrupole) while H2NP contains the additional Lennard-Jones sites.

The short-range electronic repulsion and long-range dispersion energies are included in U_{rd} by use of the Lennard-Jones 12-6 potential function. Since the Lennard-Jones r^{-6} part includes a mean-field long-range polarization, it was necessary that U_{rd} be determined. With U_{es} and U_{pol} functionally held fixed, U_{rd} was varied *via* simulated annealing such that $(U - U_{BO})^2$ was minimized (where U_{BO} is the Born-Oppenheimer energy surface). The results of the potential energy function fitting are shown in Figure 3.2. Given the anisotropic nature of the Born-Oppenheimer potential

demonstrated in Figure 3.1, Figure 3.2 demonstrates that our relatively simple and transferable potential can capture the essential anisotropy. Interestingly, while the T orientation shows the largest deviation from the *ab initio* surface, even dramatically increasing the number of Lennard-Jones parameters did not significantly improve the fit. Note, given the large variation in the potential it is critical in modeling hydrogen in highly anisotropic environments that the potential capture these structural characteristics.

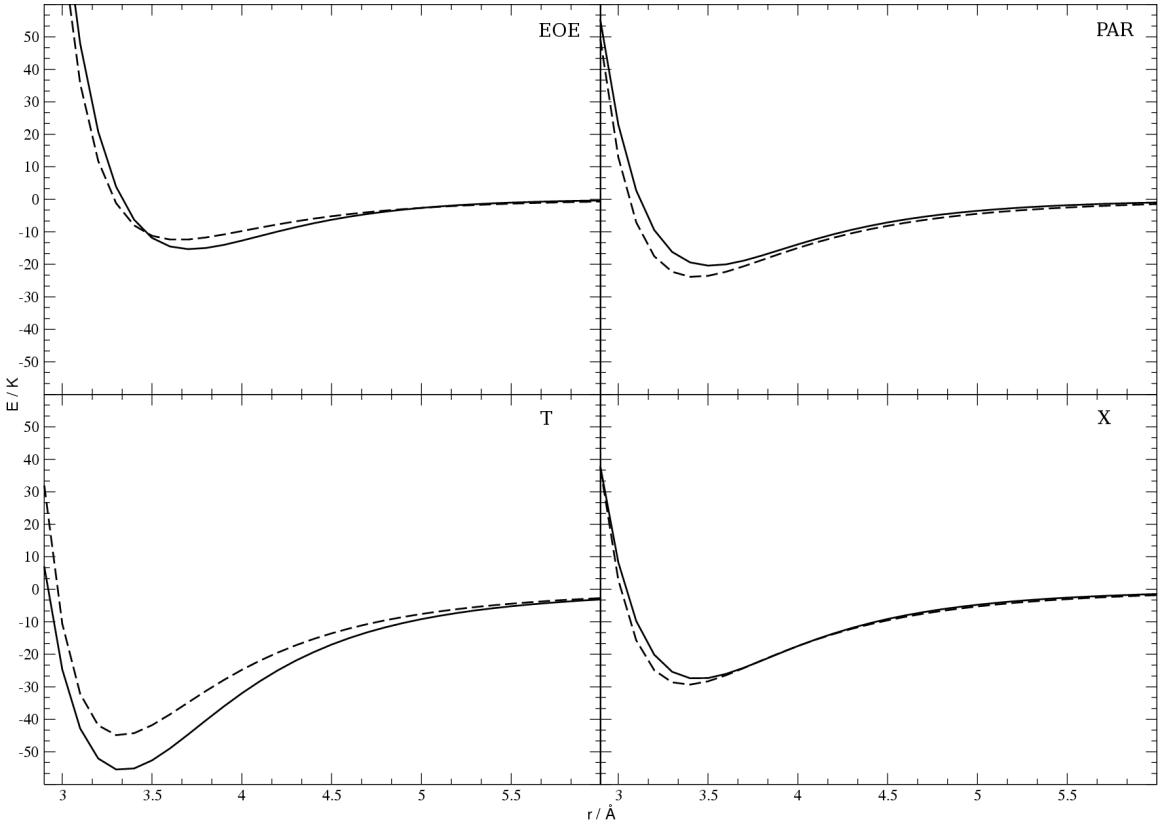


Figure 3.2: Potential energy curves found by fitting to equation 3.6 (dashed) versus the *ab initio* curves from Figure 3.1 (solid). The alternative non-polar form ($U_{pol} = 0$) that was also fit to the *ab initio* data is not shown for clarity since, with a slight exception to the T configuration, the curves are visually indistinguishable.

In the course of the parameter-space search, the Lennard-Jones sites on the hydrogen molecule were allowed to move off their nuclear centers. In addition, searches were made that constrained the sites to be both on and off the bond axis, with as

few as two and as many as nine simultaneous sites in an attempt to improve the fit; it was found, however, that three sites constrained to the bond axis were optimal in terms of having the minimized error (with only marginal improvement in the fit upon increasing the number of sites further). At the end of the fitting process, the parameters that minimized the error were found and are presented in Table 3.2. In order to also develop a non-polarizable potential, the process was repeated with $U_{pol} = 0$. In physical systems where the magnitude of electrostatic polarization is negligible, the non-polarizable parameters presented in Table 3.3 are desirable for the reduced computational cost.

site	R / Å	Q / e	ϵ / K	σ / Å
H2G	0.000	-0.7464	8.8516	3.2293
H2N	0.329	0.0000	4.0659	2.3406
H2E	0.371	0.3732	0.0000	0.0000

Table 3.3: Potential parameters for the pairwise model (neglecting polarization terms).

3.3 Model Validation

3.3.1 Second Virial Coefficient

Quantum mechanical corrections to the second virial coefficient of a gas can be systematically derived using the Wigner-Kirkwood distribution function. [138] To order h^4 , we have the semiclassical series:

$$\begin{aligned}
B_2(T) = & 2\pi \int_0^\infty dr r^2 (1 - e^{-\beta\phi}) \\
& + \frac{h^2}{24\pi\beta^3} \int_0^\infty dr r^2 e^{-\beta\phi} \left[\frac{\partial\phi}{\partial r} \right]^2 + \frac{h^4}{960\pi^3\beta^4} \int_0^\infty dr r^2 e^{-\beta\phi} \\
& \times \left\{ \left[\frac{\partial^2\phi}{\partial r^2} \right]^2 + \frac{2}{r^2} \left[\frac{\partial\phi}{\partial r} \right]^2 + \frac{10\beta}{9r} \left[\frac{\partial\phi}{\partial r} \right]^3 - \frac{5\beta^2}{36} \left[\frac{\partial\phi}{\partial r} \right]^4 \right\} \quad (3.8)
\end{aligned}$$

where $\phi(r)$ is an isotropic potential as a function of the separation. The intermolecular many-body potential developed here has been spherically averaged to produce an isotropic pair potential, $\phi(r)$, which is plotted in Figure 3.3 along with the well-known isotropic potentials of Silvera-Goldman [104] and Buch [103], both of which are known to closely match experiment.

The second virial coefficient, $B_2(T)$, was then calculated *via* equation 3.8. Numerical integration was performed for $r = 0.001 - 25 \text{ \AA}$ across a temperature range spanning from 50 to 500 K, in 10 K increments. The results are compared with experiment [133] (and the SG potential) in Figure 3.4.

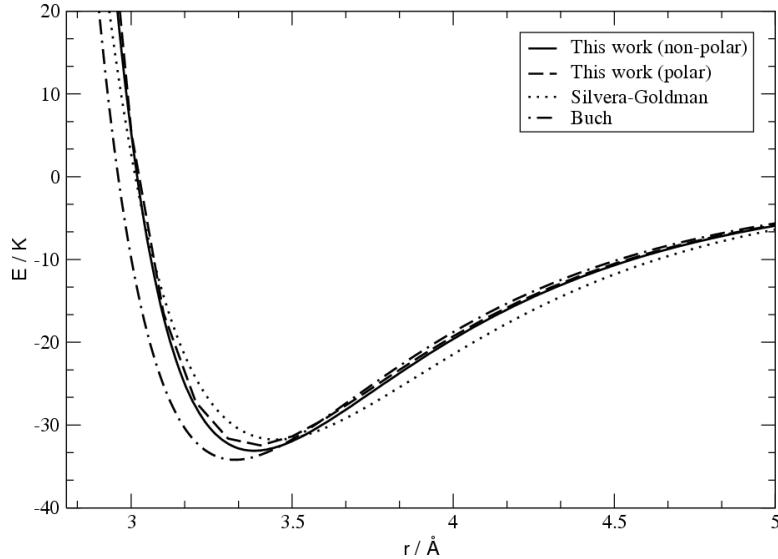


Figure 3.3: The isotropic projection $\phi(r)$ of the many-body potential U is shown along with the SG and Buch potentials.

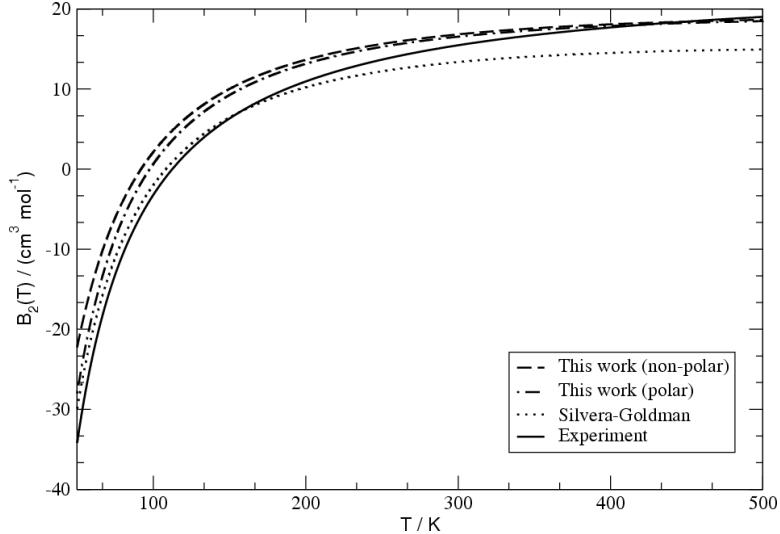


Figure 3.4: $B_2(T)$ from 50-500 K for the polarizable and non-polarizable potentials including Wigner-Kirkwood quantum corrections to order h^4 .

3.3.2 Equation of State

The equation of state was ascertained by calculating the average number of hydrogen molecules, \bar{N} , via sampling of the grand canonical ensemble for a corresponding range of chemical potential. The following statistical mechanical expression was numerically estimated by Grand Canonical Monte Carlo: [139, 140]

$$\bar{N} = \frac{1}{\Xi} \sum_{N=0}^{\infty} e^{\beta\mu N} \left\{ \prod_{i=1}^{3N} \int_{-\infty}^{\infty} dx_i \right\} N e^{-\beta U(x_1, \dots, x_{3N})}$$

where the chemical potential of the gas reservoir, μ , was determined through empirical fugacity functions [141, 142] for hydrogen at both high and low temperature (the high-pressure densities have also been verified *via* NPT molecular dynamics). Isothermal P- ρ curves were generated at temperatures of 77 K (pressure range of 0-200 atm) and 298.15 K (pressure range of 0-2000 atm) and the results are compared with experimental data [143, 144] in Figures 3.5 and 3.6. For the simulations at 77 K,

Feynman-Hibbs quantum corrections [6] were applied to the energetically dominant electronic repulsion/dispersion part of the potential to order \hbar^4 via:

$$U_{rd}^{FH} = U_{rd} + \frac{\beta\hbar^2}{24\mu} \left(U''_{rd} + \frac{2}{r} U'_{rd} \right) + \frac{\beta^2\hbar^4}{1152\mu^2} \left(\frac{15}{r^3} U'_{rd} + \frac{4}{r} U'''_{rd} + U''''_{rd} \right) \quad (3.9)$$

where U'_{rd}, U''_{rd}, \dots are the derivatives of the electronic repulsion/dispersion term of Equation 3.6 with respect to the displacement r . These corrections were not applied in the generation of the 298.15 K data since these effects are negligible at room temperature. To reduce the computational cost, these quantum corrections were only applied to the repulsion/dispersion energy because at 77K, under the pressure conditions demonstrated here, the repulsion/dispersion energy is greater than 85 % of the total energy.

It should be pointed out that the molecular hydrogen is in a supercritical phase under the conditions reported. Note the liquid hydrogen density at boiling (20 K) is 0.07 g cm^{-3} ; thus the state points considered include those representative of relatively strong intermolecular interactions for H_2 . In both instances, we see improved agreement with experiment at high-density when many-body effects are explicitly considered by inclusion of U_{pol} . It is known that many-body polarization effects are of even greater importance when considering high-density hydrogen interacting strongly with, for example, a polar adsorbing material. [100]

3.4 Conclusions

An anisotropic, many-body hydrogen potential has been derived from first principles and expressed in a functional form suitable for mixing with heterogeneous systems

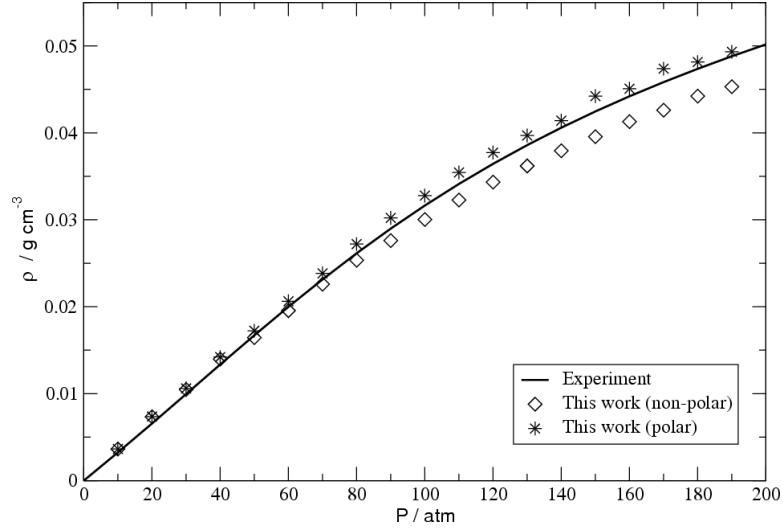


Figure 3.5: P- ρ plot of hydrogen at 77K using the first principles-derived potential energy function. The densities were calculated using GCMC over the corresponding pressure range, including empirical fugacity corrections. The electronic repulsion/dispersion energy term included Feynman-Hibbs quantum effects to order \hbar^4 ; the data generated without this correction was systematically higher by about 10%. All data points have a maximum variance of $\pm 0.001 \text{ g cm}^{-3}$.

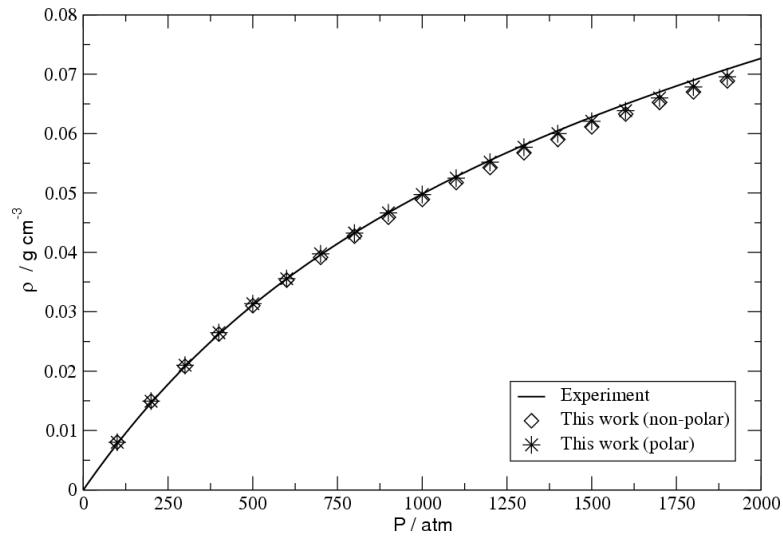


Figure 3.6: P- ρ plot of hydrogen at 298.15 K using the first principles-derived potential energy function. All data points have a maximum variance of $\pm 0.0008 \text{ g cm}^{-3}$.

consisting of partial charges, Lennard-Jones sites and atomic point polarizabilities. It has been shown to reproduce the properties of bulk hydrogen under conditions of current interest in materials research. Electrostatic quadrupolar and many-body polarization interactions are included anisotropically, thus making this model useful for high-density studies where orientation-dependence is of interest.

Next, it is planned to use the potential in modeling H₂ sorption in highly polar MOFs that have been shown to sorb hydrogen at near liquid densities at 77K. [100,115] The potential should also be useful in modeling hydrogen in any condensed phase system where the essential physics of the composite system is captured by the flexible potential energy function presented.

Chapter 4

A Predictive Model of Gas Sorption for Metal-Organic Materials

Newly developed hydrogen and MOM (Metal-Organic Materials) potential energy functions for molecular simulation are presented. They are designed to be highly transferable while still describing sorbate-MOM interactions with predictive accuracy. Specifically, they are shown to quantitatively describe hydrogen sorption, including isosteric heats, in MOF-5 over the broad temperature and pressure ranges that have been examined experimentally. The approach that is adopted is general and demonstrates that highly accurate and predictive models of molecular interaction with MOMs are quite feasible. Molecular interactions giving rise to the isosteric heat have been characterized and validated against the experimentally relevant data. Finally, inspection of the isothermal compressibility of hydrogen in MOF-5 reveals that under saturating high-pressure conditions (even at temperatures well above the neat boiling point) the state of hydrogen is characteristic of a liquid, *i.e.* with a compressibility similar to bulk hydrogen. This result is of particular relevance in developing MOMs for hydrogen-storage applications.

4.1 Introduction

Metal-Organic Materials (MOMs) are among the most widely studied structures for gas storage, and yet very few theoretical studies have investigated the accuracy of

the intermolecular potential energy functions for MOMs interacting with hydrogen. While first generation investigations have revealed reasonably good agreement with experiment using *ad hoc* parameterizations, others have raised questions as to the accuracy of the potential parameters. [16, 105] In either case, the potentials should be validated under high-pressure conditions (rather than the more typical standard pressure state point) and over a wide temperature range where discrepancies would be evident if the potentials were inaccurate. That is, in order to produce a predictive/transferable model of MOM-guest interactions for molecular simulations, the potential must include all salient intermolecular interactions.

Herein we explore the validity of a newly developed hydrogen potential [5] interacting with the prototypical MOF-5 (a.k.a. IRMOF-1) under all extant experimentally examined conditions, including high-pressure that are more relevant to the DOEs required storage conditions. [1] The excellent agreement with experimental data over the wide range of temperatures and pressures illustrates the point that quantitatively predictive models for gas sorption in MOMs are quite feasible when careful attention is paid to the intermolecular potentials.

In addition to the thermodynamic hydrogen uptake, we have investigated the isosteric heat of adsorption, [93] Q_{st} . The typical experimental approximation employed is to derive Q_{st} from isotherms performed at different temperatures. Q_{st} is determined from the Clausius-Clapeyron equation *via* a numerical derivative using two isotherms. In computer simulation, the molecular details of adsorption are accessible for further analysis and a direct statistical mechanical expression for the isosteric heat is available from the thermodynamic fluctuations in sorbate number. Here, good agreement is found for the frequently examined 77 K low-pressure range. At higher pressures, our data are consistent with the limited experimental data; unfortunately the experimental Q_{st} data at higher pressures have inherently large errors due to the range of temperatures over which the numerical derivatives are performed.

Finally, we have determined the isothermal compressibility of hydrogen in MOF-5 as a function of pressure. Interestingly, the hydrogen approaches a state resembling that of the bulk liquid as the excess sorption weight plateaus. This is evidenced by the isothermal compressibility falling rapidly over a narrow pressure range to a value characteristic of bulk hydrogen. This preliminary result is of particular importance because the DOE requirements for hydrogen storage are in excess of liquid density, and hence any interactions that may serve to stabilize the liquid state are of practical importance in elucidating MOM design principles.

4.2 Methods

4.2.1 Hydrogen Potential

Recently, an accurate and transferable hydrogen potential energy function has been developed [5] which includes quadrupolar electrostatic terms as well as many-body polarization, both of which have been shown to be important in modeling dense hydrogen interacting with a charged surface. [4] This new hydrogen potential has been shown to yield an accurate equation of state for hydrogen under high-pressure and low-temperatures. More importantly, the functional form of the potential is easily transferable in the sense that it can describe hydrogen interacting with materials that are describable by Lennard-Jones 6-12 parameters, partial charges and point polarizabilities (a form frequently used for molecular potentials). In contrast, the extant hydrogen potentials that can describe bulk hydrogen accurately (Silvera-Goldman, Buch, etc.) are not readily suitable to complex and heterogeneous condensed phase simulation and/or to a wide range of state points. We have utilized the aforementioned potential in this work to validate its accuracy on modeling various thermodynamic observables of hydrogen in MOF-5.

The effects of many-body polarization have been shown to be important for

the accurate simulation of hydrogen uptake in certain polar MOFs, [4] and for bulk hydrogen at high pressures. [5] However, MOF-5 does not possess the relatively large charge separation on the framework characteristic of some other MOFs. [19, 92, 145–147] Therefore, induction effects in modeling hydrogen in MOF-5 are negligible and, in fact, numerical analysis of the polarization in MOF-5 confirms this. When included, the polarization energy is less than 5% of the total energy at 77 K and does not significantly alter the isotherms and associated isosteric heats. Therefore, the non-polar parameters for the hydrogen potential referred to above have been used and the Thole-Applequist many-body polarization calculation was only performed in this work at selected state points as a control, verifying the negligibility of induction. While inclusion of induction is desirable, its many-bodied nature makes the simulations orders of magnitude more computationally expensive.

4.2.2 MOF-5 Potential

Permanent electrostatic interactions in atomistic simulations stem from point partial charges assigned to the coordinate corresponding to the nuclear center of each atom. Point charges were determined from electronic structure calculations on several model compounds that mimic the chemical environment of MOF-5. [35] The GAMESS *ab initio* simulation package was used to perform the Hartree-Fock quantum mechanical calculations. [61]

The structure of MOF-5 is characterized by benzene-dicarboxylate (BDC) linked zinc tetramers in a cubic octahedral net. The 424 atom unit cell may be produced from crystallographic symmetry operations applied to only 7 atoms. [20] This symmetry-unique form was used as the basis for deciding on representative chemical fragments. For the purposes of charge fitting, fragments of the infinite net were selected in a variety of ways to assess the effects of structural truncation on the fit charges. The addition of hydrogen atoms, where appropriate, was required for

Atom	Label	$\sigma / \text{\AA}$	ϵ / K	q / e^-
Zn	1	2.4616	62.3993	1.8529
O	2	3.118	30.19	-2.2568
O	3	3.118	30.19	-1.0069
C	4	3.431	52.84	1.0982
C	5	3.431	52.84	-0.1378
C	6	3.431	52.84	-0.0518
H	7	2.571	22.14	0.1489

Table 4.1: The MOF-5 potential parameters that were used in this study. The atomic labels refer to the indices depicted in Figure 4.1.

chemical termination of the fragment boundaries. Several different basis sets were chosen and results compared favorably, with the resulting charges being within $0.1 e^-$ of each other on average (the complete comparison between chemical fragments and basis sets is given in supplementary information).

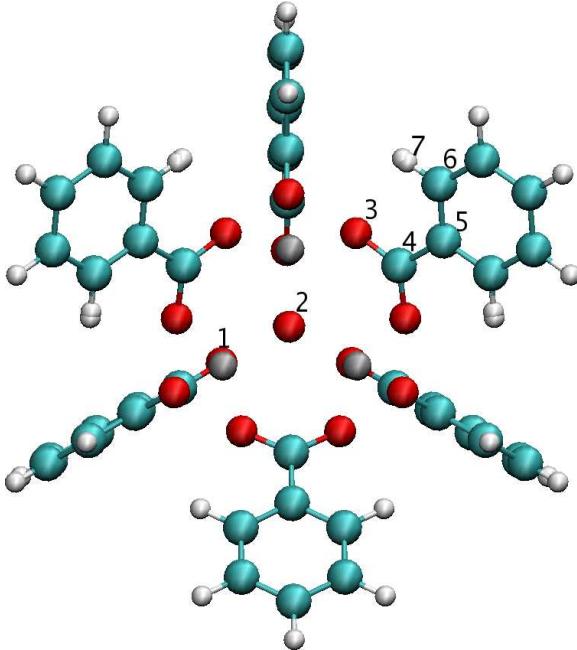


Figure 4.1: Molecular fragment of the MOF-5 framework for which potential parameters have been determined as listed in Table 4.1.

The need for intramolecular framework interactions [148] was avoided by holding the scaffold rigid during simulation. Phonons are not thought to play an im-

portant role in hydrogen sorption, especially not at the temperatures considered here. [34] Lennard-Jones parameters, representing repulsive and van der Waals interactions between hydrogen atoms and framework were taken from the Universal Force Field; [60] this set of Lennard-Jones parameters was used in earlier MOF studies as well. [16,35,44,57] The complete set of potential parameters used in this study is contained in Table 4.1 and the molecular fragment to which they refer is depicted in Figure 4.1.

4.2.3 Analysis of Potential Parameters

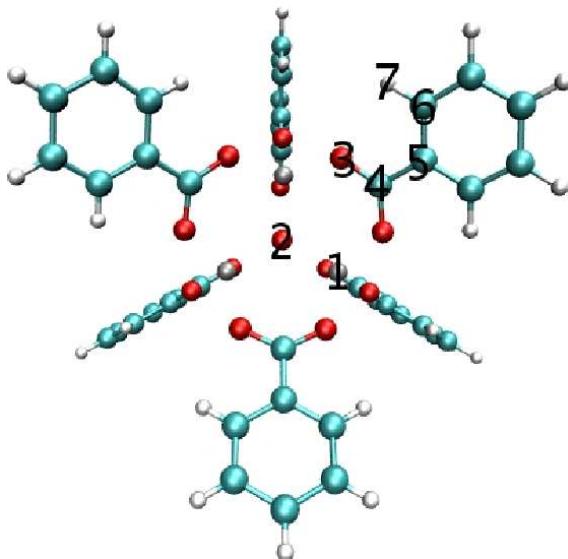


Figure 4.2: Fragment 1, used to compute atomic point charges. This fragment, extracted from the X-ray diffraction data of the crystal, contains 1 complete zinc tetramer with 6 coordinated BDC linkers. The terminal carboxylates of each BDC linker were truncated and chemically terminated with H atoms.

Table 4.2 represents the resulting charges calculated from a fit to the electrostatic potential surface generated via Hartree-Fock quantum mechanical simulation. The gas phase fragment, that the *ab initio* calculations were performed on, was taken from the crystal structure and altered only by capping the terminal benzene rings with hydrogen.

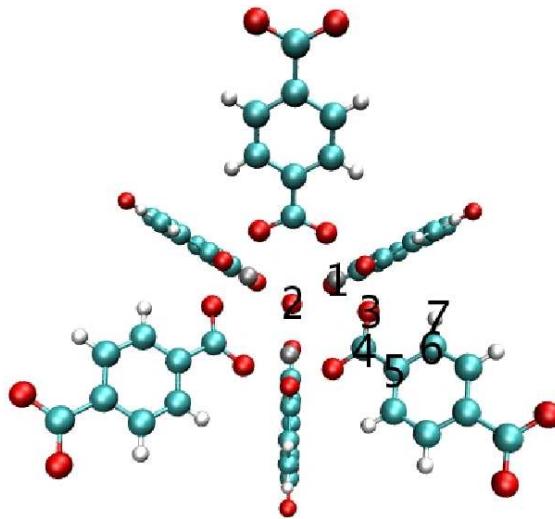


Figure 4.3: Fragment 2, used to compute atomic point charges. This fragment contains 1 SBU with 6 complete linkers including the terminal carboxylate functional groups.

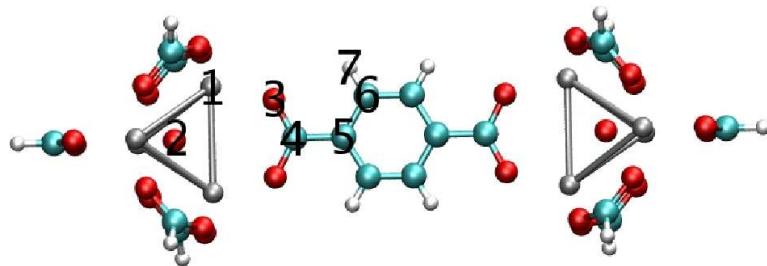


Figure 4.4: Fragment 3 used to compute atomic point charges. This fragment contains 2 SBUs connected by 1 benzene dicarboxylate linker. The SBU is capped with what is essentially formic acid.

Atom	Label	LANL2	SBKJC	6-31G*	DK3
Zn	1	1.6025	1.8529	1.8833	1.8025
O	2	-1.8432	-2.2568	-2.2684	-2.4547
O	3	-0.9036	-1.0069	-1.0144	-0.8852
C	4	1.0937	1.0982	1.1457	0.95203
C	5	-0.2668	-0.1378	-0.1787	-0.0988
C	6	-0.0388	-0.0518	-0.0659	-0.1061
H	7	0.1677	0.1489	0.1729	0.1771

Table 4.2: Partial charges (in e^-) calculated using 4 different basis sets from fragment 1.

Atom	Label	Fragment 1	Fragment 2	Fragment 3
Zn	1	1.8529	1.868	1.851
O	2	-2.2568	-2.263	-2.420
O	3	-1.0069	-1.011	-0.914
C	4	1.0982	1.138	0.958
C	5	-0.1378	-0.170	0.005
C	6	-0.0518	-0.085	-0.182
H	7	0.1489	0.146	0.201

Table 4.3: Partial charges (in e^-) calculated using the 3 different fragments.

Atom	Label	σ / Å	ϵ / K	q / e^-
Zn	1	2.4616	62.3993	1.8529
O	2	3.118	30.19	-2.2568
O	3	3.118	30.19	-1.0069
C	4	3.431	52.84	1.0982
C	5	3.431	52.84	-0.1378
C	6	3.431	52.84	-0.0518
H	7	2.571	22.14	0.1489

Table 4.4: The complete MOF-5 potential parameters that were used in this study.

4.2.4 Grand Canonical Monte Carlo

With respect to hydrogen uptake, the prime observable of interest is the average number of hydrogen molecules sorbed, $\langle N \rangle$, *via* sampling of the grand canonical ensemble over a range of chemical potentials corresponding to the equilibrium pressure of the reservoir. The following statistical mechanical expression was numerically estimated by Grand Canonical Monte Carlo [139, 140] using a code developed by our group:

$$\langle N \rangle = \frac{1}{\Xi} \sum_{N=0}^{\infty} e^{\beta \mu N} \left\{ \prod_{i=1}^{3N} \int_{-\infty}^{\infty} dx_i \right\} N e^{-\beta U_{FH}(x_1, \dots, x_{3N})}$$

where the chemical potential of the gas reservoir, μ , was determined for a broad range of temperatures (60-300 K) through the BACK equation of state. [149, 150] Quantum

mechanical dispersion effects have been included semiclassically through use of the Feynman-Hibbs effective potential [6] to order \hbar^4 via the expression:

$$U^{FH} = U + \frac{\beta\hbar^2}{24\mu} \left(U'' + \frac{2}{r}U' \right) + \frac{\beta^2\hbar^4}{1152\mu^2} \left(\frac{15}{r^3}U' + \frac{4}{r}U''' + U'''' \right) \quad (4.1)$$

and the potential energy function used amounts to:

$$U = U_{es} + U_{rd} \quad (4.2)$$

where U_{es} is the Ewald-summed electrostatic potential and U_{rd} accounts for the electronic repulsion/dispersion energy through use of the Lennard-Jones 6-12 function. The MOF-H₂ LJ interaction parameters were determined using the Lorentz-Berthelot mixing rules; pairwise σ 's are determined through an arithmetic mean and ϵ 's are formed from the geometric mean.

After obtaining $\langle N \rangle$ we proceeded to calculate both the absolute and excess weight percent of hydrogen sorbed; the excess weight calculation utilized the free volume of 11595.4 Å³ determined previously for MOF-5. [20]

Experimentally, the isosteric heat of adsorption is determined by numerical analysis of two hydrogen isotherms performed at different temperatures (typically 77 and 87 K). The isotherm data is then processed (either via curve-fitting or interpolation) and the isosteric heat of adsorption, Q_{st} , is determined over a range of densities through a finite-difference approximation to the Clausius-Clapeyron equation:

$$Q_{st} = kT^2 \frac{\partial \ln P}{\partial T} \quad (4.3)$$

While the macroscopic Clausius-Clapeyron equation can be used with GCMC isotherm data to arrive at values for Q_{st} , a more direct statistical mechanical method [151] is to relate the isosteric heat to fluctuations of a quantity involving the number of sorbed molecules, N , and the potential energy U :

$$Q_{st} = -\frac{\langle NU \rangle - \langle N \rangle \langle U \rangle}{\langle N^2 \rangle - \langle N \rangle^2} + kT \quad (4.4)$$

Another accessible, fluctuation-derived quantity is the isothermal compressibility:

$$\beta_T = -\frac{1}{V} \frac{\partial V}{\partial P} \quad (4.5)$$

which may be calculated *via* fluctuations of the number of molecules sorbed, $\langle N \rangle$, in the grand canonical ensemble through use of the statistical mechanical relation:

$$\beta_T = \frac{V}{kT} \frac{\langle N^2 \rangle - \langle N \rangle^2}{\langle N \rangle^2} \quad (4.6)$$

Both Equation 4.4 and 4.6 have been implemented into the Monte Carlo code used for this study and these quantities have been assessed for H₂ in MOF-5.

4.3 Results and Discussion

4.3.1 Hydrogen Isotherms

Experimental validation of a given MOM's hydrogen storage capability typically takes place at the liquid nitrogen boiling point (77 K). While the DOE milestones require room-temperature operation, evaluation of the surface interactions is more easily determined at lower temperature where meaningful measurements can be made with respect to hydrogen uptake for even marginally sorbing materials. In addition, the isosteric heat of adsorption is measured across the relatively small temperature variation of 77-87 K. Thus, hydrogen uptake in MOF-5, as simulated with the newly developed potentials, is reported in Figure 4.5.

Furthermore, the absolute and excess sorption isotherms for hydrogen in MOF-5 have been measured over a wide range of temperature and pressure, making this well characterized system ideal for comparison with the present theoretical model. The results are compared with experiment in Figure 4.6 and Figure 4.7.

For comparison, we note that in Figure 4.8 our simulation results at 77 K lie within the range of available experimental data. This comparison, while not available at a wide range of state points, suggests that the results in Figures 4.6 and 4.7 are accurate to within the extant experimental uncertainties (including, *e.g.*, standard measurement errors and variation in materials preparation protocol). Note, the two sets of experimental data presented are representative of some of the most careful measurements of hydrogen sorption in MOF-5 to date.

4.3.2 Isosteric Heat of Adsorption of Hydrogen

The isosteric heat of adsorption of hydrogen in MOF-5 was calculated from the fluctuation expression (Equation 4.4) over the statistical mechanical states and is shown in Figure 4.9. The low pressure results in the figure are found to be in good agreement

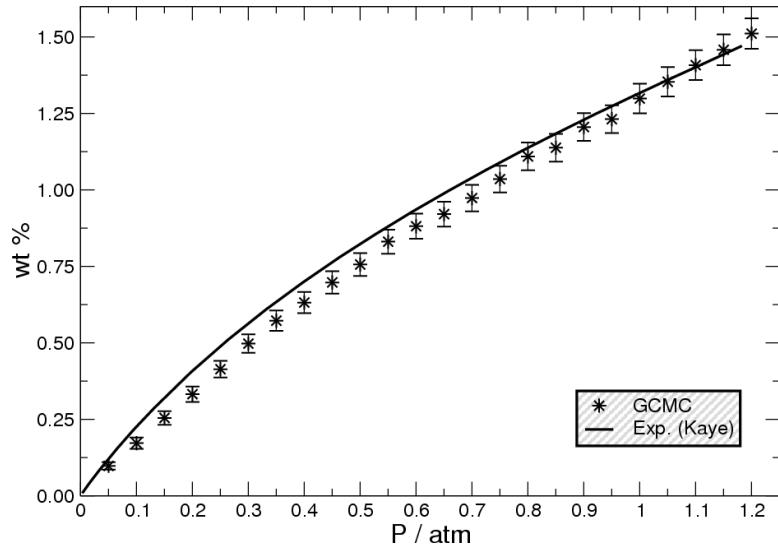


Figure 4.5: GCMC calculated low-pressure hydrogen isotherm (absolute weight percent) of MOF-5 at 77 K *vs.* experiment. [152, 153]

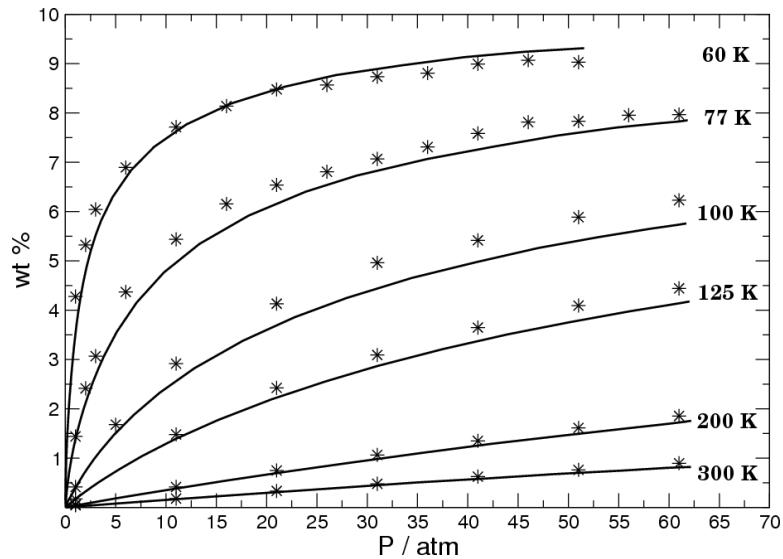


Figure 4.6: GCMC calculated (starred data) high-pressure isotherm (absolute weight percent) of MOF-5 over a wide temperature and pressure range *vs.* experimental data [154] (solid lines). Maximum calculated error is ± 0.07 wt %

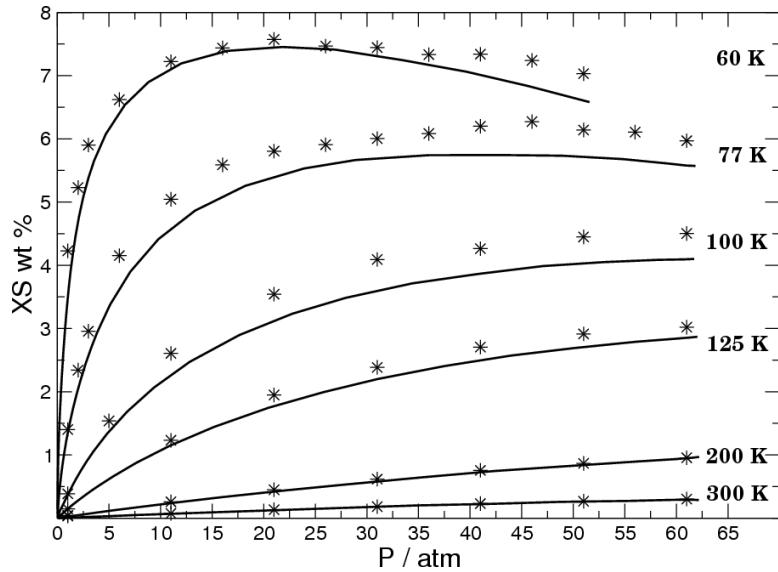


Figure 4.7: GCMC calculated (starred data) high-pressure isotherm (excess weight percent) of MOF-5 over a wide temperature and pressure range *vs.* experimental data [154] (solid lines). Maximum calculated error is ± 0.07 excess wt %

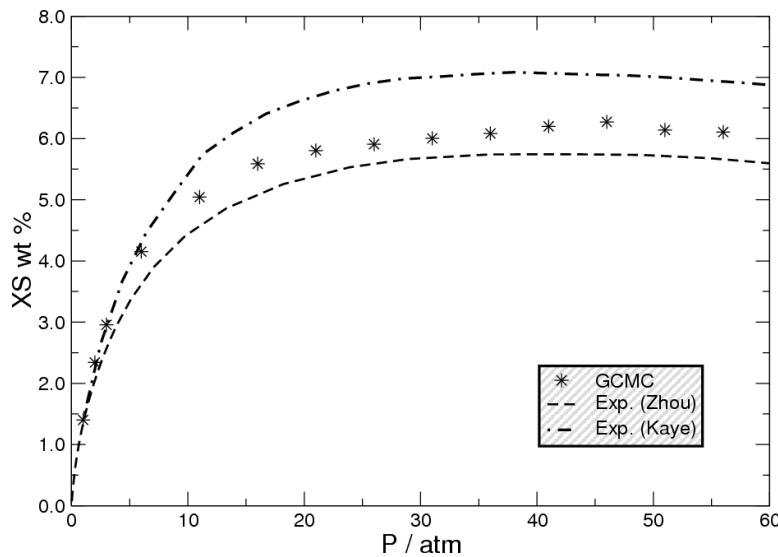


Figure 4.8: GCMC calculated high-pressure isotherm (excess weight percent) of MOF-5 at 77 K *vs.* experimental data [153, 154] (solid lines). Maximum calculated error is ± 0.068 excess wt %

with experiment (especially considering that the experimental data for MOF-5 is correspondingly similar based upon differing preparation techniques, etc.) The isosteric heat of MOF-5 is characteristic of physisorption (≈ 5 kJ/mol) and remains fairly constant over an extended range of densities. The calculated Q_{st} values for the other temperatures over broader pressure ranges are also fairly constant in a range from about 4-5 kJ/mol at low pressure, and 3.5-5 kJ/mol at the higher pressures considered. The experimental data at higher pressures and diverse temperatures [154] are similar but difficult to compare quantitatively because the finite difference approximation to Q_{st} via the Clausius-Clapeyron equation gives a range of values depending upon the two sets of temperature data that are chosen; it is also found that the theoretical isotherm-derived Q_{st} values take on a similar range of values. It would be useful if calorimetry experiments had mapped our sorption enthalpies over a range of thermodynamic conditions. [93, 94] Nonetheless, in no case are the theoretical values inconsistent with the approximate experimental values.

Further, the data also support a model where MOF-5 (with its large surface area and pore volume) is known to retain its interactions with hydrogen at 77 K across the pressure range up until surface saturation - once the surface is covered then the bulk (characteristically weak) H₂-H₂ interactions contribute a greater portion toward the statistical average of Q_{st} .

4.3.3 Isothermal Compressibility of Hydrogen

The isothermal compressibility of hydrogen in MOF-5 at 77 K was found to reach a minimum when the excess sorption isotherm saturated. Furthermore, as shown in Figure 4.10, the value of the compressibility was found to resemble that of the liquid (or supercritical state with a density similar to the normal liquid) state. This bulk compressibility value is itself relatively insensitive to the particular condensed phase state point. It is not surprising that the pore density of hydrogen under high-pressure

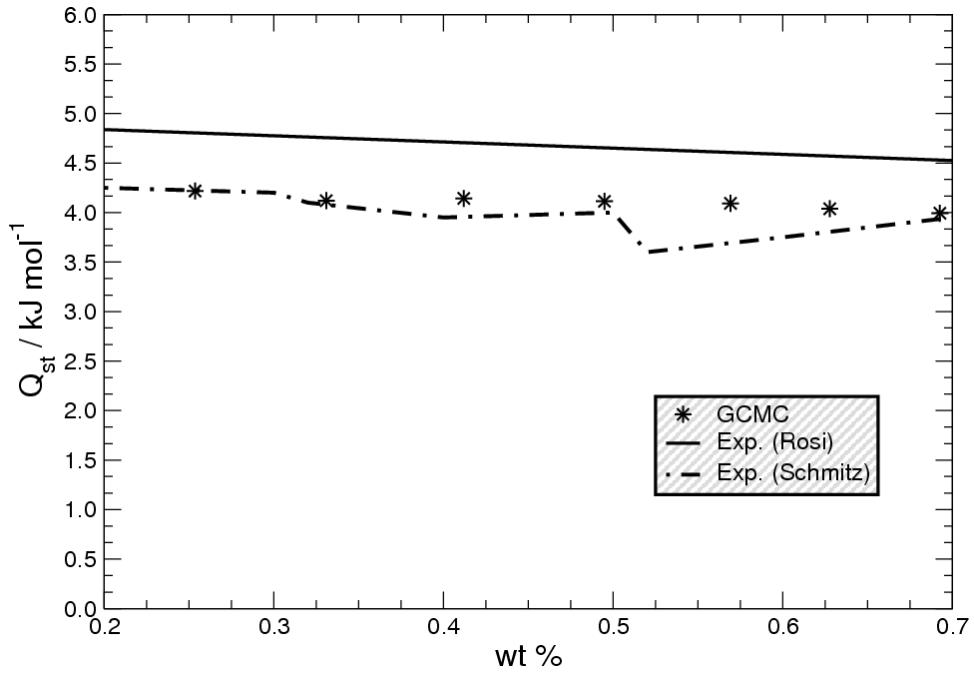


Figure 4.9: Comparison of the calculated (*via* the microscopic fluctuation equation) isosteric heat of adsorption, Q_{st} , for hydrogen in MOF-5 at 77 K *vs.* experimental [20,155] data.

in MOF-5 is characteristic of the bulk phases given the close packing of H₂ in that case. However, the compressibility has several unique properties which are of interest. For example, the compressibility will be a function of both surface interaction and bulk-like (*i.e.* pore localized hydrogen) contributions, the relative effects of which will vary as a complex function of the equilibrium state point. Therefore, given that the MOF-5 confined H₂ compressibility is comparable to bulk phases suggests a relatively minor perturbation to the structure in the weakly interacting MOM. The results demonstrate that the isothermal compressibility, directly obtainable from GCMC calculations, is an important parameter worth monitoring to assess the nature of the confined fluid and the ability to further improve sorption capacity with increased pressure.

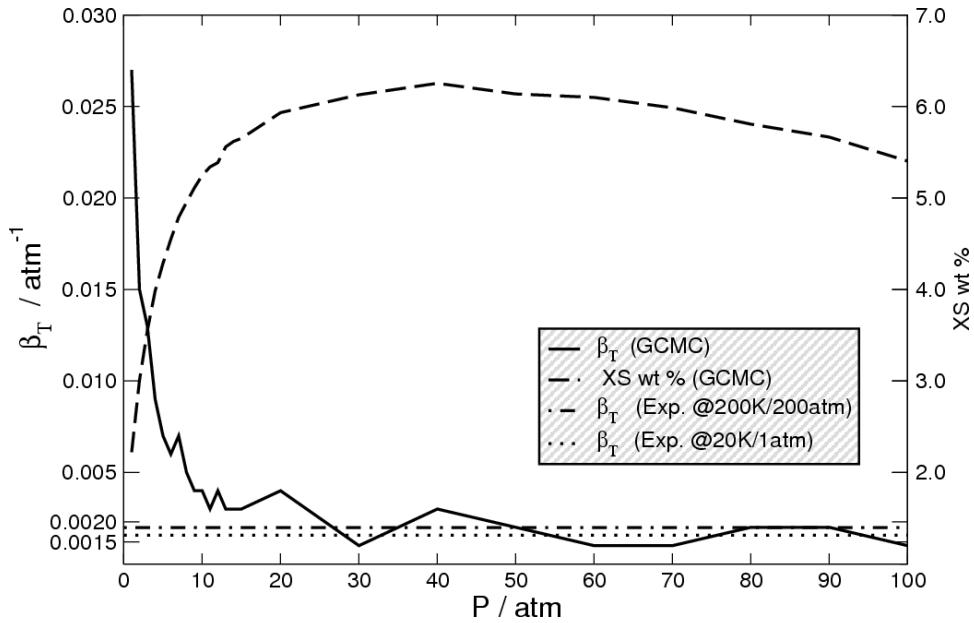


Figure 4.10: Calculated isothermal compressibility of hydrogen in MOF-5 at 77 K *vs.* experimental values of high density $\beta_T = 0.0015 \text{ atm}^{-1}$ (200 K/ 200 atm) [156–158] and $\beta_T = 0.0020$ (20 K/ 1 atm). [159] The excess weight % of hydrogen in MOF-5 at 77 K is also depicted on the opposite y-axis for comparison.

4.4 Conclusions

After careful development of both hydrogen and MOM potential energy functions, quantitatively accurate high-pressure results have been obtained (more specifically, the sorption uptake and isosteric heats). The analogous development of potential surfaces for other MOMs and possible sorbents is likely to lead to similarly accurate models of these important systems. Such predictive accuracy will aid greatly in the rational, iterative design cycle between experimental and theoretical groups that are attempting to design MOMs for a variety of purposes, including H₂ sorption and CO₂ sequestration.

Here, an accurate and transferable, anisotropic hydrogen potential has been employed in studying MOMs, and it is expected that the many-body polarizable form of the H₂ potential will shed further light on interactions in MOMs possessing open-coordination sites and charged [92,147,160] and/or polar [19] frameworks that polar-

ize hydrogen molecules (the continuing subject of future work). Such charged/polar MOMs have great potential for multiple applications due to the necessarily stronger interactions that they have with guest molecules *via* the polarization induced by the MOM framework. Finally, the benefit of liquid state analysis applied to hydrogen in the unique topological and chemical environment presented by a MOM may yield new insights into the design properties necessary for enhanced gas storage, of which analysis of the isothermal compressibility (this work) and radial distribution functions [4] have played a role thus far. Lastly, the success of the present MOM-guest interaction model suggests that molecular simulation methods, with carefully constructed potential energy surfaces, are the method of choice in modeling and predicting the properties of such systems - they are reasonable in computational cost while retaining high accuracy.

Chapter 5

Photophysical Studies of the Trans to Cis Isomerization of the Push-Pull Molecule: 1-(Pyridin-4-yl)-2-(N-methylpyrrol-2-yl)ethene (mepepy)

Organic molecules possessing intra-molecular charge transfer properties (D- π -A type molecules) are of key interest particularly in the development of new optoelectronic materials as well as photoinduced magnetism. One such class of D- π -A molecules that is of particular interest contains photo-switchable intra-molecular charge transfer states via photo-isomerizable π -system linking the donor and acceptor groups. This chapter describes a collaborative effort between both theoretical and experimental groups, with the experiments having been performed by Dr. Audrey Mokdad and Dr. Randy Larsen (USF Chemistry/SMMARTT).

The structural, energetic and electrostatic differences between the aqueous *cis*- and *trans*- isomers of 1-(pyridin-4-yl)-2-(N-methylpyrrol-2-yl)-ethene (MEPEPY) were examined using Density Functional Theory (DFT). The equilibrium geometries of the various MEPEPY states were theoretically calculated by energy minimization. The difference in potential energy and dipole moment, including mean solvation effects, between the isomers was elucidated. Unique characteristics of the potential energy surfaces were explored, most notably the existence of ground state *trans* isomers that do not thermally interconvert at room temperature.

5.1 Introduction

The importance of non-linear optical materials in high speed optical modulators, optical storage media, fast/ultrafast optical switches, etc. has stimulated research efforts into the development and characterization of novel classes of molecules which can exhibit diverse polarization properties both in the ground and excited states. [161–165] One class of molecules which have demonstrated non-linear optical (NLO) properties and are of current research interest are known as “push-pull” type molecules. The non-linear optical effects associated with these molecules are due to the presence of an electron accepting group on one side of a conjugated moiety and an electron donating group on the opposite side. This enhances the polarizability of the double bond region allowing for additional polarization to be induced in the presence of an electric field. Typically, amino, dialkylamino, ether or oxide (O-) functional groups form the electron-donating substituent while nitro, carbonyl and cyano groups are employed as the corresponding acceptor group. [166,167] To achieve the desired polarizability, these groups are separated via a conjugated linker group.

In addition to their non-linear optical properties, push-pull molecules have also been utilized as ligands for various transition metal complexes in order to photolytically alter the spin state of the complex. [168–172] In this case the push-pull ligand must possess a functional group capable of coordinating to the transition metal complex and that exhibits a shift in the basicity upon photo-excitation. Metal complexes of this type can exhibit, for example, a low spin electron configuration when the push-pull ligand is in the more basic conformation and switch to a high spin configuration upon photoconversion of the ligand to the less basic conformation. A number of such complexes have now been synthesized and their optical and magnetic properties examined. These include complexes of the $\text{Fe(II)(L}_4\text{)(X}_2\text{)}$ type in which L is the photo-isomerizable push-pull ligand and of the $\text{Fe(III)(L}_4\text{)(L')}(X)$ type in which L' is the push-pull ligand. Early studies utilized 4-styrylpyridine (Stpy) (1-phenyl-2-

(4-pyridyl) ethane) as well as several phenyl derivatives of this ligand to photo-induce the spin crossover. [170–172] In one of the first examples of ligand driven light induced spin state changes, the Fe(II)(trans Stpy)₄(NCS)₂ complex was found to undergo a thermally induced high-spin to low spin transition centered near 190 K. [170] Photoexcitation of this complex embedded within a cellulose acetate substrate at 140 K results in trans to cis isomerization of the Stpy which also induces the high-spin to low spin transition.

The success of these complexes has led to the synthesis of new compounds which exhibit ligand driven light induced spin state changes at higher temperatures. One such complex, [Fe(III)(salten)(mepepy)]BPh₄ (salten = 4-azaheptamethylene-1,7-bis(salicylideneiminate; mepepy= 1-(pyridin-4-yl)-2-(N-methylpyrrol-2-yl)-ethene; BPh₄ = tetraphenyl borate) has been shown to exhibit a high spin to low spin transition at room temperature under visible irradiation. [170, 171, 173] Unlike the Fe(II)(L₄)(X₂) type complexes discussed above which have four photo-isomerizable ligands, the Fe(III)(L₄)(L')(X) type complexes (e.g., Fe(III)(salten)(mepepy)) have only a single isomerizable ligand which is used to modulate the ligand field strength. In the case of the mepepy complex the spin state transition is triggered via the light induced trans to cis isomerization of the ligand which results in shift in electron density from the N-methylpyrrol moiety to the pyridine unit.

Although the trans/cis photoisomerization of mepepy has been shown to be effective in modulating the ligand field strength of the chromophore much less is known about the electron properties and energetics of this process. The optical spectrum of the trans conformation of mepepy in acetonitrile exhibits absorption maxima at 353 nm ($\epsilon = 22,800 \text{ M}^{-1} \text{ cm}^{-1}$) and 241 nm ($\epsilon = 7,800 \text{ M}^{-1} \text{ cm}^{-1}$) which have been tentatively assigned as π to π^* transitions. [170, 171] Upon photoisomerization, the absorption band at 353 nm exhibits a slight hypsochromic shift and decreases in extinction by nearly 50 % while the 241 nm band shows only modest changes in

extinction. In the present work, the energetics associated with the trans to cis photoisomerization as well as the potential energy surfaces of the mepepy ligand have been investigated using computational methods in an effort to better characterize the physical properties of this important ligand.

5.2 Methods

All calculations of 1-(pyridin-4-yl)-2-(N-methylpyrrol-2-yl)-ethene (MEPEPY) [171] were performed using the quantum chemistry program GAMESS. [174] Molecular properties were evaluated using Density Functional Theory because this has been shown to yield accurate results for isomerization studies of azobenzene [175–177], a molecule that is structurally similar to MEPEPY. The B3LYP hybrid exchange-correlation functional [178–180] was employed for all calculations along with the augmented correlation-consistent double-zeta basis set [78] (aug-cc-pVDZ). Solvation effects were taken into account through the use of a polarizable continuum model [181] in which the MEPEPY molecule is embedded in an isotropic dielectric field parameterized to mimick the aqueous environment. The dielectric constant was 78.39 and the solvent radius was 1.385 Å.

Geometry optimizations for both the *cis* and *trans* isomers were performed in the solvation field in order to ascertain the minimum energy geometries. The molecular electric dipole moments for the optimized structures were determined from the charge distributions.

In order to discover any additional local energy minima, the ground state *cis* and *trans* potential energy surfaces were generated as a function of the angles for the pyridine and pyrrole rings from a planar configuration. Each angle was scanned by rotating the rings along the $\alpha - \beta$ and $\gamma - \delta$ vectors, from 0° to 180° in 10° increments, while constraining the remaining degrees of freedom.

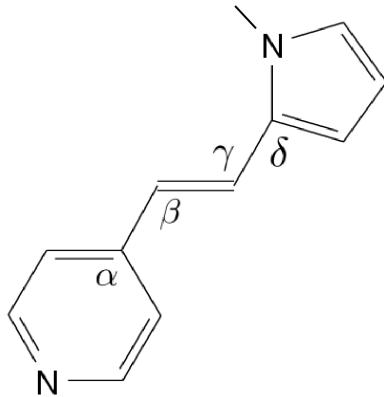


Figure 5.1: *trans* isomer of MEPEPY

isomer	$\angle\alpha\beta\gamma$ ($^{\circ}$)	$\angle\beta\gamma\delta$ ($^{\circ}$)	$d_{\alpha\beta}$ (Å)	$d_{\beta\gamma}$ (Å)	$d_{\gamma\delta}$ (Å)	σ ($^{\circ}$)
<i>cis</i>	126.647	129.507	1.487	1.348	1.450	90.0
<i>trans</i>	126.490	127.365	1.462	1.353	1.439	0.0

Table 5.1: Geometric configurations of the global energy-minimized *cis* and *trans* MEPEPY. The bond angles and distances are with respect to the labels in Figure 5.1. σ is the angle between the pyridine and pyrrole planes. Renderings of these geometries are pictured in Figure 5.2.

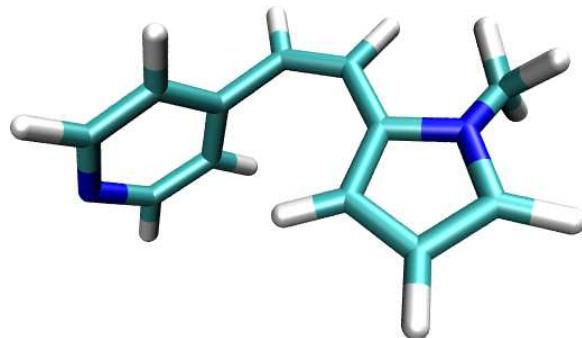
5.3 Results and Discussion

The structures of the energy minima located along the potential energy surface were found (for which the geometric details are tabulated in Table 5.1), and are also visually depicted in the rendered representation of Figure 5.2.

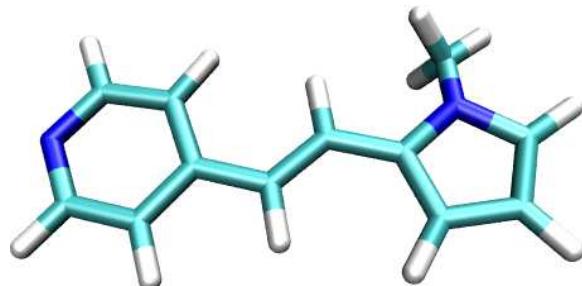
The energy difference between the global *trans* minimum and that of the *cis* form was found to be 8.237 kcal/mol, while the energy difference between the local *trans* minimum and the *cis* isomer was determined to be 7.167 kcal/mol. After evaluating the electrostatic moments, the difference in the dipole moment between the global *trans* state and the *cis* state was 0.424 Debye and the local minimum difference was 0.0398 Debye. While the relative populations of the isomers after a photoisomerization event are unknown, these determined differential values place an upper and lower bound to the true equilibrium differences.

isomer	energy (kcal mol ⁻¹)	molecular dipole moment (Debye)
<i>cis</i>	-335,330.347905	4.117
global <i>trans</i>	-335,338.585359	3.693
local <i>trans</i>	-335,337.514896	4.077

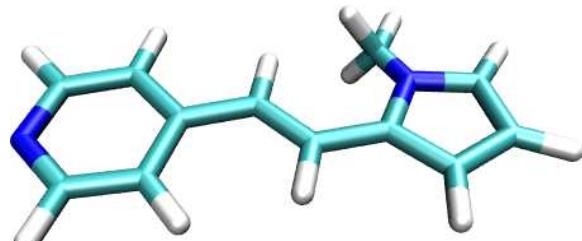
Table 5.2: Calculated energy and dipole moment of global minima *cis* and *trans* MEPEPY isomers, along with the alternate local minimum along the *trans* potential energy surface.



(a) cis geometry



(b) trans local geometry

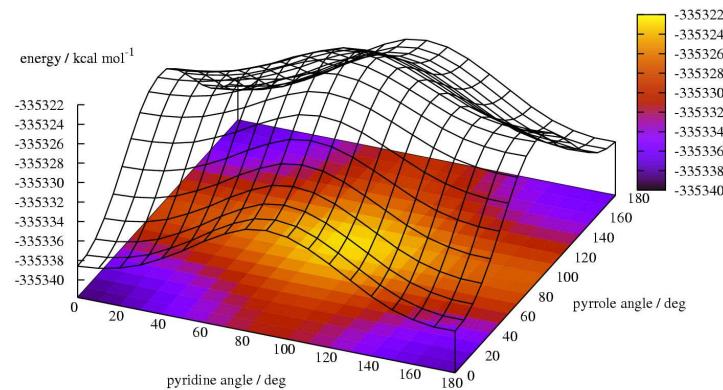


(c) trans global geometry

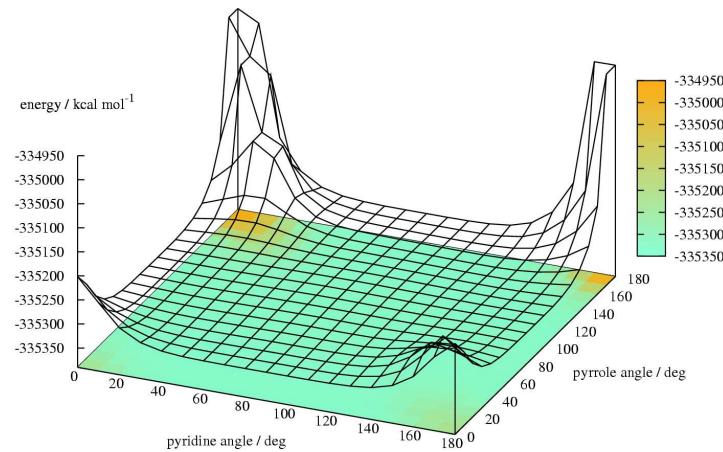
Figure 5.2: Visual renderings of the global energy minimized structures for *trans* and *cis* isomers with the values specified in Table 5.1.

The *cis* potential energy surface is a shallow basin with a single minimum, where it can be expected that the equilibrium structure will fluctuate out of plane in the solvated ground state. However, unique characteristics of the *trans* potential energy surface were found, most notably the existence of two ground state minima separated by an energy barrier. Complete determination of the isomerization mechanism for MEPEPY (which is not determined in this work) is required to know whether the transitions from the excited state to the ground state favor one isomeric *trans* form over the other. The minimum energy barrier between the congeneric *trans* states is 9.115 kcal mol⁻¹ whereas the thermal energy kT at 300K is only 0.596 kcal mol⁻¹.

In summary the results presented here demonstrate several key features of the trans to cis conversion in mepepy. First, the mepepy ligand exhibits two conformational minima for the trans state that are separated by 9 kcal/mol suggesting that the photo-induced spin state transitions associated with metal complexes containing the mepepy ligand may be energetically “tuned” based upon the nature of the trans conformer. Second, the results of the computational studies reveal very little change in dipole moment between the trans and cis conformers of mepepy indicating more subtle changes in the electronic structure of mepepy relative to other push-pull molecules and the azobenzenes.



(a) *trans* energy surface



(b) *cis* energy surface

Figure 5.3: *trans* and *cis* ground state energy surfaces as a function of the pyridine and pyrrole ring angles as defined by deviation from planar configuration.

Chapter 6

Insight into the Assembly Mechanism of Metal-Organic Materials

A collaborative theoretical/experimental study with Mohamed Alkordi of USF Chemistry/SMMARTT on the self-assembly of cobalt-containing metal-organic nanocubes is presented. This solution-based study represents the fruitful outcome of combined experiment and theory, and demonstrates the importance of determining how discrete metal-organic systems can, ultimately, form stable structures. It is hoped that this approach will motivate further work on MOM assembly.

6.1 Introduction

Metal-Organic Materials (MOMs) represent a family of functional solid state materials with great potential to answer a number of currently demanding applications such as gas separation and storage, drug delivery, catalysis, small molecules sensing, ion exchange, and CO₂ sequestration. Such attributes are pertinent to the intrinsic characteristics of MOMs including crystallinity, rigidity, and, in many cases, porosity. The crystalline nature of MOMs facilitates unambiguous structural characterization, permitting better understanding of underlying structure-function relationship and subsequently enhanced control over targeted properties. It is through identification of the underlying fundamental structural entities in MOMs, widely known as the molecular building blocks (MBBs), that rational design strategies for construction of

functional MOMs were conceived, and further successfully implemented. The MBB approach for construction of MOMs encompasses the *in situ* preparation of rigid and directional MBBs through self-assembly of judiciously designed molecular precursors encoded with appropriate information, complimentary molecular recognition sites held at specific spatial orientation, under welldefined reaction conditions. Due to relatively mild reaction conditions employed in solvothermal syntheses of MOMs, the built-in rigidity, functionality, and directionality of MBBs are maintained throughout the reaction pathway, facilitating reliable prediction of the underlying topologies in targeted constructs. Hence, geometrical design principles for construction of MOMs based on the MBB approach were identified and further successfully implemented into rational design strategies towards construction of several functional MOMs. However, despite the wide interest in developing new strategies to construct MOMs with specific topologies and/or functionalities utilizing a wide variety of chemically-accessible MBBs, systematic investigations aiming to probe the various effects associated with the wide range of reaction variables on the nature of isolated product(s) are still lacking. Valuable information obtained through such systematic investigations could provide useful insight into the crucial roles associated with various reaction variables facilitating both the *in situ* preparation of targeted MBBs and, subsequently, their self-assembly into targeted structures. Accordingly, better understanding of the complexity involved in solvothermal reaction systems could be utilized to enhance our design strategies for construction of functional MOMs.

Metal-Organic Polyhedra (MOPs), a subset of MOMs, are finite supermolecular assemblies with various functionalities, tailorabile voids, and fairly predictable geometries. Pertinent to their ability to enclose a confined space, MOPs have been utilized as “nano-reactors facilitating specific chemical transformations of encapsulated guest molecules. In addition, MOPs with peripheral functionalities can further be employed as supermolecular building blocks (SBBs) with built-in structural in-

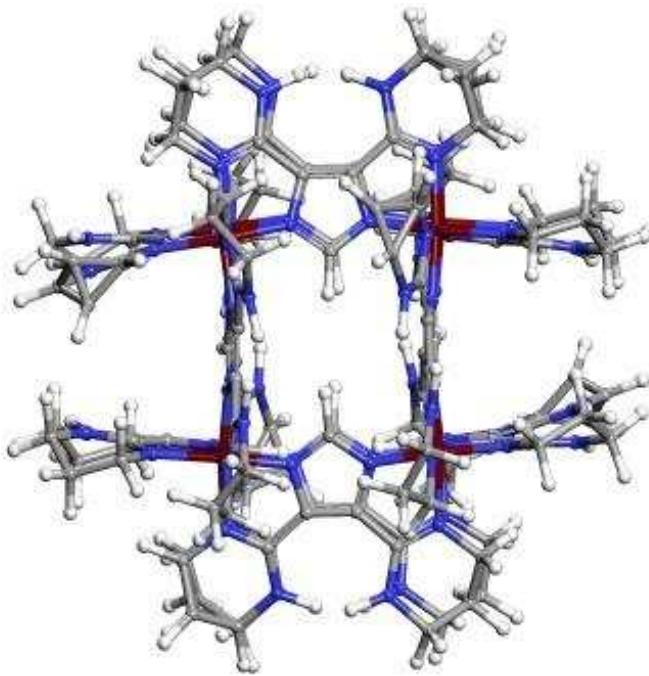


Figure 6.1: Cobalt Metal-Organic Nanocube, ME506

formation, decorating and expanding the vertices of targeted 3D nets, towards construction of metal-organic frameworks (MOFs). Among the readily accessible MOPs is the imidazolate-based metal-organic cube (MOC). The MOC, constructed through single-metal-ion MBB approach, represents a relatively simple system suited for the intended systematic study and thus is the focus of our investigations herein.

The ligand that has been used in this study has several characteristics of note. First, when coordinated the chemical environment of the ring protons is unique and

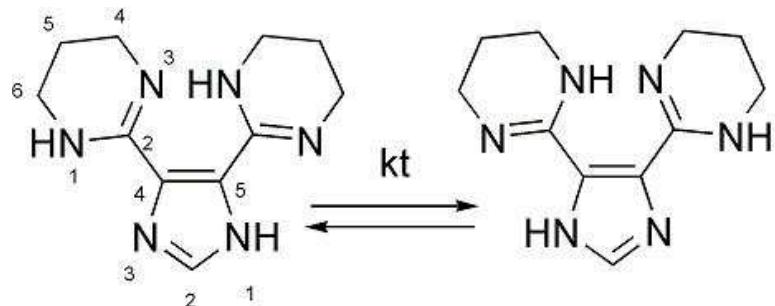


Figure 6.2: Ligand which, in complex with Co, reacts to form the nanocube

does not possess a C₂ axis of rotation; this lack of symmetry gives rise to clearly defined proton NMR peaks that have been useful in this work. Second, there is a basic site whereby (upon coordination) protonation takes place - this again yields a distinct NMR signature where deuterium is exchanged with the solvent and thus the amount of free ligand present in the sample may be accurately monitored. Finally, there is a unique intra-molecular hydrogen-bond present, as determined by both experiment and simulation.

Upon mixing of cobalt and the ligand in water, it was evident from the experiments that some form of inorganic complex was present in solution, the exact nature of which was not yet elucidated. Analysis of the time-dependent proton NMR spectra, monitored as a function of the reaction progress, showed no new peaks but only minor alterations in the intensity of peaks associated with the ligand.

Evidence pointed to the formation of a nanocube, based upon the following observations. First, the ligand-to-metal ratios that resulted in no free ligand in solution (as determined from NMR) were consistent with a cubic assembly. Second, MALDI-TOF mass spectrum fragmentation patterns were consistent with a discrete nanocube. Diffusion NMR experiments revealed a hydrodynamic radius of $\approx 9.5 \text{ \AA}$, also consistent with the cubic hypothesis. Finally, single crystals have been previously obtained with this ligand using nickel, and so it seemed plausible that a nanocube was also being formed with cobalt but simply could not be easily crystallized for some unknown reason.

6.2 Methods

In order to definitively prove the existence of a metal-organic nanocube in solution, theoretical calculations were performed on a number of possible inorganic complexes which are shown in Figure 6.4. These fragments were chosen since they represent all (reasonable) possible coordination complexes within the mass limits previously

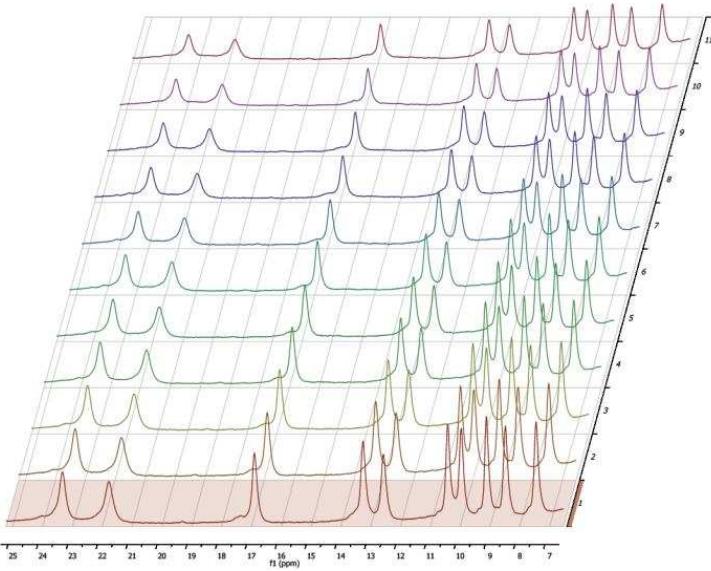


Figure 6.3: Time-series NMR spectra of reaction forming the nanocube. Note that only the intensities of the chemically unique proton shifts are altered, without additional peaks being included or lost.

determined from MALDI-TOF experiments.

The fragments were geometry optimized using the quantum chemistry code Gaussian 03. The minimized energy was calculated via Density Functional Theory (DFT) using the B3LYP hybrid exchange-correlation functional. The unrestricted calculation employed the LANL2DZ basis set with an ECP applied to the cobalt atoms. [182]

6.3 Results and Discussion

T1 relaxation experiments were performed by Mohamed Alkordi to further determine the geometry of the inorganic complex in solution. In this experiment, the NMR relaxation times are proportional (to the sixth power) to the distances from the protons (giving rise to the signal) and a high-spin metal, such as cobalt. The distances determined in this experiment were compared with the distances from DFT calculations of the possible candidate structures and is shown in Figure 6.6. Most importantly, a

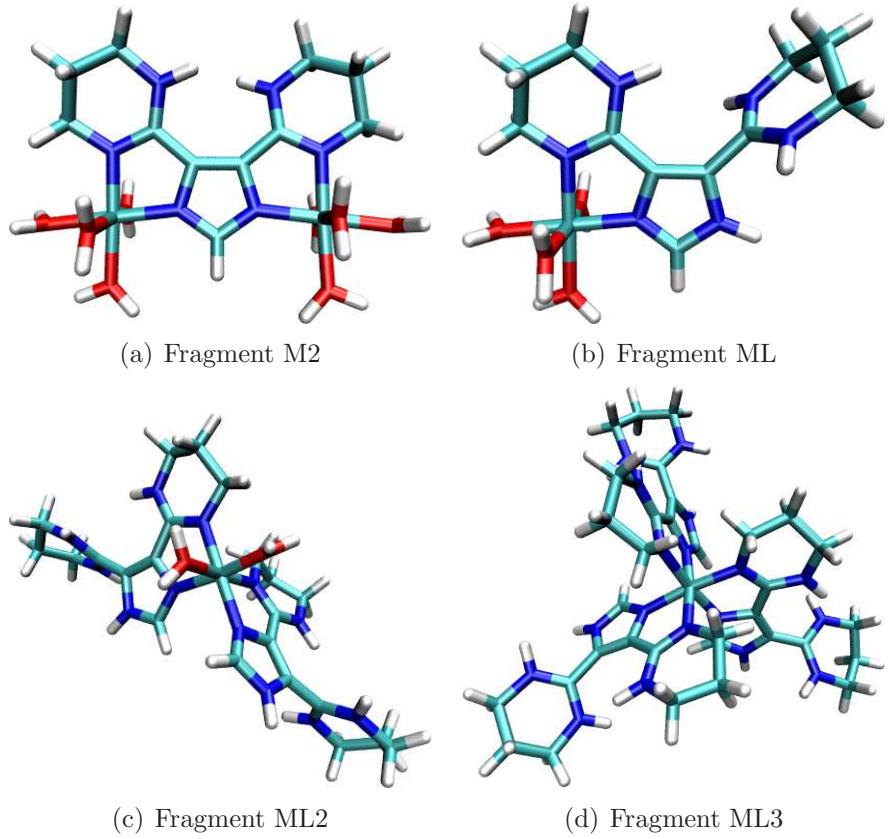


Figure 6.4: Possible inorganic complexes that have been geometry optimized using DFT.

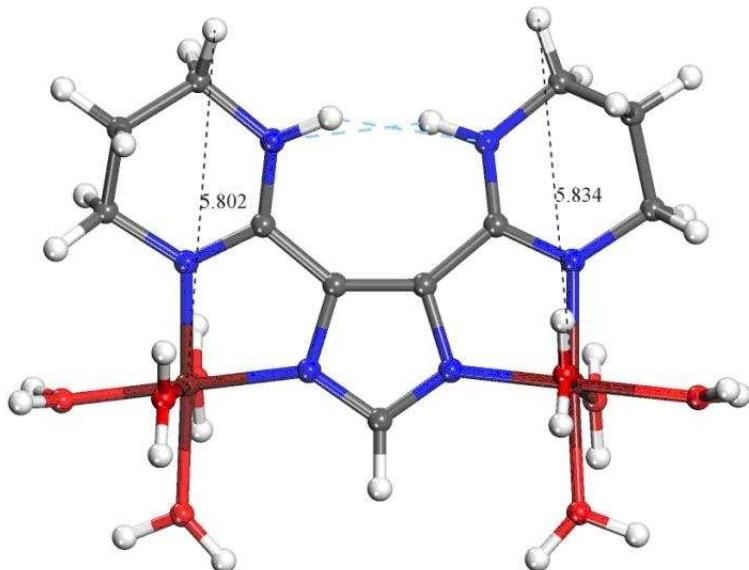


Figure 6.5: Identification of the T1 relaxation distances from the high-spin cobalt to the chemically unique protons of the ligand for the matching DFT geometry-optimized complex.

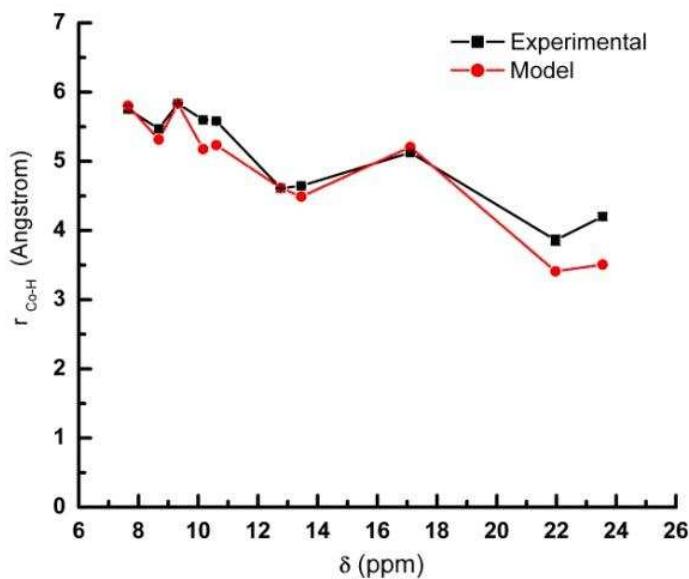


Figure 6.6: Comparison of experimental and theoretical T1 relaxation distances for the cobalt nanocube and for the theoretical inorganic complex that was optimized using DFT. None of the other possible complexes resulted in geometries given quantitative agreement with the T1 experiment.

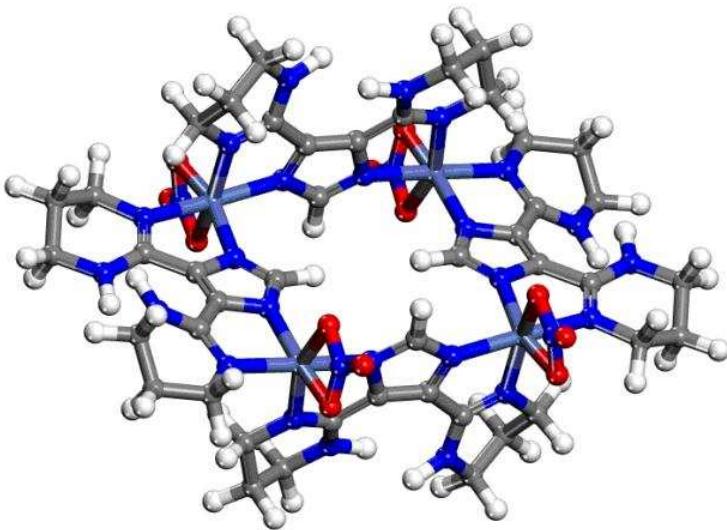


Figure 6.7: Nanosquare intermediate along the assembly route toward the nanocube.

quantitative match was found only for the structure shown in Figure 6.5, thus proving the existence of a stable nanocube in solution.

In addition, based upon the aforementioned analysis, it was proposed that the reason for the lack of crystal formation of Co-containing cubes was due to the fact that the discovered fragment (and hence the cube) would possess a net charge and that the counterions in solution were disrupting the intermolecular forces necessary to make a stable crystal. Since the inter-cube interactions are relatively weak, it is observed that alterations to the hydrogen bond network could result in varying types of packing.

Therefore, it was postulated that a solvent with the proper characteristics of high boiling point, hydrophobicity and good hydrogen bond acceptor functionality could be conducive to the formation of single crystal. N,N'-diethylformamide (DEF) is a solvent with such characteristics and experimental treatment of the nanocube-containing solution indeed resulted in pure, red single crystals. X-ray diffraction, indeed, confirmed the presence of Co-containing nanocubes, as had been determined previously by both solution phase experiment and quantum mechanical calculations.

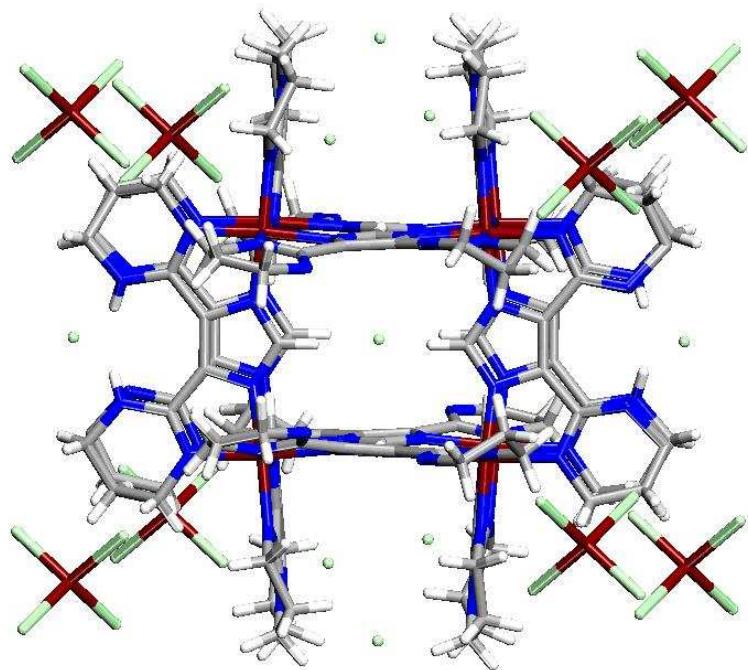


Figure 6.8: Anionic counterions depicted with the nanocube. Disruption to the hydrogen bond network necessary for orderly packing of the nanocubes was determined to be the reason for lack of a crystalline material – adjustments made to the solvent conditions resulted in pure, single crystals.

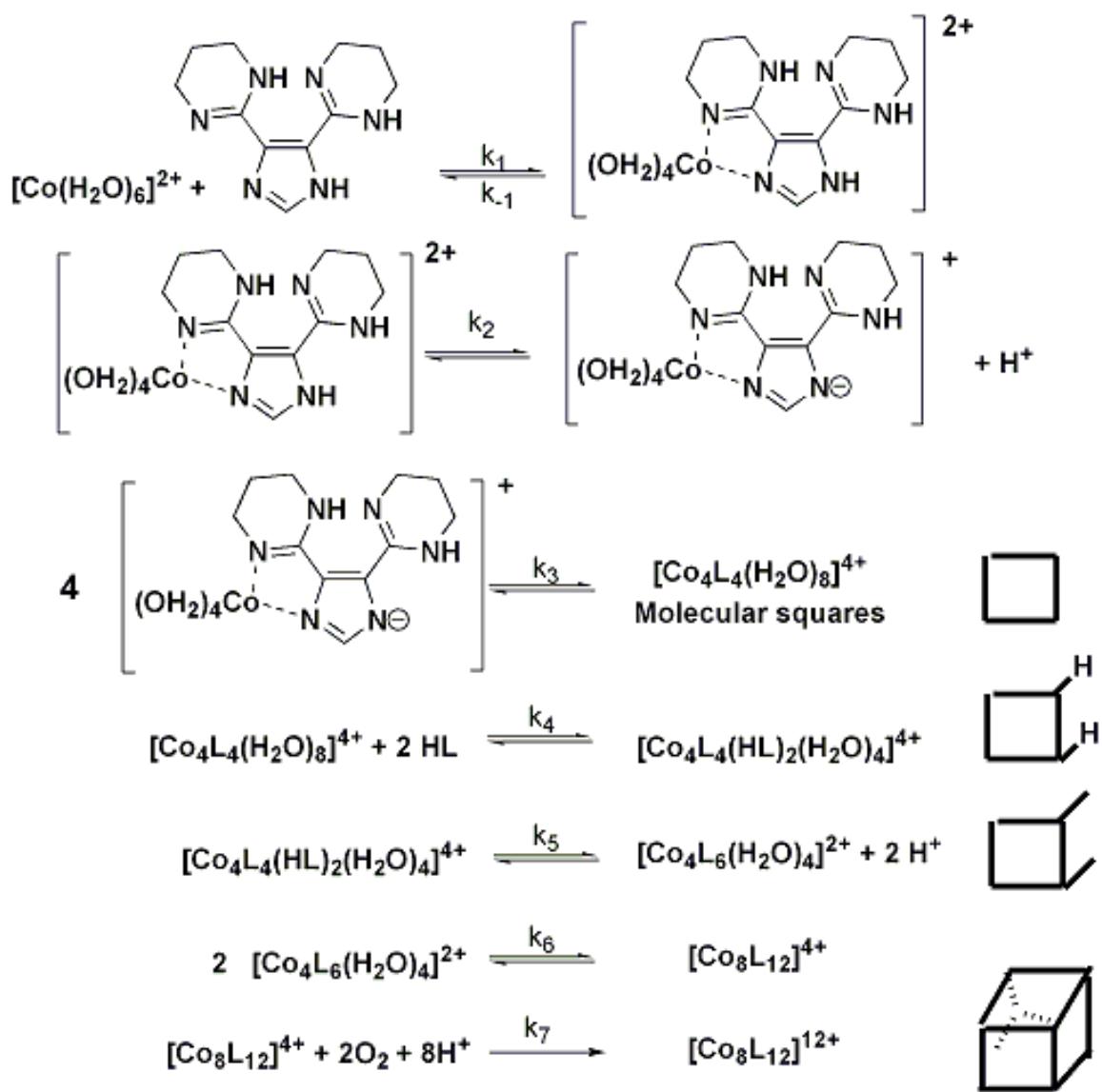


Figure 6.9: Proposed assembly mechanism of the Co nanocube.

Based upon the chemistry that has been learned regarding nanocubes (through both theory and experiment) and previous kinetic studies on the formation of metal-organic nanoballs, [183] it is hypothesized that the assembly mechanism of the MOC proceeds as shown in Figure 6.9. This mechanism remains to be proven, and will likely result in future theoretical and experimental studies.

Chapter 7

Molecular Squares: Confined Space with Specific Geometry for Hydrogen Uptake

7.1 Introduction

Metal-Organic Frameworks (MOFs), as functional solid-state materials, continue to receive wide scientific interest due to their potential applications in hydrogen storage, gas separation, carbon dioxide sequestration, enhanced catalysis, and drug delivery. Such applications are pertinent to the fundamental attributes of MOFs including dual composition, high crystallinity, and open structures. In particular, porous MOFs have been widely investigated for hydrogen storage, demonstrating reversible physisorption interactions, within available void space, amenable to a high degree of tuneability associated with the highly modular nature of MOFs.

Hydrogen interactions with metal complexes, clusters, or ions, as the inorganic part of the framework, are mostly electrostatic in nature and could play major roles in determining the H₂ uptake characteristics of a particular MOF due to their relatively significant contribution to the overall H₂ binding affinities and hence are the subject of considerable theoretical and experimental investigations. Although weaker, favorable van der Waals interactions between H₂ and the organic linkers in MOFs, best represented by benzene ring derivatives, have been theoretically investigated and experimentally documented through inelastic neutron scattering experiments. Recent

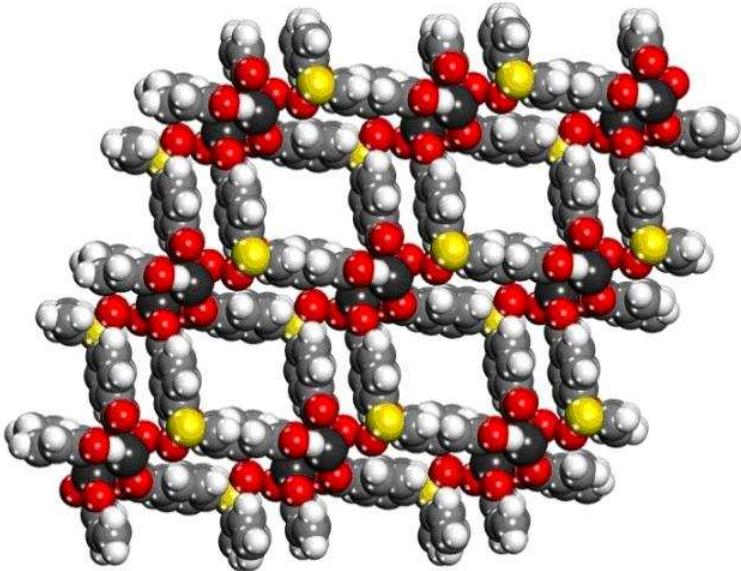


Figure 7.1: Molecular Squares: ME193

studies demonstrate that such interactions could, in principle, be enhanced through chemical modifications to the organic linkers, providing a potential strategy for a material designer to enhance H₂ sorption characteristics of MOFs. Nevertheless, no studies exist, to the best of our knowledge, addressing the possibility of improving H₂ binding affinity to the walls of MOFs through simultaneous favorable dispersive interactions, acting additively, between H₂ molecules and multiple aromatic rings placed at optimal interaction distance(s) and within a specific geometry.

7.2 Methods and Results

Therefore, we opt to explore this approach which could potentially prove useful as a viable target to consider, among others, in rational design strategies for future hydrogen storage materials. Computational studies by Head-Gordon et al, Goddard et al, and others, for the H₂ binding affinities to benzene and various aromatic rings revealed moderate binding affinities, mostly due to favorable dispersive interactions in the range of 3.4-4.0 kJ/mol for a H₂ molecule interacting with benzene ring in

terephthalic acid. Although below the estimated target of 20-40 kJ/mol, for efficient H₂ storage materials at ambient conditions, it is not yet clear if such interactions could be additive and hence can lead to enhanced favorable interactions between a H₂ molecule and multiple aromatic rings in tailored frameworks. As a test model, we envision a molecular square constructed of four benzene rings interacting simultaneously with a single H₂ molecule, resides in the center of the square, as a potential model for a material with enhanced H₂ binding affinity. In this model, the H₂ molecule is located at uniform distance, R , between its center of mass and the centroids of surrounding benzene rings. It is obvious that, due to high dependence of dispersive interactions on R , decreases as $R^{-1/6}$, any expected enhancement in H₂ binding affinity due to simultaneous dispersive interactions within the optimized molecular square geometry will be extremely sensitive to geometric deformations. Although this places a challenge on experimentally attainable structures, with such strict configurations, it provides motivation for further theoretical and experimental investigations which could eventually result in porous crystalline materials with desirable H₂ uptake characteristics.

Long-range dispersive interactions are not captured well by conventional density functional methods or by ground-state Hartree-Fock calculations. [184] To account explicitly for electron correlation, second order Møller-Plesset perturbation theory (MP2) or resolution-of-the-identity Møller-Plesset [185] (RI-MP2) is used in all the calculations conducted herein. The basis sets selected are those of Dunning [78] which present a systematic series of basis functions that allow one to estimate the energy of a system in the limit of an infinitely large basis set – the so-called complete basis set (CBS) limit. [129] Although it has been observed that CBS extrapolations including the double-zeta basis set are not optimal, quadruple-zeta calculations were too time consuming and considered unnecessary for an estimation using a model system. A two-point extrapolation was performed using the double-zeta (cc-pVDZ) and

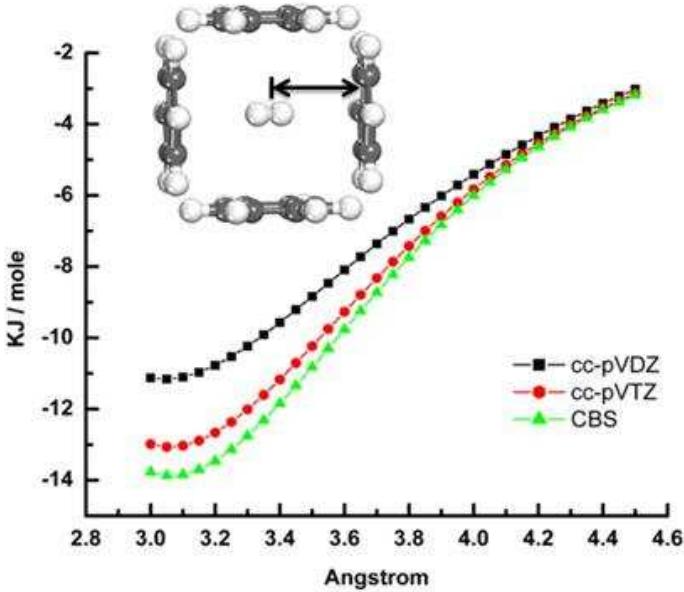


Figure 7.2: MP2/6-31G* optimized model molecular square showing the most favorable orientation of an H₂ molecule interacting simultaneously with the four benzene rings and the binding energy dependence on H₂-benzene ring separation distance, R .

triple-zeta (cc-pVTZ) basis sets. Double-zeta, triple-zeta and CBS energies are shown in Figure 7.2.

A simple model system was constructed utilizing benzene rings as building blocks. Initially, an individual benzene ring was geometrically-optimized at the MP2/6-31G* level of theory to attain a reference geometry from which the model system was constructed. Four benzene rings were arranged in a square geometry, such that the configuration roughly mimics a channel or a box in a microporous material occupied by a H₂ molecule with its center-of-mass coincides with that for the square. For simplicity, the square geometry was maintained while the dimensions were varied from 3.0 Å to 4.5 Å in increments of 0.05 Å, measured from center-of-mass of the square to center-of-mass of a benzene ring. For each step, one hydrogen molecule was positioned in the center-of-mass of the square and optimized at the MP2/6-31G* level while the square was held fixed. Using this optimized geometry,

binding energies were computed using resolution-of-the-identity MP2 with Dunning's cc-pVDZ and cc-pVTZ basis sets with their corresponding RI-fitting basis sets. [186] All binding energies were counter-poise corrected and extrapolated to the complete basis set limit. All computations were performed with NWChem. [187,188]

The calculations revealed dispersive interactions between the aromatic walls of the model box and the H₂ molecule reaching a maximum binding energy of 13.8 kJ/mol at $R = 3.05 \text{ \AA}$. Considering the configuration of the hydrogen molecule relative to the four benzene rings, the calculated energy suggests an additive behavior of aromatic ring-H₂ interactions as compared to H₂ molecule interacting with one aromatic ring.

Chapter 8

Rapidly Convergent Iterative Techniques Toward the Solution of Many-body Molecular Polarization Field Equations

8.1 Introduction

Molecular polarization was explicitly included in the Monte Carlo simulations by use of the Thole-Applequist model. [27–29] This model treats the system in terms of site (atomic) point dipoles that interact *via* many-body polarization equations. Once atomic point polarizabilities are fit to a training set of molecules the model has been shown to accurately reproduce molecular/system dipoles in a transferable (*i.e.* system-independent) manner. [27, 29] This model of explicit polarization has been successfully applied in numerous areas where inclusion of polarizable effects is paramount, such as vibrational spectroscopy, [30, 68, 69] liquid dynamics, [70–73] and biomolecules. [74, 75]

Consider a static electric field applied to a molecule. The induced dipole on this molecule will be equal to

$$\vec{\mu}_{mol} = \alpha_{mol} \vec{E}^{stat} \quad (8.1)$$

where α_{mol} is the 3×3 molecular polarizability tensor unique to that molecule. We now consider the molecular dipole as being a sum of atomic point dipoles, one for

each atom of the molecule. If we label each atomic point dipole vector $\vec{\mu}_i$ then we have

$$\begin{aligned}\vec{\mu}_i &= \alpha_i \vec{E}_i^{stat} \\ \vec{\mu}_{mol} &= \sum_i^N \vec{\mu}_i\end{aligned}\tag{8.2}$$

where α_i is the 3×3 site polarizability tensor and \vec{E}_i^{stat} is the electrostatic field at the site. In the Thole-Applequist model the system is treated as a collection of N dipoles along with a dipole field tensor $T_{ij}^{\alpha\beta}$ which contains the complete set of induced dipole-dipole interactions. This dipole field tensor, when contracted with the system dipoles, yields the (many-body) induced-dipole contribution to the electric field - this contribution is denoted here as \vec{E}^{ind} . Since the dipole field tensor (by construction) contains the entire induction contribution, we can assign a *scalar* point polarizability, α_i° , to each site rather than a polarizability tensor:

$$\alpha_i \vec{E}_i^{stat} = \alpha_i^\circ \left(\vec{E}_i^{stat} + \vec{E}_i^{ind} \right)\tag{8.3}$$

$$= \alpha_i^\circ \left(\vec{E}_i^{stat} - T_{ij}^{\alpha\beta} \vec{\mu}_j \right)\tag{8.4}$$

This equivalence can be demonstrated by reproducing the site polarizability tensors *via*

$$A\vec{\mu} = \vec{E}^{stat}\tag{8.5}$$

$$\vec{\mu} = B\vec{E}^{stat}\tag{8.6}$$

where $\vec{\mu}$ and \vec{E}^{stat} are supervectors formed by stacking the system dipole/field vectors:

$$\vec{\mu} = \begin{pmatrix} \vec{\mu}_1 \\ \vec{\mu}_2 \\ \vec{\mu}_3 \\ \vdots \\ \vdots \\ \vdots \\ \vec{\mu}_N \end{pmatrix}$$

$$\vec{E}^{stat} = \begin{pmatrix} \vec{E}_1^{stat} \\ \vec{E}_2^{stat} \\ \vec{E}_3^{stat} \\ \vdots \\ \vdots \\ \vdots \\ \vec{E}_N^{stat} \end{pmatrix}$$

and the matrices A and B are defined by

$$A = \left[(\alpha^\circ)^{-1} + T_{ij}^{\alpha\beta} \right] \quad (8.7)$$

$$B = A^{-1}$$

Just as A is a supermatrix composed of 3×3 block elements T_{ij} , B can be decomposed as [29]

$$B = \begin{pmatrix} B_{11} & B_{12} & \bullet & B_{1N} \\ B_{21} & B_{22} & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet \\ B_{N1} & \bullet & \bullet & B_{NN} \end{pmatrix}$$

where each block element B_{ij} is a 3×3 matrix. These block elements B_{ij} are the site polarizability tensors and thus characterize the site's response to an electric field. For example, the matrix B_η formed by summing the η^{th} row of B

$$B_\eta = (B_{\eta 1} + B_{\eta 2} + \dots + B_{\eta N})$$

determines the dipole response to a field for site η as a function of all N sites since (making use of Equation 8.6)

$$\begin{aligned}\vec{\mu}_\eta &= B_{\eta 1} \vec{E}_1^{stat} + B_{\eta 2} \vec{E}_2^{stat} + \dots + B_{\eta N} \vec{E}_N^{stat} \\ &= \vec{\mu}_\eta(1) + \vec{\mu}_\eta(2) + \dots + \vec{\mu}_\eta(N)\end{aligned}$$

whereby each μ term represents a contribution toward η 's dipole. Therefore, summing all of the ij blocks over the tensor components $\alpha\beta$ for an appropriate set of sites yields the molecular polarizability tensor: [29]

$$\alpha_{\alpha\beta}^{mol} = \sum_{i,j} (B_{ij})_{\alpha\beta} \quad (8.8)$$

The Applequist dipole field tensor [27] can be derived from first principles as

$$\begin{aligned}T_{ij}^{\alpha\beta} &= \nabla_\alpha \nabla_\beta \frac{1}{r_{ij}} \\ &= \frac{\delta_{\alpha\beta}}{r_{ij}^3} - \frac{3x^\alpha x^\beta}{r_{ij}^5}\end{aligned} \quad (8.9)$$

The most direct way to calculate the system dipoles is through Equation 8.6. However, since inversion of the $3N \times 3N$ matrix A is computationally efficient for only the smallest systems the dipoles must be solved for by an iterative method. The iterative method employed here makes an initial guess of $\vec{\mu}_i = \alpha_i^\circ \cdot \vec{E}_i^{stat}$ and then iteratively solves Equation 8.4 until convergence is achieved.

The Thole model introduces the additional consideration of treating each dipole as interacting with a well-behaved charge distribution $\rho(u)$ (in contrast to the Applequist model where the dipole field tensor is derived by considering a dipole interacting with a point charge, giving rise to Equation 8.9), which results in a mod-

ified form of the dipole field tensor. The net result of this modification is that the charge which induces each dipole is “smeared” at short range, and it is this additional structure that Thole added to the model which imparts transferability. One such exponential distribution [28] found to accurately and transferably [29] reproduce molecular dipoles for an associated series of dependent polarizabilities is

$$\begin{aligned}\rho(u_{ij}) &= \frac{\lambda^3}{8\pi} e^{-\lambda u_{ij}} \\ u_{ij} &= x_{ij} (\alpha_i^\circ \alpha_j^\circ)^{-\frac{1}{6}}\end{aligned}\tag{8.10}$$

where the free parameter λ has the effect of damping the dipole interactions near the regions of discontinuity that would otherwise exist in the Applequist model. The coordinate scaling of $x_{ij} \rightarrow u_{ij}$ is done for convenience in order to allow simple functional forms such as Equation 8.10 to be used for damping. Taking the exponential charge distribution into account, the modified dipole field tensor becomes [29]

$$\begin{aligned}\hat{T}_{ij}^{\alpha\beta} &= \frac{\delta_{\alpha\beta}}{r_{ij}^3} \left[1 - \left(\frac{\lambda^2 r_{ij}^2}{2} + \lambda r_{ij} + 1 \right) e^{-\lambda r_{ij}} \right] \\ &\quad - \frac{3x^\alpha x^\beta}{r_{ij}^5} \left[1 - \left(\frac{\lambda^3 r_{ij}^3}{6} + \frac{\lambda^2 r_{ij}^2}{2} + \lambda r_{ij} + 1 \right) e^{-\lambda r_{ij}} \right]\end{aligned}\tag{8.11}$$

The many-body potential energy due to the interaction of the induced dipoles (referred to as the polarization energy) is described by [28]

$$\begin{aligned}U_{pol} &= \sum_i \frac{1}{2} \vec{\mu}_i \cdot A \vec{\mu}_i - \vec{\mu}_i \cdot \vec{E}_i^{stat} \\ &= \sum_i \frac{1}{2} \vec{\mu}_i \cdot \vec{E}_i^{stat} - \vec{\mu}_i \cdot \vec{E}_i^{stat} \\ &= -\frac{1}{2} \sum_i \vec{\mu}_i \cdot \vec{E}_i^{stat}\end{aligned}\tag{8.12}$$

where it should be pointed out that \vec{E}_i^{stat} is not the total electric field, but rather only the static electric field due to the presence of the partial charges present in the system.

8.2 Iterative Methods for Many-body Polarization

Calculating the polarization energy for the system amounts to self-consistently solving the dipole field equation for each atomic dipole vector $\vec{\mu}_i$ through an iterative process until a sufficient degree of precision is achieved. Thus, to make the calculations practical, efficient methods of solving the field equations were required. Typically, a simultaneous over-relaxation scheme (*i.e.* linear solution mixing),

$$\mu_i^{k+1} = \gamma \mu_i^k + (1 - \gamma) \mu_i^{k-1} \quad (8.13)$$

is used to improve the convergence rate for the iterative method of solution. [71] However, recently a number of multigrid techniques [76] have been applied to the Thole model for a one-dimensional system [77] and a similar Gauss-Seidel smoothing technique was applied here and found to reduce the number of iterations required for convergence by two-fold over linear mixing. The applied Gauss-Seidel numerical iteration method for a slowly converging process consists of updating the current dipole vector set for the k^{th} iteration step as the new dipole vectors become available:

$$\begin{aligned} \vec{\mu}_i^k &= \alpha_i^\circ \left(\vec{E}_i^{stat} - \hat{T}_{ij}^{\alpha\beta} \vec{\mu}_j^{k-1+\zeta} \right) \\ \zeta &= \begin{cases} 0, & \text{if } i < j \\ 1, & \text{if } i > j \end{cases} \end{aligned} \quad (8.14)$$

While the Gauss-Seidel iterative method has been used in numerous areas of science and mathematics for decades, a novel approach was taken here to further

improve the iterative convergence of the polarization model employed. [4] It may be noticed that the *order* in which the Gauss-Seidel updates are made available is completely unspecified, since there are no unique ordering of indices implied in Equation 8.14. In this work, an algorithmic approach has been taken whether the dipoles that stand to vary the most between iterations are updated first.

The algorithms proceed by assigning a metric to each dipole in order to approximately predict it's variance throughout the iterative subspace. One such algorithm is:

```
for i, j
    if r(ij) < r(min)
        metric(i)++
```

where $r(min)$ is the minimum separate of polarizable sites found for that configuration.

A slightly different variation takes into account the polarizability of both sites:

```
for i, j
    if r(ij) < r(min)
        metric(i) += \alpha(i) \alpha(j)
```

Since the pairs i, j must be evaluated in the code anyway, there is little performance penalty is calculating these metrics. However, once the metrics have been found the dipoles must then be sorted, in order of greatest metric to least metric, prior to being utilized in the SCF. (In this work a simple bubble sort was used for convenience, however production HPC work would benefit from better scaling sorting methods – in either case, there is a substantial net computational savings by virtue of the fact that fewer iterations will need to be done with this method).

Furthermore, it has been repeatedly observed that the tendency is for the dipoles to grow in magnitude as a function of iterative progress. Therefore, a precondition transformation of the following form has been applied to the dipole vectors:

$$\mu^0 = P \cdot (\alpha E^{stat}) \quad (8.15)$$

$$P = \gamma I \quad (8.16)$$

$$(8.17)$$

where γ is typically between 1.0-1.4. This is the most simple preconditioning step one can take, and its effects (as shown in the next section) are fairly dramatic. There are more sophisticated preconditioners, one can imagine, that would improve the convergence properties even further (e.g. a preconditioning matrix P as a diagonal form of the dipole field tensor, etc.).

It has been reported in the literature that if a perturbative correction to the polarization energy is made, that the energy will converge much more rapidly than the dipoles. [189, 190] We will refer to this method, due to Palmo and Krimm, as the “PK” method and it results in the following energy expression as distinct from Equation 8.12:

$$U_{pol}^k = -\frac{1}{2} \sum_i \mu_i^k E_i^{stat} - \frac{1}{2} \sum_i \mu_i^k E_i^{k+1} \quad (8.18)$$

$$(8.19)$$

In procedure, this means that the energy is dependent upon the dipoles on the k -th iteration step with the induced field from the $k+1$ -th step. Since the polarization energy is of prime importance to the Monte Carlo methods widely used here, this PK technique has been implemented and further enhanced with the Gauss-Seidel scheme

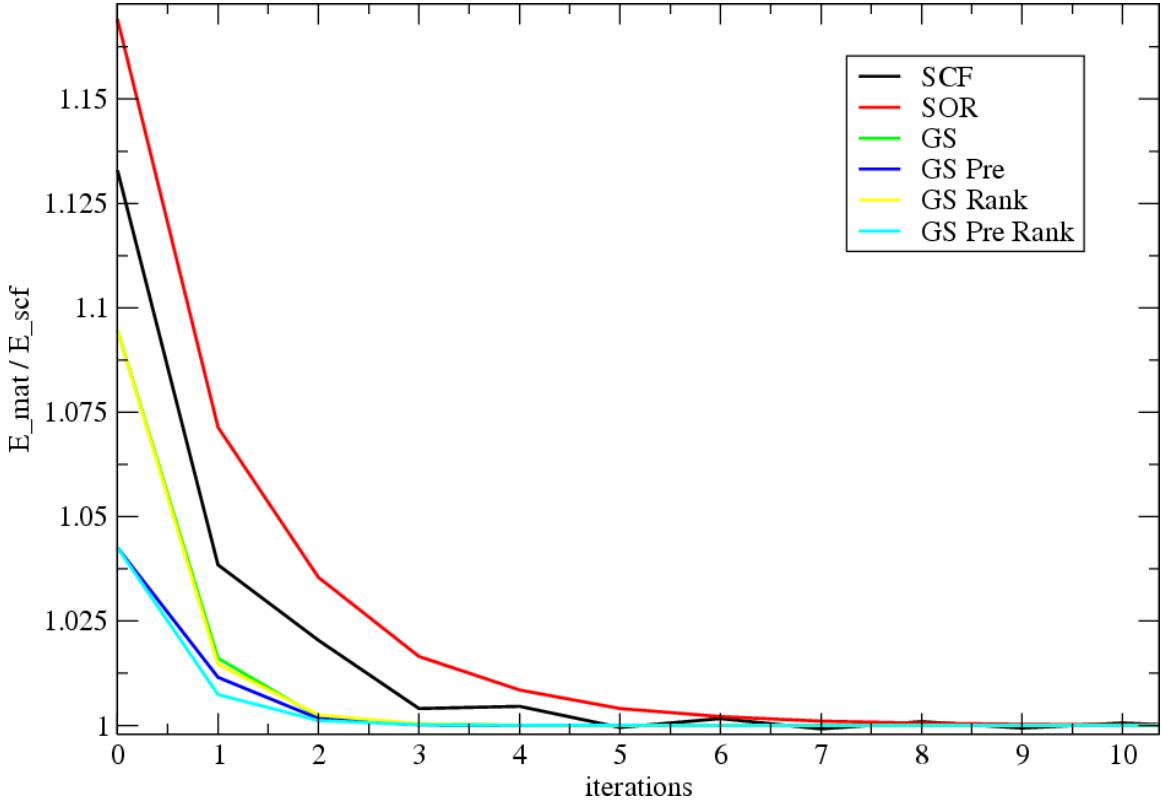


Figure 8.1: The iterated energy vs. the exact matrix inversion energy as a function of iterative convergence for various SCF methods.

described above.

8.3 Results

As a test case, states have been generated for 64 SPCF water molecules fixed to the liquid density at 298 K using NVT Monte Carlo. Water is an excellent test case for investigating electrostatic induction since the dipole magnitude of the condensed phase is dramatically different from the gas phase (≈ 2.8 D vs. 1.5 D).

Shown in Figure 8.1 is a comparison of the SCF energy vs. the exact matrix inversion energy for a number of iterative scheme previously described. It should noted that the best performing SCF technique is Gauss-Seidel iterations with the metric ranking scheme and preconditioning. It is also found that when applied to the

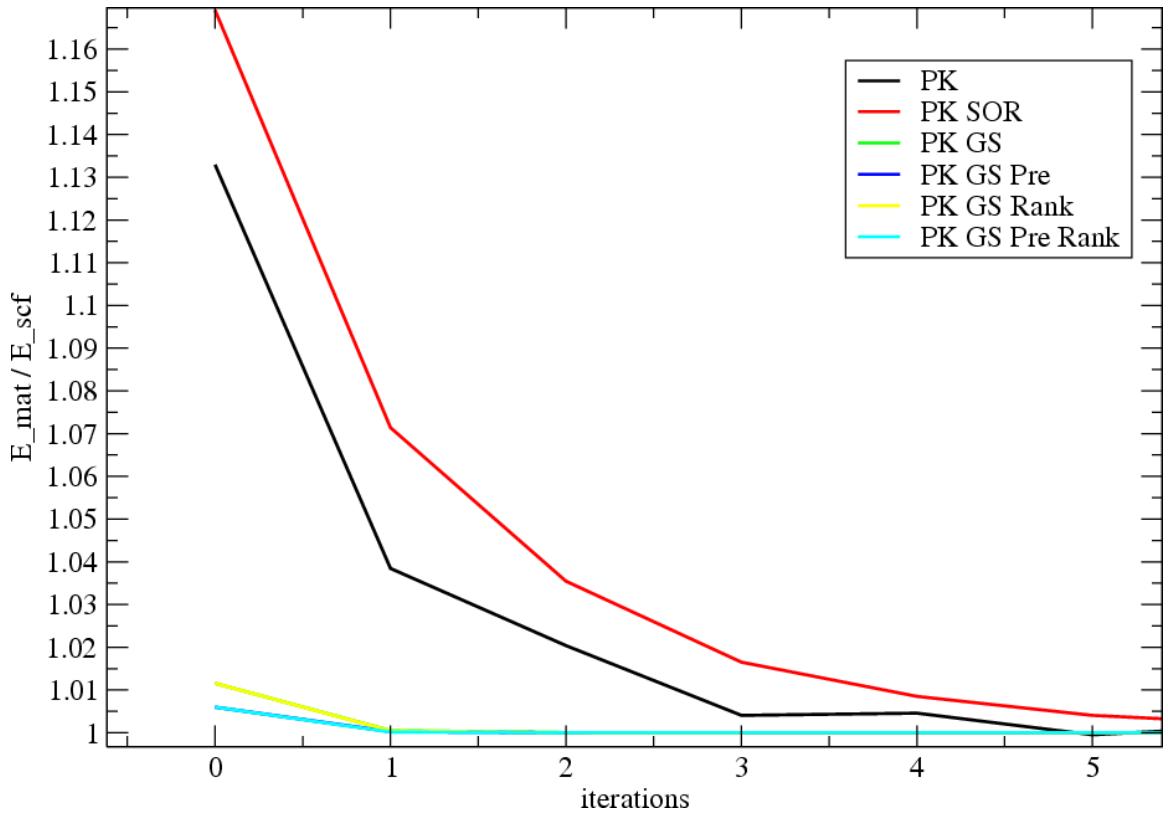


Figure 8.2: The iterated PK energy vs. the exact matrix inversion energy as a function of iterative convergence for various SCF methods.

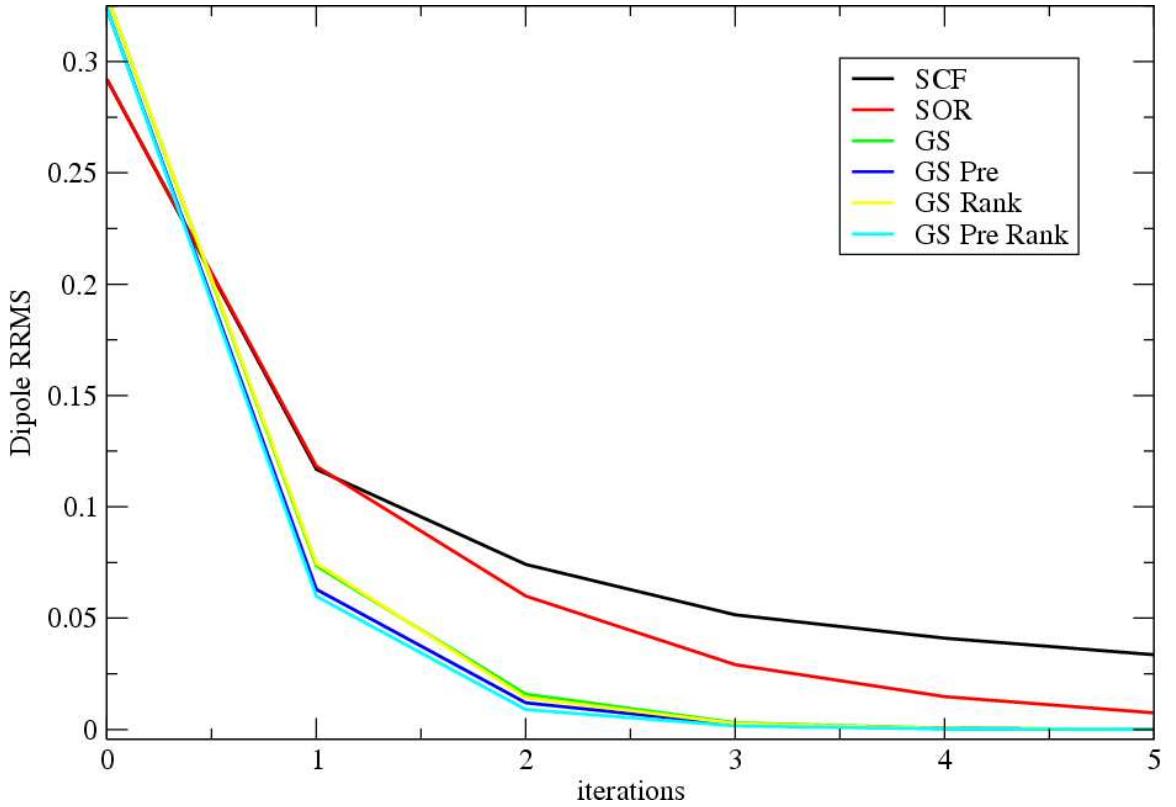


Figure 8.3: Relative root mean-square deviation of the dipoles as a function of iterative progress for various SCF methods.

PK energy correction method, the energy is dramatically better converged, as shown in Figure 8.2.

Furthermore, as is evident in Figure 8.3, not only is the polarization energy more convergent but the relative root mean square of the dipoles is greatly reduced. This example illustrates that with very little extra computation work both the energy and dipoles are more accurately converged with the novel iteration scheme used here.

Table 8.1 shows that as a 2-step iteration scheme, the iterative techniques employed have resulted in convergence of the energy to within 0.02 %, as compared to the PK method (which was previously considered the most effective technique) which is less than 4 %. In addition, the RRMS of the dipoles is less than 6 %, with further reduction in RRMS proceeding with increased iterations. The computational savings that have been resulted from these techniques have been enormous, as the number

Iter type	PK E _{mat} /E _{scf} %	Error RRMS %
SCF	3.84	11.7
SOR	7.14	11.8
GS	0.049	7.4
GS Pre	0.024	6.3
GS Rank	0.045	7.5
GS Pre Rank	0.016	5.9

Table 8.1: Comparison of error for the various SCF methods as a 2-step iteration scheme

of iterations necessary to get well-converged energies for hydrogen in polar MOFs is typically 10 – but, as has been shown here, the same level of accuracy can be had with a 2-step iteration scheme.

Chapter 9

Calculation of Rotational Spectra for Sorbed Hydrogen in Metal-Organic Materials

This chapter described collaborative work that is ongoing with Zlatko Bačić (NYU), Ivana Matanović (NYU) and Juergen Eckert (UCSB/LANL) to explore the quantum mechanical aspects of hydrogen rotation near surfaces. Using newly developed potentials that described the hydrogen-MOM interface, we solve the Schrodinger equation for each H_2 and thus obtain the rotational energy levels. The methodology and code-base developed allow for the inclusion of *ortho/para* nuclear spin effects in Monte Carlo simulation and comparison with experimental rotational spectra.

9.1 Introduction

The quantum mechanical aspects of the rotation of a hydrogen molecule are fascinating and present an interesting historical perspective into the great success of quantum statistical mechanics at resolving the observed heat capacity of hydrogen.

Consider protium H_2 , whose nuclei are fermions and thus the overall symmetry of the hydrogen wavefunction must obey the Pauli principle by changing sign upon nuclear inversion. Writing the total wavefunction as a product:

$$\psi_{H_2} = \psi_{nuc}\psi_{elec}\psi_{trans}\psi_{vib}\psi_{rot} \quad (9.1)$$

where we are considering the various contributions to the hamiltonian as being seperable. We note that, in the electronic ground state, ψ_{elec} is symmetric under inversion. Translational/vibration symmetry analysis results in the same conclusion.

However, the nuclear spin eigenfunctions consist of 3 symmetric solutions ($\alpha\alpha$, $\beta\beta$, $\alpha\beta + \beta\alpha$) and 1 antisymmetric solution ($\alpha\beta - \beta\alpha$). Since ψ_{H_2} must remain antisymmetric overall, this requires that *the rotational eigenfunctions may only be symmetric or antisymmetric, depending upon the nuclear spin state.*

For a rigid rotor, the rotational eigenfunctions are the familiar spherical harmonics, whose symmetry alternates from g/u along the energy spectrum. Therefore, if the nuclear spins happen to be parallel then the molecule will be in a rotational state corresponding to $J = \text{even}$, and $J = \text{odd}$ for antiparallel nuclear spins. These distinct hydrogen molecules are called *para* or *ortho* hydrogen, respectively, and for highly excited hydrogen will occur in the 1:3 ratio corresponding to the nuclear spin degeneracies. [191] Following the same considerations for deuterium, tritium or combinations thereof, produce different population ratios.

Nearly all substances exist in the solid state at temperatures low enough to observe selective occupation of rotational energy levels due to these nuclear spin effects. However, hydrogen is unique in that it is still a gas where these effects are dominant! (The characteristic rotational temperature of hydrogen is 85 K).

Of further interest is the fact that the barrier to *ortho/para* interconversion is extremely high, and thus it is possible to cool hydrogen from a gas to a liquid and still have 1:3 ratio of populations, despite the fact that all of the hydrogen should be the $J = 0$ (*para*) state. This non-equilibrium coexistence will occur typically for weeks

before magnetic interactions with the environment drive interconversion to the ground state. This poses an enormous engineering challenge for cryogenic hydrogen storage applications, since the heat released upon interconversion larger than the enthalpy of vaporization and so there are constant LH₂ boil-off issues. Typically, metal oxide catalysts are employed to help the interconversion process along as the hydrogen is condensed.

Selective adsorption and/or interconversion of hydrogen nuclear spin isomers in Metal-Organic Frameworks has not been previously explored theoretically (although such studies have been undertaken for hydrogen in zeolites [18,36]). Herein, a description of the methodology and source code for allowing such calculations is explained along with results of this applied system.

9.2 Methods and Results

For each H₂ in the system, the goal is to diagonalize the rotor hamiltonian in a spherical harmonic basis. Each matrix element, $\langle Y_{lm}|V(\theta, \phi)|Y_{lm}\rangle$ is constructed by Gauss-Legendre quadrature (16 point integration) using a moderately sized basis set consisting of both +m,-m functions. The purely kinetic energy (i.e. rigid rotor part) $l(l+1)$ is added to the diagonal elements. The hermitian matrix is then diagonalized using the LAPACK linear algebra package, yielding the rotational energy eigenvalues and the eigenvector coefficients.

For example, the energy levels of the hindered rotor potential of Curl et al, [192]

$$V(\theta, \phi) = V_0 \sin^2 \theta \quad (9.2)$$

have been found using the MPMC code (described in Appendix L) and the levels are shown in Figure 9.1. In this Figure, we observe the well-known rotational tunnel splitting effect whereby in the presence of a rotational barrier the rotorsational energy

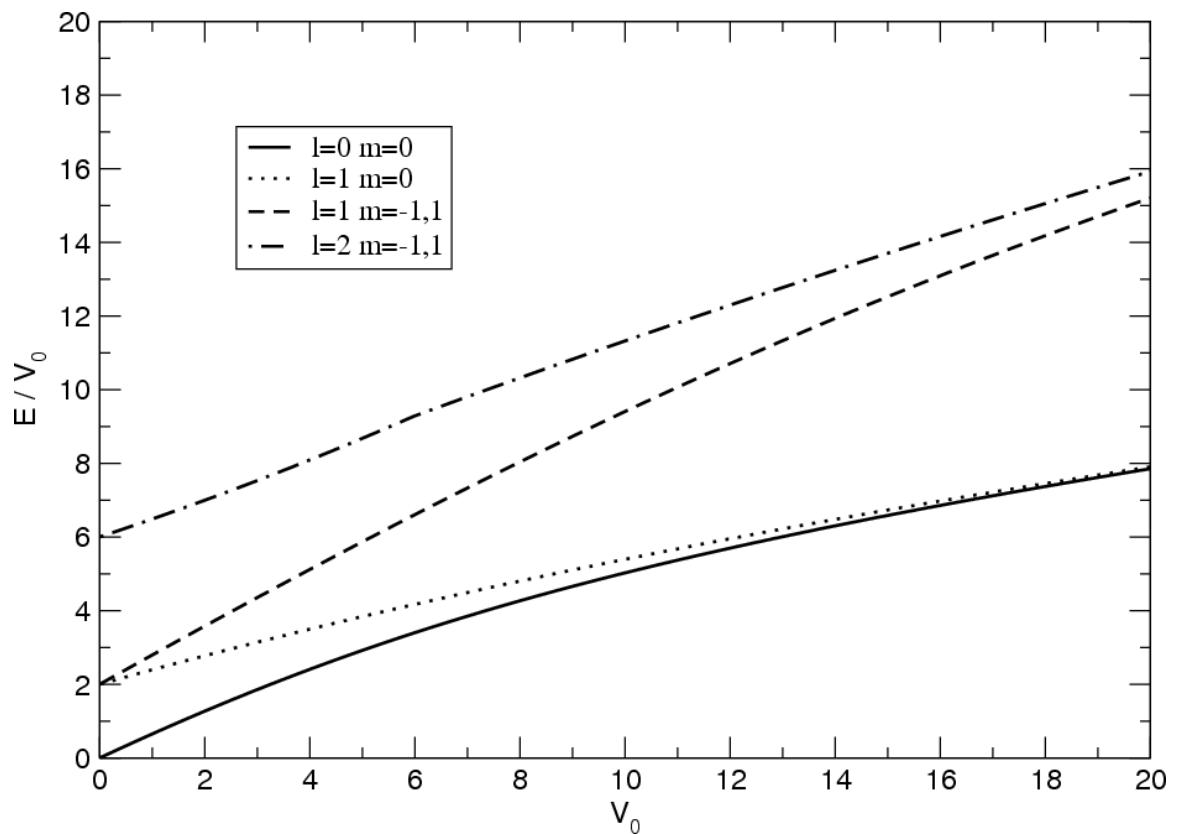


Figure 9.1: Rotational tunnel splitting for a hindered rotor

levels split and group according to their m values (depending upon the symmetry of the barrier). Note that the $m = 0$ levels split off from the other m values since there is no ϕ dependence in the potential.

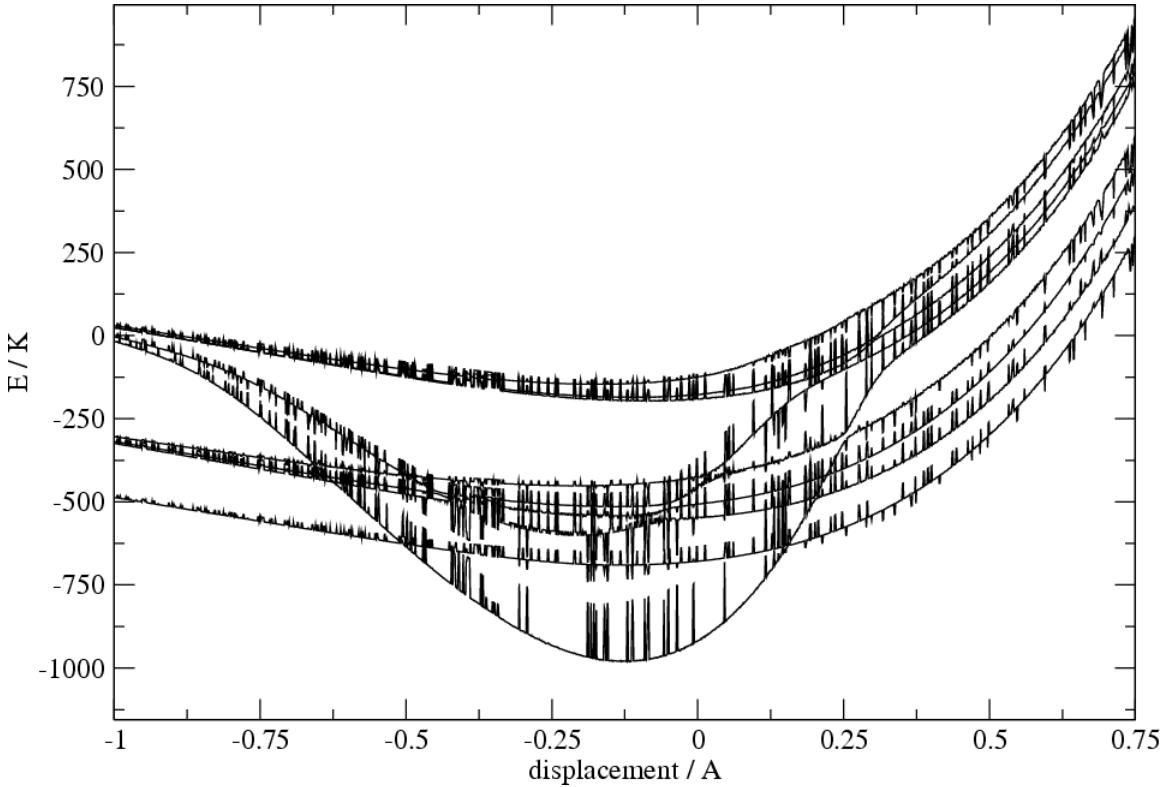


Figure 9.2: Rotational tunnel splitting for H_2 along a displacement vector from the global minimum energy adsorption site.

In a MOF, however, the energy levels may split in more complex ways given the curvature of the potential energy surface. The rotational energy levels of hydrogen in the vicinity of the global energy minimum binding site in MOF-5 have been calculated as a function of center-of-mass displacement from this site, and are shown in Figure 9.2. The global energy minimum configuration had been previously determined *via* simulated annealing. It should be pointed out that along the negative displacement (i.e. in the pore) the hydrogen is rigid rotor-like, with its associated degeneracies. To the far right, up against a potential barrier, we see complete splitting of energy levels. However, in the region near the minimum we see more complicated spacing

of the levels, with some levels completely shifting with respect to one another. (As a result, the MPMC codebase determines the symmetry of each eigenvector individually). Transitions between such energy levels (observed in, for example, INS spectra) can now be associated with specific barriers to the binding site in the MOF using this methodology.

Chapter 10

Microcanonical Effective Partition Function

The microcanonical effective partition function, constructed from a Feynman-Hibbs-Kleinert potential, is derived using generalized ensemble theory. The form of the potential is amenable to *NVE* Monte Carlo simulation techniques and the relevant Metropolis function is presented. The low-temperature entropy of a proton in an anharmonic potential is numerically evaluated from the microcanonical effective partition function and found to be accurate compared to the canonical technique.

10.1 Introduction

The Green's function for the quantum propagator,

$$G(x', t'; x, t) = \langle x' | e^{-i\hat{H}(t' - t)/\hbar} | x \rangle \quad (10.1)$$

can be expressed in it's path integral form [193], after Trotter factorization and making use of the resolution of the identity, as: [194]

$$\begin{aligned} G(x', t'; x, t) &= \lim_{P \rightarrow \infty} \int_{-\infty}^{\infty} dx_1 \dots dx_{P-1} \left(\frac{m}{2\pi i \hbar \epsilon} \right)^{P/2} \\ &\times e^{\frac{i\epsilon}{\hbar} \int_t^{t'} d\tau \left\{ \frac{1}{2} m \left(\frac{dx}{d\tau} \right)^2 - V[x(\tau)] \right\}} \end{aligned} \quad (10.2)$$

where the time interval $\epsilon = (t' - t)/P$ and the path from $x \rightarrow x'$ has been discretized among P points. Analytically continuing Eq. (10.2) *via* the substitution $\beta = it/\hbar$ and letting the initial time $t = 0$ results in

$$G(x', -i\beta\hbar; x) = \lim_{P \rightarrow \infty} \int_{-\infty}^{\infty} dx_1 \dots dx_{P-1} \left(\frac{m}{2\pi\hbar^2\beta} \right)^{P/2} \times e^{-\frac{1}{\hbar} \int_0^{\beta\hbar} d\tau \left\{ \frac{1}{2}m \left(\frac{dx}{d\tau} \right)^2 + V[x(\tau)] \right\}} \quad (10.3)$$

It can be shown that the canonical partition function $Q(N, V, \beta)$ results from taking the trace of expression (10.3), where the paths propagate from $x \rightarrow x$:

$$\begin{aligned} Q(N, V, \beta) &= \int_{-\infty}^{\infty} dx G(x, -i\beta\hbar; x) \\ &= \int_{-\infty}^{\infty} dx \lim_{P \rightarrow \infty} \int_{-\infty}^{\infty} dx_1 \dots dx_{P-1} \left(\frac{m}{2\pi\hbar^2\beta} \right)^{P/2} \times e^{-\frac{1}{\hbar} \int_0^{\beta\hbar} d\tau \left\{ \frac{1}{2}m \left(\frac{dx}{d\tau} \right)^2 + V[x(\tau)] \right\}} \end{aligned} \quad (10.4)$$

and the integration is done over all possible closed paths that start and end at x . A great deal of applied research has proceeded from the approximation whereby Eq. (10.4) is closed for a finite Trotter number P . Indeed, in such a form expression (10.4) then looks very much like a classical partition function (albeit with an odd-looking β -dependent harmonic term) and can be numerically evaluated by many-dimensional integration techniques such as Monte Carlo.

However, an alternative approach is to approximate the integrals over $\int_{-\infty}^{\infty} dx_1 \dots dx_{P-1}$ *via* a variational principle [195] the result of which is then expressed as an exponential of an *effective potential* $\widetilde{W}(x)$:

$$Q(N, V, \beta) \approx \tilde{Q}(N, V, \beta) = \int_{-\infty}^{\infty} dx \sqrt{\frac{m}{2\pi\hbar^2\beta}} e^{-\beta\tilde{W}(x)} \quad (10.5)$$

where $\tilde{Q}(N, V, \beta)$ is the *canonical effective partition function*; in the high-temperature limit \tilde{Q} is equivalent to Eq. (10.4). In its original formulation, $\tilde{W}(x)$ minimizes the variation if taken to be a gaussian-smeared potential,

$$\tilde{W}(x) = \int_{-\infty}^{\infty} dy \frac{1}{\sqrt{2\pi a^2}} e^{\frac{(x-y)^2}{2a^2}} U(y) \quad (10.6)$$

where in its first approximation the gaussian width $a^2 = \beta\hbar^2/12m$ and the same fixed-width approximation is made here. Techniques that improve upon the fixed-width approximation have been previously made. [196, 197] The Taylor series expansion of expression (10.6) yields the familiar terms commonly used in molecular simulation.

The canonical effective partition function has been of significant value in numerical statistical mechanics since it includes quantum fluctuations while preserving the classical mathematical structure of the partition function. In the simplest approximation of a fixed-gaussian width of $\beta\hbar/12m$ the effective approach provides accuracy amenable to the semiclassical regime.

Curiously, while the path integral expression for the microcanonical partition function has been derived [198], to our knowledge the microcanonical effective partition function has not been reported in the literature. While the canonical ensemble is quite natural for the study of many physical systems, there are cases where the microcanonical ensemble is more convenient since the thermodynamic energy can be fixed. Our intention here is to develop the microcanonical effective partition function such that quantum fluctuations may be included in, for instance, *NVE* Monte

Carlo [7, 8] simulations.

10.2 Microcanonical Derivation

We derive the microcanonical effective partition function through application of generalized ensemble theory [199–201] which allows us to relate the constant energy shell ensembles *directly* to other ensembles in which thermal energy can flow between system and bath. Of immediate interest is the Laplace transform relationship between the canonical and microcanonical partition functions, so we demonstrate with the following example:

$$Q(N, V, \beta) = \int_0^\infty dE e^{-\beta E} \Omega(N, V, E) \quad (10.7)$$

since the energy spectrum may always be shifted such that the lower bound is zero; additionally, of course $\beta \in \mathbb{C}$. In solving for the microcanonical partition function, the inverse Laplace transform yields:

$$\begin{aligned} \Omega(N, V, E) &= \frac{1}{2\pi i} \oint d\beta e^{\beta E} Q(N, V, \beta) \\ &= \int_{-\infty}^{\infty} d\Omega \frac{1}{2\pi i} \int_{\gamma-i\infty}^{\gamma+i\infty} d\beta e^{\beta(E-H)} \end{aligned} \quad (10.8)$$

where $d\Omega$ is the phase space differential form $h^{-1}dx dp$. Since $\beta = \sigma + i\tau$ and no singularity is present in the right-half of the complex plane, we may take the contour vertically through $\gamma = 0$. Since $\text{Re}(\beta) = 0$ along the integration we may make the substitution $\beta = i\tau$:

$$\begin{aligned}
\Omega(N, V, E) &= \int_{-\infty}^{\infty} d\Omega \frac{1}{2\pi} \int_{-\infty}^{\infty} d\tau e^{i\tau(E-H)} \\
&= \int_{-\infty}^{\infty} d\Omega \delta(E - H)
\end{aligned} \tag{10.9}$$

which is the microcanonical partition function, as it should be. Another simple example is the quantum harmonic oscillator:

$$\begin{aligned}
\Omega_{HO} &= \frac{1}{2\pi i} \oint d\beta e^{\beta E} Q_{HO} = \frac{1}{2\pi} \int_{-\infty}^{\infty} d\tau e^{i\tau E} \frac{e^{-\frac{1}{2}i\tau\hbar\omega}}{1 - e^{-i\tau\hbar\omega}} \\
&= \frac{1}{2\pi} \int_{-\infty}^{\infty} d\tau e^{i\tau(E - \frac{1}{2}\hbar\omega)} \sum_n e^{-i\tau\hbar\omega n} \\
&= \sum_n \delta \left[E - \hbar\omega \left(n + \frac{1}{2} \right) \right]
\end{aligned} \tag{10.10}$$

10.3 Anharmonic Oscillator and Numerical Evaluation

Along similar lines, we wish to construct the microcanonical effective partition function $\tilde{\Omega}(N, V, E)$ from $\tilde{Q}(N, V, \beta)$ through use of the same Laplace structure. Proceeding in this manner,

$$\begin{aligned}
\tilde{\Omega}(N, V, E) &= \frac{1}{2\pi i} \oint d\beta e^{\beta E} \tilde{Q}(N, V, \beta) \\
&= \frac{1}{2\pi i} \oint d\beta e^{\beta E} \int_{-\infty}^{\infty} dx \sqrt{\frac{m}{2\pi\hbar^2\beta}} e^{-\beta\tilde{W}}
\end{aligned} \tag{10.11}$$

Using Eq. (10.6), the gaussian smeared version of the anharmonic potential $U_{AHO} = \frac{1}{2}kx^2 + \frac{1}{4}gx^4$ is exactly integrable and yields

$$\widetilde{W}_{AHO} = U_{AHO} + \frac{\beta\hbar^2}{24m} (k + 3gx^2) + \left(\frac{\beta\hbar^2}{24m}\right)^2 3g \quad (10.12)$$

$$= U_{AHO} + W_1(\beta) \quad (10.13)$$

for the effective potential. With the momentum integration having been undone, the inverse Laplace transform now becomes:

$$\begin{aligned} \widetilde{\Omega}_{AHO}(E) &= \int_{-\infty}^{\infty} d\Omega \frac{1}{2\pi i} \oint d\beta e^{\beta[E-H-W_1(\beta)]} \\ &= \int_{-\infty}^{\infty} d\Omega \frac{1}{2\pi i} \oint d\beta e^{\beta(E-H)} e^{-\beta^2 \left[\frac{\hbar^2}{24m} (k+3gx^2) \right]} e^{-\beta^3 \left(\frac{\hbar^2}{24m} \right)^2 3g} \\ &= \int_{-\infty}^{\infty} d\Omega \frac{1}{2\pi i} \oint d\beta e^{a\beta - b\beta^2 - c\beta^3} \end{aligned} \quad (10.14)$$

where the contour integral can be approximated by the method of steepest descent to yield

$$= \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} d\Omega \frac{e^{a\beta_0 - b\beta_0^2 - c\beta_0^3}}{(4b^2 + 12ac)^{\frac{1}{4}}} \quad (10.15)$$

where the saddle points, β_0 , are

$$\beta_0 = \frac{\pm\sqrt{4b^2 + 12ac} - 2b}{6c} \quad (10.16)$$

recalling that the following substitutions have been made,

$$a = E - H$$

$$b = \frac{\hbar^2}{24m} (k + 3gx^2)$$

$$c = \left(\frac{\hbar^2}{24m} \right)^2 3g$$

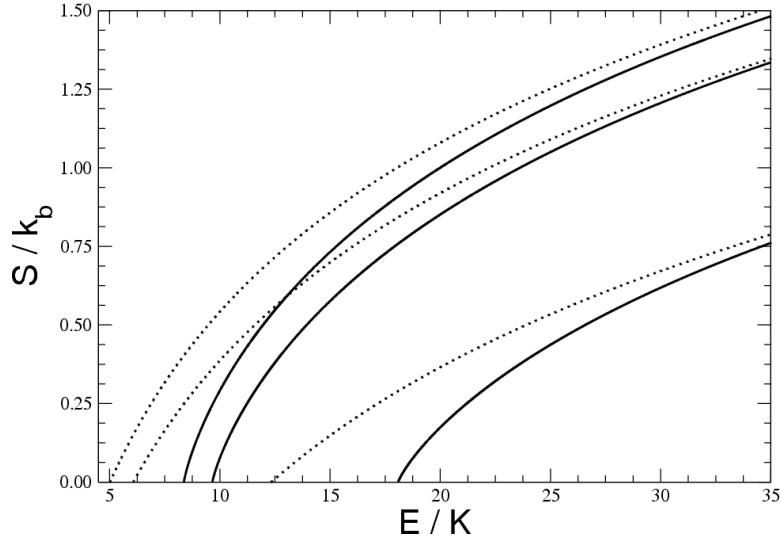


Figure 10.1: Classical entropy for a proton in an anharmonic well vs. the quantum mechanic result (canonical ensemble).

Evaluation of expression (10.15) has been performed to obtain the entropy of a proton in an anharmonic well. In Fig. 10.1, the Feynman-Hibbs-Kleinert micro-canonical entropy, $S = k \ln \tilde{\Omega}_{AHO}$, is compared with the exact classical and quantum mechanical [202] entropy found from the canonical ensemble, $S = k \ln Q + E/T$.

Chapter 11

A Student-Friendly Derivation of the Partition Function for Generalized Ensembles

The connections between the combinatoric formula $S = k \ln W$, the Gibbs entropy, $S = -k \sum_i p_i \ln p_i$, and the microcanonical entropy expression $S = k \ln \Omega$ are clarified, including addressing their range of validity. The condition for microcanonical equilibrium, and the associated role of the entropy as the thermodynamic potential is shown to arise from the postulate of equal *a priori* states. The derivation of the canonical partition function follows by invoking the Gibbs ensemble construction and the first and second law of thermodynamics (*via* the fundamental equation $dE = TdS - PdV + \mu dN$) that incorporate the conditions of conservation of energy and composition without the needs for explicit constraints. The need for explicit maximization of any function is thus also avoided. By Legendre transforming the relevant thermodynamic function to an ensemble of interest, an analogous procedure leads to the corresponding partition function. Connections to generalized ensemble theory also arise in this context. The central role of the entropy in establishing equilibrium for a given ensemble emerges naturally from the current approach.

11.1 Introduction

In deriving the partition function for a desired ensemble, the most common approach is to maximize an entropy function with constraints appropriate to the thermodynamic condition. While equivalent to the approach proposed below, such a method (called the traditional approach hereafter) does not make explicit the central role of the assumption of equal *a priori* states and the corresponding role of the entropy as the thermodynamic potential for the microcanonical ensemble. Indeed, $S = k \ln \Omega$ is often taken as a postulate [203] and its connection to the statistical formula $S = k \ln W$ (appearing on Boltzmann's tombstone) is not evident. Further, in the traditional approach, the role of the entropy in understanding equilibrium in non-isolated, open ensembles can be confusing.

Also, frequently omitted is the Gibbs entropy, $S = -k \sum_i p_i \ln p_i$, where p_i is the probability of finding a system in a given state, which can be invoked for any equilibrium ensemble and associated state probabilities. It is a direct consequence of the statistical entropy formula, $S = k \ln W$, in conjunction with the Gibbs construction of an ensemble with a large number of macroscopic subsystems, each consistent with the desired thermodynamic variables; W gives the number of possible realizations within the Gibbs construction for the ensemble under consideration. The Gibbs entropy also permits the derivation of the connection between the characteristic thermodynamic function and the partition function for a given ensemble without further appeal to thermodynamic expressions, as is required in the traditional approach.

In the present approach, first, the connection between the statistical formula $S = k \ln W$, the Gibbs entropy, $S = -k \sum_i p_i \ln p_i$, and the microcanonical entropy expression $S = k \ln \Omega$ are clarified including addressing their range of validity. The condition for microcanonical equilibrium, and the associated role of the entropy as the thermodynamic potential then arises from the postulate of equal *a priori* states. The derivation of the canonical partition function follows by invoking the Gibbs construc-

tion and the first and second law of thermodynamics *via* the fundamental equation, $dE = TdS - PdV + \mu dN$, that incorporates the conditions of conservation of energy and composition without the needs for explicit constraints. The role of the temperature (coming from the constraint of total energy and an appeal to appropriate thermodynamic relationships in the traditional approach) is immediately apparent and also introduced *via* the fundamental equation. The need for explicit maximization of any function is thus also avoided. By Legendre transforming a characteristic thermodynamic function to an ensemble of interest, an analogous procedure leads to the corresponding partition function. [204] Connections to generalized ensemble theory also arise in this context. The central role of the entropy in establishing equilibrium for a given ensemble also emerges naturally from the current approach.

The present approach is novel in proving clarity as to the roles played by the different formulas and physical quantities of interest while drawing upon portions of several existing derivations. Further, it makes explicit the assumptions inherent in deriving the partition function for an ensemble and provides its direct connection to the relevant thermodynamic potential. This approach also makes deriving the partition function for a given ensemble a simplified, straight-forward process, even for more obscure examples such as the isothermal-isobaric ensemble.

11.2 The Gibbs Entropy and the Microcanonical Ensemble

We begin by introducing the concept of an ensemble of replicas that describe the molecular states corresponding to a given macrostate; this picture is referred to as the “Gibbs construction” herein, due to its original introduction by Gibbs [205] who addressed many of the subtleties inherent [203] in the formulation of statistical mechanics. Consider a collection of macroscopic molecular subsystems of N molecules within a volume V , each of which is part of the larger Gibbs construction. No other constraints have yet been imposed, *i.e.* the system’s macrostate is otherwise unspec-

ified. It is desirable to define the microscopic statistics of this system as thoroughly as possible and then apply any other constraints at the end.

Let the total number of subsystems in our collection be known as Ω . Then let ω_i , the occupation number, denote the number of subsystems from this collection that are in the same thermodynamic state. These occupations will thus take on a large value in the thermodynamic limit and they obeys a sum rule, $\sum_i \omega_i = \Omega$. Note that technically the energy is course-grained, *i.e.* specified to within a small but otherwise arbitrary range (these arguments are presented in detail elsewhere [203, 206]) and the results are insensitive to this choice.

First, consider the following combinatoric formula:

$$S = k \ln W \{\omega\} = k \ln \frac{\Omega!}{\omega_1! \omega_2! \dots} \quad (11.1)$$

$W \{\omega\}$ is the number of ways in which the set of occupations $\{\omega\}$ may be arranged. First, it is to be shown that when evaluated at fixed energy, this quantity S may be identified with the thermodynamic entropy of a system at equilibrium.

Proceeding, by applying the Sterling approximation [207] to the factorial function gives:

$$\begin{aligned} S &= k \left\{ \Omega \ln \Omega - \Omega - \sum_i \omega_i \ln \omega_i + \sum_i \omega_i \right\} \\ &= k \{ \Omega \ln \Omega - \Omega - \omega_1 \ln \omega_1 + \omega_1 - \omega_2 \ln \omega_2 + \omega_2 - \dots \} \end{aligned} \quad (11.2)$$

Using the above sum rule eliminates the middle two terms. Further rewriting

the above sums using the property of the natural log gives:

$$\begin{aligned}
&= k \left\{ \underbrace{\ln \Omega + \ln \Omega + \dots}_{\Omega \text{ times}} - \Omega + \underbrace{\omega_1 + \omega_2 + \dots}_{\Omega} - \omega_1 \ln \omega_1 - \dots \right\} \\
&= k \left\{ -\omega_1 \ln \frac{\omega_1}{\Omega} - \omega_2 \ln \frac{\omega_2}{\Omega} - \dots \right\}
\end{aligned} \tag{11.3}$$

In Equation 11.3 the $\ln \Omega$'s have been distributed amongst the occupation terms.

Identifying the probability $p_i = \omega_i/\Omega$ leads to the Gibbs entropy for the as yet unspecified ensemble; the Gibbs entropy is an entirely general definition that, for any equilibrium ensemble, specifies the relationship between the partition function and the associated characteristic thermodynamic function.

$$\frac{S}{\Omega} = -k \sum_i p_i \ln p_i \tag{11.4}$$

Now, specializing to a set of microcanonical subsystems, and invoking the equilibrium principle of equal *a priori* states, *i.e.* $p_i = 1/\Omega$, gives the well known result:

$$\frac{S}{\Omega} = -k \sum_i \frac{1}{\Omega} \ln \frac{1}{\Omega} = k \ln \Omega \tag{11.5}$$

It is also simple and useful to show that the Gibbs entropy, and thus the thermodynamic entropy, is maximized microcanonically [208] by the state-independent probabilities $p = p_i = 1/\Omega$. Proceeding, taking the derivative of Equation 11.4 and setting it to zero as

$$\frac{\partial}{\partial p_j} \left(-k \sum_i p_i \ln p_i \right) = 0 \tag{11.6}$$

gives $p_j = 1/e$, a constant value independent of the summation index. Thus, normalizing the probabilities as, $\sum_i^{\Omega} p_i = 1$ immediately yields $p_i = 1/\Omega$.

Thus, for an isolated system, the assumption of equal *a priori* states leads to a probability p_i that is independent of index, *i.e.* every subsystem has energy E by construction. Further, the characteristic maximum entropy in the microcanonical equilibrium state also follows. Then applying the Gibbs entropy expression leads to the identification of the thermodynamic entropy as the microcanonical characteristic function and gives its relationship to the N, V, E partition function, $\Omega(E)$, which can also be interpreted as the density of subsystems [203] (states) at that energy.

11.3 A Simplified Derivation of the Canonical Partition Function

Specializing the Gibbs construction from the previous section to include temperature – N, V, T are now defined for each subsystem. This can be thought of by placing the subsystems in contact with a large heat bath of temperature T . [203, 208]

Using the earlier result, the entropy for a collection of subsystems with a specified energy E_i is $S_i = k \ln \Omega(E_i)$.

Consider the ratio of the density at energies $E_{i+1} > E_i$:

$$\frac{\Omega(E_i)}{\Omega(E_{i+1})} = \frac{e^{\frac{1}{k}S_i}}{e^{\frac{1}{k}S_{i+1}}} = e^{-\frac{1}{k}(S_{i+1}-S_i)} \quad (11.7)$$

The fundamental equation of thermodynamics [209] is now invoked:

$$dE(S, V, N) = TdS - PdV + \mu dN \quad (11.8)$$

The canonical ensemble is given by a state with well defined thermodynamic

variables, N, V, T . So the energy function, $E(S, V, N)$ is Legendre transformed to a new thermodynamic function, the Helmholtz free energy, $A(T, V, N)$ via:

$$A = LT \{E\} = E - S \frac{\partial E}{\partial S} = E - ST \quad (11.9)$$

$$dA = dE - TdS - SdT \quad (11.10)$$

where the condition for canonical equilibrium is that $dA = 0$ and N, V, T are constant, giving:

$$0 = dE - TdS \quad (11.11)$$

$$dS = \frac{1}{T} dE \quad (11.12)$$

Integrating between two state points gives:

$$\int_i^{i+1} dS = \frac{1}{T} \int_i^{i+1} dE \quad (11.13)$$

$$S_{i+1} - S_i = \frac{1}{T} (E_{i+1} - E_i) \quad (11.14)$$

Note, the constraints of fixed temperature, particle number and volume have been explicitly enforced by using the fundamental equation, Equation 11.8 and dropping differential terms that are fixed canonically.

Substituting Equation 11.14 into Equation 11.7 gives:

$$\frac{\Omega(E_i)}{\Omega(E_{i+1})} = e^{-\frac{1}{k}(S_{i+1} - S_i)} = \frac{e^{-\beta E_{i+1}}}{e^{-\beta E_i}} \quad (11.15)$$

where $\beta = 1/kT$.

Equation 11.15 is clearly suggestive of canonical forms and allows the definition of the normalized probability, $p_i = \omega_i / \sum_i \omega_i = e^{-\beta E_i} / Q$, i.e the canonical probability expression, where the normalization factor, $Q = \sum_i e^{-\beta E_i}$, is identified as the canonical partition function.

We can now proceed to use the Gibbs entropy expression, Equation 11.4, substituting the canonical expression for p_i to obtain:

$$\begin{aligned} S &= -k \sum_i \frac{e^{-\beta E_i}}{Q} \ln \frac{e^{-\beta E_i}}{Q} \\ S &= k \frac{\ln Q}{Q} \sum_i e^{-\beta E_i} + k\beta \sum_i \frac{E_i e^{-\beta E_i}}{Q} \\ TS &= kT \ln Q + \langle E \rangle_{NVT} \end{aligned} \quad (11.16)$$

Above, $\langle E \rangle_{NVT}$ represents the canonical average energy that is identified with the thermodynamic energy, E . [191, 203] Thus, the relationship $A = E - TS = -kT \ln Q$ is obtained directly from the Gibbs entropy. Note, the Gibbs entropy is defined for any set of probabilities and, as was shown above, is simply a consequence of the combinatoric formula $S = k \ln W$ interpreted in the context of the Gibbs construction. Thus, an entropy can be associated even with nonequilibrium probabilities. However, in that case, the entropy does not play the role of being the constrained maximized quantity that it does at equilibrium and its utility, in such circumstances, is unclear.

Further, note that the role of temperature is introduced *via* the fundamental equation without further appeal to thermodynamic relationships. This emphasizes the role of temperature as the system is in contact with a heat bath – different energy ranges are now accessible with canonical probabilities. The ability of a diathermal system to exchange energy with its surroundings also clarifies how the concepts of

work and entropy make sense for an open system and provides their relationship to the temperature.

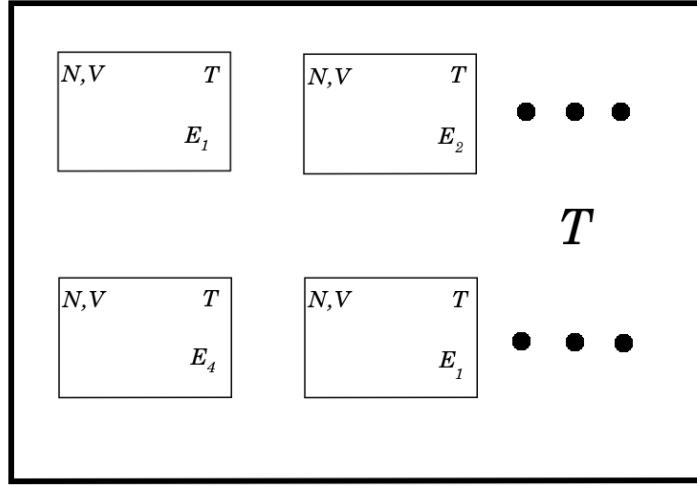


Figure 11.1: Gibbs construction for the canonical ensemble. The subsystems of identical N, V are in thermal equilibrium with a large bath at temperature T .

11.4 Grand Canonical Partition Function

Using the Gibbs construction from the previously derived canonical ensemble, the constraint that all subsystems possess identical N can now be relaxed. We now consider the ratio of subsystem, having chosen a particular system from the energy level E_i with N_j molecules:

$$\frac{\Omega(N_{j+1}, E_{i+1})}{\Omega(N_j, E_i)} = e^{-\frac{1}{k}(S_{i,j} - S_{i+1,j+1})} \quad (11.17)$$

As before, the fundamental equation of thermodynamics can be relied upon to relate the change in entropy to the other variables of our ensemble. In this case, the probabilities that will generate the macrostate corresponding to constant μ, V, T

are desired. After doing so, the entropy *via the Boltzmann law* is used to determine the microscopic states.

Legendre transforming the canonical thermodynamic equation to substitute μ for N :

$$\begin{aligned} G &= LT \{A\} = A - N \frac{\partial A}{\partial N} \\ &= A - \mu N \end{aligned} \quad (11.18)$$

$$\begin{aligned} dG &= dA - Nd\mu - \mu dN \\ &= dE - TdS - SdT - Nd\mu - \mu dN \end{aligned} \quad (11.19)$$

where at equilibrium $dG = 0$ and with constant μ, V, T :

$$0 = dE - TdS - \mu dN \quad (11.20)$$

$$dS(\mu, V, T) = \frac{1}{T} (dE - \mu dN) \quad (11.21)$$

Upon integrating, the difference equation for the entropy is found:

$$\int_{i,j}^{i+1,j+1} dS = \frac{1}{T} \left(\int_{i,j}^{i+1,j+1} dE - \mu \int_{i,j}^{i+1,j+1} dN \right) \quad (11.22)$$

$$S_{i+1,j+1} - S_{i,j} = \frac{1}{T} [E_{i+1} - E_i - \mu (N_{j+1} - N_j)] \quad (11.23)$$

The ratio of microstates then becomes:

$$\begin{aligned}
\frac{\Omega(N_j, E_i)}{\Omega(N_{j+1}, E_{i+1})} &= e^{-\frac{1}{k}(S_{i+1,j+1}-S_{i,j})} \\
&= e^{-\frac{1}{kT}(E_{i+1}-E_i-\mu N_{j+1}+\mu N_j)} \\
&= \frac{e^{-\beta E_{i+1}} e^{\beta \mu N_{j+1}}}{e^{-\beta E_i} e^{\beta \mu N_j}}
\end{aligned} \tag{11.24}$$

Again, having achieved an expression where the numerator and denominator are of the same functional form and are thus normalized with respect to both N_j and E_i to find the probability:

$$\begin{aligned}
Pr(N_j, E_i) &= \frac{e^{\beta \mu N_j} e^{-\beta E_i}}{\sum_{N_j=1}^{\infty} e^{\beta \mu N_j} \sum_i e^{-\beta E_i}} \\
&= \frac{e^{\beta \mu N_j} e^{-\beta E_i}}{\Xi}
\end{aligned} \tag{11.25}$$

where the normalization factor Ξ is the *grand canonical partition function*. It may be noted that in this case the constraint of constant N was merely relaxed, Legendre transformed to the corresponding macrostate, and the partition function then followed quite naturally and simply.

11.5 Isothermal-Isobaric Partition Function

Having successfully derived the grand canonical partition function by relaxing the constant N constraint, it can now be shown that the isothermal-isobaric ensemble is generated by starting with constant NVT and relaxing the condition of constant V . Consider the number of subsystems with both volume V_i and energy E_j :

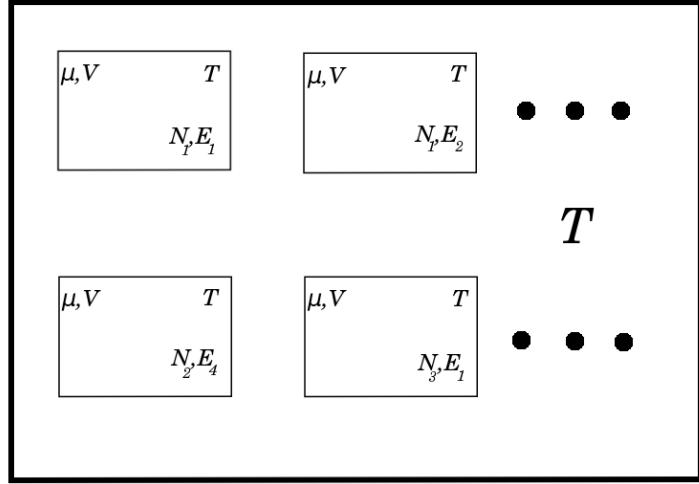


Figure 11.2: Gibbs construction for the grand canonical ensemble. μ has been Legendre transformed to replace N as the macroscopic constant, and so the set of subsystems includes those of differing N values.

$$\frac{\Omega(V_j, E_i)}{\Omega(V_{j+1}, E_{i+1})} = e^{-\frac{1}{k}(S_{i+1,j+1} - S_{i,j})} \quad (11.26)$$

Legendre transforming our desired variables to a new characteristic function J , and then applying the condition of equilibrium $dJ = 0$ and our constant differential terms:

$$J = A - V \frac{\partial A}{\partial V} = A + PV \quad (11.27)$$

$$dJ = dA + PdV + VdP = 0$$

$$= dE - TdS - SdT + PdV + VdP \quad (11.28)$$

$$\rightarrow dS = \frac{1}{T} (dE + PdV) \quad (11.29)$$

and so the ratio of observable subsystems becomes:

$$\begin{aligned}
\frac{\Omega(V_j, E_i)}{\Omega(V_{j+1}, E_{i+1})} &= e^{-\frac{1}{k}(S_{i+1,j+1} - S_{i,j})} \\
&= e^{-\frac{1}{kT}[E_{i+1} - E_i + P(V_{i+1} - V_i)]} \\
&= \frac{e^{-\beta E_{i+1}} e^{-\beta P V_{j+1}}}{e^{-\beta E_i} e^{-\beta P V_j}}
\end{aligned} \tag{11.30}$$

Upon normalizing, the probabilities are:

$$Pr(V_j, E_i) = \frac{e^{-\beta E_i} e^{-\beta P V_j}}{\Delta} \tag{11.31}$$

where the normalization factor,

$$\Delta = \sum_j e^{-\beta P V_j} \sum_i e^{-\beta E_i} \tag{11.32}$$

is the *isothermal-isobaric partition function*.

11.6 Connection with Generalized Ensemble Theory

Based on the previous sections, it may be noted that if a partition function for *any* macrostate in thermal equilibrium is to be derived, one shall *always* arrive at an expression that involves an exponential function; this follows as a consequence of the Boltzmann Law. Furthermore, the partition function being the normalization factor to express this as a probability means that one will always have a discrete sum. Therefore, our partition functions will always be some variation on a theme amounting to a “sum of exponentials”.

In the continuum limit, it can be shown that the canonical partition function

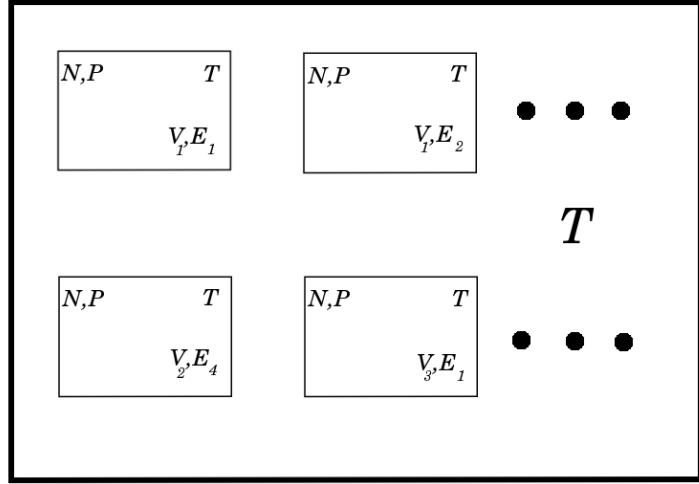


Figure 11.3: Gibbs construction for the isothermal-isobaric ensemble. P has been Legendre transformed to replace V as the macroscopic constant, and so the set of subsystems includes those of differing V values.

Q (written as a sum over energy levels) transforms as:

$$Q(N, V, \beta) = \sum_i e^{-\beta E_i} \Omega(E_i)$$

$$\rightarrow \int_0^\infty dE e^{-\beta E} \Omega(N, V, E) \quad (11.33)$$

or, in other words, the canonical partition function is the Laplace transform of the microcanonical partition function Ω .

How does it work the other way? Let's apply the inverse Laplace transform to Q :

$$\begin{aligned}
\Omega(N, V, E) &= \frac{1}{2\pi i} \oint d\beta e^{\beta E} Q(N, V, \beta) \\
&= \int_{-\infty}^{\infty} d\Omega \frac{1}{2\pi i} \int_{\gamma-i\infty}^{\gamma+i\infty} d\beta e^{\beta(E-H)}
\end{aligned} \tag{11.34}$$

where the phase space differential form $d\Omega = (h^{3N} N!)^{-1} dx_1 \dots dx_{3N} dp_1 \dots dp_{3N}$. Now, $\beta = \sigma + i\tau$ and because no singularity is present in the right-half of the complex plane, the contour may be taken vertically through $\gamma = 0$. Since $\text{Re}(\beta) = 0$ along the integration, the substitution $\beta = i\tau$ can be made:

$$\begin{aligned}
\Omega(N, V, E) &= \int_{-\infty}^{\infty} d\Omega \frac{1}{2\pi} \int_{-\infty}^{\infty} d\tau e^{i\tau(E-H)} \\
&= \int_{-\infty}^{\infty} d\Omega \delta(E - H)
\end{aligned} \tag{11.35}$$

where indeed Equation 11.35 can be identified as the microcanonical partition function. Thus, any constant energy shell ensemble may be Laplace transformed to an ensemble of a new intensive variable. As the partition functions are related to one another through the Laplace transform, this is isomorphic to the thermodynamic potentials (to each of which may be associated a particular partition function) being related through the Legendre transform. [204]

The quantum harmonic oscillator is an illustrative example of how the canonical partition function may be transformed to the microcanonical case:

$$\begin{aligned}
\Omega_{HO} &= \frac{1}{2\pi i} \oint d\beta e^{\beta E} Q_{HO} \\
&= \frac{1}{2\pi} \int_{-\infty}^{\infty} d\tau e^{i\tau E} \frac{e^{-\frac{1}{2}i\tau\hbar\omega}}{1 - e^{-i\tau\hbar\omega}} \\
&= \frac{1}{2\pi} \int_{-\infty}^{\infty} d\tau e^{i\tau(E - \frac{1}{2}\hbar\omega)} \sum_n e^{-i\tau\hbar\omega n} \\
&= \sum_n \delta \left[E - \hbar\omega \left(n + \frac{1}{2} \right) \right]
\end{aligned} \tag{11.36}$$

The reason that the aforementioned “recipe” for generating the partition function (as outlined in Sections 11.3, 11.4 and 11.5) in an arbitrary ensemble works is due to the thermodynamic relations and the Boltzmann law. The underlying mathematical structure that allows this has also been previously formulated as generalized ensemble theory. [199–201, 210, 211]

11.7 Conclusions

We’ve presented an approach toward deriving the partition function that is an alternative to the standard derivation provided by most textbooks. This approach emphasizes the central role that the Boltzmann entropy plays in connecting the molecular states of the system to the observable thermodynamics. It has been shown how this can be a useful technique to easily derive expressions for the partition function, even for the less common ensembles. Finally, the similarities between the derivation method demonstrated and the relations known from generalized ensemble theory have been noted. It is our hope that the formulaic approach presented here will be of utility

in both research and pedagogy.

Chapter 12

Volume Determination of Globular Proteins by Molecular Dynamics

12.1 Introduction

Experimental techniques for determining the partial specific volume and partial specific adiabatic compressibility of proteins in solution have provided key insight into structural and catalytic events. The application of these methods has resulted in a broad base of knowledge about solvation effects, ligand binding and dissociation, the influence of protein domains on catalytic events, and protein folding pathways.

Theoretical methods offer the potential to link specific events to the thermodynamic observables that experimentalists measure in the course of their research. In particular, molecular dynamics (MD) can provide a powerful approach to correlating the molecular trajectories of proteins that may give rise to an experimentally measured molecular volume or change thereof.

The state-of-the-art method to date for determining the apparent volume of a protein is the method of “accessible surface area, [212] a method that employs a spherically-accessible surface integration of the proteins crystal structure. This method is attractive in that it is not computationally demanding (and is thus ac-

cessible to the typical computational equipment of an experimental lab), however it calculates volumes that can be significantly different from those measured in solution.

Prior research on the volume of small molecules has been performed by our group [213, 214] and the methodology presented here is an application of those techniques to protein systems. This work demonstrates the validity of the MD approach toward determining the apparent molecular volume of globular proteins.

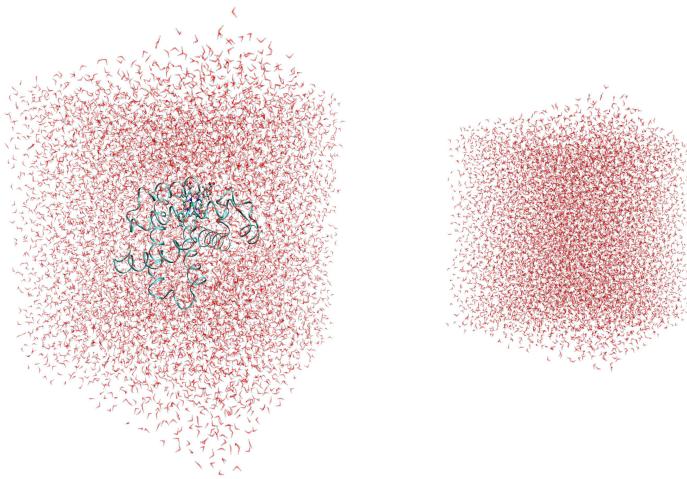


Figure 12.1: A periodic cell containing horse-heart myoglobin solvated with 10,796 waters, in comparison with a system containing only the bulk TIP3P water. The difference in the volume between each cell under NPT dynamics results in the apparent molecular volume of the myoglobin.

12.2 Methods

The apparent molecular volume of a protein is calculated as:

$$V_p = V_{p+w} - V_w \quad (12.1)$$

The NAMD molecular dynamics package uses a Langevin-Hoover hybrid method where a piston is coupled to the equations of motion for a particle in the isothermal-isobaric ensemble.

A simulated annealing algorithm was implemented to perform a global energy minimization. The system is brought to a higher energy state and allowed to randomly walk through the phase space for a hold time before being brought back down to the initial energy state.

Post-simulation analysis consisted of calculating the correlation time of the volume signal using a block-averaging method in order to calculate an unbiased error in the volume, as well as structural analysis of the equilibrium structures (RMSD, Ramachandran plots) versus their crystal structures.

12.3 Results and Discussion

The apparent molecular volumes of wild-type myoglobin and aspartate aminotransferase (AspAT) were calculated and compared with experimental values.

12.3.1 Horse-heart myoglobin

Preparatory simulation stages (NAMD [215]) consisted of constructing the solvated protein system and local energy minimization, followed by heating and equilibration steps. Initial myoglobin structural coordinates were obtained from the RCSB protein data bank (crystal structure entry 1DWR4), solvated with 10,796 TIP3P water molecules and then minimized by the method of conjugate gradients. The system was then heated to 300 K over a period of 1.2 ns, followed by equilibration in the NPT

ensemble for 0.2 ns.

A production NPT (300 K/1 atm/2.0 fs timestep) run of 10.0 ns resulted in an average volume for the protein-water system of 341,875 Å³; simulating the bulk water alone over a trajectory of equal time duration yielded an average system volume of 319,775 Å³. The difference of these values gives an apparent molecular volume of wild-type myoglobin corresponding to 22,100 Å³ 0.747 cm³ / g. The correlation times of both volume signals were determined by block-averaging and the signals were then uncorrelated to provide an unbiased volume error of ± 0.001 cm³ / g. This computationally determined apparent volume of 0.747 ± 0.001 cm³ / g agrees precisely with experimentally reported sound velocity measurements [216] and is within the experimental error of that study. The equilibrium protein structure in solution was aligned and compared with the crystal structure of myoglobin and while the overall RMSD was minimal, a local region of amino acids near GLY80 was found to be displaced by 6.0 .

12.3.2 Aspartate aminotransferase

The halozyme of E. coli aspartate aminotransferase (RCSB entry 1ASM), [217, 218] a large dimer consisting of identical 404 amino acid subunits complete with LYS258-bound pyridoxal-5-phosphate cofactors, was simulated in the NPT ensemble with 58,361 water molecules. An average apparent volume of 0.733 cm³ / g was calculated at the end of a 0.5 ns run, a result that is in good agreement with the experimentally measured value of 0.731 cm³ / g. Work is in progress to obtain a longer timescale trajectory and calculate the associated volumetric error.

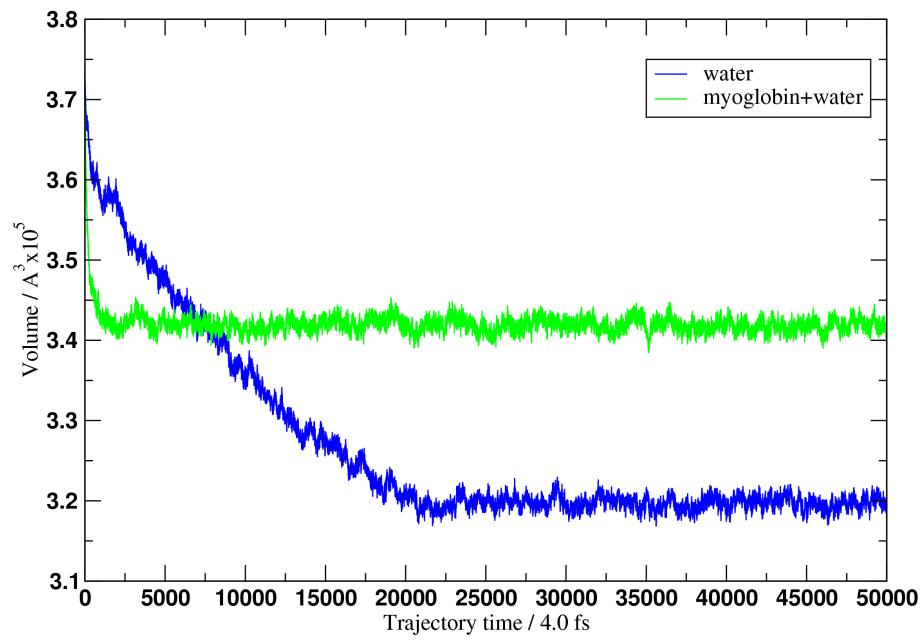


Figure 12.2: Volume signals for the solvated myoglobin system vs. the bulk water. After initial equilibration the volumes fluctuate about their average values for the 10 ns trajectory the difference of which gives an apparent molecular volume of 0.747 cm^3 / g .

A single-point mutation (VAL39 to GLY) has been experimentally shown to induce a large conformational change in the dimer [219] and an associated change in the apparent volume by $-0.035 \text{ cm}^3 / \text{g}$ one of the largest molecular volume changes observed due to a single residue mutation.

Since the crystallographic structure of the V39G mutant has not yet been elucidated, manual alteration of the VAL39 sidechain of the 1ASM crystal structure was performed to give the V39G initial configuration. A simulated annealing algorithm was developed and performed on the mutant to help facilitate the adoption of its new equilibrium conformation prior to performing production NPT runs.

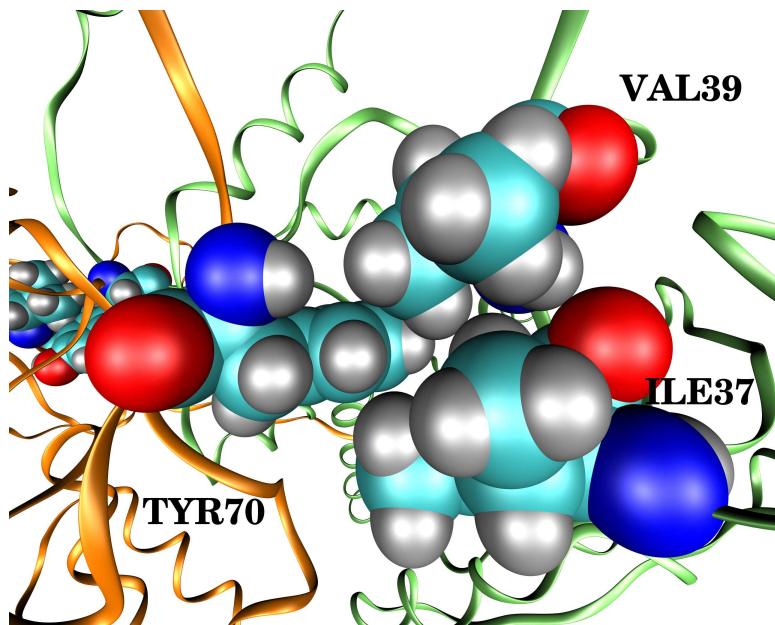


Figure 12.3: Van der Waals representation showing the steric effects of the valine (V39), tyrosine (Y70 on the second subunit of the dimer) and isoleucine (I37) residues of interest for the V39G mutant active site. Substitution of the valine for a glycine reduces the steric interactions leading to a large scale conformational change as well as a drastic change in catalytic activity. The same triad can be seen in the distance to the left on the opposite side of the dimer.

12.4 Conclusions

The application of molecular dynamics for studying the volume of globular proteins can accurately model experimental data. The methods presented can offer insight into structural protein studies since conformational changes can be examined and correlated with experimentally observed volume changes in solution. The volumetric contribution of various regions of a protein (including the more difficult case of a single-point mutation) can be elucidated through use of this practical and consistent methodology.

Chapter 13

Massively Parallel Monte Carlo (MPMC)

A statistical mechanics simulation code named MPMC (**M**assively **P**arallel **M**onte **C**arlo) has been developed to perform simulations of the fluid/material interface. The vast majority of calculations that this codebase provides can be trivially parallelized, and so it's massively parallel nature is reflected in the name. The source code of this simulation package is listed in Appendix L.

The canonical, grand canonical and microcanonical ensemble Monte Carlo schemes can be performed through keyword specification in the input file. Cavity biasing schemes (both grid-based and distance based) have been found useful in GCMC simulations of dense systems. The equation of state for hydrogen (for mapping the applied pressure to the chemical potential used in GCMC) has been included for a wide range of conditions.

A variety of potential energy functions have been implemented, including Lennard-Jones and Silvera-Goldman. Long-range electrostatics are calculating using the Ewald summation method. Morse and harmonic bond potentials are included for specific vibrational surface calculations. Many-body polarization of the point in-

duced dipole type can be enabled and the polarization energy calculated based upon the induced dipole contribution to the electrostatic field. A variety of advanced iterative methods for solving the self-consistent field equations, as outlined in Chapter 8, are employed to accelerate convergence of the induced field.

Quantum fluctuations may be included in simulation through both finite Trotter Path Integral Monte Carlo. In addition, all of the above potentials have Feynman-Hibbs effective potential analogues to quartic order that may be activated. The more advanced iterative scheme of Feynman and Kleinert has also been implemented.

Rotational excited states are calculated by direct diagonalization of the hamiltonian for a diatomic rotor subject to an external field. Integration is performed over a Gauss-Lengdre quadrature grid, and then the hamiltonian (in a spherical harmonic basis) is diagonalized using the LAPACK linear algebra package. The inversion symmetry operator is acted on the resulting eigenvectors in order to determine whether the quantum states are symmetric or antisymmetric. Having determined the symmetry of the rotational wavefunctions, Boltzmann factors for distinct nuclear spin states may be included in the Monte Carlo Metropolis function.

References

- [1] Jeffrey R. Long, A. Paul Alivisatos, Marvin L. Cohen, Jean M. J. Fréchet, Martin Head-Gordon, Steven G. Louie, Alex Zettl, Thomas J. Richardson, Samuel S. Mao, Carole Read and Jim Alkire, A Synergistic Approach to the Development of New Classes of Hydrogen Storage Materials, FY 2005 Progress Report, DOE Hydrogen Program, 2005.
- [2] J. L. Belof, A. C. Stern, and B. Space, *J. Phys. Chem. C* **113**, 9316 (2009).
- [3] S. K. Bhatia and A. L. Myers, *Langmuir* **22**, 1688 (2006).
- [4] J. L. Belof, A. C. Stern, M. Eddaoudi, and B. Space, *J. Am. Chem. Soc.* **129**, 15202 (2007).
- [5] J. L. Belof, A. C. Stern, and B. Space, *J. Chem. Theory Comput.* **4**, 1332 (2008).
- [6] R. Feynman and A. Hibbs, *Quantum Mechanics and Path Integrals* (McGraw-Hill, New York, 1965), "Chapt. 10, Sect. 3, pg. 281".
- [7] M. Creutz, *Phys. Rev. Lett.* **50**, 1411 (1983).
- [8] J. R. Ray, *Phys. Rev. A* **44**, 4061 (1991).

- [9] R. C. Weast, *CRC Handbook of Chemistry and Physics* (CRC Press, Inc., West Palm Beach, 1994).
- [10] J. L. Rowsell and O. M. Yaghi, *Angew. Chem. Int. Ed.* **44**, 4670 (2005).
- [11] A. M. Seayad and D. M. Antonelli, *Adv. Mater.* **16**, 765 (2004).
- [12] V. Berube, G. Radtke, M. Dresselhaus, and G. Chen, *Int. J. Energy Research* **6-7**, 637 (2007).
- [13] E. David, *J. Mater. Process. Technol.* **162**, 169 (2005).
- [14] E. Fakioglu, Y. Yurum, and T. Veziroglu, *Int. J. Hydrogen Energy* **13**, 1371 (2004).
- [15] A. Seayad and D. Antonelli, *Adv. Mater.* **9-10**, 765 (2004).
- [16] G. Garberoglio, A. I. Skouidas, and J. K. Johnson, *J. Phys. Chem. B* **109**, 13094 (2005).
- [17] J. Eckert, *Spectrochim. Acta* **48A**, 363 (1992).
- [18] C.-R. Anderson, D. F. Coker, J. Eckert, and A. L. R. Bug, *J. Chem. Phys.* **111**, 7599 (1999).
- [19] Y. Liu *et al.*, *Angew. Chem. Int. Ed.* **46**, 3278 (2007).
- [20] N. L. Rosi *et al.*, *Science* **300**, 1127 (2003).
- [21] B. Moulton and M. J. Zaworotko, *Chem. Rev.* **101**, 1629 (2001).

- [22] M. O'Keeffe and O. M. Yaghi, see the web site: <http://rcsr.anu.edu.au/>, 2003.
- [23] O. Delgado-Friedrichs, M. O'Keeffe, and O. M. Yaghi, *Acta crystallographica A* **62**, 350 (2006).
- [24] R. Haber, B. Lucas, and N. Collins, *Proc. IEEE Vis.* **1**, 298 (1991).
- [25] B. Lucas *et al.*, *Proc. IEEE Vis.* **92**, 107 (1992).
- [26] G. Abram and L. Treinish, *Proc. IEEE Comp. Soc.* **95**, 263 (1995).
- [27] J. Applequist, J. R. Carl, and K.-K. Fung, *J. Am. Chem. Soc.* **94**, 2952 (1972).
- [28] B. Thole, *Chem. Phys.* **59**, 341 (1981).
- [29] P. T. van Duijen and M. Swart, *J. Phys. Chem. A* **102**, 2399 (1998).
- [30] A. Perry, C. Neipert, P. Moore, and B. Space, *Chem. Rev.* **106**, 1234 (2006).
- [31] F. Mulder, T. Dingemans, M. Wagemaker, and G. Kearley, *Chem. Phys.* **317**, 113 (2005).
- [32] Q. Wang and J. K. Johnson, *J. Phys. Chem. B* **103**, 4809 (1999).
- [33] G. E. Froudakis, *J. Phys. Condens. Matter* **14**, R453 (2002).
- [34] P. Demontis and G. B. Suffritti, *Chem. Rev.* **97**, 2845 (1997).
- [35] T. Sagara, J. Klassen, and E. Ganz, *J. Chem. Phys.* **121**, 12543 (2004).
- [36] J. A. MacKinnon, J. Eckert, D. F. Coker, and A. L. Bug, *J. Chem. Phys.* **114**, 10137 (2001).

- [37] A. Larin and E. Cohen De Lara, J. Chem. Phys. **101**, 8130 (1994).
- [38] D. F. Coker, D. Thirumalai, and B. J. Berne, J. Chem. Phys. **86**, 5689 (1987).
- [39] D. F. Coker and B. J. Berne, J. Chem. Phys. **89**, 2128 (1988).
- [40] F. Darkrim and D. Levesque, J. Chem. Phys. **109**, 4981 (1998).
- [41] K. Shirono, A. Endo, and H. Daiguji, J. Phys. Chem. B **109**, 3446 (2005).
- [42] Q. Fu *et al.*, Phys. Rev. B **65**, 0753181 (2002).
- [43] Q. Sun, Q. Wang, P. Jena, and Y. Kawazoe, J. Am. Chem. Soc. **127**, 14582 (2005).
- [44] T. Kawakami *et al.*, Polyhedron **20**, 1197 (2001).
- [45] T. Mueller and G. Cedar, J. Phys. Chem. B **109**, 17974 (2005).
- [46] Q. Wang and J. K. Johnson, J. Chem. Phys. **110**, 577 (1999).
- [47] V. V. Simonyan, P. Diep, and J. K. Johnson, J. Chem. Phys. **111**, 9778 (1999).
- [48] Q. Wang and J. K. Johnson, J. Phys. Chem. B **103**, 277 (1999).
- [49] S. M. Lee and Y. H. Lee, App. Phys. Lett. **76**, 2877 (2000).
- [50] F. L. Darkrim, P. Makbrunot, and G. P. Tartaglia, Int. J. Hydrogen Energy **27**, 193 (2002).
- [51] L. Schlapbach and A. Zuttel, Nature **414**, 353 (2001).
- [52] S.-H. Jhi, Micro. Meso. Materials **89**, 138 (2006).

- [53] D.-Y. Hwang and A. M. Mebel, Chem. Phys. Lett. **321**, 95 (2000).
- [54] Q. Yang and C. Zhong, Phys. Chem. B **110**, 655 (2006).
- [55] R. A. Zidan and R. E. Rocheleau, J. Mater. Res. **14**, 286 (1999).
- [56] S. Bordiga *et al.*, J. Phys. Chem. B **109**, 18237 (2005).
- [57] T. Duren, L. Sarkisov, O. M. Yaghi, and R. Q. Snurr, Langmuir **20**, 2683 (2004).
- [58] M. P. Allen and D. J. Tildesley, *Computer Simulation of Liquids* (Clarendon Press, Oxford, 1989).
- [59] D. Frenkel and B. Smit, *Understanding Molecular Simulation: From Algorithms to Applications* (Academic Press, New York, 2002).
- [60] A. Rappe *et al.*, J. Am. Chem. Soc. **114**, 10024 (1992).
- [61] M.W. Schmidt, K.K. Baldridge, J.A. Boatz, S.T. Elbert, M.S. Gordon, J.J. Jensen, S. Koseki, N. Matsunaga, K.A. Nguyen, S. Su, T.L. Windus, M. Dupuis and J.A. Montgomery, J. Comput. Chem. **14**, 1347 (1993).
- [62] U. Singh and P. Kollman, J.Comput.Chem. **5**, 129 (1984).
- [63] W. J. Stevens, H. Basch, and M. Krauss, J. Chem. Phys. **81**, 6026 (1984).
- [64] L. A. LaJohn *et al.*, J. Chem. Phys. **87**, 2812 (1987).
- [65] P. J. Hay and W. R. Wadt, J. Chem. Phys. **82**, 270 (1985).
- [66] W. D. Cornell *et al.*, J. Am. Chem. Soc. **117**, 5179 (1995).

- [67] T. Tsuchiya, M. Abe, T. Nakajima, and K. Hirao, J. Chem. Phys. **115**, 4463 (2001).
- [68] R. DeVane *et al.*, J. Chem. Phys. **121**, 3688 (2004).
- [69] J. Applequist and C. O. Quicksall, J. Chem. Phys. **66**, 3455 (1977).
- [70] P. Ren and J. W. Ponder, J. Phys. Chem. B **107**, 5933 (2003).
- [71] S. S. X. Christian J. Burnham, Jichen Li and M. Leslie, J. Chem. Phys. **110**, 4566 (1999).
- [72] M. Souaille and J. C. Smith, Mol. Phys. **87**, 1333 (1996).
- [73] D. N. Bernardo, Y. Ding, K. Krogh-Jespersen, and R. M. Levy, J. Phys. Chem. **98**, 4180 (1994).
- [74] P. Ren and J. W. Ponder, J. Comput. Chem. **23**, 1497 (2002).
- [75] K. A. Bode and J. Applequist, J. Phys. Chem. **100**, 17825 (1996).
- [76] M. Griebel *et al.*, *Discretization Methods and Iterative Solvers Based on Domain Decomposition, Lecture Notes in Computational Science and Engineering* (Springer, Berlin, 2001).
- [77] T. Dinh and G. A. Huber, J. Math. Modelling and Algorithms **4**, 111 (2005).
- [78] T. H. Dunning, Jr., J. Chem. Phys. **90**, 1007 (1989).
- [79] N. C. C. Comparison and N. S. R. D. N. . Benchmark Database, <http://srdata.nist.gov/cccdb>, Release 12, Aug 2005.

- [80] T. Guella *et al.*, Phys. Rev. A **29**, 2977 (1984).
- [81] N. Metropolis *et al.*, Phys. Lett. B **21**, 1087 (1953).
- [82] S. Duane, A. Kennedy, B. J. Pendleton, and D. Roweth, Phys. Lett. B **195**, 216 (1987).
- [83] B. Mehlig, D. Heermann, and B. Forrest, Phys. Rev. B **45**, 679 (1992).
- [84] P. Moore, C. F. Lopez, and M. L. Klein, Biophys. J. **81**, 24842494 (2001).
- [85] P. Moore, Q. Zhong, T. Husslein, and M. L. Klein, FEBS Lett. **431**, 143148 (1998).
- [86] Q. Zhong, P. B. Moore, and M. L. Klein, FEBS Lett. **427**, 267274 (1998).
- [87] M. Tarek, K. Tu, M. L. Klein, and D. J. Tobias, Biophys. J. **77**, 964972 (1999).
- [88] S. Wolfram, Rev. Mod. Phys. **55**, 601 (1983).
- [89] S. Wolfram, *A New Kind of Science* (Wolfram Media, Champaign, IL, 2002).
- [90] A. M. Horowitz, Phys. Lett. B **268**, 247 (1991).
- [91] E. Akhmatskaya and S. Reich, Lecture Notes in Computational Science and Engineering **49**, 141 (2005).
- [92] M. Dinca *et al.*, J. Am. Chem. Soc. **128**, 16876 (2006).
- [93] A. Meyers, American Institute of Chemical Engineers **48**, 145 (2002).
- [94] A. L. Meyers and F. Siperstein, Colloids and Surfaces A **187-188**, 73 (2001).

- [95] B. Widom, J. Chem. Phys. **39**, 2808 (1963).
- [96] B. Widom, J. Phys. Chem. **86**, 869 (1982).
- [97] V. Bhethanabotla and W. Steele, J. Phys. Chem. **92**, 3285 (1988).
- [98] J. Van Kranendonk, *Solid Hydrogen: Theory of the Properties of Solid H₂, HD, and D₂*, 2nd ed. (Plenum Press, New York, 1983), p. 45.
- [99] G. Garberoglio, A. I. Skouidas, and J. K. Johnson, J. Phys. Chem. B **109**, 13094 (2005).
- [100] J. Belof, A. Stern, M. Eddaoudi, and B. Space, J. Am. Chem. Soc. **129**, 15202 (2007).
- [101] J. Schaefer and R. Watts, Mol. Phys. **47**, 933 (1982).
- [102] H. Knaap and J. Beenakker, Physica **27**, 523 (1961).
- [103] V. Buch, J. Chem. Phys. **100**, 7610 (1994).
- [104] I. F. Silvera and V. V. Goldman, J. Chem. Phys. **69**, 4209 (1978).
- [105] P. Diep and J. K. Johnson, J. Chem. Phys. **112**, 4465 (2000).
- [106] P. Diep and J. K. Johnson, J. Chem. Phys. **113**, 3480 (2000).
- [107] G. Gallup, Mol. Phys. **33**, 943 (1977).
- [108] R. Etters, R. Danilowicz, and W. England, Phys. Rev. A **12**, 2199 (1975).
- [109] P. Wind and I. Røeggen, Chem. Phys. **211**, 179 (1996).

- [110] F. Mulder, A. van der Avoird, and P. Wormer, Mol. Phys. **37**, 157 (1979).
- [111] F. Darkrim and D. Levesque, J. Chem. Phys. **109**, 4981 (1998).
- [112] Z. Zhang and Z. Duan, J. Chem. Phys. **122**, 214507 (2005).
- [113] M. K. Kostov *et al.*, Chem. Phys. Lett. **332**, 26 (2000).
- [114] B. Axilrod and E. Teller, J. Chem. Phys. **11**, 299 (1943).
- [115] Y. Liu *et al.*, Angew. Chem. Int. Ed. **46**, 1433 (2007).
- [116] D. Hartree, Proc. Camb. Phil. Soc. **24**, 89 (1928).
- [117] V. Fock, Zeitschrift für Physik A **62**, 795 (1930).
- [118] J. Slater, Phys. Rev. **42**, 33 (1932).
- [119] J. Slater, Phys. Rev. **81**, 385 (1950).
- [120] C. Roothaan, Rev. Mod. Phys. **23**, 69 (1951).
- [121] R. D. J. III, NIST Computational Chemistry Comparison and Benchmark Database: NIST Standard Reference Database Number 101, <http://srdata.nist.gov/cccldb>, Release 14, Sept 2006.
- [122] G. Herzberg, *Molecular Spectra and Molecular Structure, Second Edition* (Krieger Publishing Co., Malabar, FL, 1989), "Vol. I, p. 532".
- [123] A. V. Nemukhin, B. L. Grigorenko, and A. A. Granovsky, Moscow University Chemistry Bulletin **45**, 75 (2004).

- [124] C. Møller and M. S. Plesset, Phys. Rev. **46**, 618 (1934).
- [125] J. Binkley and J. Pople, Int. J. Quant. Chem. **9**, 229 (1975).
- [126] R. Krishnan and J. Pople, Int. J. Quant. Chem. **14**, 91 (1978).
- [127] R. Krishnan and J. Pople, J. Chem. Phys. **72**, 4244 (1980).
- [128] J. T.H. Dunning, J. Chem. Phys. **90**, 1007 (1989).
- [129] T. Helgaker, W. Klopper, H. Koch, and J. Noga, J. Chem. Phys. **106**, 9639 (1997).
- [130] S. Boys and F. Bernardi, Mol. Phys. **19**, 553 (1970).
- [131] J. Poll and L. Wolniewicz, J. Chem. Phys. **68**, 3053 (1978).
- [132] S. Karna and M. Dupuis, J. Comp. Chem. **12**, 487 (1991).
- [133] D. R. Lide, *Handbook of Chemistry and Physics* (CRC Press, Inc., Boca Raton, FL, 2003).
- [134] W. Kolos and L. Wolniewicz, J. Chem. Phys. **46**, 1426 (1967).
- [135] K. A. Bode and J. Applequist, J. Phys. Chem. **100**, 17820 (1996).
- [136] B. Thole, Chem. Phys. **59**, 341 (1981).
- [137] P. T. van Duijnen and M. Swart, J. Phys. Chem. A **102**, 2399 (1998).
- [138] M. Brack, R. Bhaduri, and R. K. Bhaduri, *Semiclassical Physics* (Westview Press, Boulder, CO, 2003), "Chapt. 4, Sect. 2, pg. 148".

- [139] N. Metropolis *et al.*, Phys. Lett. B **21**, 1087 (1953).
- [140] D. Frenkel and B. Smit, *Understanding Molecular Simulation: From Algorithms to Applications* (Academic Press, New York, 2002), "Chapt. 5, Sect. 6.1, p.129".
- [141] H. Shaw and D. Wones, Amer. J. Sci. **262**, 918 (1964).
- [142] L. Zhou and Y. Zhou, Int. J. Hydrogen Energy **26**, 597 (2001).
- [143] E. Lemmon, M. McLinden, and D. Friend, in *NIST Chemistry WebBook*, edited by P. Linstrom and W. Mallard (National Institute of Standards and Technology, Gaithersburg, MD, 2005), Vol. 69, <http://webbook.nist.gov>.
- [144] B. Younglove, J. Phys. Chem. Ref. Data **11**, 1 (1982).
- [145] M. H. Alkordi *et al.*, J. Am. Chem. Soc. **130**, 12639 (2008).
- [146] J. Eckert and G. J. Kubas, J. Phys. Chem. **97**, 2378 (1993).
- [147] E. Hao, G. Schatz, R. Johnson, and J. Hupp, J. Chem. Phys. **117**, 5961 (2006).
- [148] J. A. Greathouse and M. D. Allendorf, J. Phys. Chem. C **112**, 5795 (2008).
- [149] T. Boublík, Fluid Phase Equil. **240**, 96 (2005).
- [150] B. J. Alder, D. A. Young, and M. A. Mark, J. Chem. Phys. **56**, 3013 (1972).
- [151] D. Nicholson and N. G. Parsonage, *Computer Simulation and the Statistical Mechanics of Adsorption* (Academic Press, London, 1982), "Chapt. 3, Sect. 4, pg. 97".

- [152] S. S. Kaye and J. R. Long, unpublished results.
- [153] S. S. Kaye, A. Dailly, O. M. Yaghi, and J. R. Long, *J. Am. Chem. Soc. Comm.* **129**, 14176 (2007).
- [154] W. Zhou, H. Wu, M. R. Hartman, and T. Yildirim, *J. Phys. Chem. C* **111**, 16131 (2007).
- [155] B. Schmitz *et al.*, *Chem. Phys. Chem.* **9**, 2181 (2008).
- [156] G. K. Platt, Space Vehicle Low Gravity Fluid Mechanics Problems and the Feasibility of their Experimental Investigation, NASA Technical Memorandum, TM X-53589, 1967.
- [157] A. A. Sheuinina, N. G. Bereznyak, V. P. Vorob'eva, and M. A. Khadzhmuradov, *Low Temp. Phys.* **19**, 356 (1993).
- [158] E. S. Yakub, *Int. J. Thermophys.* **22**, 505 (2001).
- [159] R. D. McCarty, Hydrogen Technological Survey - Thermophysical Properties, NASA SP-3089, p. 518-519, 1975.
- [160] Y. Liu, V. C. Kravtsov, R. W. Larsen, and M. Eddaoudi, *Chem. Commun.* **14**, 1488 (2006).
- [161] S. Bradamante, A. Facchetti, and G. Pagani, *J. Phys. Org. Chem.* **10**, 514 (1997).
- [162] I. Isaksson and J. Sandstrom, *Acta Chim. Scand.* **27**, 1183 (1973).

- [163] D. Chemla and J. Zyss, *Non-Linear Optical Properties of Organic Molecules and Crystals Vol. 1 and 2* (Academic Press, New York, 1986).
- [164] L. Cheng *et al.*, *J. Phys. Chem.* **95**, 10631 (1991).
- [165] J. Oudar, *J. Chem. Phys.* **67**, 446 (1977).
- [166] P. Prasad and D. Williams, *Introduction to nonlinear optical effects in molecules and polymers*. (John Wiley, New York, 1994).
- [167] J. Quenneville and T. Martinez, *J. Phys. Chem.* **107**, 829 (2003).
- [168] J. Zarembowitch *et al.*, *Mol. Cryst. Liq. Cryst.* **234**, 247 (1993).
- [169] S. Decurtis *et al.*, *Chem. Phys. Lett.* **105**, 1 (1984).
- [170] M. Boillot *et al.*, *J. Inorg. Chem.* **35**, 3975 (1996).
- [171] A. Sour, M. Boillet, E. Riviere, and P. Lesot, *Eur. J. Inorg. Chem.* 2117 (1999).
- [172] M. Boillot, S. Chantraine, J. Z. J. Lallemand, and J. Prunet, *New J. Chem.* 179 (1999).
- [173] C. Faulmann *et al.*, *Eur. J. Inorg. Chem.* 3261 (2005).
- [174] M. W. Schmidt *et al.*, *J. Comput. Chem.* **14**, 1347 (1993).
- [175] C. R. Crecca and A. E. Roitberg, *J. Phys. Chem.* **110**, 8188 (2006).
- [176] A. Cembran *et al.*, *J. Am. Chem. Soc.* **126**, 3234 (2004).
- [177] M. L. Tiago, S. Ismail-Beigi, and S. G. Louie, *J. Chem. Phys.* **122**, 94311 (2005).

- [178] A. D. Becke, J. Chem. Phys. **98**, 5648 (1993).
- [179] P. Stephens, F. Devlin, C. Chabalowski, and M. Frisch, J. Phys. Chem. **98**, 11623 (1994).
- [180] R. Hertwig and W. Koch, Chem. Phys. Lett. **268**, 345 (1997).
- [181] R. Semiat *et al.*, Chem. Phys. Lett. **255**, 327 (1996).
- [182] P. J. Hay and W. R. Wadt, J. Chem. Phys. **82**, 270 (1985).
- [183] R. W. Larsen, J. Am. Chem. Soc. **130**, 11246 (2008).
- [184] S. Kristyan and P. Pulay, Chem. Phys. Lett. **229**, 175 (1994).
- [185] F. Weigend, M. Haser, H. Patzelt, and R. Ahrichs, Chem. Phys. Lett. **294**, 143 (1998).
- [186] F. Weigend, A. Kohn, and C. Hattig, J. Chem. Phys. **116**, 3175 (2002).
- [187] B. et al., NWChem, A Computational Chemistry Package for Parallel Computers, Version 5.1, Pacific Northwest National Laboratory, 2007.
- [188] R. A. K. et al., Computer Phys. Comm. **128**, 260 (2000).
- [189] K. Palmo and S. Krimm, Chem. Phys. Lett. **395**, 133 (2004).
- [190] H. Yu, T. Hansson, and W. F. van Gunsteren, J. Chem. Phys. **118**, 221 (2003).
- [191] D. A. McQuarrie, *Statistical Mechanics* (University Science Books, Sausalito, CA, 2000).

- [192] R. Curl, H. P. Hopkins, and K. Pitzer, J. Chem. Phys. **48**, 4064 (1968).
- [193] R. P. Feynman, Rev. Mod. Phys. **20**, 367 (1948).
- [194] L. Schulman, *Techniques and Applications of Path Integration* (Dover, New York, 2005).
- [195] R. Feynman and A. Hibbs, *Quantum Mechanics and Path Integrals* (McGraw-Hill, New York, 1965).
- [196] R. Feynman and H. Kleinert, Phys. Rev. A **34**, 5080 (1986).
- [197] H. Kleinert, Phys. Lett. B **280**, 251 (1992).
- [198] J. W. Lawson, Phys. Rev. E **61**, 61 (2000).
- [199] R. A. Sack, Mol. Phys. **2**, 8 (1959).
- [200] E. A. Guggenheim, J. Chem. Phys. **7**, 103 (1939).
- [201] E. M. Pearson, T. Halicioglu, and W. A. Tiller, Phys. Rev. A **32**, 3030 (1985).
- [202] F. Hioe, D. MacMillen, and E. Montroll, Phys. Rep. **43**, 305 (1978).
- [203] R. S. Berry, S. A. Rice, and J. Ross, *Physical Chemistry, Second Edition* (Oxford University Press, New York, 2000).
- [204] R. K. P. Zia, E. F. Redish, and S. R. McKay, Am. J. Phys. **77**, 614 (2009).
- [205] J. W. Gibbs, in *The Collected Works of J. Willard Gibbs*, edited by W. Longley and R. V. Name (Longmans, Green and Co., New York, 1928).

- [206] K. Huang, *Statistical Mechanics* (John Wiley and Sons, New York, 1963).
- [207] H. Eyring, D. Henderson, B. J. Stover, and E. M. Eyring, *Statistical Mechanics and Dynamics* (John Wiley and Sons, New York, 1964).
- [208] D. Chandler, *Introduction to Modern Statistical Mechanics* (Oxford University Press, New York, 1987).
- [209] R. A. Alberty, Pure Appl. Chem. **73**, 1349 (2001).
- [210] J. Haile and H. Graben, Mol. Phys. **40**, 1433 (1980).
- [211] J. R. Ray, H. Graben, and J. M. Haile, J. Chem. Phys. **75**, 4076 (1981).
- [212] T. Chalikian, M. Totrov, R. Abagyan, and K. Breslauer, J. Mol. Biol. **260**, 588 (1996).
- [213] R. Devane *et al.*, Biophys. J. **85**, 2801 (2003).
- [214] C. Ridley *et al.*, Chem. Phys. Lett. **418**, 137 (2006).
- [215] J. Phillips *et al.*, J. Comput. Chem. **26**, 1781 (2004).
- [216] K. Chu *et al.*, Nature **403**, 921 (2000).
- [217] K. Gekko and Y. Hasegawa, Biochemistry **25**, 6563 (1986).
- [218] K. Gekko *et al.*, Protein Science **5**, 542 (1996).
- [219] J. Jager, M. Moser, U. Sauder, and J. Jansonius, J. Mol. Biol. **239**, 285 (1994).

Appendices

Appendix A. Virial Equation for the Pressure

The canonical expression for the pressure is

$$\begin{aligned} P &= -kT \frac{1}{Q} \frac{\partial Q}{\partial V} \\ &= -kT \frac{1}{Q} \frac{\partial}{\partial V} \frac{1}{N! \Lambda^{3N}} \int_V dV_1 \dots \int_V dV_N e^{-\beta U(r_1, \dots, r_N)} \end{aligned} \quad (13.1)$$

where

$$\int_V dV_i = \int_0^L dr_i^x \int_0^L dr_i^y \int_0^L dr_i^z \quad (13.2)$$

Switching to scaled coordinates, we note the following two useful items:

$$\overline{r_i^x} = \frac{r_i^x}{V^{1/3}} \quad (13.3)$$

$$\frac{\partial r}{\partial V} = \frac{r}{3V} \quad (13.4)$$

Expressed in scaled coordinates, the pressure becomes:

$$\begin{aligned} P &= -kT \frac{1}{Q} \frac{\partial}{\partial V} \frac{1}{N! \Lambda^{3N}} \overline{V^{1/3}} \int_0^1 \overline{dr_1^x} \dots \overline{V^{1/3}} \int_0^1 \overline{dr_N^z} e^{-\beta U(\overline{r_1}, \dots, \overline{r_N})} \\ &= kT \frac{1}{Q} N \frac{1}{V^{N+1}} \frac{1}{N! \Lambda^{3N}} \int_0^1 dr_1 \dots \int_0^1 dr_N e^{-beta U(\overline{r_1}, \dots, \overline{r_N})} \\ &\quad - kT \frac{1}{Q} \frac{1}{N! \Lambda^{3N}} \overline{V^N} \int_0^1 dr_1 \dots \int_0^1 dr_N - \beta \frac{\partial U}{\partial r} \frac{\partial r}{\partial V} e^{-beta U(\overline{r_1}, \dots, \overline{r_N})} \\ &= \rho kT + \frac{1}{Q} \frac{1}{N! \Lambda^{3N}} \overline{V^N} \int_0^1 dr_1 \dots \int_0^1 dr_N - \beta \frac{\partial U}{\partial r} \frac{r}{3V} e^{-\beta U} \\ &= \rho kT + \frac{1}{3V} \langle r \frac{\partial U}{\partial r} \rangle = \rho kT - \frac{1}{3V} \langle \vec{r} \cdot \vec{F} \rangle \end{aligned} \quad (13.5)$$

Appendix B. Frenkel Volume Derivative

The pressure is canonically identified as:

$$P = -\frac{\partial A}{\partial V} \quad (13.6)$$

where the particle number, N , and the temperature T are held constant.

This free energy derivative can be approximated by central difference:

$$\begin{aligned} P &\approx -\frac{1}{2} \left\{ \frac{A(V + \Delta V) - A(V)}{\Delta V} + \frac{A(V - \Delta V) - A(V)}{-\Delta V} \right\} \\ &= -\frac{1}{2} \left\{ \frac{-kT \ln Q(V + \Delta V) + kT Q(V)}{\Delta V} - \frac{-kT \ln Q(V - \Delta V) + kT \ln Q(V)}{\Delta V} \right\} \\ &= \frac{kT}{2\Delta V} \left\{ \ln \left[\frac{Q(V + \Delta V)}{Q(V)} \right] - \ln \left[\frac{Q(V - \Delta V)}{Q(V)} \right] \right\} \\ &= \frac{kT}{2\Delta V} \ln \left\{ \frac{\frac{Q(V+\Delta V)}{Q(V)}}{\frac{Q(V-\Delta V)}{Q(V)}} \right\} \\ &= \frac{kT}{2\Delta V} \ln \left\{ \frac{\frac{\int_{V+\Delta V} dV_1 \dots e^{-\beta U(V+\Delta V)}}{\int_V dV_1 \dots e^{-\beta U(V)}}}{\frac{\int_{V-\Delta V} dV_1 \dots e^{-\beta U(V-\Delta V)}}{\int_V dV_1 \dots e^{-\beta U(V)}}} \right\} \quad (13.7) \\ &\quad (13.8) \end{aligned}$$

since we can write $U(V + \Delta V) = U(V) + U(\Delta V)$ we have:

$$\begin{aligned} P &= \frac{kT}{2\Delta V} = \ln \left[\frac{\langle e^{-\beta U(+\Delta V)} \rangle \left(\frac{V+\Delta V}{V} \right)^N}{\langle e^{-\beta U(-\Delta V)} \rangle \left(\frac{V-\Delta V}{V} \right)^N} \right] \\ &= \frac{kT}{2\Delta V} \left\{ N \ln \left(\frac{V + \Delta V}{V - \Delta V} \right) + \ln \left(\frac{\langle e^{-\beta U(+\Delta V)} \rangle}{\langle e^{-\beta U(-\Delta V)} \rangle} \right) \right\} \quad (13.9) \end{aligned}$$

However, we note that the Taylor expansion $\ln(1 + x) = x + O(3)$ and so

$$\begin{aligned} \ln\left(\frac{V + \Delta V}{V - \Delta V}\right) &= \ln\left(\frac{1 + \frac{\Delta V}{V}}{1 - \frac{\Delta V}{V}}\right) \\ &= \ln\left(1 + \frac{\Delta V}{V}\right) - \ln\left(1 - \frac{\Delta V}{V}\right) = \frac{2\Delta V}{V} \end{aligned} \quad (13.10)$$

$$P = \frac{NkT}{V} + \frac{kT}{2\Delta V} \ln\left(\frac{\langle e^{-\beta U(+\Delta V)} \rangle}{\langle e^{-\beta U(-\Delta V)} \rangle}\right) \quad (13.11)$$

where the Boltzmann factors that arise due to performing volume changes to the system may be evaluated by Monte Carlo.

Appendix C. Widom Potential Distribution Theorem

The following is a derivation of the Potential Distribution Theorem due to Benjamin Widom. [96] This result is remarkable in that it expresses the chemical potential of the system as a locally invariant quantity, *even in the absence of homogeneity.*

We begin by considering the density function for a system of N molecules under conditions of constant V, T :

$$\rho(r_1) = N \frac{\int dr_2 \dots \int dr_N e^{-\beta U(r_1, \dots, r_N)}}{\int dr_1 \dots \int dr_N e^{-\beta U(r_1, \dots, r_N)}} \quad (13.12)$$

In addition, consider a Boltzmann average of the quantity $\psi(r_1)$, the energy of inserting an additional molecule into a system of $N - 1$ molecules:

$$\langle e^{-\beta\psi(r_1)} \rangle = \frac{\int dr_2 \dots \int dr_N e^{-\beta U(r_2, \dots, r_N)} e^{-\beta\psi(r_1)}}{\int dr_2 \dots \int dr_N e^{-\beta U(r_2, \dots, r_N)}} \quad (13.13)$$

Dividing Equation 13.13 by Equation 13.12 we obtain:

$$\begin{aligned} \frac{\langle e^{-\beta\psi(r_1)} \rangle}{\rho(r_1)} &= \frac{\int dr_2 \dots \int dr_N e^{-\beta U(r_2, \dots, r_N)} e^{-\beta\psi(r_1)}}{\int dr_2 \dots \int dr_N e^{-\beta U(r_2, \dots, r_N)}} \\ &\div N \frac{\int dr_2 \dots \int dr_N e^{-\beta U(r_1, \dots, r_N)}}{\int dr_1 \dots \int dr_N e^{-\beta U(r_1, \dots, r_N)}} = \frac{1}{N} \frac{Z_N}{Z_{N-1}} \end{aligned} \quad (13.14)$$

Notice that Equation 13.14 is a constant. It is interesting that in an interfacial system, where the quantity $\langle e^{-\beta\psi(r_1)} \rangle$ will be quite small in a region of high repulsion, there is an exact cancellation with the density in that region such that a constant is obtained in all regions of the interface. More specifically,

$$\mu = -kT \ln \left\{ \frac{\langle e^{-\beta\psi(r_1)} \rangle}{\Lambda^3 \rho(r_1)} \right\} \quad (13.15)$$

Appendix D. Fugacity

Making use of the Gibbs free energy,

$$dG = -SdT + VdP + \mu dN \quad (13.16)$$

and integrating over an adiabatic pressure change, we arrive at:

$$\int_{\mu_0}^{\mu} d\mu = \int_{P_0}^P \overline{V} dP \quad (13.17)$$

We note that for an ideal gas:

$$\int_{\mu_0}^{\mu} d\mu = \int_{P_0}^P \frac{kT}{P} dP = kT \ln \frac{P}{P_0} \quad (13.18)$$

$$\mu = \mu_0 + kT \ln \frac{P}{P_0} \quad (13.19)$$

$$(13.20)$$

If we have the experimental molar volume of a non-ideal gas (as a function of the applied pressure) then we would have all of the information necessary to obtain μ and thus evaluate properties in the grand canonical ensemble.

For a non-ideal gas, we define a correction to the pressure called the fugacity:

$$\mu = \mu_0 + kT \ln \frac{f}{f_0} \quad (13.21)$$

where in the ideal gas limit $f = P$

We define the function

$$\Phi = \frac{P\bar{V} - kT}{P} \quad (13.22)$$

and so the work equation of interest becomes:

$$\begin{aligned} \int_{P_0}^P \bar{V} dP &= \int_{P_0}^P \left(\frac{kT}{P} + \Phi \right) dP \\ &= kT \ln \frac{P}{P_0} + \int_{P_0}^P \Phi dP \end{aligned} \quad (13.23)$$

$$kT \ln \frac{f}{f_0} = \ln \frac{P}{P_0} + \int_{P_0}^P \Phi dP \quad (13.24)$$

$$\mu = kT \ln \frac{fP_0}{f_0 P} = \int_{P_0}^P \Phi dP \quad (13.25)$$

(13.26)

but we must obtain an ideal gas in the low pressure limit, i.e.:

$$\lim_{P \rightarrow 0} \frac{f}{P} = 1 \quad (13.27)$$

and this can only happen if $\int_{P_0}^P \Phi dP = 0$

Therefore,

$$\mu = kT \ln \frac{f}{P} = \int_{P_0}^P \Phi dP \quad (13.28)$$

We define $\phi = \frac{f}{P}$ and arrive at the equation for the fugacity coefficient

$$\ln \phi = \beta \int_0^P \frac{P\bar{V} - kT}{P} dP \quad (13.29)$$

Appendix E. Isosteric Heat of Adsorption (Q_{st})

We consider a closed system as having adsorbed N_a molecules in equilibrium with a gaseous phase of N_g molecules. Starting from the fundamental equation of thermodynamics

$$dE = TdS - PdV + \mu dN \quad (13.30)$$

we can identify the heat adsorbed by the system as $dQ = -TdS$

The isosteric heat of adsorption, Q_{st} , is defined as

$$Q_{st} = \frac{\partial Q}{\partial N_a} \quad (13.31)$$

We note that $dN^g = -dN^a$

$$\begin{aligned} \frac{\partial Q}{\partial N_a} &= -T \frac{\partial S}{\partial N_a} = -T \frac{\partial}{\partial N_a} (S_g + S_a) \\ &= T \left(\frac{\partial S_g}{\partial N_g} - \frac{\partial S_a}{\partial N_a} \right) \end{aligned} \quad (13.32)$$

We note that, at equilibrium, the Gibbs free energy is:

$$dG = -SdT + VdP + \mu dN = 0 \quad (13.33)$$

$$\mu dN = -SdT - VdP \quad (13.34)$$

$$\mu = \frac{\partial S}{\partial N}dT - \frac{\partial V}{\partial N}dP \quad (13.35)$$

$$(13.36)$$

Since the gaseous and adsorbed phases are in equilibrium, $\mu_g = \mu_a$. Therefore,

$$\frac{\partial S_g}{\partial N_g}dT - \frac{\partial V_g}{\partial N_g}dP = \frac{\partial S_a}{\partial N_a}dT - \frac{\partial V_a}{\partial N_a}dP \quad (13.37)$$

$$(13.38)$$

Upon arranging for dP/dT , we have:

$$\frac{dP}{dT} = \frac{\frac{\partial S_g}{\partial N_g} - \frac{\partial S_a}{\partial N_a}}{\frac{\partial V_g}{\partial N_g} - \frac{\partial V_a}{\partial N_a}} \quad (13.39)$$

where, making use of Equation 13.32, we have:

$$\frac{dP}{dT} = \frac{Q_{st}}{T(\overline{V}_g - \overline{V}_a)} \quad (13.40)$$

If we assume that (a) the molar volume of the adsorbed phase is negligible ($V_a = 0$) and (b) that the reservoir is within the ideal gas regime, $\overline{V}_g = \overline{V}_{ideal} =$

kT/P_{ideal} :

$$\frac{dP}{dT} = \frac{Q_{st}}{T \frac{kT}{P_{ideal}}} = \frac{P_{ideal} Q_{st}}{kT^2} \quad (13.41)$$

$$Q_{st} = RT^2 \frac{1}{P} \frac{dP}{dT} = RT^2 \frac{\partial \ln P}{\partial T} \quad (13.42)$$

As a final note, we point out that the isosteric heat of adsorption is distinct from the *adsorption enthalpy*

$$\frac{\partial H}{\partial N_a} = -RT^2 \frac{\partial \ln f}{\partial T} \quad (13.43)$$

where f is the fugacity of the gas under consideration.

Appendix F. autocorr.c

```
*****
/* Determines the correlation time interval for a random sequence */
/*
/* The input signal is specified in an input file where each line holds a single */
/* floating point value. The signal values are all shifted by the average value in */
/* order that the signal may fluctuate about the x-axis. The normalization constant is */
/* then calculated so that the autocorrelation function will be normalized. */
/* A random signal will see a rapid decay time, whereas a signal with greater */
/* periodicity would be evident by judging decay time until crossing the x-axis, thus */
/* determining the correlation interval. The default method is to integrate the */
/* autocorrelation function to get the correlation time - an alternative method */
/* based upon the slope of a semilog scale can also be calculated by the -l option */
/* on the command line. Use of both methods yields a reasonable value for the */
/* correlation time of the data. */
/*
/*
/* This implementation utilizes the autocorrelation function as follows. The j-th value */
/* of the autocorrelation function (in latex) is given by:
/*
/*           ac(j) =  $\sum_{i=j}^{N-i} s(i)s(i+j)$ 
/*
/* where s(k) is the value of the signal at time k and N is the total number of data */
/* points in the signal. The average of this sum is then taken by dividing by */
/* the number of points processed and a normalization constant, where the */
/* normalization constant is given by:
/*
/*
/* usage:
/*     autocorr <-l> [datfile] > output_file
/*
/*
/* compilation:
/*     gcc -o autocorr autocorr.c -lm
/*
/*
/* References:
/*     Ifeachor, Jervis "Digital Signal Processing: A Practical Approach"
/*     Mitra, "Digital Signal Processing"
/*     Frenkel, "Understanding Molecular Simulation"
/*
/*
/* ©2005 Jonathan Belof
/* Space Research Group
/* Department of Chemistry
/* University of South Florida
*****
```

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <math.h>
#include <time.h>

#define NUM_LINES    4000000
#define THRESHOLD   0.01    /* if the ac gets close to 0 within this amount, consider a "hit" */
#define DECAY_POINT  2       /* if this many "hits" occurs, consider the ac function decayed */

int autocorr(double *dat, int num, int log_scale) {
    int i, j;          /* working indices */
    double avg = 0;    /* pre-processed average for the signal */
    double norm = 0;   /* normalization constant */
    double ac = 0;     /* current autocorrelation value */
    int hits = 0;      /* number of times the ac has passed the x-axis */
    int decayed = 0;   /* used to flag when ac is decayed */
    double ac_integral; /* integration value */
    int corr_time = 0;  /* correlation time estimated by integration */
    int log_corr_time = 0; /* correlation time estimated by semilog extrapolation */

    /* calculate the uncorrelated average */
    for(i = 0; i < num; i++)
        avg += *(dat + i);
    avg /= num;

    /* translate the signal by the average to fluctuate about the x-axis */
    for(i = 0; i < num; i++)
        *(dat + i) -= avg;

    /* calculate the normalization constant */
    for(i = 0; i < num; i++) {
        norm += *(dat + i) * *(dat + i));
    }
    norm /= num;

    /* autocorrelate the signal */
    for(i = 0; i < (num/2); i++) {
        ac = 0;
        for(j = 0; j < (num - i); j++) {
            ac += *(dat + j) * *(dat + i + j));
        }
        /* normalize the autocorrelation */
        ac /= (num - i) * norm;

        /* determine if it is crossing the x-axis */
        if(fabs(ac) <= THRESHOLD) ++hits;

        /* determine if we have sufficiently decayed */
        if(hits >= DECAY_POINT) decayed = 1;

        /* generate semilog scale, and determine correlation time from its inverse slope */
    }
}
```

```

    if(log_scale) {
        printf("%d %f\n", i, exp(ac));
        if(decayed && !log_corr_time) { /* take inverse slope until decay point */
            log_corr_time = (int)(ceil(((double)i)/(exp(1.0) - exp(ac))));
        }
    } else {
        printf("%d %f\n", i, ac);
        /* sum this contribution into the integral, until we get to the noisy part */
        if(!decayed) ac_integral += ac;
    }
}

/* return the estimated correlation time */
if(log_scale)
    corr_time = log_corr_time;
else
    corr_time = (int)ceil(ac_integral);

return(corr_time);
}

void usage(char *progrname) {
    fprintf(stderr, "usage: %s <-> [datafile]\n", progrname);
    fprintf(stderr, "options: -l output logarithmic scale\n");
    exit(1);
}

int main(int argc, char **argv) {
    int i, n = 0, log_scale = 0;
    char *datfile;
    FILE *fp;
    double *dat;
    int c;
    extern char *optarg;
    extern int optind, getopt;
    if(argc < 2) usage((char *)argv[0]);
    while ((c = getopt(argc, (char **)argv, "l")) != -1) {
        switch (c) {
            case 'l':
                log_scale = 1;
                break;
            case '?':
                usage((char *)argv[0]);
        }
    }
    datfile = argv[optind];
    if(!datfile) {
        fprintf(stderr, "error: invalid datafile\n");
        usage((char *)argv[0]);
    }
    else
        fp = fopen(datfile, "r");
    if(!fp) {
        fprintf(stderr, "error: could not open datafile %s\n", datfile);
        usage((char *)argv[0]);
    }
    /* load in the data - XXX do without num_lines */
    dat = (double *)calloc(NUM_LINES, sizeof(double));
    if(!dat) {
        fprintf(stderr, "couldn't allocate working memory buffer\n");
        exit(1);
    }
    for(i = 0; n != EOF; i++) {
        n = fscanf(fp, "%lg", (dat + i));
    }
    fclose(fp);
    printf("# correlation time = %d\n", autocorr(dat, (i - 1), log_scale));
    free(dat);
    exit(0);
}

```

Appendix G. get_virial_coefficients.c

```

/*
Calculate the quantum second virial coefficient to second-order via the Wigner-Kirkwood
semiclassical expansion, including the third-order ideal quantum exchange term

©2007, Jonathan Belof
Space Research Group
Department of Chemistry
University of South Florida

*/
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>

#define H 6.626068e-34 /* Planck's constant in J s */
#define HBAR 1.054571e-34 /* above divided by 2pi in J s */
#define KB 1.3806503e-23 /* Boltzmann's constant in J/K */
#define NA 6.0221415e23 /* Avogadro's number */
#define H2_MASS 3.348e-27 /* mass of H2 molecule in kg */

#define MAXLINE 256

void usage(char *progname) {
    fprintf(stderr, "usage: %s <min temp> <max temp> <inc temp> <pe file>\n", progname);
    exit(1);
}

int main(int argc, char **argv) {
    int i, N;
    FILE *fp_fit;
    char linebuf[MAXLINE];
    double r, r_min, r_max, r_inc;
    double first_derivative, second_derivative, integrand;
    double *r_input, *fit_input;
    char fit_file[MAXLINE];
    double temperature, temperature_min, temperature_max, temperature_inc;
    double B, B_classical, B_quantum_first, B_quantum_second, B_quantum_ideal;

    if(argc != 5)
        usage(argv[0]);

    temperature_min = atof(argv[1]);
    temperature_max = atof(argv[2]);
    temperature_inc = atof(argv[3]);
    strcpy(fit_file, argv[4]);

    /* read in the number of lines */
    fp_fit = fopen(fit_file, "r");
    for(N = 0; fgets(linebuf, MAXLINE, fp_fit); N++);
    fclose(fp_fit);

    /* allocate space */
    r_input = calloc(N, sizeof(double));
    fit_input = calloc(N, sizeof(double));

    /* read in the isotropic function */
    fp_fit = fopen(fit_file, "r");
    for(i = 0; i < N; i++)
        fscanf(fp_fit, "%lg\n", &r_input[i], &fit_input[i]);
    fclose(fp_fit);

    /* determine independent domain */
    r_min = r_input[0];
    r_max = r_input[N-1];
    r_inc = r_input[1] - r_input[0];

    for(temperature = temperature_min; temperature <= temperature_max; temperature += temperature_inc) {

        /* calculate the classical part */
        B_classical = 0;
        for(r = r_min, i = 0; r < r_max; r += r_inc, i++) {
            integrand = (exp(-fit_input[i]/temperature) - 1.0)*pow(r, 2.0)*r_inc;
            B_classical += integrand;
        }
        B_classical *= -2.0*M_PI;
        B_classical *= 1.0e-24*NA; /* convert to cm^3/mol */

        /* calculate the first order quantum correction */
        B_quantum_first = 0;
        for(r = r_min, i = 0; r < r_max; r += r_inc, i++) {

            /* take the central derivative */
            first_derivative = KB*(fit_input[i+1] - fit_input[i])/r_inc;
            integrand = exp(-fit_input[i]/temperature)*pow(first_derivative*r, 2.0)*r_inc;
            B_quantum_first += integrand;
        }
        B_quantum_first *= (H*H/H2_MASS)/(24.0*M_PI*pow(KB*temperature, 3.0));
        B_quantum_first *= 1.0e-4*NA; /* convert to cm^3/mol */

        /* calculate the second order quantum correction */
        B_quantum_second = 0;
        for(r = r_min, i = 0; r < (r_max - r_inc); r += r_inc, i++) {
    }
}

```

```

first_derivative = KB*(fit_input[i+1] - fit_input[i])/r_inc;
second_derivative = KB*((fit_input[i+2] - fit_input[i+1])/r_inc - first_derivative)/r_inc;

integrand = pow(second_derivative, 2.0) + 2.0*pow((first_derivative/r), 2.0);
integrand += (10.0/(9.0*KB*temperature))*pow(first_derivative, 3.0)/r;
integrand -= (5.0/(36.0*pow(KB*temperature, 2.0)))*pow(first_derivative, 4.0)/r;
integrand *= exp(-fit_input[i]/temperature)*pow(r, 2.0)*r_inc;

B_quantum_second += integrand;
}
B_quantum_second *= pow(H*H/H2_MASS, 2.0)/(960.0*M_PI*M_PI*M_PI*pow(KB*temperature, 4.0));
B_quantum_second *= -1.0e11*NA;

/* the exchange term for an ideal Bose-Einstein gas */
B_quantum_ideal = -NA*pow((H*H/(2.0*M_PI*M2_MASS*KB*temperature)), (3.0/2.0))*pow(2.0, (-5.0/2.0));
B_quantum_ideal *= 1.0e6;

/* the total virial coefficient */
B = B_classical + B_quantum_first + B_quantum_second + B_quantum_ideal;
printf("%f %f\n", temperature, B);
/*printf("%f %f %f %f %f %f\n", temperature, B, B_classical, B_quantum_first, B_quantum_second, B_quantum_ideal);*/
}

return(0);
}

```

Appendix H. virial_iso.c

```
/****************************************************************************
 * Calculate the virial isotherm using the "a" *
 * coefficients from the CRC      */
 * Jonathan Belf          */
 * Department of Chemistry      */
 * University of South Florida */
 * compile with:             */
 * gcc -std=c99 -o v ./virial_iso.c -lm */
 */

#include <sys/types.h>
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <math.h>

#define T0      298.15      /* standard temperature (K) */
#define NA     6.02214e23    /* Avogadro's number */
#define KB     1.3806503e-23 /* Boltzmann's constant */

#define ATM2PSC 101325      /* convert from atm to Pascals */
#define DP      100.0        /* small interval in Pascals */

void usage(char *progname) {
    fprintf(stderr, "usage: %s <mw> <temp> <start pressure> <stop pressure> <points> <a1> [a2] [a3] ...\\n", progname);
    fprintf(stderr, "\\t<mw>: molecular/atomic weight (g/mol)\\n");
    fprintf(stderr, "\\t<temp>: temperature of isotherm (K)\\n");
    fprintf(stderr, "\\t<start pressure>: starting pressure of isotherm (atm)\\n");
    fprintf(stderr, "\\t<stop pressure>: stopping pressure of isotherm (atm)\\n");
    fprintf(stderr, "\\t<points>: number of data points to calculate (integer)\\n");
    fprintf(stderr, "\\t<a1>: the i-th a-coefficient from the CRC\\n");
    fprintf(stderr, "\\nuseful parameters:\\n");
    fprintf(stderr, "\\tH2:\\ta1=15.4 a2=-9.0 a3=-0.2\\n");
    fprintf(stderr, "\\tHe:\\ta1=12 a2=-1\\n");
    fprintf(stderr, "\\tH2O:\\ta1=-1158 a2=-5157 a3=-10301 a4=-10597 a5=-4415\\n");
    fprintf(stderr, "\\tUF6:\\ta1=-1204 a2=-2690 a3=-2144\\n");
    fprintf(stderr, "\\tN2:\\ta1=-4 a2=-56 a3=-12\\n");
    fprintf(stderr, "\\tO2:\\ta1=-16 a2=-62 a3=-4 a4=-3\\n");
    fprintf(stderr, "\\tCO:\\ta1=-9 a2=-58 a3=-18\\n");
    fprintf(stderr, "\\tNO:\\ta1=-12 a2=-119 a3=89 a4=-73\\n");
    fprintf(stderr, "\\tCO2:\\ta1=-127 a2=-288 a3=-118\\n");
    fprintf(stderr, "\\tAr:\\ta1=-16 a2=-60 a3=-10\\n");
    fprintf(stderr, "\\tKr:\\ta1=-51 a2=-118 a3=-29 a4=-5\\n");
    fprintf(stderr, "\\tXe:\\ta1=-130 a2=-262 a3=-87\\n");

    exit(1);
}

int main(int argc, char **argv) {
    double mw;           /* molecular weight (g/mol) */
    double temperature; /* temperature to perform the isotherm at (K) */
    double KT;           /* thermal energy (J) */
    double P;            /* current pressure (Pascals) */
    double dP;           /* pressure interval (Pascals) */
    double pressure_lb, pressure_ub; /* lower and upper pressure bounds (atm) */
    int points;          /* number of data points to calculate */
    double B;            /* 2nd virial coefficient (cm^3/mol) */
    double density;      /* density, to be solved for (g/cm^3) */
    int i;               /* used for arg parsing */
    double current_a;   /* used in parsing out the "a" coefficients */

    if(argc < 7)
        usage(argv[0]);
    else { ++argv; --argc; }

    /* read in the main arguments */
    mw = atof(*argv); ++argv; --argc;
    temperature = atof(*argv); ++argv; --argc;
    pressure_lb = atof(*argv); ++argv; --argc;
    pressure_ub = atof(*argv); ++argv; --argc;
    points = atoi(*argv); ++argv; --argc;

    /* argument checking */
    if((mw < 1) || (mw > 267)) { /* synthesize a new isotope lately? */
        fprintf(stderr, "%s: invalid molecular weight\\n", argv[0]);
        usage(argv[0]);
    }

    if(temperature < 0) {
        fprintf(stderr, "%s: invalid temperature\\n", argv[0]);
        usage(argv[0]);
    }

    if((pressure_lb > pressure_ub) || (pressure_lb < 0) || (pressure_ub < 0)) {
        fprintf(stderr, "%s: invalid pressure range specified\\n", argv[0]);
        usage(argv[0]);
    }

    if(points < 1) {
        fprintf(stderr, "%s: invalid number of data points\\n", argv[0]);
        usage(argv[0]);
    }

    /* read in the "a" coefficients */
    for(i = 0, B = 0; i < argc; i++) {
        current_a = atof(*argv + i));
    }
}
```

```

        B += current_a*(pow((T0/temperature - 1), i));
    }

/* unit conversions to SI */
pressure_lb *= ATM2PSC;           /* convert from atm to Pascals */
pressure_ub *= ATM2PSC;           /* ditto */
dP = (pressure_ub - pressure_lb)/points; /* interval to use in calculating the function */
B /= NA*1.0e6;                   /* convert from cm^3/mol to m^3/molec*/
kT = KB*temperature;             /* calculate KT (J) */

for(P = pressure_lb; P <= pressure_ub; P += dP) {
    /* solve for the density quadratically */
    density = (-1 + sqrt(1 + 4*B*P/kT))/(2*B);
    density *= mw/(NA*1.0e6); /* convert to g/cm^3 */

    if(!finite(density)) /* if we're in non-physical territory, don't bother with NaN's */
        break;
    else
        printf("%f %f\n", P/ATM2PSC, density); /* output the pressure in atm, density in g/cm^3 */
}

exit(0);
}

```

Appendix I. structure_factor.c

```

/****************************************************************************
 * Calculates the structure factor S(k) from a g(r)      */
/*          */
/* In latex, the structure factor is:          */
/* S(k)=1+4\pi\frac{\rho_0}{k}\int_{r_0}^{\infty} r[g(r)-1]\sin(kr)dr   */
/*          */
/* where rho is the atomic density, k is in distance units */
/*          */
/* In order to eliminate noise form the S(k), the tail end of */
/* the g(r) needs to be extrapolated to a cleaner analytical */
/* form. The form used here is:          */
/* h(r) = \frac{A}{r} e^{-\frac{r}{r_0}} \sin(\frac{r}{r_0})   */
/*          */
/* where the parameters A, r0, r1 as well as the extrapolation */
/* point, are determined from fitting to the g(r)      */
/*          */
/* Fitting procedure:          */
/* Set the extrapolation point to the third zero value in the */
/* h(r). Set A to the height of the next peak. Set r0 to a */
/* large value so that the exponential damping is minimal - */
/* then play with r1 until the phase of the sin wave is a close */
/* fit. Then decrease r0 to increase the exponential damping */
/* until the fit is final - A may need minor tweaking at that */
/* point.          */
/*          */
/* usage: ./sf <-f> [density] [xp] [A] [r0] [r1] [datafile] */
/* <-f> - turns on fitting mode, h(r) extrapolated is output */
/* side-by-side with the g(r)-based h(r)          */
/* [density] - density in units of molecules/A^3          */
/* [datfile] - two column file containing the g(r) and domain */
/*          */
/* compilation: gcc -o sf sf.c -lm          */
/*          */
/* @2006 */
/* Jonathan Belof          */
/* Space Research Group          */
/* Department of Chemistry          */
/* University of South Florida          */
/****************************************************************************

#define RESOLUTION 0.001
#define MAXLINE 4000000
#define SK_LOWER_BOUND 0
#define SK_UPPER_BOUND 10
#define H_LOWER_BOUND 0
#define H_UPPER_BOUND 100

struct gor_t {
    double domain;
    double range;
};

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <math.h>
#include <time.h>

/* generate a curve to fit A, r0 and r1 */
/* first column output is g(r) - 1, the second column is the extrapolated form */
void fit_gor(struct gor_t *gor, int num_gor, double extrapolation_point, double A, double r0, double r1) {

    int i, j; /* working indices */
    double *h; /* h(r) = g(r) - 1 */
    double *h_x; /* h(r) = extrapolated */
    int lower_bound, upper_bound; /* lower/upper bounds for the domain of h(r) */
    double resolution = RESOLUTION;

    lower_bound = (int)rint(((double)H_LOWER_BOUND) / resolution);
    upper_bound = (int)rint(((double)H_UPPER_BOUND) / resolution);
    h = calloc(upper_bound, sizeof(double));
    h_x = calloc(upper_bound, sizeof(double));

    printf("# extrapolation point = %f\n", extrapolation_point);
    printf("# A = %f\n", A);
    printf("# r0 = %f\n", r0);
    printf("# r1 = %f\n", r1);

    /* generate h(r) = g(r) - 1 */
    for(i = 0; i < num_gor; i++) {
        for(j = lower_bound; j < upper_bound; j++) {
            if(fabs((gor + i)->domain - ((double)j)*resolution) < resolution)
                h[j] = (gor + i)->range - 1.0;
        }
    }

    /* generate h(r) = extrapolated */
    for(i = lower_bound; ((double)i)*resolution < extrapolation_point; i++) { /* use the g(r) data */
        h_x[i] = h[i];
    }
    for(; i < upper_bound; i++) { /* start extrapolating */
        h_x[i] = (A / (((double)i)*resolution))*exp(-(double)i)*resolution/r0)*sin(((double)i)*resolution/r1);
    }

    /* output the two h(r)'s */
    for(i = lower_bound; i < upper_bound; i++)
        printf("%f %f\n", ((double)i)*resolution, h[i], h_x[i]);
}

/* generate the extrapolated h(r) and return a point to it */

```

```

double *make_h(struct gor_t *gor, int num_gor, double extrapolation_point, double A, double r0, double r1) {
    int i, j; /* working indices */
    double *h; /* h(r) = g(r) - 1 */
    double *h_x; /* h(r) = extrapolated */
    int lower_bound, upper_bound; /* lower/upper bounds for the domain of h(r) */
    double resolution = RESOLUTION;

    lower_bound = (int)rint(((double)H_LOWER_BOUND) / resolution);
    upper_bound = (int)rint(((double)H_UPPER_BOUND) / resolution);
    h = calloc(upper_bound, sizeof(double));
    h_x = calloc(upper_bound, sizeof(double));

    printf("# extrapolation point = %f A\n", extrapolation_point);
    printf("# A = %f\n", A);
    printf("# r0 = %f\n", r0);
    printf("# r1 = %f\n", r1);

    /* generate h(r) = g(r) - 1 */
    for(i = 0; i < num_gor; i++) {
        for(j = lower_bound; j < upper_bound; j++) {
            if(fabs((gor + i)->domain - ((double)j)*resolution) < resolution)
                h[j] = (gor + i)->range - 1.0;
        }
    }

    /* generate h(r) = extrapolated */
    for(i = lower_bound; ((double)i)*resolution < extrapolation_point; i++) { /* use the g(r) data */
        h_x[i] = h[i];
    }
    for(; i < upper_bound; i++) { /* start extrapolating */
        h_x[i] = (A / (((double)i)*resolution))*exp(-((double)i)*resolution/r0)*sin(((double)i)*resolution/r1);
    }

    free(h);
    return(h_x);
}

/* this function calculates the structure factor S(k) */
void structure_factor(struct gor_t *gor, int num_gor, double density, double extrapolation_point, double A, double r0, double r1) {
    int i, j;
    double *h;
    int sk_lower_bound, sk_upper_bound;
    int h_lower_bound, h_upper_bound;
    double integral, k, r;
    double resolution = RESOLUTION;

    /* get the integration boundaries */
    /* S(k) boundaries */
    sk_lower_bound = (int)rint(((double)SK_LOWER_BOUND) / resolution);
    /* advance the lower_bound so that we don't divide by zero */
    ++sk_lower_bound;
    sk_upper_bound = (int)rint(((double)SK_UPPER_BOUND) / resolution);
    /* h(r) boundaries */
    h_lower_bound = (int)rint(((double)H_LOWER_BOUND) / resolution);
    ++h_lower_bound;
    h_upper_bound = (int)rint(((double)H_UPPER_BOUND) / resolution);

    /* get the extrapolated h(r) */
    h = make_h(gor, num_gor, extrapolation_point, A, r0, r1);

    /* get the structure factor S(k) */
    for(i = sk_lower_bound; i < sk_upper_bound; i++) {

        k = ((double)i)*resolution;
        /* integrate h(r)*r*sin(kr) trapezoidally */
        integral = 0.0;
        for(j = h_lower_bound ; j < h_upper_bound; j++) {
            r = ((double)j)*resolution;
            integral += h[j]*r*sin(k*r);
        }
        /* finish the integral */
        integral -= 0.5*(h[h_lower_bound] + h[h_upper_bound - 1]);
        integral *= resolution;

        /* multiply coefficients */
        integral *= (4.0*M_PI*density / k);
        integral += 1.0;

        /* output the current S(k) value */
        printf("%f %f\n", k, integral);
    }

    free(h);
}

void usage(char *progname) {
    fprintf(stderr, "usage: %s <-f> [density] [xp] [A] [r0] [r1] [datafile]\n", progname);
    fprintf(stderr, "<-f> - turns on fitting mode, h(r) extrapolated is output\n");
    fprintf(stderr, "side-by-side with the g(r)-based h(r)\n");
    fprintf(stderr, "[density] - density in units of molecules/A^3\n");
    fprintf(stderr, "[datfile] - two column file containing the g(r) and domain\n");
    exit(1);
}

int main(int argc, char **argv) {
    int i, n, c, fitting = 0;
    FILE *fp;
    char *datfile;
    double density, xp, A, r0, r1;

```

```

struct gor_t *gor;
extern char *optarg;
extern int optind, getopt;

if(argc < 7) usage((char *)argv[0]);

/* get arguments */
c = getopt(argc, (char **)argv, "f");
if(c != -1) {
    switch (c) {
        case 'f':
            fitting = 1;
            break;
        case '?':
            usage((char *)argv[0]);
    }
}

/* get the density of particles for S(k) */
density = atof(argv[optind]);

/* get the parameters */
xp = atof(argv[++optind]);
A = atof(argv[++optind]);
r0 = atof(argv[++optind]);
r1 = atof(argv[++optind]);

/* get the filename containing the g(r) */
datfile = argv[++optind];
if(!datfile) {
    fprintf(stderr, "error: invalid datafile\n");
    usage((char *)argv[0]);
}
else
    fp = fopen(datfile, "r");

if(!fp) {
    fprintf(stderr, "error: could not open datafile %s\n", datfile);
    usage((char *)argv[0]);
}

/* load in the data */
gor = (struct gor_t *)calloc(MAXLINE, sizeof(struct gor_t));
if(!gor) {
    fprintf(stderr, "error: couldn't allocate working memory buffer\n");
    exit(1);
}

for(i = 0; n != EOF; i++) {
    n = fscanf(fp, "%lg %lg", &(gor + i)->domain, &(gor + i)->range);
}
fclose(fp);
-i;

if(!i) {
    fprintf(stderr, "error: datfile %s is empty\n", datfile);
    exit(1);
}

/* realloc to save memory */
gor = realloc(gor, sizeof(struct gor_t)*i);
if(!gor) {
    fprintf(stderr, "error: couldn't realloc working memory buffer\n");
    exit(1);
}

if(fitting)
    fit_gor(gor, i, xp, A, r0, r1); /* get the fitted h(r) */
else
    structure_factor(gor, i, density, xp, A, r0, r1); /* get the structure factor */

/* free our g(r) structure and exit */
free(gor);
exit(0);
}

```

Appendix J. rng.c

```

        else
            printf("0");
        in <= RHS_ONE;
    }

}

void print_binary_double(double in_double) {
    int i;
    long long in;
    long long out = 0;
    in = (long long)in_double;
    for(i = 0; i < 64; i++) {
        out = in & LHS_ONE;
        if(out & LHS_ONE)
            printf("1");
        else
            printf("0");
        in <= RHS_ONE;
    }
}

#endif WORDSIZE == 64 /* 64-bit */

double rule30_rng(unsigned long int seed) {
    register unsigned long int rule = RULE30; /* the rule to enforce */
    register unsigned long int in_reg1 = 0, /* input registers */
        in_reg2 = 0,
        in_reg3 = 0,
        in_reg4 = 0,
        in_reg5 = 0,
        in_reg6 = 0,
        in_reg7 = 0;
    register unsigned long int out_reg1 = 0, /* output registers */
        out_reg2 = 0,
        out_reg3 = 0,
        out_reg4 = 0,
        out_reg5 = 0,
        out_reg6 = 0,
        out_reg7 = 0;
    register unsigned long int mp = 0; /* multi-purpose register:
        /* - the right-most 8 bits are for the inner loop counter */
        /* - the next 16 bits are for the outer loop counter */
        /* - the left-most bit is for carries */
    static unsigned long int last_reg1, /* static memory addrs to store results from the current run */
        last_reg2,
        last_reg3,
        last_reg4,
        last_reg5,
        last_reg6,
        last_reg7;
    double random_result = 0; /* return a double from 0.0 to 1.0 */
    unsigned long int random_result_int = 0; /* integer version of the above for boolean ops */

    /* start with initial config */
    if(seed) {
        in_reg1 = in_reg2 = in_reg3 = in_reg4 = in_reg5 = in_reg6 = in_reg7 = seed;
    } else {
        #ifdef DEBUG
            /* set up the canonical state for debugging */
            in_reg4 = CENTER_MASK;
        #else
            /* already seeded - restore state from memory */
            in_reg1 = last_reg1;
            in_reg2 = last_reg2;
            in_reg3 = last_reg3;
            in_reg4 = last_reg4;
            in_reg5 = last_reg5;
            in_reg6 = last_reg6;
            in_reg7 = last_reg7;
        #endif /* DEBUG */
    }

    #ifdef DEBUG
        /* the current cellular automata rule being imposed */
        printf("current rule: %d\n", rule);
        /* print initial line */
        print_binary(in_reg4); printf("\n");
    #endif /* DEBUG */

    for((mp &= OUTER_ZERO); ((mp & OUTER_COUNT) >> DELTA_COUNT) < DELTA_MANTISSA; mp += OUTER_ONE) /*-- notice the fact that the
increment here */
        for((mp &= INNER_ZERO); (mp & INNER_COUNT) < WORDSIZE; mp += INNER_ONE) /* will blow away the low-order bits
doesn't matter */
            /* mask off first three bits and compare with rule */
            /* set the output register bit appropriately */
            out_reg1 |= ((rule >> (in_reg1 & CELL_MASK)) & RHS_ONE) << (mp & INNER_COUNT);
            out_reg2 |= ((rule >> (in_reg2 & CELL_MASK)) & RHS_ONE) << (mp & INNER_COUNT);
            out_reg3 |= ((rule >> (in_reg3 & CELL_MASK)) & RHS_ONE) << (mp & INNER_COUNT);
            out_reg4 |= ((rule >> (in_reg4 & CELL_MASK)) & RHS_ONE) << (mp & INNER_COUNT);
            out_reg5 |= ((rule >> (in_reg5 & CELL_MASK)) & RHS_ONE) << (mp & INNER_COUNT);
            out_reg6 |= ((rule >> (in_reg6 & CELL_MASK)) & RHS_ONE) << (mp & INNER_COUNT);
            out_reg7 |= ((rule >> (in_reg7 & CELL_MASK)) & RHS_ONE) << (mp & INNER_COUNT);
}

```

```

/* rotate all input registers one bit to the right, preserve carry */
mp &= LHS_ZERO; /* clear the carry bit */
mp |= ((in_reg7 & RHS_ONE) << (WORDSIZE - 1)); /* set carry bit if needed */
in_reg7 >>= RHS_ONE;
in_reg6 |= ((in_reg6 & RHS_ONE) << (WORDSIZE - 1));
in_reg5 |= ((in_reg5 & RHS_ONE) << (WORDSIZE - 1));
in_reg5 >>= RHS_ONE;
in_reg4 |= ((in_reg4 & RHS_ONE) << (WORDSIZE - 1));
in_reg4 >>= RHS_ONE;
in_reg3 |= ((in_reg3 & RHS_ONE) << (WORDSIZE - 1));
in_reg3 >>= RHS_ONE;
in_reg2 |= ((in_reg2 & RHS_ONE) << (WORDSIZE - 1));
in_reg2 >>= RHS_ONE;
in_reg1 |= ((in_reg1 & RHS_ONE) << (WORDSIZE - 1));
in_reg1 >>= RHS_ONE;
in_reg1 |= mp & LHS_ONE;

}

/* now must rotate output registers one bit to the left */
mp &= LHS_ZERO; /* clear the carry bit */
mp |= out_reg1 & LHS_ONE; /* set the carry bit if needed */
out_reg1 <<= RHS_ONE;
out_reg1 |= ((out_reg2 & LHS_ONE) >> (WORDSIZE - 1));
out_reg2 <<= RHS_ONE;
out_reg2 |= ((out_reg3 & LHS_ONE) >> (WORDSIZE - 1));
out_reg3 <<= RHS_ONE;
out_reg3 |= ((out_reg4 & LHS_ONE) >> (WORDSIZE - 1));
out_reg4 <<= RHS_ONE;
out_reg4 |= ((out_reg5 & LHS_ONE) >> (WORDSIZE - 1));
out_reg5 <<= RHS_ONE;
out_reg5 |= ((out_reg6 & LHS_ONE) >> (WORDSIZE - 1));
out_reg6 <<= RHS_ONE;
out_reg6 |= ((out_reg7 & LHS_ONE) >> (WORDSIZE - 1));
out_reg7 <<= RHS_ONE;
out_reg7 |= ((mp & LHS_ONE) >> (WORDSIZE - 1));

/* set output bits of random number */
(unsigned long int)random_result_int |= ((out_reg4 & CENTER_MASK) >> DELTA_CENTER) << ((DELTA_MANTISSA - 1) - ((mp & OUTER_COUNT) >> DELTA_COUNT));

#endif /* DEBUG */

/* give visual output */
print_binary(out_reg4); printf("\t%d\n", (mp & OUTER_COUNT) >> DELTA_COUNT);

#endif /* DEBUG */

/* swap the input and output registers */
in_reg1 = out_reg1;
in_reg2 = out_reg2;
in_reg3 = out_reg3;
in_reg4 = out_reg4;
in_reg5 = out_reg5;
in_reg6 = out_reg6;
in_reg7 = out_reg7;

/* clear output registers */
out_reg1 = out_reg2 = out_reg3 = out_reg4 = out_reg5 = out_reg6 = out_reg7 = 0;

}

/* save last state point to static memory */
last_reg1 = in_reg1;
last_reg2 = in_reg2;
last_reg3 = in_reg3;
last_reg4 = in_reg4;
last_reg5 = in_reg5;
last_reg6 = in_reg6;
last_reg7 = in_reg7;

random_result = (double)random_result_int;
random_result /= (double)MAX_MANTISSA; /* ensure that result is normalized from 0 to 1 */
return(random_result);

}

#else /* 32-bit */

double rule30_rng(unsigned long int seed) {

register unsigned long int rule = RULE30; /* the rule to enforce */
register unsigned long int in_reg1 = 0, /* input registers */
    in_reg2 = 0,
    in_reg3 = 0;
register unsigned long int out_reg1 = 0, /* output registers */
    out_reg2 = 0,
    out_reg3 = 0;
register unsigned long int mp = 0; /* multi-purpose register: */
/* - the right-most 8 bits are for the inner loop counter */
/* - the next 16 bits are for the outer loop counter */
/* - the left-most bit is for carries */
static unsigned long int last_reg1, /* static memory addrs to store results from the current run */
    last_reg2,
    last_reg3;

double random_result = 0; /* return a double from 0.0 to 1.0 */
unsigned long long int random_result_int = 0; /* integer version of the above for boolean ops */

/* start with initial config */
if(seed) {
    in_reg1 = in_reg2 = in_reg3 = seed;
}
else {
#ifndef DEBUG
    /* set up the canonical state for debugging */
    in_reg2 = CENTER_MASK;
#endif
}

```

```

/* already seeded - restore state from memory */
in_reg1 = last_reg1;
in_reg2 = last_reg2;
in_reg3 = last_reg3;
#endif /* DEBUG */
}

#ifndef DEBUG
/* the current cellular automata rule being imposed */
printf("current rule: %d\n", rule);

/* print initial line */
print_binary(in_reg1); print_binary(in_reg2); print_binary(in_reg3); printf("\n");
#endif /* DEBUG */

for((mp &= OUTER_ZERO); ((mp & OUTER_COUNT) >> DELTA_COUNT) < DELTA_MANTISSA; mp += OUTER_ONE) {/* -- notice the fact that the
increment here */
    for((mp &= INNER_ZERO); (mp & INNER_COUNT) < WORDSIZE; mp += INNER_ONE) { /* will blow away the low-order bits
doesn't matter */

        /* mask off first three bits and compare with rule */
        /* set the output register bit appropriately */
        out_reg1 |= ((rule >> (in_reg1 & CELL_MASK)) & RHS_ONE) << (mp & INNER_COUNT);
        out_reg2 |= ((rule >> (in_reg2 & CELL_MASK)) & RHS_ONE) << (mp & INNER_COUNT);
        out_reg3 |= ((rule >> (in_reg3 & CELL_MASK)) & RHS_ONE) << (mp & INNER_COUNT);

        /* rotate all input registers one bit to the right, preserve carry */
        mp &= LHS_ZERO; /* clear the carry bit */
        mp |= ((in_reg3 & RHS_ONE) << (WORDSIZE - 1)); /* set carry bit if needed */
        in_reg3 >>= RHS_ONE;
        in_reg3 |= ((in_reg2 & RHS_ONE) << (WORDSIZE - 1));
        in_reg2 >>= RHS_ONE;
        in_reg2 |= ((in_reg1 & RHS_ONE) << (WORDSIZE - 1));
        in_reg1 >>= RHS_ONE;
        in_reg1 |= mp & LHS_ONE;

    }

    /* now must rotate output registers one bit to the left */
    mp &= LHS_ZERO; /* clear the carry bit */
    mp |= out_reg1 & RHS_ONE; /* set the carry bit if needed */
    out_reg1 |= ((out_reg2 & RHS_ONE) >> (WORDSIZE - 1));
    out_reg2 <<= RHS_ONE;
    out_reg2 |= ((out_reg3 & RHS_ONE) >> (WORDSIZE - 1));
    out_reg3 <<= RHS_ONE;
    out_reg3 |= ((mp & RHS_ONE) >> (WORDSIZE - 1));

    /* set output bits of random sequence */
    random_result_int |= ((out_reg2 & CENTER_MASK) >> DELTA_CENTER) << ((DELTA_MANTISSA - 1) - ((mp & OUTER_COUNT) >>
DELTA_COUNT));

#endif /* DEBUG */

/* give visual output */
print_binary(out_reg1); print_binary(out_reg2); print_binary(out_reg3); printf("\t%d\n", (mp & OUTER_COUNT) >> DELTA_COUNT);
#endif /* DEBUG */

/* swap the input and output registers */
in_reg1 = out_reg1;
in_reg2 = out_reg2;
in_reg3 = out_reg3;

/* clear the output registers */
out_reg1 = out_reg2 = out_reg3 = 0;

}
/* save the last state point in static memory */
last_reg1 = in_reg1;
last_reg2 = in_reg2;
last_reg3 = in_reg3;

random_result = (double)random_result_int;
random_result /= (double)MAX_MANTISSA; /* ensure that result is normalized from 0 to 1 */
return(random_result);

}
#endif /* WORDSIZE == 64 */

int main() {

    int i;
    unsigned long int seed = 1234523;
    double rand;
    clock_t initial_time, final_time;

    rule30_rng(seed);
#ifndef BENCHMARK
    while(1 {

        initial_time = final_time = clock();
        for(i = 0; (final_time - initial_time) / CLOCKS_PER_SEC < 1.0; i++) {
            final_time = clock();
            rand = rule30_rng(0);
        }
        fprintf(stderr, "# performance: %d doubles/sec\n", i);
    }
#else
    for(i = 0; i < 1000000; i++) {
        rand = rule30_rng(0);
        printf("%.16f\n", rand);
    }
#endif /* BENCHMARK */
    exit(0); /* NOT REACHED */
}

```


Appendix K. Q_{st} R Code

```

*** import Long's Torr (p/p0) vs. mmol/g isotherm ***

iso77 <- read.table("S8.77K.torr_vs_mmolg.dat")
iso87 <- read.table("S8.87K.torr_vs_mmoig.dat")

*** import Long's normalized units ***
iso77 <- read.table("S8.77K.pp0_vs_n0.dat")
iso87 <- read.table("S8.87K.pp0_vs_n0.dat")

*** import Long's percent wt isotherms ***
iso77 <- read.table("S8.77K.pp0_vs_percwt.dat")
iso87 <- read.table("S8.87K.pp0_vs_percwt.dat")

*** import ME080 normalized units of the same isotherms ***
iso77 <- read.table("ME080.77K.pp0_vs_n0.dat")
iso87 <- read.table("ME080.87K.pp0_vs_n0.dat")

*** import ME080 percent wt units ***
iso77 <- read.table("ME080.77K.pp0_vs_percwt.dat")
iso87 <- read.table("ME080.87K.pp0_vs_percwt.dat")

*** fit isotherm ***
x77 <- iso77$V1
y77 <- iso77$V2
x87 <- iso87$V1
y87 <- iso87$V2
isofn <- function(p) sum( (( log(x77) - (log(y77) + (p[1]*y77^0 + p[2]*y77^1 + p[3]*y77^2 + p[4]*y77^3 + p[5]*y77^4)/77.0 + p[6]*y77^0 + p[7]*y77^1 + p[8]*y77^2 )^2) + (( log(x87) - (log(y87) + (p[1]*y87^0 + p[2]*y87^1 + p[3]*y87^2 + p[4]*y87^3 + p[5]*y87^4)/87.0 + p[6]*y87^0 + p[7]*y87^1 + p[8]*y87^2 )^2) )
fitparams <- nlmnb(c(1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0), isofn)
fitiso77 <- function(N) exp( log(N) + (fitparams$par[1]*N^0 + fitparams$par[2]*N^1 + fitparams$par[3]*N^2 + fitparams$par[4]*N^3 + fitparams$par[5]*N^4)/77.0 + fitparams$par[6]*N^0 + fitparams$par[7]*N^1 + fitparams$par[8]*N^2 )
fitiso87 <- function(N) exp( log(N) + (fitparams$par[1]*N^0 + fitparams$par[2]*N^1 + fitparams$par[3]*N^2 + fitparams$par[4]*N^3 + fitparams$par[5]*N^4)/87.0 + fitparams$par[6]*N^0 + fitparams$par[7]*N^1 + fitparams$par[8]*N^2 )
#independent <- seq(0.0, 110.0, 0.01)
#independent <- seq(0.0, 3.0, 0.001)
independent <- seq(0.0, 3.0, 0.001)

plot(x77, y77)
points(x87, y87)
plot(fitiso77(independent), independent)
points(fitiso87(independent), independent)

*** output isotherms to a file ***
output77 <- data.frame(fitiso77(independent), independent)
output87 <- data.frame(fitiso87(independent), independent)
write.table(output77, "fit_isotherm77.dat", row.names=FALSE, col.names=FALSE)
write.table(output87, "fit_isotherm87.dat", row.names=FALSE, col.names=FALSE)

*** generate Long qst ***
qst <- function(N) -0.008314472*(fitparams$par[1]*N^0 + fitparams$par[2]*N^1 + fitparams$par[3]*N^2 + fitparams$par[4]*N^3 + fitparams$par[5]*N^4)
independent <- seq(0.0, 3.0, 0.01)
plot(independent, qst(independent))

output to file:
output <- data.frame(independent, qst(independent))
write.table(output, "fit_qst.dat", row.names=FALSE, col.names=FALSE)

*** finite difference qst ***
lniso77 <- function(N) (log(N) + (fitparams$par[1]*N^0 + fitparams$par[2]*N^1 + fitparams$par[3]*N^2 + fitparams$par[4]*N^3 + fitparams$par[5]*N^4)/77.0 + fitparams$par[6]*N^0 + fitparams$par[7]*N^1 + fitparams$par[8]*N^2)
lniso87 <- function(N) (log(N) + (fitparams$par[1]*N^0 + fitparams$par[2]*N^1 + fitparams$par[3]*N^2 + fitparams$par[4]*N^3 + fitparams$par[5]*N^4)/87.0 + fitparams$par[6]*N^0 + fitparams$par[7]*N^1 + fitparams$par[8]*N^2)
qst <- function(T1, T2, N) -0.008314472*(lniso77(N) - lniso87(N))/(1.0/T1 - 1.0/T2)
#independent <- seq(0.0, 100.0, 0.01)
independent <- seq(0.0, 3.0, 0.01)
plot(independent, qst(77.0, 87.0, independent))

*** output qst to a file ***
output <- data.frame(independent, qst(77.0, 87.0, independent))
write.table(output, "fit_qst.dat", row.names=FALSE, col.names=FALSE)

***** XXX EXPERIMENTAL fd qst with choice of temperature *****

```

```

lniso77 <- function(N) (log(N) + (fitparams$par[1]*N^0 + fitparams$par[2]*N^1
+ fitparams$par[3]*N^2 + fitparams$par[4]*N^3 + fitparams$par[5]*N^4)/77.0 +
fitparams$par[6]*N^0 + fitparams$par[7]*N^1 + fitparams$par[8]*N^2)
lniso87 <- function(N) (log(N) + (fitparams$par[1]*N^0 + fitparams$par[2]*N^1
+ fitparams$par[3]*N^2 + fitparams$par[4]*N^3 + fitparams$par[5]*N^4)/87.0 +
fitparams$par[6]*N^0 + fitparams$par[7]*N^1 + fitparams$par[8]*N^2)
qst <- function(T1, T2, N) 0.008314472*82.0*82.0*(lniso77(N) - lniso87(N))/(T1 - T2)
#independent <- seq(0.0, 100.0, 0.01)
independent <- seq(0.0, 3.0, 0.01)
plot(independent, qst(77.0, 87.0, independent))

***** XXX EXPERIMENTAL virial qst with varying coefficients *****
# ADD ANOTHER 'A'
# *** fit isotherm ***

iso77 <- read.table("S8.77K.pp0_vs_percwt.dat")
iso87 <- read.table("S8.87K.pp0_vs_percwt.dat")

x77 <- iso77$V1
y77 <- iso77$V2
x87 <- iso87$V1
y87 <- iso87$V2
isofn <- function(p) sum( (( log(x77) - (log(y77) + (p[1]*y77^0 + p[2]*y77^1 + p[3]*y77^2 + p[4]*y77^3 + p[5]*y77^4 + p[6]*y77^5)/77.0 +
p[7]*y77^0 + p[8]*y77^1 + p[9]*y77^2 )^2) + (( log(x87) - (log(y87) + (p[1]*y87^0 + p[2]*y87^1 + p[3]*y87^2 + p[4]*y87^3 + p[5]*y87^4 +
p[6]*y87^5)/87.0 + p[7]*y87^0 + p[8]*y87^1 + p[9]*y87^2 )^2) )
fitparams <- nlmnb(c(1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0), isofn)

fitiso77 <- function(N) exp( log(N) + (fitparams$par[1]*N^0 + fitparams$par[2]*N^1 + fitparams$par[3]*N^2 + fitparams$par[4]*N^3 +
fitparams$par[5]*N^4 + fitparams$par[6]*N^5)/77.0 + fitparams$par[7]*N^0 + fitparams$par[8]*N^1 + fitparams$par[9]*N^2 )
fitiso87 <- function(N) exp( log(N) + (fitparams$par[1]*N^0 + fitparams$par[2]*N^1 + fitparams$par[3]*N^2 + fitparams$par[4]*N^3 +
fitparams$par[5]*N^4 + fitparams$par[6]*N^5)/87.0 + fitparams$par[7]*N^0 + fitparams$par[8]*N^1 + fitparams$par[9]*N^2 )
#independent <- seq(0.0, 110.0, 0.01)
independent <- seq(0.0, 3.0, 0.001)

plot(x77, y77)
points(x87, y87)
plot(fitiso77(independent), independent)
points(fitiso87(independent), independent)

# *** generate Long qst ***
qst <- function(N) -0.008314472*(fitparams$par[1]*N^0 + fitparams$par[2]*N^1 + fitparams$par[3]*N^2 + fitparams$par[4]*N^3 +
fitparams$par[5]*N^4 + fitparams$par[6]*N^5)
independent <- seq(0.0, 3.0, 0.01)
plot(independent, qst(independent))

# *** output to file ***
output <- data.frame(independent, qst(independent))
write.table(output, "fit_qst.dat", row.names=FALSE, col.names=FALSE)

# REMOVE AN 'A'
# *** fit isotherm ***

iso77 <- read.table("S8.77K.pp0_vs_percwt.dat")
iso87 <- read.table("S8.87K.pp0_vs_percwt.dat")

x77 <- iso77$V1
y77 <- iso77$V2
x87 <- iso87$V1
y87 <- iso87$V2
isofn <- function(p) sum( (( log(x77) - (log(y77) + (p[1]*y77^0 + p[2]*y77^1 + p[3]*y77^2 )/77.0 + p[4]*y77^3 + p[5]*y77^4 + p[6]*y77^5 )^2) +
(( log(x87) - (log(y87) + (p[1]*y87^0 + p[2]*y87^1 + p[3]*y87^2 )/87.0 + p[4]*y87^3 + p[5]*y87^4 + p[6]*y87^5 )^2) )
fitparams <- nlmnb(c(1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0), isofn)

fitiso77 <- function(N) exp( log(N) + (fitparams$par[1]*N^0 + fitparams$par[2]*N^1 + fitparams$par[3]*N^2 )/77.0 + fitparams$par[4]*N^3 +
fitparams$par[5]*N^4 + fitparams$par[6]*N^5 )
fitiso87 <- function(N) exp( log(N) + (fitparams$par[1]*N^0 + fitparams$par[2]*N^1 + fitparams$par[3]*N^2 )/87.0 + fitparams$par[4]*N^3 +
fitparams$par[5]*N^4 + fitparams$par[6]*N^5 )
#independent <- seq(0.0, 110.0, 0.01)
independent <- seq(0.0, 3.0, 0.001)

plot(x77, y77)
points(x87, y87)
plot(fitiso77(independent), independent)
points(fitiso87(independent), independent)

# *** generate Long qst ***
qst <- function(N) -0.008314472*(fitparams$par[1]*N^0 + fitparams$par[2]*N^1 + fitparams$par[3]*N^2 )
independent <- seq(0.0, 3.0, 0.01)
plot(independent, qst(independent))

# *** output to file ***
output <- data.frame(independent, qst(independent))
write.table(output, "fit_qst.dat", row.names=FALSE, col.names=FALSE)

```

Appendix L. MPMC

```

#!/bin/sh

if [ $# != 5 ]
then
    echo "usage: $0 <hostname> <compiler> <serial | parallel> <CUDA: yes|no> <QM Rotation: yes|no>"
    echo "      hostnames: bigben circe galactic dirac lonestar ranger spin time generic"
    echo "      compilers: gcc pgi icc"
    exit 1
fi

HOSTNAME=$1
COMPILER=$2
MODE=$3
POLARIZATION_CUDA=$4
QM_ROTATION=$5

if [ $QM_ROTATION == "yes" ]
then
    DEFINES="-DQM_ROTATION"
fi

case "$HOSTNAME" in
    "bigben")
        case "$COMPILER" in
            "gcc")
                echo "error: compiler $COMPILER not supported for hostname $HOSTNAME"
                exit 1
            ;;
            "pgi")
                CC="cc"
                CFLAGS="-O3 -fastsse -Mnontemporal -Mprefetch=distance:8,nta"
            ;;
            "icc")
                echo "error: compiler $COMPILER not supported for hostname $HOSTNAME"
                exit 1
            ;;
            *)
                echo "error: unknown compiler $COMPILER"
                exit 1
            ;;
        esac
    ;;
    "circe")
        case "$COMPILER" in
            "gcc")
                echo "error: compiler $COMPILER not supported for hostname $HOSTNAME"
                exit 1
            ;;
            "pgi")
                CC="pgcc"
                CFLAGS="-O3 -fastsse -Mipa=fast"
                EXTRA_LIBS="-L/usr/local/pgi/7.2-5/linux86-64/7.2-5/libso -lacml -lacml_mv -lacml_mp -lpgmp -lpgftnrtl"
            ;;
            "icc")
                echo "error: compiler $COMPILER not supported for hostname $HOSTNAME"
                exit 1
            ;;
            *)
                echo "error: unknown compiler $COMPILER"
                exit 1
            ;;
        esac
    ;;
    "dirac")
        case "$COMPILER" in
            "gcc")
                CC="gcc"
                CFLAGS="-O3 -std=c99"
                EXTRA_LIBS="-L/usr/local/pgi/linux86-64/7.0-6/libso/ -lacml -lacml_mv -lacml_mp -lpgmp -lpgftnrtl -lrt"
            ;;
            "pgi")
                CC="pgcc"
                CFLAGS="-O3 -fastsse -Mipa=fast"
                EXTRA_LIBS="-L/usr/local/pgi/linux86-64/7.0-6/libso/ -lacml -lacml_mv -lacml_mp -lpgmp -lpgftnrtl -lrt"
            ;;
            "icc")
                echo "error: compiler $COMPILER not supported for hostname $HOSTNAME"
                exit 1
            ;;
            *)
                echo "error: unknown compiler $COMPILER"
                exit 1
            ;;
        esac
    ;;
    "time")
        case "$COMPILER" in
            "gcc")
                CC="gcc"
                CFLAGS="-O3"
                EXTRA_LIBS="-lm"
            ;;
            "pgi")
                CC="pgcc"
                CFLAGS="-O3 -fastsse -Mipa=fast"
                EXTRA_LIBS="-L/usr/local/pgi/linux86-64/7.0-6/libso/ -lacml -lacml_mv -lacml_mp -lpgmp -lpgftnrtl -lrt"
            ;;
            "icc")
                echo "error: compiler $COMPILER not supported for hostname $HOSTNAME"
                exit 1
            ;;
            *)
                echo "error: unknown compiler $COMPILER"
                exit 1
            ;;
        esac
    ;;
esac

```

```

;;
"galactic")
case "$COMPILER" in
  "gcc")
    echo "error: compiler $COMPILER not supported for hostname $HOSTNAME"
    exit 1
  ;;
  "pgi")
    echo "error: compiler $COMPILER not supported for hostname $HOSTNAME"
    exit 1
  ;;
  "icc")
    CC="icc"
    CFLAGS="-O3 -prefetch -mp -axP -ip -unroll"
  ;;
  *)
    echo "error: unknown compiler $COMPILER"
    exit 1
esac
;;
"lonestar")
case "$COMPILER" in
  "gcc")
    echo "error: compiler $COMPILER not supported for hostname $HOSTNAME"
    exit 1
  ;;
  "pgi")
    echo "error: compiler $COMPILER not supported for hostname $HOSTNAME"
    exit 1
  ;;
  "icc")
    CC="icc"
    CFLAGS="-O3 -prefetch -mp -ip -unroll"
  ;;
  *)
    echo "error: unknown compiler $COMPILER"
    exit 1
esac
;;
"ranger")
case "$COMPILER" in
  "gcc")
    echo "error: compiler $COMPILER not supported for hostname $HOSTNAME"
    exit 1
  ;;
  "pgi")
    CC="pgcc"
    CFLAGS="-O3 -fastsse -tp barcelona-64"
  ;;
  "icc")
    echo "error: compiler $COMPILER not supported for hostname $HOSTNAME"
    exit 1
  ;;
  *)
    echo "error: unknown compiler $COMPILER"
    exit 1
esac
;;
"spin")
case "$COMPILER" in
  "gcc")
    CC="gcc"
    CFLAGS="-g3 -O3"
  ;;
  "pgi")
    CC="pgcc"
    CFLAGS="-g -O3 -fastsse -Mipa=fast"
  ;;
  "icc")
    CC="icc"
    CFLAGS="-O3 -prefetch -mp -axN -ip -unroll"
  ;;
  *)
    echo "error: unknown compiler $COMPILER"
    exit 1
esac
;;
"generic")
case "$COMPILER" in
  "gcc")
    CC="gcc"
    CFLAGS="-O3"
    EXTRA_LIBS="-lm"
  ;;
  "pgi")
    CC="pgcc"
    CFLAGS="-O3"
    EXTRA_LIBS="-lm"
  ;;
  "icc")
    CC="icc"
    CFLAGS="-O3"
    EXTRA_LIBS="-lm"
  ;;
  *)
    echo "error: unknown compiler $COMPILER"
    exit 1
esac
;;
*)
  echo "error: unknown hostname $HOSTNAME"
  exit 1
;;
esac
case "$MODE" in
  "serial")

```

```

    DEFINES="$DEFINES"
;;
"parallel")
    DEFINES="$DEFINES -DMPI"
# handle special cases for parallelization
case "$HOSTNAME" in
    "bigben")
        CC="cc"
;;
"dirac")
    EXTRA_INCLUDES="-I/usr/include/openmpi/1.2.3-gcc"
    EXTRA_LIBS="-L/usr/lib64/openmpi/1.2.3-gcc -lmpi"
;;
"time")
    EXTRA_INCLUDES="-I/usr/include/openmpi/1.2.3-gcc"
    EXTRA_LIBS="-L/usr/lib64/openmpi/1.2.3-gcc -lmpi"
;;
"spin")
    EXTRA_INCLUDES="-I/usr/include/openmpi"
    EXTRA_LIBS="-L/usr/lib64/openmpi -lmpi"
;;
*)
    CC="mpicc"
;;
esac
;;
*)
    echo "error: unknown mode, must be serial or parallel"
    exit 1
;;
esac

echo > Makefile
echo "#HOSTNAME=$HOSTNAME COMPILER=$COMPILER MODE=$MODE POLARIZATION_CUDA=$POLARIZATION_CUDA QM_ROTATION=$QM_ROTATION" >> Makefile
echo "CC=$CC" >> Makefile
echo "CFLAGS=$CFLAGS" >> Makefile
echo "DEFINES=$DEFINES" >> Makefile
echo "EXTRA_DEFINES=$EXTRA_DEFINES" >> Makefile
echo "EXTRA_INCLUDES=$EXTRA_INCLUDES" >> Makefile
echo "EXTRA_LIBS=$EXTRA_LIBS" >> Makefile
echo "POLARIZATION_CUDA=$POLARIZATION_CUDA" >> Makefile
echo "QM_ROTATION=$QM_ROTATION" >> Makefile
echo >> Makefile

echo "include Makefile.common" >> Makefile
echo >> Makefile

#####
# !!! common Makefile - DO NOT EDIT !!! #
#####

BUILD=.
TOP=$(BUILD)/..
HEADERS=$(TOP)/include/function_prototypes.h $(TOP)/include/mc.h $(TOP)/include/physical_constants.h $(TOP)/include/structs.h
INCLUDES=-I$(TOP)/include

ENERGY=$(TOP)/energy
IO=$(TOP)/io
MAIN=$(TOP)/main
MC=$(TOP)/mc
ifeq ($(POLARIZATION_CUDA),yes)
    POLARIZATION=$(TOP)/polarization_cuda
else
    POLARIZATION=$(TOP)/polarization
endif
HISTOGRAM=$(TOP)/histogram
QUANTUM_ROTATION=$(TOP)/quantum_rotation

ifeq ($(QM_ROTATION),yes)
all: energy.o pairs.o pbc.o lj.o sg.o coulombic.o polar.o bond.o bessel.o \
    input.o output.o average.o mpi.o dxwrite.o \
    main.o cleanup.o rng.o \
    mc.o mc_moves.o pimc.o surface.o cavity.o \
    thole_iterative.o thole_matrix.o thole_field.o thole_polarizability.o \
    histogram.o \
    rotational_eigenspectrum.o rotational_basis.o rotational_potential.o rotational_integrate.o \
    $(CC) $(CFLAGS) $(DEFINES) $(EXTRA_DEFINES) $(LIBS) $(EXTRA_LIBS) *.o -o mpmpc
else
all: energy.o pairs.o pbc.o lj.o sg.o coulombic.o polar.o bond.o bessel.o \
    input.o output.o average.o mpi.o dxwrite.o \
    main.o cleanup.o rng.o \
    mc.o mc_moves.o pimc.o surface.o cavity.o \
    thole_iterative.o thole_matrix.o thole_field.o thole_polarizability.o \
    histogram.o \
    $(CC) $(CFLAGS) $(DEFINES) $(EXTRA_DEFINES) $(LIBS) $(EXTRA_LIBS) *.o -o mpmpc
endif

clean:
rm -f *.o *.oo *.ipa mpmpc

# energy routines
energy: energy.o pairs.o pbc.o lj.o sg.o coulombic.o polar.o bond.o bessel.o
energy.o: $(ENERGY)/energy.c $(HEADERS)
    $(CC) $(CFLAGS) $(DEFINES) $(EXTRA_DEFINES) $(INCLUDES) $(EXTRA_INCLUDES) -c $(ENERGY)/energy.c
pairs.o: $(ENERGY)/pairs.c $(HEADERS)
    $(CC) $(CFLAGS) $(DEFINES) $(EXTRA_DEFINES) $(INCLUDES) $(EXTRA_INCLUDES) -c $(ENERGY)/pairs.c
pbc.o: $(ENERGY)/pbc.c $(HEADERS)
    $(CC) $(CFLAGS) $(DEFINES) $(EXTRA_DEFINES) $(INCLUDES) $(EXTRA_INCLUDES) -c $(ENERGY)/pbc.c
lj.o: $(ENERGY)/lj.c $(HEADERS)
    $(CC) $(CFLAGS) $(DEFINES) $(EXTRA_DEFINES) $(INCLUDES) $(EXTRA_INCLUDES) -c $(ENERGY)/lj.c
sg.o: $(ENERGY)/sg.c $(HEADERS)
    $(CC) $(CFLAGS) $(DEFINES) $(EXTRA_DEFINES) $(INCLUDES) $(EXTRA_INCLUDES) -c $(ENERGY)/sg.c
coulombic.o: $(ENERGY)/coulombic.c $(HEADERS)
    $(CC) $(CFLAGS) $(DEFINES) $(EXTRA_DEFINES) $(INCLUDES) $(EXTRA_INCLUDES) -c $(ENERGY)/coulombic.c
polar.o: $(ENERGY)/polar.c
    $(CC) $(CFLAGS) $(DEFINES) $(EXTRA_DEFINES) $(INCLUDES) $(EXTRA_INCLUDES) -c $(ENERGY)/polar.c

```

```

bond.o: $(ENERGY)/bond.c
$(CC) $(CFLAGS) $(DEFINES) $(EXTRA_DEFINES) $(INCLUDES) $(EXTRA_INCLUDES) -c $(ENERGY)/bond.c
bessel.o: $(ENERGY)/bessel.c
$(CC) $(CFLAGS) $(DEFINES) $(EXTRA_DEFINES) $(INCLUDES) $(EXTRA_INCLUDES) -c $(ENERGY)/bessel.c

# input/output functions
io: input.o output.o average.o mpi.o dxwrite.o
input.o: $(IO)/input.c $(HEADERS)
$(CC) $(CFLAGS) $(DEFINES) $(EXTRA_DEFINES) $(INCLUDES) $(EXTRA_INCLUDES) -c $(IO)/input.c
output.o: $(IO)/output.c $(HEADERS)
$(CC) $(CFLAGS) $(DEFINES) $(EXTRA_DEFINES) $(INCLUDES) $(EXTRA_INCLUDES) -c $(IO)/output.c
average.o: $(IO)/average.c $(HEADERS)
$(CC) $(CFLAGS) $(DEFINES) $(EXTRA_DEFINES) $(INCLUDES) $(EXTRA_INCLUDES) -c $(IO)/average.c
mpi.o: $(IO)/mpi.c $(HEADERS)
$(CC) $(CFLAGS) $(DEFINES) $(EXTRA_DEFINES) $(INCLUDES) $(EXTRA_INCLUDES) -c $(IO)/mpi.c
dxwrite.o: $(IO)/dxwrite.c $(HEADERS)
$(CC) $(CFLAGS) $(DEFINES) $(EXTRA_DEFINES) $(INCLUDES) $(EXTRA_INCLUDES) -c $(IO)/dxwrite.c

# main associated things
main: main.o cleanup.o rng.o
main.o: $(MAIN)/main.c $(HEADERS)
$(CC) $(CFLAGS) $(DEFINES) $(EXTRA_DEFINES) $(INCLUDES) $(EXTRA_INCLUDES) -c $(MAIN)/main.c
cleanup.o: $(MAIN)/cleanup.c $(HEADERS)
$(CC) $(CFLAGS) $(DEFINES) $(EXTRA_DEFINES) $(INCLUDES) $(EXTRA_INCLUDES) -c $(MAIN)/cleanup.c
rng.o: $(MAIN)/rng.c $(HEADERS)
$(CC) $(CFLAGS) $(DEFINES) $(EXTRA_DEFINES) $(INCLUDES) $(EXTRA_INCLUDES) -c $(MAIN)/rng.c

# monte carlo subtree
mc: mc.o mc_moves.o pimc.o surface.o cavity.o
mc.o: $(MC)/mc.c $(HEADERS)
$(CC) $(CFLAGS) $(DEFINES) $(EXTRA_DEFINES) $(INCLUDES) $(EXTRA_INCLUDES) -c $(MC)/mc.c
mc_moves.o: $(MC)/mc_moves.c $(HEADERS)
$(CC) $(CFLAGS) $(DEFINES) $(EXTRA_DEFINES) $(INCLUDES) $(EXTRA_INCLUDES) -c $(MC)/mc_moves.c
pimc.o: $(MC)/pimc.c $(HEADERS)
$(CC) $(CFLAGS) $(DEFINES) $(EXTRA_DEFINES) $(INCLUDES) $(EXTRA_INCLUDES) -c $(MC)/pimc.c
surface.o: $(MC)/surface.c $(HEADERS)
$(CC) $(CFLAGS) $(DEFINES) $(EXTRA_DEFINES) $(INCLUDES) $(EXTRA_INCLUDES) -c $(MC)/surface.c
cavity.o: $(MC)/cavity.c $(HEADERS)
$(CC) $(CFLAGS) $(DEFINES) $(EXTRA_DEFINES) $(INCLUDES) $(EXTRA_INCLUDES) -c $(MC)/cavity.c

# many-body polarization
polarization: thole_iterative.o thole_matrix.o thole_field.o thole_polarizability.o
thole_iterative.o: $(POLARIZATION)/thole_iterative.c $(HEADERS)
$(CC) $(CFLAGS) $(DEFINES) $(EXTRA_DEFINES) $(INCLUDES) $(EXTRA_INCLUDES) -c $(POLARIZATION)/thole_iterative.c
thole_matrix.o: $(POLARIZATION)/thole_matrix.c $(HEADERS)
$(CC) $(CFLAGS) $(DEFINES) $(EXTRA_DEFINES) $(INCLUDES) $(EXTRA_INCLUDES) -c $(POLARIZATION)/thole_matrix.c
thole_field.o: $(POLARIZATION)/thole_field.c $(HEADERS)
$(CC) $(CFLAGS) $(DEFINES) $(EXTRA_DEFINES) $(INCLUDES) $(EXTRA_INCLUDES) -c $(POLARIZATION)/thole_field.c
thole_polarizability.o: $(POLARIZATION)/thole_polarizability.c $(HEADERS)
$(CC) $(CFLAGS) $(DEFINES) $(EXTRA_DEFINES) $(INCLUDES) $(EXTRA_INCLUDES) -c $(POLARIZATION)/thole_polarizability.c

# 3D histograms
histogram: histogram.o
histogram.o: $(HISTOGRAM)/histogram.c $(HEADERS)
$(CC) $(CFLAGS) $(DEFINES) $(EXTRA_DEFINES) $(INCLUDES) $(EXTRA_INCLUDES) -c $(HISTOGRAM)/histogram.c

ifeq ($(QM_ROTATION),yes)
# quantum rotational diagonalization
quantum_rotation: rotational_eigenspectrum.o rotational_basis.o rotational_potential.o rotational_integrate.o
rotational_eigenspectrum.o: $(QUANTUM_ROTATION)/rotational_eigenspectrum.c $(HEADERS)
$(CC) $(CFLAGS) $(DEFINES) $(EXTRA_DEFINES) $(INCLUDES) $(EXTRA_INCLUDES) -c $(QUANTUM_ROTATION)/rotational_eigenspectrum.c
rotational_basis.o: $(QUANTUM_ROTATION)/rotational_basis.c $(HEADERS)
$(CC) $(CFLAGS) $(DEFINES) $(EXTRA_DEFINES) $(INCLUDES) $(EXTRA_INCLUDES) -c $(QUANTUM_ROTATION)/rotational_basis.c
rotational_potential.o: $(QUANTUM_ROTATION)/rotational_potential.c $(HEADERS)
$(CC) $(CFLAGS) $(DEFINES) $(EXTRA_DEFINES) $(INCLUDES) $(EXTRA_INCLUDES) -c $(QUANTUM_ROTATION)/rotational_potential.c
rotational_integrate.o: $(QUANTUM_ROTATION)/rotational_integrate.c $(HEADERS)
$(CC) $(CFLAGS) $(DEFINES) $(EXTRA_DEFINES) $(INCLUDES) $(EXTRA_INCLUDES) -c $(QUANTUM_ROTATION)/rotational_integrate.c
endif

#include <mc.h>

/* returns the modified Bessel K function of fractional order */
/* wrapper for the NR routine */
double besselK(double order, double x) {
    float function_value_i, function_value_k;
    float rip, rpk;
    bessik((float)x, (float)order, &function_value_i, &function_value_k, &rip, &rpk);
    return((double)function_value_k);
}

float chebev(float a, float b, float c[], int m, float x)
{
    void nrerror(char error_text[]);
    float d=0.0,dd=0.0,sv,y,y2;
    int j;
    if ((x-a)*(x-b) > 0.0) return(-1);
    y2=2.0*(y=(2.0*x-a-b)/(b-a));
    for (j=m-1;j>1;j--) {
        sv=d;
        d=y2*d-dd+c[j];
        dd=sv;
    }
    return y*d-dd+0.5*c[0];
}

#define NUSE1 5
#define NUSE2 5

```

```

void beschb(double x, double *gam1, double *gam2, double *gampl, double *gammi)
{
    float chebev(float a, float b, float c[], int m, float x);
    float xx;
    static float c1[] = {
        -1.42022680371172e-0, 6.516511267076e-3,
        3.08709017308e-4, -3.470626964e-6, 6.943764e-9,
        3.6780e-11, -1.36e-13};
    static float c2[] = {
        1.843740587300906e0, -0.076852840844786e0,
        1.271927136655e-3, -4.971736704e-6, -3.3126120e-8,
        2.42310e-10, -1.70e-13, -1.0e-15};

    xx=8.0*x*x-1.0;
    *gam1=chebev(-1.0,1.0,c1,NUSE1,xx);
    *gam2=chebev(-1.0,1.0,c2,NUSE2,xx);
    *gampl= *gam2+**(*gam1);
    *gammi= *gam2+**(*gam1);
}

#define NUSE1
#define NUSE2
#include <math.h>
#define EPS 1.0e-10
#define FPMIN 1.0e-30
#define MAXIT 10000
#define XMIN 2.0
#define PI 3.141592653589793

/* returns error status */
int bessik(float x, float xnu, float *ri, float *rk, float *rip, float *rkp)
{
    int i,l,nl;
    double a,b,c,d,del,del1,delh,dels,e,f,fact,fact2,ff,gam1,gam2,
          gammi,gampl,h,p,pimu,q,q1,q2,qnew,ril,rili,rimu,ripl,ripl1,
          ritemp,rk1,rkmu,rkmp,rktemp,s,sum,sum1,x2,xi,xi2,xmu,xmu2;

    if (x <= 0.0 || xnu < 0.0) return(-1);
    nl=(int)(xnu+0.5);
    xmu=xnu-nl;
    xmu2=xmu*xmu;
    xi=1.0/x;
    xi2=2.0*xi;
    h=xmu*xi;
    if (h < FPMIN) h=FPMIN;
    b=xi2*xmu;
    d=0.0;
    c=0.0;
    for (i=1;i<=MAXIT;i++) {
        b += xi2;
        d=1.0/(b+d);
        c+=1.0/c;
        del=c*d;
        h=del*h;
        if (fabs(del-1.0) < EPS) break;
    }
    if (i > MAXIT) return(-1);
    ril=FPMIN;
    rip1=ril;
    rili=ril;
    rip1=rip1;
    fact=xmu*xi;
    for (l=nl;l>=1;l--) {
        ritemp=fact*ril+rip1;
        fact -= xi;
        rip1=fact*ritemp+ril;
        ril=ritemp;
    }
    f=rip1/ril;
    if (x < XMIN) {
        x2=0.5*x;
        pimu=PI*xmu;
        fact = (fabs(pimu) < EPS ? 1.0 : pimu/sin(pimu));
        d = -log(x2);
        e=xmu*d;
        fact2 = (fabs(e) < EPS ? 1.0 : sinh(e)/e);
        beschb(xmu,&gam1,&gam2,&gampl,&gammi);
        ff=fact*(gam1*cosh(e)+gam2*fact2*d);
        sum=ff;
        ee=exp(e);
        p=0.5*ee/gampl;
        q=0.5/(ee*gammi);
        c=1.0;
        d=x2*x2;
        sum1=p;
        for (i=1;i<=MAXIT;i++) {
            ff=(i*ff+p+q)/(i*i-xmu2);
            c *= (d/i);
            p /= (i-xmu);
            q /= (i+xmu);
            del=c*ff;
            sum += del;
            deli=c*(p-i*ff);
            sum1 += deli;
            if (fabs(del) < fabs(sum)*EPS) break;
        }
        if (i > MAXIT) return(-1);
        rkmu=sum;
        rk1=sum1*xi2;
    } else {
        b=2.0*(1.0+x);
        d=1.0/b;
        h=delh=d;
        q1=0.0;
        q2=1.0;
        a1=0.25-xmu2;
    }
}

```

```

q=c=a1;
a = -a1;
s=1.0+q*delh;
for (i=2;i<=MAXIT;i++) {
    a -= 2*(i-1);
    c = -a*c/i;
    qnew=(q1-b*q2)/a;
    q1=q2;
    q2=qnew;
    q += c*qnew;
    b += 2.0;
    d=1.0/(b+a*d);
    delh=(b*d-1.0)*delh;
    h += delh;
    dels=q*delh;
    s += dels;
    if (fabs(dels/s) < EPS) break;
}
if (i > MAXIT) return(-1);
h=a1*h;
rkmu=sqrt(PI/(2.0*x))*exp(-x)/s;
rk1=rkmu*(xmu+x*0.5-h)*xi;
}
rkmup=xmu*xi*rkmu-rk1;
rimu=xi/(f*rkmu-rkmup);
*ri=(rimu*ril1)/ril;
*rip=(rimu*rip1)/ril;
for (i=1;i<=n1;i++) {
    rktemp=(xmu+i)*xi2*rk1+rkmu;
    rkmu=rk1;
    rk1=rktemp;
}
*rk=rkmu;
*rkp=xmu*xi*rkmu-rk1;
return(0);
}

/*
@2007, Jonathan Belof
Space Research Group
Department of Chemistry
University of South Florida
*/
#include <mc.h>

/* energy of molecule in a 1D anharmonic well */
double anharmonic(system_t *system) {

    double k, g, x;
    double energy;
    molecule_t *molecule_ptr;
    atom_t *atom_ptr;

    k = system->rd_anharmonic_k;
    g = system->rd_anharmonic_g;

    energy = 0;
    for(molecule_ptr = system->molecules; molecule_ptr; molecule_ptr = molecule_ptr->next) {
        for(atom_ptr = molecule_ptr->atoms; atom_ptr; atom_ptr = atom_ptr->next) {
            x = atom_ptr->pos[0];
            energy += anharmonic_energy(k, g, x);

            if(system->feynman_hibbs) {
                if(system->feynman_kleinert) {
                    energy = anharmonic_fk(system->temperature, atom_ptr->mass, k, g, x);
                } else {
                    if(system->feynman_hibbs_order == 2)
                        energy += anharmonic_fh_second_order(system->temperature, atom_ptr->mass, k, g, x);
                    else if(system->feynman_hibbs_order == 4)
                        energy += anharmonic_fh_fourth_order(system->temperature, atom_ptr->mass, k, g, x);
                }
            }
        }
    }
    return(energy);
}

/* Feynman-Kleinert iterative method of effective potential */
double anharmonic_fk(double temperature, double mass, double k, double g, double x) {

    int keep_iterating;
    double a_sq; /* width a^2 (A^2) */
    double omega, omega_sq; /* spacing Omega^2 (K/A^2) */

```

```

double prev_a_sq; /* last a_sq */
double tolerance; /* iterative tolerance */
double V_a; /* V_a^2 (K) */
double potential; /* W_1 (K) */
double conversion_factor; /* hbar^2*(m2A)^2/(k*m) */

/* convert the mass to kg */
mass *= AMU2KG;

conversion_factor = pow(METER2ANGSTROM, 2.0)*pow(HBAR, 2.0)/(KB*mass);

/* initial guess a^2 = beta/12 */
a_sq = pow(METER2ANGSTROM, 2.0)*pow(HBAR, 2.0)/(12.0*KB*temperature*mass);

/* solve self-consistently */
keep_iterating = 1;
while(keep_iterating) {

    /* save the last a_sq for tolerance */
    prev_a_sq = a_sq;

    omega_sq = conversion_factor*(k + 3.0*g*a_sq + 3.0*g*pow(x, 2.0)); omega = sqrt(omega_sq);
    a_sq = conversion_factor*(temperature/omega_sq)*((omega/(2.0*temperature))*(1.0/tanh(omega/(2.0*temperature))) - 1.0);

    tolerance = fabs(prev_a_sq - a_sq);
    if(tolerance < FEYNMAN_KLEINERT_TOLERANCE)
        keep_iterating = 0;
}

V_a = 0.5*a_sq*k + 0.75*g*pow(a_sq, 2.0) + 0.5*(k + 3.0*g*a_sq)*pow(x, 2.0) + 0.25*g*pow(x, 4.0);

potential = temperature*log(sinh(omega/(2.0*temperature))/(omega/(2.0*temperature))) - 0.5*omega_sq*a_sq/conversion_factor + V_a;

return(potential);
}

/* up to FH h^2 effective potential term */
double anharmonic_fh_second_order(double temperature, double mass, double k, double g, double x) {

    double first_derivative;
    double second_derivative;
    double potential;

    mass *= AMU2KG;

    first_derivative = k*x + g*pow(x, 3.0);
    second_derivative = k + 3.0*g*pow(x, 2.0);

    potential = pow(METER2ANGSTROM, 2.0)*pow(HBAR, 2.0)/(24.0*KB*temperature*mass)*(second_derivative + 2.0*first_derivative/x);

    return(potential);
}

/* up to FH h^4 effective potential term */
double anharmonic_fh_fourth_order(double temperature, double mass, double k, double g, double x) {

    double first_derivative, second_derivative;
    double other_derivatives;
    double potential;

    mass *= AMU2KG;

    first_derivative = k*x + g*pow(x, 3.0);
    second_derivative = k + 3.0*g*pow(x, 2.0);
    other_derivatives = 15.0*k/pow(x, 2.0) + 45.0*g;

    potential = pow(METER2ANGSTROM, 2.0)*pow(HBAR, 2.0)/(24.0*KB*temperature*mass)*(second_derivative + 2.0*first_derivative/x);
    potential += pow(METER2ANGSTROM, 4.0)*pow(HBAR, 4.0)/(1152.0*pow(KB*temperature*mass, 2.0))*other_derivatives;

    return(potential);
}

/* anharmonic potential */
double anharmonic_energy(double k, double g, double x) {

    double potential;

    potential = 0.5*k*pow(x, 2.0) + 0.25*g*pow(x, 4.0);

    return(potential);
}

/* H2 specific function */
double h2_bond_energy(double r) {

    return(morse_energy(H2_SPRING_CONSTANT, H2_De, H2_R0, r));
}

/* morse potential */
/* input args are spring constant k in (eV*A^-2), */
/* dissociation energy (in eV), equil r (in A) and the r value (in A) */
double morse_energy(double k, double de, double r0, double r) {

    double a, potential;

    a = sqrt(k/(2.0*de));
    potential = (EV2K*de)*pow((1.0 - exp(-a*(r - r0))), 2.0);

    return(potential);
}

```

```

}

/*
@2007, Jonathan Belof
Space Research Group
Department of Chemistry
University of South Florida
*/
#include <mc.h>

/* total ES energy term */
double coulombic(system_t *system) {

    molecule_t *molecule_ptr;
    atom_t *atom_ptr;
    pair_t *pair_ptr;
    double real, reciprocal;
    double potential;

    /* construct the relevant ewald terms */
    real = coulombic_real(system);
    reciprocal = coulombic_reciprocal(system);

    /* return the total electrostatic energy */
    potential = real + reciprocal;

    return(potential);
}

/* fourier space sum */
double coulombic_reciprocal(system_t *system) {

    molecule_t *molecule_ptr;
    atom_t *atom_ptr;
    int q, p;
    int kmax;
    double alpha;
    double l[3], k[3], k_squared, norm;
    double gaussian, position_product;
    double SF_real, SF_imaginary; /* structure factor */
    double recip_potential, self_potential, potential;

    alpha = system->ewald_alpha;
    kmax = system->ewald_kmax;

    recip_potential = 0; self_potential = 0;
    /* perform the fourier sum over the reciprocal lattice for each particle */
    for(l[0] = 0; l[0] <= kmax; l[0]++) {
        for(l[1] = (!l[0] ? 0 : -kmax); l[1] <= kmax; l[1]++) {
            for(l[2] = ((!l[0] && !l[1]) ? 1 : -kmax); l[2] <= kmax; l[2]++) {

                /* compare the norm */
                for(p = 0, norm = 0; p < 3; p++)
                    norm += l[p]*l[p];

                /* get the reciprocal lattice vectors */
                for(p = 0, k_squared = 0; p < 3; p++) {
                    for(q = 0, k[q] = 0; q < 3; q++)
                        k[p] += system->pbc->reciprocal_basis[p][q]*2.0*M_PI*((double)l[q]);
                    k_squared += k[p]*k[p];
                }

                /* make sure we are within k-max */
                if((norm <= kmax*kmax) && (k_squared > 0.0)) {

                    gaussian = exp(-k_squared/(4.0*alpha*alpha))/k_squared;

                    /* structure factor */
                    SF_real = 0; SF_imaginary = 0;
                    for(molecule_ptr = system->molecules; molecule_ptr = molecule_ptr->next) {
                        for(atom_ptr = molecule_ptr->atoms; atom_ptr; atom_ptr = atom_ptr->next) {
                            if(!atom_ptr->frozen) {

                                /* the inner product of the position vector and the k vector */
                                for(p = 0, position_product = 0; p < 3; p++)
                                    position_product += k[p]*atom_ptr->pos[p];

                                SF_real += atom_ptr->charge*cos(position_product);
                                SF_imaginary += atom_ptr->charge*sin(position_product);

                                /* include the self-interaction term */
                                /* this only need to happen once, on the first k-vec */
                                if((l[0] && !l[1] && (l[2] == 1)) {
                                    atom_ptr->es_self_point_energy =
                                        alpha*atom_ptr->charge*sqrt(M_PI);
                                    self_potential += atom_ptr->es_self_point_energy;
                                }
                            } /* !frozen */
                        } /* atom */
                    } /* molecule */
                    recip_potential += gaussian*(SF_real*SF_real + SF_imaginary*SF_imaginary);
                } /* end if norm */
            } /* end for n */
        } /* end for m */
    } /* end for l */
}

```

```

recip_potential *= 4.0*M_PI/system->pbc->volume;
potential = recip_potential - self_potential;

return(potential);
}

/* real space sum */
double coulombic_real(system_t *system) {

molecule_t *molecule_ptr;
atom_t *atom_ptr;
pair_t *pair_ptr;
double alpha, r, erfc_term, gaussian_term;
double potential;
double potential_classical, potential_fh_second_order, potential_fh_fourth_order;
double first_derivative, second_derivative, third_derivative, fourth_derivative;
double reduced_mass;

alpha = system->ewald_alpha;

potential = 0;
for(molecule_ptr = system->molecules; molecule_ptr; molecule_ptr = molecule_ptr->next) {
    for(atom_ptr = molecule_ptr->atoms; atom_ptr; atom_ptr = atom_ptr->next) {
        for(pair_ptr = atom_ptr->pairs; pair_ptr; pair_ptr = pair_ptr->next) {

            if(pair_ptr->recalculate_energy) {

                pair_ptr->es_real_energy = 0;
                if(!pair_ptr->frozen) {
                    r = pair_ptr->rimg;
                    if(!(r > system->pbc->cutoff) || pair_ptr->es_excluded) { /* unit cell part */

                        erfc_term = erfc(alpha*r);
                        gaussian_term = exp(-alpha*alpha*r*r);

                        potential_classical = atom_ptr->charge*pair_ptr->charge*erfc_term/r;
                        pair_ptr->es_real_energy += potential_classical;

                        if(system->feynman_hibbs) {

                            reduced_mass =
AMU2KG*molecule_ptr->mass*pair_ptr->molecule->mass/(molecule_ptr->mass*pair_ptr->molecule->mass);

                            /* FIRST DERIVATIVE */
                            first_derivative = -2.0*alpha*gaussian_term/(r*sqrt(M_PI)) -
erfc_term/(r*r);

                            /* SECOND DERIVATIVE */
                            second_derivative = (4.0/sqrt(M_PI))*gaussian_term*(pow(alpha, 3.0) + pow(r,
-2.0));
                            second_derivative += 2.0*erfc_term/pow(r, 3.0);

                            potential_fh_second_order = pow(METER2ANGSTROM,
2.0)*(HBAR*HBAR/(24.0*KB*system->temperature*reduced_mass))*(second_derivative + 2.0*first_derivative/r);
                            pair_ptr->es_real_energy += potential_fh_second_order;

                            if(system->feynman_hibbs_order >= 4) {

                                /* THIRD DERIVATIVE */
                                third_derivative = (gaussian_term/sqrt(M_PI))*(-8.0*pow(alpha,
5.0)*r - 8.0*pow(alpha, 3.0)/r - 12.0*alpha/pow(r, 3.0));
                                third_derivative -= 6.0*erfc(alpha*r)/pow(r, 4.0);

                                /* FOURTH DERIVATIVE */
                                fourth_derivative = (gaussian_term/sqrt(M_PI))*( 8.0*pow(alpha, 5.0)
+ 16.0*pow(alpha, 7.0)*r*r + 32.0*pow(alpha, 3.0)/pow(r, 2.0) + 48.0/pow(r, 4.0));
                                fourth_derivative += 24.0*erfc_term/pow(r, 5.0);

                                potential_fh_fourth_order = pow(METER2ANGSTROM, 4.0)*(pow(HBAR,
4.0)/(1152.0*pow(KB*system->temperature*reduced_mass, 2.0)))*(15.0*first_derivative/pow(r, 3.0) + 4.0*third_derivative/r +
fourth_derivative);
                                pair_ptr->es_real_energy += potential_fh_fourth_order;
                            }
                        }
                    }
                }
            }
        }
    }
}

/* else if(pair_ptr->es_excluded) /* calculate the self-intra part */
pair_ptr->es_self_intra_energy =
atom_ptr->charge*pair_ptr->charge*erf(alpha*pair_ptr->r)/pair_ptr->r;

} /* frozen */

} /* recalculate */

/* sum all of the pairwise terms */
potential += pair_ptr->es_real_energy - pair_ptr->es_self_intra_energy;

} /* pair */
} /* atom */
} /* molecule */

return(potential);
}

/* no ewald summation - regular accumulation of Coulombic terms without out consideration of PBC */
/* only used by surface module */
double coulombic_nopbc(molecule_t * molecules) {

molecule_t *molecule_ptr;

```

```

atom_t *atom_ptr;
pair_t *pair_ptr;
double pe, total_pe;

total_pe = 0;
for(molecule_ptr = molecules; molecule_ptr; molecule_ptr = molecule_ptr->next) {
    for(atom_ptr = molecule_ptr->atoms; atom_ptr; atom_ptr = atom_ptr->next) {
        for(pair_ptr = atom_ptr->pairs; pair_ptr; pair_ptr = pair_ptr->next) {

            if(!pair_ptr->es_excluded) {
                pe = atom_ptr->charge*pair_ptr->charge/pair_ptr->r;
                total_pe += pe;
            }

        }
    }
}

return(total_pe);
}

/*
@2007, Jonathan Belof
Space Research Group
Department of Chemistry
University of South Florida
*/
#include <mc.h>

/* returns the total potential energy for the system and updates our observables */
double energy(system_t *system) {

    atom_t *atom_ptr;
    molecule_t *molecule_ptr;
    double potential_energy, rd_energy, coulombic_energy, polar_energy;

    /* zero the initial values */
    potential_energy = 0;
    rd_energy = 0;
    coulombic_energy = 0;
    polar_energy = 0;

    /* get the periodic boundary conditions */
    if(system->wpi || system->fvm) pbc(system->pbc); /* do this each time only for fvm and wpi */

    /* get the pairwise terms necessary for the energy calculation */
    pairs(system);

    /* only on the first simulation step, make sure that all recalculate flags are set */
    if(system->observables->energy == 0.0) flag_all_pairs(system);

    /* get the repulsion/dispersion potential */
    if(system->rd_anharmonic)
        rd_energy = anharmonic(system);
    else if(system->sg)
        rd_energy = sg(system);
    else
        rd_energy = lj(system);
    system->observables->rd_energy = rd_energy;

    /* get the electrostatic potential */
    if(!(system->sg || system->rd_only)) {

        coulombic_energy = coulombic(system);
        system->observables->coulombic_energy = coulombic_energy;

        /* get the polarization potential */
        if(system->polarization) {

            polar_energy = polar(system);
            system->observables->polarization_energy = polar_energy;
        }
    }

    /* sum the total potential energy */
    potential_energy = rd_energy + coulombic_energy + polar_energy;
    system->observables->energy = potential_energy;

    /* count the number of molecules currently in the system */
    for(molecule_ptr = system->molecules, system->observables->N = 0; molecule_ptr; molecule_ptr = molecule_ptr->next) {
        if(!(molecule_ptr->frozen || molecule_ptr->adiabatic)) system->observables->N += 1.0;
        if(system->ensemble == ENSEMBLE_NVE) system->N = system->observables->N;
    }

    /* for NVE */
    if(system->ensemble == ENSEMBLE_NVE) {
        system->observables->kinetic_energy = system->total_energy - potential_energy;
        system->observables->temperature = (2.0/3.0)*system->observables->kinetic_energy/system->observables->N;
    }

    /* SA */
    if(system->simulated_annealing)
        system->observables->temperature = system->temperature;

    /* need this for the isosteric heat */
    system->observables->NU = system->N*system->observables->energy;
}

```

```

    return(potential_energy);

}

/* returns the total potential energy for the system */
/* this function is meant to be called by routines that do not */
/* require observables to be averaged in, i.e. quantum integration */
/* routines, widow insertion, etc. */
double energy_no_observables(system_t *system) {

    atom_t *atom_ptr;
    molecule_t *molecule_ptr;
    double potential_energy, rd_energy, coulombic_energy, polar_energy;

    /* zero the initial values */
    potential_energy = 0;
    rd_energy = 0;
    coulombic_energy = 0;
    polar_energy = 0;

    /* get the periodic boundary conditions */
    if(system->wpi || system->fvm) pbc(system->pbc); /* do this each time only for fvm and wpi */

    /* get the pairwise terms necessary for the energy calculation */
    pairs(system);

    /* get the repulsion/dispersion potential */
    if(system->sg)
        rd_energy = sg(system);
    else
        rd_energy = lj(system);

    /* get the electrostatic potential */
    if(!(system->sg || system->rd_only)) {
        coulombic_energy = coulombic(system);

        /* get the polarization potential */
        if(system->polarization)
            polar_energy = polar(system);
    }

    /* sum the total potential energy */
    potential_energy = rd_energy + coulombic_energy + polar_energy;

    return(potential_energy);
}

/*
@2007, Jonathan Belof
Space Research Group
Department of Chemistry
University of South Florida
*/
#include <mc.h>

/* Lennard-Jones repulsion/dispersion */
double lj(system_t *system) {

    molecule_t *molecule_ptr;
    atom_t *atom_ptr;
    pair_t *pair_ptr;
    double sigma_over_r, term12, term6;
    double first_derivative, second_derivative, third_derivative, fourth_derivative;
    double potential, potential_classical, potential_fh_second_order, potential_fh_fourth_order;
    double reduced_mass;
    double sig3, sig_cut, sig_cut3, sig_cut9;

    potential = 0;
    for(molecule_ptr = system->molecules; molecule_ptr; molecule_ptr = molecule_ptr->next) {
        for(atom_ptr = molecule_ptr->atoms; atom_ptr; atom_ptr = atom_ptr->next) {
            for(pair_ptr = atom_ptr->pairs; pair_ptr; pair_ptr = pair_ptr->next) {
                if(pair_ptr->recalculate_energy) {
                    pair_ptr->rd_energy = 0;

                    /* make sure we're not excluded or beyond the cutoff */
                    if(!((pair_ptr->rimg > system->pbc->cutoff) || pair_ptr->rd_excluded || pair_ptr->frozen)) {
                        sigma_over_r = pair_ptr->sigma/pair_ptr->rimg;

                        /* the LJ potential */
                        term6 = pow(sigma_over_r, 6.0);
                        term6 *= system->scale_rd;
                        term12 = pow(term6, 2.0);
                        potential_classical = 4.0*pair_ptr->epsilon*(term12 - term6);
                        pair_ptr->rd_energy += potential_classical;

                        if(system->feynman_hibbs) {
                            reduced_mass =
                                AMU2KG*molecule_ptr->mass*pair_ptr->molecule->mass/(molecule_ptr->mass+pair_ptr->molecule->mass);
                            /* FIRST DERIVATIVE */
                            first_derivative = -24.0*pair_ptr->epsilon*(2.0*term12 - term6)/pair_ptr->rimg;
                        }
                    }
                }
            }
        }
    }
}

```

```

        /* SECOND DERIVATIVE */
        second_derivative = 24.0*pair_ptr->epsilon*(26.0*term12 -
7.0*term6)/pow(pair_ptr->ring, 2.0);

        potential_fh_second_order = pow(METER2ANGSTROM,
2.0)*(HBAR*HBAR/(24.0*KB*system->temperature*reduced_mass))*(second_derivative + 2.0*first_derivative/pair_ptr->rimg);
        pair_ptr->rd_energy += potential_fh_second_order;

        if(system->feynman_hibbs_order >= 4) {
            /* THIRD DERIVATIVE */
            third_derivative = -1344.0*pair_ptr->epsilon*(6.0*term12 -
term6)/pow(pair_ptr->rimg, 3.0);

            /* FOURTH DERIVATIVE */
            fourth_derivative = 12096.0*pair_ptr->epsilon*(10.0*term12 -
term6)/pow(pair_ptr->rimg, 4.0);

            potential_fh_fourth_order = pow(METER2ANGSTROM, 4.0)*(pow(HBAR,
4.0)/(1152.0*pow(KB*system->temperature*reduced_mass, 2.0))*(15.0*first_derivative/pow(pair_ptr->rimg, 3.0) +
4.0*third_derivative/pair_ptr->rimg + fourth_derivative));
            pair_ptr->rd_energy += potential_fh_fourth_order;
        }
    }

    /* cause an autoreject on insertions closer than 0.5*sigma */
    if(system->cavity_autoreject) {
        if(pair_ptr->rimg < system->cavity_autoreject_scale*pair_ptr->sigma)
            pair_ptr->rd_energy = MAXVALUE;
    }
}

/* include the long-range correction */
if(!(pair_ptr->rd_excluded || pair_ptr->frozen) && (pair_ptr->lrc == 0.0) && system->rd_lrc) {
    sig_cut = pair_ptr->sigma/system->pbc->cutoff;
    sig3 = pow(pair_ptr->sigma, 3.0);
    sig_cut3 = pow(sig_cut, 3.0);
    sig_cut9 = pow(sig_cut, 9.0);
    pair_ptr->lrc = ((16.0/3.0)*M_PI*pair_ptr->epsilon*sig3)*((1.0/3.0)*sig_cut9 -
sig_cut3)/system->pbc->volume;
}

/* if recalculate */
/* sum all of the pairwise terms */
potential += pair_ptr->rd_energy + pair_ptr->lrc;

} /* pair */
} /* atom */
} /* molecule */

return(potential);
}

/* same as above, but no periodic boundary conditions */
double lj_nopbc(molecule_t *molecules) {

molecule_t *molecule_ptr;
atom_t *atom_ptr;
pair_t *pair_ptr;
double sigma_over_r, term12, term6;
double potential;

for(molecule_ptr = molecules, potential = 0; molecule_ptr; molecule_ptr = molecule_ptr->next) {
    for(atom_ptr = molecule_ptr->atoms; atom_ptr; atom_ptr = atom_ptr->next) {
        for(pair_ptr = atom_ptr->pairs; pair_ptr; pair_ptr = pair_ptr->next) {
            /* make sure we're not excluded or beyond the cutoff */
            if(!pair_ptr->rd_excluded) {
                sigma_over_r = pair_ptr->sigma/pair_ptr->r;

                /* the LJ potential */
                term6 = pow(sigma_over_r, 6.0);
                term12 = pow(term6, 2.0);
                potential += 4.0*pair_ptr->epsilon*(term12 - term6);
            }
        }
    }
}

return(potential);
}

/*
@2007, Jonathan Belof
Space Research Group
Department of Chemistry
University of South Florida
*/
#include <mcl.h>

```

```

/* flag all pairs to have their energy calculated */
/* needs to be called at simulation start, or can */
/* be called to periodically keep the total energy */
/* from drifting */
void flag_all_pairs(system_t *system) {

    molecule_t *molecule_ptr;
    atom_t *atom_ptr;
    pair_t *pair_ptr;

    for(molecule_ptr = system->molecules; molecule_ptr; molecule_ptr = molecule_ptr->next)
        for(atom_ptr = molecule_ptr->atoms; atom_ptr; atom_ptr = atom_ptr->next)
            for(pair_ptr = atom_ptr->pairs; pair_ptr; pair_ptr = pair_ptr->next)
                pair_ptr->recalculate_energy = 1;
}

/* set the exclusions and LJ mixing for relevant pairs */
void pair_exclusions(system_t *system, molecule_t *molecule_i, molecule_t *molecule_j, atom_t *atom_i, atom_t *atom_j, pair_t *pair_ptr) {

    /* recalculate exclusions */
    if(molecule_i == molecule_j) { /* if both on same molecule, exclude all interactions */

        pair_ptr->rd_excluded = 1;
        pair_ptr->es_excluded = 1;
    } else {

        /* exclude null repulsion/dispersion interations */
        if((atom_i->epsilon == 0.0) || (atom_i->sigma == 0.0) || (atom_j->epsilon == 0.0) || (atom_j->sigma == 0.0))
            pair_ptr->rd_excluded = 1;
        else
            pair_ptr->rd_excluded = 0;

        /* exclude null electrostatic interactions */
        if((atom_i->charge == 0.0) || (atom_j->charge == 0.0))
            pair_ptr->es_excluded = 1;
        else
            pair_ptr->es_excluded = 0;
    }

    /* get the frozen interactions */
    pair_ptr->frozen = atom_i->frozen && atom_j->frozen;

    /* get the mixed LJ parameters */
    if(!system->sg) {

        /* Lorentz-Berthelot mixing rules */
        pair_ptr->sigma = 0.5*(atom_i->sigma + atom_j->sigma);
        pair_ptr->epsilon = sqrt(atom_i->epsilon*atom_j->epsilon);

        /* get the neighbor charge */
        pair_ptr->charge = atom_j->charge;

        /* get the neighbor polarizability */
        pair_ptr->polarizability = atom_j->polarizability;
    }
}

/* set the link */
pair_ptr->atom = atom_j;
pair_ptr->molecule = molecule_j;

}

/* perform the modulo minimum image for displacements */
void minimum_image(system_t *system, atom_t *atom_i, atom_t *atom_j, pair_t *pair_ptr) {

    int p, q;
    double img[3];
    double d[3], r, r2;
    double di[3], ri, ri2;

    /* get the real displacement */
    pair_ptr->recalculate_energy = 0; /* reset the recalculate flag */
    for(p = 0; p < 3; p++) {
        d[p] = atom_i->pos[p] - atom_j->pos[p];
        pair_ptr->d[p] = d[p];
    }

    /* this pair changed and so it will have it's energy recalculated */
    if(pair_ptr->d[p] != pair_ptr->d_prev[p]) {

        pair_ptr->recalculate_energy = 1;
        pair_ptr->d_prev[p] = pair_ptr->d[p]; /* reset */
    }
}

/* matrix multiply with the inverse basis and round */
for(p = 0; p < 3; p++) {
    for(q = 0, img[p] = 0; q < 3; q++) {
        img[p] += system->pbc->reciprocal_basis[p][q]*d[q];
    }
    img[p] = rint(img[p]);
}

/* matrix multiply to project back into our basis */
for(p = 0; p < 3; p++) {
    for(q = 0, di[p] = 0; q < 3; q++)
        di[p] += system->pbc->basis[p][q]*img[q];
}

```

```

/* now correct the displacement */
for(p = 0; p < 3; p++)
    di[p] = d[p] - di[p];

/* pythagorean terms */
for(p = 0, r2 = 0, ri2 = 0; p < 3; p++) {
    r2 += d[p]*d[p];
    ri2 += di[p]*di[p];
}
r = sqrt(r2);
ri = sqrt(ri2);

/* store the results for this pair */
pair_ptr->r = r;
pair_ptr->rimg = ri;
for(p = 0; p < 3; p++)
    pair_ptr->dimg[p] = di[p];

/* store the 3rd and 5th powers for A matrix stuff */
if(system->polarization) {
    pair_ptr->r2 = r*r;
    pair_ptr->r3 = r*r*r;
    pair_ptr->r5 = pair_ptr->r3*r*r;
    pair_ptr->r2img = ri*ri;
    pair_ptr->r3img = ri*ri*ri;
    pair_ptr->r5img = pair_ptr->r3img*ri*ri;
}

}

/* update everything necessary to describe the complete pairwise system */
void pairs(system_t *system) {

int i, j, n;
molecule_t *molecule_ptr;
atom_t *atom_ptr;
pair_t *pair_ptr;
molecule_t **molecule_array;
atom_t **atom_array;
/* needed for GS ranking metric */
int p;
double r, rmin;

/* generate an array of atom ptrs */
for(molecule_ptr = system->molecules, n = 0, atom_array = NULL, molecule_array = NULL; molecule_ptr; molecule_ptr = molecule_ptr->next) {
    for(atom_ptr = molecule_ptr->atoms; atom_ptr; atom_ptr = atom_ptr->next, n++) {
        molecule_array = realloc(molecule_array, sizeof(molecule_t *)*(n + 1));
        molecule_array[n] = molecule_ptr;

        atom_array = realloc(atom_array, sizeof(atom_t *)*(n + 1));
        atom_array[n] = atom_ptr;
    }
}

/* loop over all atoms and pair */
for(i = 0; i < (n - 1); i++) {
    for(j = (i + 1), pair_ptr = atom_array[i]->pairs; j < n; j++, pair_ptr = pair_ptr->next) {

        /* set any necessary exclusions */
        pair_exclusions(system, molecule_array[i], molecule_array[j], atom_array[i], atom_array[j], pair_ptr);

        /* get the minimum image displacement */
        minimum_image(system, atom_array[i], atom_array[j], pair_ptr);

        } /* for j */
    } /* for i */

/* update the com of each molecule */
update_com(system->molecules);

/* store wrapped coords */
wrapall(system->molecules, system->pbc);

/* rank metric */
if(system->polar_iterative && system->polar_gs_ranked) {

    /* determine smallest polarizable separation */
    rmin = MAXVALUE;
    for(i = 0; i < n; i++) {

        for(j = 0; j < n; j++) {

            if(i != j) && (atom_array[i]->polarizability != 0.0) && (atom_array[j]->polarizability != 0.0) {
                for(p = 0, r = 0; p < 3; p++)
                    r += pow( ( atom_array[i]->wrapped_pos[p] - atom_array[j]->wrapped_pos[p] ), 2.0);
                r = sqrt(r);

                if(r < rmin) rmin = r;
            }
        }
    }

    for(i = 0; i < n; i++) {

        atom_array[i]->rank_metric = 0;
        for(j = 0; j < n; j++) {

            if((i != j) && (atom_array[i]->polarizability != 0.0) && (atom_array[j]->polarizability != 0.0) ) {
                for(p = 0, r = 0; p < 3; p++)
                    r += pow( ( atom_array[i]->wrapped_pos[p] - atom_array[j]->wrapped_pos[p] ), 2.0);
            }
        }
    }
}
}

```

```

        r = sqrt(r);
        if(r <= 1.5*rmin)
            atom_array[i]->rank_metric += 1.0;
    }
}
}

/* free our temporary arrays */
free(atom_array);
free(molecule_array);

}

/* molecular center of mass */
void update_com(molecule_t *molecules) {

    int i;
    molecule_t *molecule_ptr;
    atom_t *atom_ptr;

    for(molecule_ptr = molecules; molecule_ptr; molecule_ptr = molecule_ptr->next) {
        for(i = 0; i < 3; i++)
            molecule_ptr->com[i] = 0;

        for(atom_ptr = molecule_ptr->atoms, molecule_ptr->mass = 0; atom_ptr; atom_ptr = atom_ptr->next) {
            molecule_ptr->mass += atom_ptr->mass;

            for(i = 0; i < 3; i++)
                molecule_ptr->com[i] += atom_ptr->mass*atom_ptr->pos[i];
        }

        for(i = 0; i < 3; i++)
            molecule_ptr->com[i] /= molecule_ptr->mass;
    }
}

/* add new pairs for when a new molecule is created */
void update_pairs_insert(system_t *system) {

    int i, n;
    molecule_t *molecule_ptr;
    atom_t *atom_ptr;
    pair_t *pair_ptr;

    /* count the number of atoms per molecule */
    for(atom_ptr = system->checkpoint->molecule_altered->atoms, n = 0; atom_ptr; atom_ptr = atom_ptr->next, n++);

    /* add n number of pairs to altered and all molecules ahead of it in the list */
    for(molecule_ptr = system->molecules; molecule_ptr != system->checkpoint->tail; molecule_ptr = molecule_ptr->next) {
        for(atom_ptr = molecule_ptr->atoms; atom_ptr; atom_ptr = atom_ptr->next) {

            /* go to the end of the pair list */
            if(atom_ptr->pairs) {
                /* go to the end of the pair list */
                for(pair_ptr = atom_ptr->pairs; pair_ptr->next; pair_ptr = pair_ptr->next);

                /* tag on the extra pairs */
                for(i = 0; i < n; i++) {
                    pair_ptr->next = calloc(1, sizeof(pair_t));
                    pair_ptr = pair_ptr->next;
                }
            } else {

                /* needs a new list */
                atom_ptr->pairs = calloc(1, sizeof(pair_t));
                pair_ptr = atom_ptr->pairs;
                for(i = 0; i < (n - 1); i++) {
                    pair_ptr->next = calloc(1, sizeof(pair_t));
                    pair_ptr = pair_ptr->next;
                }
            }
        }
    }
}

} /* for atom */
} /* for molecule */

}

/* remove pairs when a molecule is deleted */
void update_pairs_remove(system_t *system) {

    int i, n, m;
    molecule_t *molecule_ptr;
    atom_t *atom_ptr;
    pair_t *pair_ptr, **pair_array;

    /* count the number of atoms per molecule */
    for(atom_ptr = system->checkpoint->molecule_backup->atoms, n = 0; atom_ptr; atom_ptr = atom_ptr->next, n++);

    /* remove n number of pairs for all molecules ahead of the removal point */

```

```

pair_array = calloc(1, sizeof(pair_t *));
for(molecule_ptr = system->molecules; molecule_ptr != system->checkpoint->tail; molecule_ptr = molecule_ptr->next) {
    for(atom_ptr = molecule_ptr->atoms; atom_ptr; atom_ptr = atom_ptr->next) {

        /* build the pair pointer array */
        for(pair_ptr = atom_ptr->pairs, m = 0; pair_ptr; pair_ptr = pair_ptr->next, m++) {

            pair_array = realloc(pair_array, sizeof(pair_t *)*(m + 1));
            pair_array[m] = pair_ptr;
        }

        for(i = (m - n); i < m; i++)
            free(pair_array[i]);

        /* handle the end of the list */
        if((m - n) > 0)
            pair_array[(m - n - 1)]->next = NULL;
        else
            atom_ptr->pairs = NULL;
    }
}

/* free our temporary array */
free(pair_array);
}

/* if an insert move is rejected, remove the pairs that were previously added */
void unupdate_pairs_insert(system_t *system) {

int i, n, m;
molecule_t *molecule_ptr;
atom_t *atom_ptr;
pair_t *pair_ptr, **pair_array;

/* count the number of atoms per molecule */
for(atom_ptr = system->checkpoint->molecule_altered->atoms, n = 0; atom_ptr; atom_ptr = atom_ptr->next)
    ++n;

/* remove n number of pairs for all molecules ahead of the removal point */
pair_array = calloc(1, sizeof(pair_t *));
for(molecule_ptr = system->molecules; molecule_ptr != system->checkpoint->tail; molecule_ptr = molecule_ptr->next) {
    for(atom_ptr = molecule_ptr->atoms; atom_ptr; atom_ptr = atom_ptr->next) {

        /* build the pair pointer array */
        for(pair_ptr = atom_ptr->pairs, m = 0; pair_ptr; pair_ptr = pair_ptr->next, m++) {

            pair_array = realloc(pair_array, sizeof(pair_t *)*(m + 1));
            pair_array[m] = pair_ptr;
        }

        for(i = (m - n); i < m; i++)
            free(pair_array[i]);

        /* handle the end of the list */
        if((m - n) > 0)
            pair_array[(m - n - 1)]->next = NULL;
        else
            atom_ptr->pairs = NULL;
    }
}

/* free our temporary array */
free(pair_array);
}

/* if a remove is rejected, then add back the pairs that were previously deleted */
void unupdate_pairs_remove(system_t *system) {

int i, n;
molecule_t *molecule_ptr;
atom_t *atom_ptr;
pair_t *pair_ptr;

/* count the number of atoms per molecule */
for(atom_ptr = system->checkpoint->molecule_backup->atoms, n = 0; atom_ptr; atom_ptr = atom_ptr->next)
    ++n;

/* add n number of pairs to altered and all molecules ahead of it in the list */
for(molecule_ptr = system->molecules; molecule_ptr != system->checkpoint->molecule_backup; molecule_ptr = molecule_ptr->next) {
    for(atom_ptr = molecule_ptr->atoms; atom_ptr; atom_ptr = atom_ptr->next) {

        /* go to the end of the pair list */
        if(atom_ptr->pairs) {

            /* go to the end of the pair list */
            for(pair_ptr = atom_ptr->pairs; pair_ptr->next; pair_ptr = pair_ptr->next);

            /* tag on the extra pairs */
            for(i = 0; i < n; i++) {
                pair_ptr->next = calloc(1, sizeof(pair_t *));
                pair_ptr = pair_ptr->next;
            }
        } else {

            /* needs a new list */
            atom_ptr->pairs = calloc(1, sizeof(pair_t *));
            pair_ptr = atom_ptr->pairs;
        }
    }
}
}

```

```

        for(i = 0; i < (n - 1); i++) {
            pair_ptr->next = calloc(1, sizeof(pair_t));
            pair_ptr = pair_ptr->next;
        }
    }

} /* for atom */
} /* for molecule */

}

/* allocate the pair lists */
void setup_pairs(molecule_t *molecules) {

    int i, j, n;
    molecule_t *molecule_ptr, **molecule_array;
    atom_t *atom_ptr, **atom_array;
    pair_t *pair_ptr, *prev_pair_ptr;

    /* generate an array of atom ptrs */
    for(molecule_ptr = molecules, n = 0, atom_array = NULL, molecule_array = NULL; molecule_ptr; molecule_ptr = molecule_ptr->next) {
        for(atom_ptr = molecule_ptr->atoms; atom_ptr; atom_ptr = atom_ptr->next) {
            atom_array = realloc(atom_array, sizeof(atom_t *)*(n + 1));
            atom_array[n] = atom_ptr;
        }
        molecule_array = realloc(molecule_array, sizeof(molecule_t *)*(n + 1));
        molecule_array[n] = molecule_ptr;
        ++n;
    }

    /* setup the pairs, lower triangular */
    for(i = 0; i < (n - 1); i++) {
        atom_array[i]->pairs = calloc(1, sizeof(pair_t));
        pair_ptr = atom_array[i]->pairs;
        prev_pair_ptr = pair_ptr;

        for(j = (i + 1); j < n; j++) {
            pair_ptr->next = calloc(1, sizeof(pair_t));
            prev_pair_ptr = pair_ptr;
            pair_ptr = pair_ptr->next;
        }

        prev_pair_ptr->next = NULL;
        free(pair_ptr);
    }
}

#endif DEBUG
void test_pairs(molecule_t *molecules) {

    molecule_t *molecule_ptr;
    atom_t *atom_ptr;
    pair_t *pair_ptr;

    for(molecule_ptr = molecules; molecule_ptr; molecule_ptr = molecule_ptr->next) {
        for(atom_ptr = molecule_ptr->atoms; atom_ptr; atom_ptr = atom_ptr->next) {
            for(pair_ptr = atom_ptr->pairs; pair_ptr; pair_ptr = pair_ptr->next) {
                if(!!(pair_ptr->frozen || pair_ptr->rd_excluded || pair_ptr->es_excluded)) printf("%d: charge = %f, epsilon = %f, sigma = %f, r = %f, rimg = %f\n", atom_ptr->id, pair_ptr->charge, pair_ptr->epsilon, pair_ptr->sigma, pair_ptr->r, pair_ptr->rimg); fflush(stdout);
            }
        }
    }
}

#endif
/*
@2007, Jonathan Belof
Space Research Group
Department of Chemistry
University of South Florida
*/
#include <mc.h>

/* calculate the minimum cutoff radius from the basis lattice */
/* generalized to handle skewed unit cells */
/* there is most likely a more elegant way to find this */
double pbc_cutoff(pbc_t *pbc) {

    int P, Q;
    int v1, v2;          /* any two basis vectors */
    double v1_magnitude; /* magnitude of the first basis */
    double component;   /* the terms a*a + b*a */
    double r_vector[3], r_magnitude; /* the new vector */
    double rmin[3];      /* the radial cutoff for each basis pair */
    double cutoff;       /* the minimal cutoff */

    /* for each pair of basis vectors */
    for(p = 0; p < 3; p++) {

```

```

/* the unique basis pairs forming the parallelogram */
v1 = p;
v2 = (p + 1) % 3;

/* calculate the first basis magnitude */
v1_magnitude = 0;
for(q = 0; q < 3; q++) {
    vi_magnitude += pbc->basis[v1][q]*pbc->basis[v1][q];
}
v1_magnitude = sqrt(v1_magnitude);

/* compute vector components of aa + ba */
component = 0;
for(q = 0; q < 3; q++) {
    component += pbc->basis[v1][q]*pbc->basis[v1][q];
    component += pbc->basis[v2][q]*pbc->basis[v1][q];
}
component /= v1_magnitude;

/* finally, the r vector itself */
/* r = 1/2[a + b - (aa + ba)a] */
r_magnitude = 0;
for(q = 0; q < 3; q++) {
    r_vector[q] = pbc->basis[v1][q] + pbc->basis[v2][q];
    r_vector[q] -= component*pbc->basis[v1][q]/v1_magnitude;
    r_vector[q] *= 0.5;
    r_magnitude += r_vector[q]*r_vector[q];
}
r_magnitude = sqrt(r_magnitude);

/* store the result for this parallelogram - sort it out when done */
rmin[p] = r_magnitude;
}

/* sort out the smallest radial cutoff */
cutoff = MAXVALUE;
for(p = 0; p < 3; p++) {
    if(rmin[p] < cutoff) cutoff = rmin[p];
}

return(cutoff);
}

/* take the determinant of the basis matrix */
double pbc_volume(pbc_t *pbc) {

    double volume;

    volume = pbc->basis[0][0]*(pbc->basis[1][1]*pbc->basis[2][2] - pbc->basis[1][2]*pbc->basis[2][1]);
    volume += pbc->basis[0][1]*(pbc->basis[1][2]*pbc->basis[2][0] - pbc->basis[1][0]*pbc->basis[2][2]);
    volume += pbc->basis[0][2]*(pbc->basis[1][0]*pbc->basis[2][1] - pbc->basis[2][1]*pbc->basis[2][0]);

    return(volume);
}

/* get the reciprocal space basis */
void pbc_reciprocal(pbc_t *pbc) {

    int p, q;
    double inverse_volume;

    inverse_volume = 1.0/pbc_volume(pbc);

    pbc->reciprocal_basis[0][0] = inverse_volume*(pbc->basis[1][1]*pbc->basis[2][2] - pbc->basis[1][2]*pbc->basis[2][1]);
    pbc->reciprocal_basis[0][1] = inverse_volume*(pbc->basis[0][2]*pbc->basis[2][1] - pbc->basis[0][1]*pbc->basis[2][2]);
    pbc->reciprocal_basis[0][2] = inverse_volume*(pbc->basis[0][1]*pbc->basis[1][2] - pbc->basis[0][2]*pbc->basis[1][1]);

    pbc->reciprocal_basis[1][0] = inverse_volume*(pbc->basis[1][2]*pbc->basis[2][0] - pbc->basis[1][0]*pbc->basis[2][2]);
    pbc->reciprocal_basis[1][1] = inverse_volume*(pbc->basis[0][0]*pbc->basis[2][2] - pbc->basis[0][2]*pbc->basis[2][0]);
    pbc->reciprocal_basis[1][2] = inverse_volume*(pbc->basis[0][2]*pbc->basis[1][2] - pbc->basis[0][0]*pbc->basis[1][2]);

    pbc->reciprocal_basis[2][0] = inverse_volume*(pbc->basis[1][0]*pbc->basis[2][1] - pbc->basis[1][1]*pbc->basis[2][0]);
    pbc->reciprocal_basis[2][1] = inverse_volume*(pbc->basis[0][1]*pbc->basis[2][0] - pbc->basis[0][0]*pbc->basis[2][1]);
    pbc->reciprocal_basis[2][2] = inverse_volume*(pbc->basis[0][0]*pbc->basis[1][1] - pbc->basis[0][1]*pbc->basis[1][0]);

}

void pbc(pbc_t *pbc) {

    /* get the unit cell volume and cutoff */
    pbc->volume = pbc_volume(pbc);
    pbc->cutoff = pbc_cutoff(pbc);

    /* get the reciprocal space lattice */
    pbc_reciprocal(pbc);
}

/*
@2007, Jonathan Belof
Space Research Group
Department of Chemistry
University of South Florida
*/
#include <mc.h>

/* get the induction energy */

```

```

double polar(system_t *system) {
    int i, num_iterations;
    molecule_t *molecule_ptr;
    atom_t *atom_ptr;
    double dipole_rrms, N, potential;

    /* take measures to let N fluctuate */
    if((system->ensemble == ENSEMBLE_UVT) && !system->polar_zodid)
        thole_resize_matrices(system);

    /* get the A matrix */
    if(!system->polar_zodid) {
        thole_amatrix(system);
        if(system->polarizability_tensor) {
            output("POLAR: A matrix:\n");
            print_matrix(3*((int)system->checkpoint->N_atom), system->A_matrix);
        }
    }

    /* calculate the field vectors */
    thole_field(system);

    /* find the dipoles */
    if(system->polar_iterative) { /* solve the self-consistent field... */
        num_iterations = thole_iterative(system);
        system->nodestats->polarization_iterations = (double)num_iterations;

        /* RRMS of dipoles */
        N = 0; dipole_rrms = 0;
        for(molecule_ptr = system->molecules; molecule_ptr; molecule_ptr = molecule_ptr->next) {
            for(atom_ptr = molecule_ptr->atoms; atom_ptr; atom_ptr = atom_ptr->next) {
                if(isinfinite(atom_ptr->dipole_rrms)) dipole_rrms += atom_ptr->dipole_rrms;
                N += 1.0;
            }
        }
        dipole_rrms /= N;
        system->observables->dipole_rrms = dipole_rrms;

        } else { /* ...or do matrix inversion */
        thole_bmatrix(system);
        thole_bmatrix_dipoles(system);

        /* output the 3x3 molecular polarizability tensor */
        if(system->polarizability_tensor) {
            output("POLAR: B matrix:\n");
            print_matrix(3*((int)system->checkpoint->N_atom), system->B_matrix);
            thole_polarizability_tensor(system);
            exit(0);
        }
    }

    /* calculate the polarization energy as 1/2 mu*E */
    for(molecule_ptr = system->molecules, potential = 0; molecule_ptr; molecule_ptr = molecule_ptr->next) {
        for(atom_ptr = molecule_ptr->atoms; atom_ptr; atom_ptr = atom_ptr->next) {
            for(i = 0; i < 3; i++) {
                potential += atom_ptr->mu[i]*atom_ptr->ef_static[i];
                if(system->polar_paldo)
                    potential += atom_ptr->mu[i]*atom_ptr->ef_induced_change[i];
            }
        }
        potential *= -0.5;
    }
    return(potential);
}

/*
@2007, Jonathan Belof
Space Research Group
Department of Chemistry
University of South Florida
*/
#include <mc.h>

/* Silvera-Goldman H2 potential */
double sg(system_t *system) {
    molecule_t *molecule_ptr;
    atom_t *atom_ptr;
    pair_t *pair_ptr;
    double rring, r6, r8, r10, r_rm;
    double repulsive_term, multipole_term, exponential_term;
    double first_r_diff_term, second_r_diff_term;
    double first_derivative, second_derivative;
    double potential_classical, potential_fh_second_order;
    double potential;
    double temperature;
}

```

```

temperature = system->temperature;
for(molecule_ptr = system->molecules; molecule_ptr; molecule_ptr = molecule_ptr->next) {
    for(atom_ptr = molecule_ptr->atoms; atom_ptr; atom_ptr = atom_ptr->next) {
        for(pair_ptr = atom_ptr->pairs; pair_ptr; pair_ptr = pair_ptr->next) {
            if(pair_ptr->recalculate_energy) {
                pair_ptr->rd_energy = 0;
                rimg = pair_ptr->rimg;
                if(rimg < system->pbc->cutoff) {
                    /* convert units to Bohr radii */
                    rimg /= AU2ANGSTROM;

                    /* classical pairwise part */
                    repulsive_term = exp(ALPHA - BETA*rimg - GAMMA*rimg*rimg);

                    r6 = pow(rimg, 6.0);
                    r8 = pow(rimg, 8.0);
                    r10 = pow(rimg, 10.0);
                    multipole_term = C6/r6 + C8/r8 + C10/r10;

                    r_rm = RM/rimg;
                    if(rimg < RM)
                        exponential_term = exp(-pow((r_rm - 1.0), 2.0));
                    else
                        exponential_term = 1.0;

                    potential_classical = (repulsive_term - multipole_term*exponential_term);
                    pair_ptr->rd_energy += potential_classical;

                    if(system->feynman_hibbs) {
                        /* FIRST DERIVATIVE */
                        first_derivative = (-BETA - 2.0*GAMMA*rimg)*repulsive_term;
                        first_derivative += (6.0*C6/pow(rimg, 7.0) + 8.0*C8/pow(rimg, 9.0) +
                        10.0*C10/pow(rimg, 11.0))*exponential_term;
                        first_r_diff_term = (r_rm*r_rm - r_rm)/rimg;
                        first_derivative += -2.0*multipole_term*exponential_term*first_r_diff_term;

                        /* SECOND DERIVATIVE */
                        second_derivative = (pow((BETA + 2.0*GAMMA*rimg), 2.0) - 2.0*GAMMA)*repulsive_term;
                        second_derivative += (-exponential_term)*(42.0*C6/pow(rimg, 8.0) + 72.0*C8/pow(rimg,
                        10.0) + 110.0*C10/pow(rimg, 10.0));
                        second_derivative += exponential_term*first_r_diff_term*(12.0*C6/pow(rimg, 7.0) +
                        16.0*C8/pow(rimg, 9.0) + 20.0*C10/pow(rimg, 11.0));
                        second_derivative += exponential_term*pow(first_r_diff_term,
                        2.0)*4.0*multipole_term;
                        second_r_diff_term = (3.0*r_rm*r_rm - 2.0*r_rm)/(rimg*rimg);
                        second_derivative += exponential_term*second_r_diff_term*2.0*multipole_term;

                        potential_fh_second_order = pow(METER2ANGSTROM,
                        2.0)*(HBAR*HBAR/(24.0*KB*temperature*(AMU2KG*molecule_ptr->mass)))*(second_derivative +
                        2.0*first_derivative/rimg);
                        pair_ptr->rd_energy += potential_fh_second_order;
                    }
                }
            }
        }
    }
}

/* convert units from Hartrees back to Kelvin */
pair_ptr->rd_energy *= HARTREE2KELVIN;

}

} /* */ recalculate */
} /* */ pair */
} /* */ atom */
} /* */ molecule */

potential = 0;
for(molecule_ptr = system->molecules; molecule_ptr; molecule_ptr = molecule_ptr->next)
    for(atom_ptr = molecule_ptr->atoms; atom_ptr; atom_ptr = atom_ptr->next)
        for(pair_ptr = atom_ptr->pairs; pair_ptr; pair_ptr = pair_ptr->next)
            potential += pair_ptr->rd_energy;

return(potential);
}

/* same as above, but no periodic boundary conditions */
double sg_nopbc(molecule_t *molecules) {

molecule_t *molecule_ptr;
atom_t *atom_ptr;
pair_t *pair_ptr;
double r, r6, r8, r10, r9;
double r_rm, r_rm_2, r_exp;
double potential, multipole_term;
double result, exp_result;

for(molecule_ptr = molecules, potential = 0; molecule_ptr; molecule_ptr = molecule_ptr->next) {
    for(atom_ptr = molecule_ptr->atoms; atom_ptr; atom_ptr = atom_ptr->next) {
        for(pair_ptr = atom_ptr->pairs; pair_ptr; pair_ptr = pair_ptr->next) {
            if(pair_ptr->recalculate_energy) {
                r = pair_ptr->r/AU2ANGSTROM;

```

```

r6 = pow(r, 6.0);
r8 = pow(r, 8.0);
r10 = pow(r, 10.0);
r9 = pow(r, 9.0);

multipole_term = C6/r6 + C8/r8 + C10/r10 - C9/r9;

if(r < RM) {
    r_rm = RM/r;
    r_rm -= 1.0;
    r_rm_2 = pow(r_rm, 2.0);
    r_rm_2 *= -1.0;
    r_exp = exp(r_rm_2);

    multipole_term *= r_exp;
}

result = ALPHA - BETA*r - GAMMA*r*r;

exp_result = exp(result);
exp_result -= multipole_term;
pair_ptr->rd_energy = HARTREE2KELVIN*exp_result;

} /* pair */
} /* atom */
} /* molecule */

potential = 0;
for(molecule_ptr = molecules; molecule_ptr; molecule_ptr = molecule_ptr->next)
    for(atom_ptr = molecule_ptr->atoms; atom_ptr; atom_ptr = atom_ptr->next)
        for(pair_ptr = atom_ptr->pairs; pair_ptr; pair_ptr = pair_ptr->next)
            potential += pair_ptr->rd_energy;

return(potential);
}

#include <mc.h>

/* to use this routine, compile with -g defined.
 * call this routine at any point in the execution
 * that you would like to attach. when execution
 * begins, the process should write out the PIDS
 * of all processes and hang, open a new terminal
 * and start gdb. Attach to the PID of choice. go
 * up the function stack until you are in the scope
 * of the
 * gdb> attach 21456
 */
void attach(void)
{
    /* debug routine to attach to gdb */
    static int i=0;
    printf("RANK: %d PID: %d\n",rank,getpid());
    fflush(NULL);
    while(!i) sleep(1);
}

/* update the avg_histogram stored on root */
void update_root_histogram(system_t *system)
{
    int i,j,k;
    int xdim=system->grids->histogram->x_dim;
    int ydim=system->grids->histogram->y_dim;
    int zdim=system->grids->histogram->z_dim;

    for(k=0; k < zdim; k++){
        for(j=0; j < ydim; j++){
            for(i=0; i < xdim; i++){
                system->grids->avg_histogram->grid[i][j][k] += system->grids->histogram->grid[i][j][k];
                /* norm_total is updated here to normalize upon write out */
                system->grids->avg_histogram->norm_total += system->grids->histogram->grid[i][j][k];
            }
        }
    }
}

/* zero out the histogram grid */
void zero_grid(int ***grid, system_t *system)
{
    int i,j,k;
    int xdim=system->grids->histogram->x_dim;
    int ydim=system->grids->histogram->y_dim;
    int zdim=system->grids->histogram->z_dim;

    for(k=0; k < zdim; k++){
        for(j=0; j < ydim; j++){
            for(i=0; i < xdim; i++){
                grid[i][j][k]=0;
            }
        }
    }
}

/* compute the histogram bin number to place this molecule in */
void compute_bin(double *cart_coords, system_t *system, int *bin_vector)
{
    double frac_coords[3];
    int Abin,Bbin,Cbin;
    int index;
    int xdim=system->grids->histogram->x_dim;
    int ydim=system->grids->histogram->y_dim;
    int zdim=system->grids->histogram->z_dim;
}

```

```

/* we need the fractional coords to simplify the bin calculation */
cart2frac(frac_coords,cart_coords,system);

/* the coordinate system in the simulation is from -0.5 to 0.5 */
/* so we need to correct for this: add 0.5 to each dimension */
frac_coords[0]+=0.5;
frac_coords[1]+=0.5;
frac_coords[2]+=0.5;

/* compute bin in each dimension */
Abin=(int)floor(frac_coords[0]*system->grids->histogram->x_dim);
Bbin=(int)floor(frac_coords[1]*system->grids->histogram->y_dim);
Cbin=(int)floor(frac_coords[2]*system->grids->histogram->z_dim);

/* return result to the bin_vector passed in */
bin_vector[0]=Abin;
bin_vector[1]=Bbin;
bin_vector[2]=Cbin;
}

void wrap1coord(double *unwrapped, double *wrapped, system_t *system)
{
    double unit[3],offset[3],frac[3];
    int i,j;

    /* zero the vectors */
    for(i=0;i<3;i++) { offset[i]=0; unit[i]=0; frac[i]=0; }

    /* put coords in fractional representation */
    for(i=0;i<3;i++)
        for(j=0;j<3;j++)
            frac[i]+=system->pbc->reciprocal_basis[i][j]*unwrapped[j];

    /* any fractional coord > .5 or < -.5 round to 1,-1 etc. */
    for(i=0;i<3;i++) unit[i] = rint(frac[i]);

    /* multiply this rounded fractional unit vector by basis */
    for(i=0;i<3;i++)
        for(j=0;j<3;j++)
            offset[i]+=system->pbc->basis[i][j]*unit[j];

    /* subtract this distance from the incoming vector */
    for(i=0;i<3;i++) wrapped[i] = unwrapped[i] - offset[i];
}

/* population histogram should be performed only every corr time
 * individual nodes store the data on grids->histogram
 * only root will compile the sum into grids->avg_histogram
 * the node only grid->histogram will be rezeroed at every corr time
 * to prevent overflow.
 * NOTE: The histogram is normalized to 1.
 * e.g. if there are a total of 328 bin counts -> every bin is
 * divided by 328. */
void population_histogram(system_t *system)
{
    molecule_t *mol_p;
    char linebuf[MAXLINE];
    int bin[3];
    double wrappedcoords[3];
    double wrappedcoords[3];
    for(mol_p = system->molecules; mol_p = mol_p->next){
        if(!mol_p->frozen){
            /* wrap the coordinates of mol_p */
            wrap1coord(mol_p,&wrappedcoords,system);
            /* compute what bin to increment. store answer in bin[] */
            compute_bin(wrappedcoords,system,bin);
            /* increment the bin returned in bin[] */
            /*(system->grids->histogram->grid[(bin[0])][(bin[1])][(bin[2])]]++;*/
        }
    }
}

/* This writes out the grid with the Cbin (last index) varying
 * the fastest. Line break between dimensions, double line
 * break between ZY sheets, #####'s between complete sets
 * Remember, for this to work, we had to offset the origin
 * by 1/2 a bin width */
void write_histogram(FILE *fp_out, int ***grid, system_t *system)
{
    int i,j,k;
    int xdim,ydim,zdim;
    int count=0;

    xdim=system->grids->histogram->x_dim;
    ydim=system->grids->histogram->y_dim;
    zdim=system->grids->histogram->z_dim;

    rewind(fp_out);
    fprintf(fp_out,"# OpenDX format population histogram\n");
    fprintf(fp_out,"object 1 class gridpositions counts %d %d %d\n",xdim,ydim,zdim);
    fprintf(fp_out,"origin %f %f %f\n",system->grids->histogram->origin[0],system->grids->histogram->origin[1],system->grids->histogram->origin[2]);
    fprintf(fp_out,"delta %f %f %f\n",system->grids->histogram->delta[0][0],system->grids->histogram->delta[0][1],system->grids->histogram->delta[0][2]);
    fprintf(fp_out,"delta %f %f %f\n",system->grids->histogram->delta[1][0],system->grids->histogram->delta[1][1],system->grids->histogram->delta[1][2]);
    fprintf(fp_out,"delta %f %f %f\n",system->grids->histogram->delta[2][0],system->grids->histogram->delta[2][1],system->grids->histogram->delta[2][2]);
    fprintf(fp_out,"#\n");
    fprintf(fp_out,"object 2 class gridconnections counts %d %d %d\n",xdim,ydim,zdim);
    fprintf(fp_out,"#\n");
    fprintf(fp_out,"object 3 class array type float rank 0 items %d data follows\n",system->grids->histogram->n_data_points);

    for(i=0; i < xdim; i++){
        for(j=0; j < ydim; j++){
            for(k=0; k < zdim; k++){

```

```

        fprintf(fp_out,"%f ",(float)(grid[i][j][k]) / (float)(system->grids->avg_histogram->norm_total));
        count+=grid[i][j][k];
    }
    fprintf(fp_out,"\\n");
}
fprintf(fp_out,"\\n");

fprintf(fp_out,"# count=%d\\n",count);
fprintf(fp_out,"attribute \"dep\" string \"positions\\n\"");
fprintf(fp_out,"object \"regular positions regular connections\" class field\\n");
fprintf(fp_out,"component \"positions\" value 1\\n");
fprintf(fp_out,"component \"connections\" value 2\\n");
fprintf(fp_out,"component \"data\" value 3\\n");
fprintf(fp_out,"\\nend\\n");
fflush(fp_out);
}

/* Take a vector in fractional coordinates (frac[]) and
 * convert it to cartesian coordinates.
 * store the answer in answer[]      */
int frac2cart(double *answer, double *frac, system_t *system)
{
    int i,j;
    for(i=0;i<3;i++) answer[i]=0.0;
    for(i=0;i<3;i++){
        for(j=0;j<3;j++){
            answer[i]+=system->pbc->basis[i][j]*frac[j];
        }
    }
    return 1;
}

/* Take a vector in cartesian coordinates (cart[]) and
 * convert it to fractional coordinates.
 * store the answer in anwer[]      */
int cart2frac(double *answer, double *cart, system_t *system){
    int i,j;
    for(i=0;i<3;i++) answer[i]=0.0;
    for(i=0;i<3;i++){
        for(j=0;j<3;j++){
            answer[i]+=system->pbc->reciprocal_basis[i][j]*cart[j];
        }
    }
    return 1;
}

/* Returns the magnitude of a 3-vector */
double magnitude(double *vector){
    int i;
    double mag=0;
    for(i=0;i<3;i++) mag+=vector[i]*vector[i];
    return sqrt(mag);
}

void allocate_histogram_grid(system_t *system)
{
    int i,j,k;
    int x_dim=system->grids->histogram->x_dim;
    int y_dim=system->grids->histogram->y_dim;
    int z_dim=system->grids->histogram->z_dim;

    /* allocate a 3D grid for the histogram */
    system->grids->histogram->grid = calloc(x_dim, sizeof(int *));
    for(i=0; i<x_dim; i++)
        system->grids->histogram->grid[i] = calloc(y_dim, sizeof(int *));
    for(i=0; i<x_dim; i++){
        for(j=0; j<y_dim; j++){
            system->grids->histogram->grid[i][j] = calloc(z_dim, sizeof(int));
        }
    }

    /* if root, allocate an avg_histogram grid */
    if(!rank){
        system->grids->avg_histogram->grid = calloc(x_dim, sizeof(int *));
        for(i=0; i<x_dim; i++)
            system->grids->avg_histogram->grid[i] = calloc(y_dim, sizeof(int *));
        for(i=0; i<x_dim; i++){
            for(j=0; j<y_dim; j++){
                system->grids->avg_histogram->grid[i][j] = calloc(z_dim, sizeof(int));
            }
        }
    }
}

/* These are needed by dxwrite routines.
 * delta is a transformation matrix that defines the
 * step size for setting up a grid in OpenDX
 * see the DX userguide.pdf appendix B for
 * more details. */
void setup_deltas(histogram_t *hist, system_t *system)
{
    int i,j;
    for(i=0;i<3;i++)
        for(j=0;j<3;j++)

```

```

hist->delta[i][j]=system->pbc->basis[j][i]/hist->count[i];
/* we divide by the count to get our actual step size in each dimension */

/* Because OpenDX puts our data values at the points of the grid,
 * we need to offset them by half a bin width to reflect the fact that
 * we have a histogram. Our data really lies between each point. */
void offset_dx_origin(double *real_origin_cartesian, histogram_t *hist, system_t *system)
{
    double fractional_binwidth[3];
    double cart_halfbin[3];

    /* figure out how wide each bin is in each dimension */
    fractional_binwidth[0] = 1.0 / hist->x_dim;
    fractional_binwidth[1] = 1.0 / hist->y_dim;
    fractional_binwidth[2] = 1.0 / hist->z_dim;

    /* figure out how wide half a binwidth is in cartesians */
    fractional_binwidth[0]/=2.0;
    fractional_binwidth[1]/=2.0;
    fractional_binwidth[2]/=2.0;
    frac2cart(cart_halfbin, fractional_binwidth, system);

    /* add this value to the origin */
    real_origin_cartesian[0]+=cart_halfbin[0];
    real_origin_cartesian[1]+=cart_halfbin[1];
    real_origin_cartesian[2]+=cart_halfbin[2];
}

/* Variables needed upon printing in openDX native format.
 * see DX userguide.pdf appendix B for details. */
void setup_dx_variables(histogram_t *hist, system_t *system)
{
    int i,j,k;
    double vec[3],origin[3];

    /* setup counts */
    hist->count[0]=hist->x_dim;
    hist->count[1]=hist->y_dim;
    hist->count[2]=hist->z_dim;

    /* setup origin */
    vec[0]=-0.5; vec[1]=-0.5; vec[2]=-0.5;
    frac2cart(origin, vec, system);

    /* IMPORTANT!!! I am offsetting the origin by 1/2 a bin in each dimension */
    /* the result of origin is not the true origin!!! */
    offset_dx_origin(origin,hist,system);

    hist->origin[0]=origin[0];
    hist->origin[1]=origin[1];
    hist->origin[2]=origin[2];

    /* setup deltas */
    setup_deltas(system->grids->histogram,system);

    /* setup N data points */
    hist->n_data_points = hist->x_dim * hist->y_dim * hist->z_dim;
}

/* Setup the various quantities that define the histogram grid */
void setup_histogram(system_t *system){

    char linebuf[256];
    double trial_vec1[3];
    double trial_vec2[3];
    double magA,magB,magC;
    int Nbins,i,j,x_dim,y_dim,z_dim;

    /* get the magnitudes of all the basis vectors and test the frac2cart routine.
     * define a fractional vector (1,0,0) and transform it with our basis.
     * then calculate its magnitude. Do this in all 3 dimensions */
    trial_vec1[0]=1.0; trial_vec1[1]=0.0; trial_vec1[2]=0.0;
    frac2cart(trial_vec2,trial_vec1,system);
    magA = magnitude(trial_vec2);

    trial_vec1[0]=0.0; trial_vec1[1]=1.0; trial_vec1[2]=0.0;
    frac2cart(trial_vec2,trial_vec1,system);
    magB = magnitude(trial_vec2);

    trial_vec1[0]=0.0; trial_vec1[1]=0.0; trial_vec1[2]=1.0;
    frac2cart(trial_vec2,trial_vec1,system);
    magC = magnitude(trial_vec2);

    /* calculate the number of bins in each fractional coordinate */
    x_dim=rint(magA/system->hist_resolution);
    y_dim=rint(magB/system->hist_resolution);
    z_dim=rint(magC/system->hist_resolution);
    sprintf(linebuf,"HISTOGRAM: %f resolution -> %d bins(A) * %d bins(B) * %d
bins(C)\n",system->hist_resolution,x_dim,y_dim,z_dim); output(linebuf);

    Nbins=x_dim * y_dim * z_dim;
    sprintf(linebuf,"HISTOGRAM: Total Bins = %d\n",Nbins); output(linebuf);

    system->grids->histogram->x_dim = x_dim;
    system->grids->histogram->y_dim = y_dim;
    system->grids->histogram->z_dim = z_dim;
    system->grids->avg_histogram->x_dim = x_dim;
    system->grids->avg_histogram->y_dim = y_dim;
    system->grids->avg_histogram->z_dim = z_dim;
    system->grids->histogram->n_data_points = Nbins;
    system->n_histogram_bins = Nbins;
    system->grids->avg_histogram->norm_total=0;
    system->grids->histogram->norm_total=0;
}

```

```

    setup_dx_variables(system->grids->histogram,system);
}

/* energy */
double energy(system_t *);
double energy_no_observables(system_t *);
double lj(system_t *);
double lj_nopbc(molecule_t *);
void update_com(molecule_t *);
void flag_all_pairs(system_t *);
void pair_exclusions(system_t *, molecule_t *, molecule_t *, atom_t *, atom_t *, pair_t *);
void minimum_image(system_t *, atom_t *, atom_t *, pair_t *);
void pairs(system_t *);
void setup_pairs(molecule_t *);
void update_pairs_insert(system_t *);
void update_pairs_remove(system_t *);
void unupdate_pairs_insert(system_t *);
void unupdate_pairs_remove(system_t *);
double pbc_cutoff(pbc_t *);
double pbc_volume(pbc_t *);
void pbc(pbc_t *);
double sg(system_t *);
double sg_nopbc(molecule_t *);
double coulombic(system_t *);
double coulombic_real(system_t *);
double coulombic_reciprocal(system_t *);
double coulombic_nopbc(molecule_t *);
double morse_energy(double, double, double, double);
double anharmonic(system_t *);
double anharmonic_energy(double, double, double);
double anharmonic_fk(double, double, double, double, double);
double anharmonic_fh_second_order(double, double, double, double, double);
double anharmonic_fh_fourth_order(double, double, double, double, double);
double h2_bond_energy(double r);
int bessik(float, float, float *, float *, float *, float *); /* NR function */
double besselK(double, double);

/* io */
system_t *read_config(char *);
int check_config(system_t *);
molecule_t *read_molecules(system_t *);
system_t *setup_system(char *);
void error(char *);
void output(char *);
void clear_nodestats(nodestats_t *);
void clear_node_averages(avg_nodestats_t *);
void clear_observables(observables_t *);
void clear_root_averages(avg_observables_t *);
void track_ax(nodestats_t *);
void update_nodestats(nodestats_t *, avg_nodestats_t *);
void update_root_averages(system_t *, observables_t *, avg_nodestats_t *, avg_observables_t *);
int write_performance(int, system_t *);
int write_averages(avg_observables_t *);
int write_molecules(system_t *, char *);
void write_observables(FILE *, observables_t *);
void write_states(FILE *, molecule_t *);
int wrappall(molecule_t *, pbc_t *);
double co2_fugacity(double, double);
double h2_fugacity(double, double);
double h2_fugacity_shaw(double, double);
double h2_fugacity_zhou(double, double);
double h2_fugacity_back(double, double);
double h2_comp_back(double, double);
int open_files(system_t *);
void close_files(system_t *);

/* main */
void usage(char *);
double get_rand(void);
double rule30_rng(long int);
void seed_rng(long int);
#ifndef QM_ROTATION
void free_rotation(system_t *);
#endif /* QM_ROTATION */
void free_pairs(molecule_t *);
void free_atoms(molecule_t *);
void free_molecule(system_t *, molecule_t *);
void free_molecules(molecule_t *);
void free_averages(system_t *system);
void free_matrices(system_t *system);
void free_cavity_grid(system_t *system);
void cleanup(system_t *);
void terminate_handler(int, system_t *);

/* mc */
void boltzmann_factor(system_t *, double, double);
void register_accept(system_t *);
void register_reject(system_t *);
int mc(system_t *);
molecule_t *copy_molecule(system_t *, molecule_t *);
void translate(molecule_t *, pbc_t *, double);
void rotate(molecule_t *, pbc_t *, double);
void displace(molecule_t *, pbc_t *, double, double);
void displace_1D(system_t *, molecule_t *, double);
void make_move(system_t *);
void checkpoint(system_t *);
void restore(system_t *);
double surface_energy(system_t *, int);
void molecule_rotate(molecule_t *, double, double, double);
int surface_dimer_geometry(system_t *, double, double, double, double, double, double, double, double);
int surface_dimer_parameters(system_t *, param_t *);
void surface_curve(system_t *, double, double, double, double *);
int surface(system_t *);
int surface_fit(system_t *);
int surface_virial(system_t *);
void setup_cavity_grid(system_t *);
void cavity_volume(system_t *);

```

```

void cavity_probability(system_t *);
void cavity_update_grid(system_t *);

/* polarization */
double polar(system_t *);
void thole_amatrix(system_t *);
void thole_bmatrix(system_t *);
void thole_bmatrix_dipoles(system_t *);
void thole_polarizability_tensor(system_t *);
void thole_field(system_t *);
void thole_field_nopbc(system_t *);
void thole_field_real(system_t *);
void thole_field_recip(system_t *);
void thole_field_self(system_t *);
int thole_iterative(system_t *);
int invert_matrix(int **, double **, double **);
int num_atoms(system_t *);
void thole_resize_matrices(system_t *);

/* pimc */
int pimc(system_t *);

/* histogram */
int histogram(system_t *);
int cart2frac(double *, double *, system_t *);
int frac2cart(double *, double *, system_t *);
void setup_histogram(system_t *);
void allocate_histogram_grid(system_t *);
void write_histogram(FILE *, int ***, system_t *);
void zero_grid(int ***, system_t *);
void population_histogram(system_t *);
void mpi_copy_histogram_to_sendbuffer(char *, int ***, system_t *);
void mpi_copy_rcv_histogram_to_data(char *, int ***, system_t *);
void update_root_histogram(system_t *);
void write_histogram(FILE *, int ***, system_t *);

/* dxwrite */
void write_frozen(FILE *fp_frozen, system_t *system);

#ifndef QM_ROTATION
/* quantum rotation */
void quantum_system_rotational_energies(system_t *);
void quantum_rotational_energies(system_t *, molecule_t *, int, int);
void quantum_rotational_grid(system_t *, molecule_t *);
complex_t **rotational_hamiltonian(system_t *, molecule_t *, int, int);
int determine_rotational_eigensymmetry(molecule_t *, int, int);
double rotational_basis(int, int, int, double, double);
double rotational_potential(system_t *, molecule_t *, double, double);
double hindered_potential(double);
double rotational_integrate(system_t *, molecule_t *, int, int, int, int, int);
#endif /* QM_ROTATION */

#ifndef DEBUG
void test_pairs(molecule_t *);
void test_molecule(molecule_t *);
void test_list(molecule_t *);
void test_cavity_grid(system_t *);
#endif /* DEBUG */

extern int rank, size;

#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <values.h>
#include <time.h>
#include <signal.h>

#define MAXLINE      512
#define MAXVALUE     1.0e200
#define DARTSCALE    0.1
#define QUANTUM_ROTATION_SYMMETRIC   0
#define QUANTUM_ROTATION_ANTISYMMETRIC 1
#define QUANTUM_ROTATION_SYMMETRY_POINTS 64

#define QUANTUM_ROTATION_LEVEL_MAX     36
#define QUANTUM_ROTATION_L_MAX        5
#define QUANTUM_ROTATION_GRID         16
#define QUANTUM_ROTATION_THETA_MAX    QUANTUM_ROTATION_GRID
#define QUANTUM_ROTATION_PHI_MAX      QUANTUM_ROTATION_GRID

#define FEYNMAN_KLEINERT_TOLERANCE   1.0e-12      /* tolerance in A^2 */

#include <math.h>

#ifndef MPI
#include <mpi.h>
#endif /* MPI */

#include <structs.h>
#include <physical_constants.h>
#include <function_prototypes.h>

#define ENSEMBLE_UVT      1
#define ENSEMBLE_NVT      2
#define ENSEMBLE_SURF     3
#define ENSEMBLE_SURF_FIT 4
#define ENSEMBLE_NVE      5

#define MOVETYPE_INSERT    1
#define MOVETYPE_REMOVE    2
#define MOVETYPE_DISPLACE   3
#define MOVETYPEADIABATIC 4

```

```

#define MAX_ITERATION_COUNT 128
#define DAMPING_LINEAR 1
#define DAMPING_EXPONENTIAL 2
#define EWALD_ALPHA 0.5
#define EWALD_KMAX 7
#define REAL 0
#define IMAGINARY 1

#ifndef M_PI
#define M_PI 3.14159265358979323846
#endif /* M_PI */

#define H 6.626068e-34 /* Planck's constant in J s */
#define HBAR 1.054571e-34 /* above divided by 2pi in J s */
#define KB 1.3806503e-23 /* Boltzmann's constant in J/K */
#define NA 6.0221415e23 /* Avogadro's number */
#define C 2.99792458e8 /* speed of light in vacuum in m/s */

/* H2 specific constants */
#define H2_MASS 3.348e-27 /* mass of H2 molecule in kg */
#define H2_REDUCED_MASS 0.25*H2_MASS /* reduced mass of H2 molecule in kg */
#define H2_De 4.72 /* dissociation energy in eV */
#define H2_R0 0.742 /* equilibrium bond distance in angstroms */
#define H2_SPRING_CONSTANT 35.2 /* harmonic spring constant at well minimum */

/* Silvera-Goldman parameters */
#define ALPHA 1.713 /* unitless */
#define BETA 1.5671 /* 1/a.u. */
#define GAMMA 0.00993 /* 1/a.u.^2 */
#define C6 12.14 /* multipole term1 a.u.^6 */
#define C8 215.2 /* multipole term2 a.u.^8 */
#define C10 4813.9 /* multipole term3 a.u.^10 */
#define C9 143.1 /* 3-body term a.u.^9 */
#define RM 8.321 /* position of max well depth (a.u.) times 1.28 */

/* conversion factors */
#define AU2ANGSTROM 0.529177249 /* convert from Bohr radii to angstroms */
#define METER2ANGSTROM 1.0e10 /* convert from meters to angstroms */
#define HARTREE2KELVIN 3.15774655e5 /* convert from Hartrees to Kelvin */
#define E2REDUCED 408.7816 /* convert from e to sqrt(K*A) */
#define ATM2REDUCED 0.0073389366 /* convert from atm to K/A^3 */
#define ATM2PASCALS 101325.0 /* convert from atm to Pascals */
#define ATM2PSI 14.6959488 /* convert from atm to psi */
#define A3CM3 1.0e-24 /* convert from A^3 to cm^3 */
#define AMU2KG 1.66053873e-27 /* convert amu's to kg */
#define DEBYE2SKA 85.10597636 /* convert from Debye to sqrt(KA)*A */
#define EV2K 1.160444e4 /* convert eV to K */
#define K2WN 0.695039 /* convert K to cm^-1 */

/* complex value */
typedef struct _complex_t {
    double real;
    double imaginary;
} complex_t;

typedef struct _pair {
    int frozen;
    int rd_excluded, es_excluded;
    double lrc;
    double charge;
    double polarizability;
    double epsilon;
    double sigma;
    double r;
    double d[3];
    double d_prev[3];
    int recalculate_energy;
    double ring;
    double dimg[3];
    double r2, r3, r5; /* 2nd, 3rd and 5th powers for A matrix calc */
    double r2img, r3img, r5img;
    double rd_energy, es_real_energy, es_self_intra_energy;
    struct _atom *atom;
    struct _molecule *molecule;
    struct _pair *next;
} pair_t;

typedef struct _atom {
    int id;
    char atomtype[MAXLINE];
    int frozen, adiabatic;
    double mass;
    double charge;
    double polarizability;
    double epsilon;
    double sigma;
    double es_self_point_energy;
    double pos[3];
    double wrapped_pos[3];
    double ef_static[3];
    double ef_induced[3];
    double ef_induced_change[3];
    double mu[3];
    double old_mu[3];
    double new_mu[3];
    double dipole_rrms;
} atom_t;

```

```

    double rank_metric;
    pair_t *pairs;
    struct _atom *next;
} atom_t;

typedef struct _molecule {
    int id;
    char moleculetype[MAXLINE];
    double mass;
    int frozen, adiabatic;
    double com[3];
    double wrapped_com[3];
#ifndef QM_ROTATION
    double *quantum_rotational_energies;
    complex_t **quantum_rotational_eigenvectors;
    int *quantum_rotational_eigensymmetry;
    double quantum_rotational_potential_grid[QUANTUM_ROTATION_GRID][QUANTUM_ROTATION_GRID];
#endif /* QM_ROTATION */
#ifndef XXX
/* XXX - vib work in progress */
    double *quantum_vibrational_energies;
    complex_t **quantum_vibrational_eigenvectors;
    int *quantum_vibrational_eigensymmetry;
#endif /* XXX */
    atom_t *atoms;
    struct _molecule *next;
} molecule_t;

typedef struct _pbc {
    double basis[3][3]; /* unit cell lattice (A) */
    double reciprocal_basis[3][3]; /* reciprocal space lattice (1/A) */
    double cutoff; /* radial cutoff (A) */
    double volume; /* unit cell volume (A^3) */
} pbc_t;

typedef struct _cavity {
    int occupancy;
    double pos[3];
} cavity_t;

typedef struct _histogram {
    int ***grid;
    int x_dim,y_dim,z_dim;
    double origin[3];
    double delta[3][3];
    int count[3];
    int n_data_points;
    int norm_total;
} histogram_t;

/* used in quantum rotation code */
typedef struct _spherical_harmonic {

    double legendre; /* legendre polynomial part */
    double real; /* real part of the spherical harmonic */
    double imaginary; /* imaginary part of the spherical harmonic */
} spherical_harmonic_t;

typedef struct _nodestats {

    int accept, reject;
    int accept_insert, reject_insert;
    int accept_remove, reject_remove;
    int accept_displace, reject_displace;
    int accept_adiabatic, reject_adiabatic;

    double boltzmann_factor;
    double acceptance_rate;
    double acceptance_rate_insert;
    double acceptance_rate_remove;
    double acceptance_rate_displace;
    double acceptance_rate_adiabatic;
    double cavity_bias_probability;
    double polarization_iterations;
} nodestats_t;

typedef struct _avg_nodestats {

    double boltzmann_factor;
    double boltzmann_factor_sq;

    double acceptance_rate;
    double acceptance_rate_insert;
    double acceptance_rate_remove;
    double acceptance_rate_displace;
    double acceptance_rate_adiabatic;

    double cavity_bias_probability;
    double cavity_bias_probability_sq;

    double polarization_iterations;
    double polarization_iterations_sq;
} avg_nodestats_t;

typedef struct _observables {

    double energy;
    double coulombic_energy;
    double rd_energy;
    double polarization_energy;
    double dipole_rrms;
}

```

```

    double kinetic_energy; /* for NVE */
    double temperature; /* for NVE */
    double N;
    double NU;

} observables_t;

typedef struct _avg_observables {

/* these are uncorrelated averages of the observables */
    double energy;
    double energy_sq;
    double energy_error;

/* needed for heat capacity error propagation */
    double energy_sq_sq;
    double energy_sq_error;

    double coulombic_energy;
    double coulombic_energy_sq;
    double coulombic_energy_error;

    double rd_energy;
    double rd_energy_sq;
    double rd_energy_error;

    double polarization_energy;
    double polarization_energy_sq;
    double polarization_energy_error;

    double dipole_rrms;
    double dipole_rrms_sq;
    double dipole_rrms_error;

/* for NVE MC */
    double kinetic_energy;
    double kinetic_energy_sq;
    double kinetic_energy_error;

    double temperature;
    double temperature_sq;
    double temperature_error;

    double N;
    double N_sq;
    double N_error;

/* needed for qst */
    double NU;

/* these quantities are node stats, not observables */
    double boltzmann_factor;
    double boltzmann_factor_sq;
    double boltzmann_factor_error;

    double acceptance_rate;
    double acceptance_rate_insert;
    double acceptance_rate_remove;
    double acceptance_rate_displace;
    double acceptance_rate_adiabatic;

    double cavity_bias_probability;
    double cavity_bias_probability_sq;
    double cavity_bias_probability_error;

    double polarization_iterations;
    double polarization_iterations_sq;
    double polarization_iterations_error;

    double qst;
    double qst_sq;
    double qst_error;

    double heat_capacity;
    double heat_capacity_sq;
    double heat_capacity_error;

    double compressibility;
    double compressibility_sq;
    double compressibility_error;

/* the error of these propagates */
    double density;
    double density_error;

    double pore_density;
    double pore_density_error;

    double percent_wt;
    double percent_wt_error;

    double percent_wt_me;
    double percent_wt_me_error;

    double excess_ratio;
    double excess_ratio_error;

} avg_observables_t;

typedef struct _grid {
    histogram_t *histogram;
    histogram_t *avg_histogram;
} grid_t;

/* begin mpi message struct */
typedef struct _message {

```

```

/* observables */
double energy;
double coulombic_energy;
double rd_energy;
double polarization_energy;
double kinetic_energy;
double temperature;
double N;
double NU;
/* avg_nodestats */
double boltzmann_factor;
double boltzmann_factor_sq;

double acceptance_rate;
double acceptance_rate_insert;
double acceptance_rate_remove;
double acceptance_rate_displace;

double cavity_bias_probability;
double cavity_bias_probability_sq;

double polarization_iterations;
double polarization_iterations_sq;

/* histogram */
int ***grid;
} message_t;

typedef struct _checkpoint {
    int movetype, biased_move;
    int N_atom, N_atom_prev;
    molecule_t *molecule_backup, *molecule_altered;
    molecule_t *head, *tail;
    observables_t *observables;
} checkpoint_t;

typedef struct _param {
    char atomtype[MAXLINE];
    int ntypes;
    double charge;
    double epsilon;
    double sigma;
    double dr;
    int axis;
    double last_charge;
    double last_epsilon;
    double last_sigma;
    double last_dr;
    struct _param *next;
} param_t;

typedef struct _file_pointers {
    FILE *fp_energy;
    FILE *fp_traj;
    FILE *fp_dipole;
    FILE *fp_field;
    FILE *fp_histogram;
    FILE *fp_frozen;
} file_pointers_t;

typedef struct _system {
    int ensemble;
    double surf_min, surf_max, surf_inc, surf_ang;
    int surf_preserve, surf_decomp;
    long int seed;
    int numsteps, corrtime;
    double move_probability, rot_probability, insert_probability, adiabatic_probability;
    int cavity_bias, cavity_grid_size;
    cavity_t ***cavity_grid;
    int cavities_open;
    double cavity_radius, cavity_volume;
    int cavity_autoreject;
    double cavity_autoreject_scale;
    double temperature, pressure, fugacity, free_volume, total_energy, N;
    int simulated_annealing;
    double simulated_annealing_schedule;
    int rd_only, rd_anharmonic, rd_lrc;
    int h2_fugacity, co2_fugacity;
    double rd_anharmonic_k, rd_anharmonic_g;
    int feynman_hibbs, feynman_kleinert, feynman_hibbs_order;
    int sg, fvm, wpi, wpi_grid;
    int wrapall;
    double scale_charge, scale_rd;
    double evald_alpha;
    int evald_kmax;
    int polarization, polarizability_tensor;
    int polar_iterative, polar_ewald, polar_zodid, polar_self, polar_palmo, polar_gs, polar_gs_ranked, polar_sor, polar_esor,
    polar_max_iter;
    double polar_gamma;
    double polar_damp, field_damp, polar_precision;
    int quantum_rotation, quantum_rotation_hindered;
    double quantum_rotation_B;
    double quantum_rotation_hindered_barrier;
    int quantum_rotation_level_max, quantum_rotation_l_max, quantum_rotation_theta_max, quantum_rotation_phi_max;
    int quantum_vibration;
    int damp_type;
    double **A_matrix, **B_matrix, C_matrix[3][3]; /* A matrix, B matrix and polarizability tensor */
    char *pdb_input, *pdb_output, *pdb_restart, *traj_output, *energy_output;
    char *dipole_output, *field_output, *histogram_output, *frozen_output;
    char *fit_input;

    grid_t *grids;
    int calc_hist; /* flag to calculate a 3D histogram */
}

```

```

double hist_resolution;
int n_histogram_bins;

double max_bondlength; /* threshold to bond (re:output files) */

pbct *pbct;
molecule_t *molecules;

nodestats_t *nodestats;
avg_nodestats_t *avg_nodestats;
observables_t *observables;
avg_observables_t *avg_observables;

checkpoint_t *checkpoint;

file_pointers_t file_pointers;

} system_t;

/*
@2007, Jonathan Belof
Space Research Group
Department of Chemistry
University of South Florida
*/
#include <mc.h>

void clear_nodestats(nodestats_t *stats) {
    memset(stats, 0, sizeof(nodestats_t));
}

void clear_node_averages(avg_nodestats_t *avgstats) {
    memset(avgstats, 0, sizeof(avg_nodestats_t));
}

void clear_observables(observables_t *obs) {
    memset(obs, 0, sizeof(observables_t));
}

void clear_root_averages(avg_observables_t *avgobs) {
    memset(avgobs, 0, sizeof(avg_observables_t));
}

/* determine the acceptance rate */
void track_ar(nodestats_t *ns) {

    if(ns->accept + ns->reject)
        ns->acceptance_rate = ((double)ns->accept) / ((double)(ns->accept + ns->reject));
    else
        ns->acceptance_rate = 0;

    if(ns->accept_insert + ns->reject_insert)
        ns->acceptance_rate_insert = ((double)ns->accept_insert) / ((double)(ns->accept_insert + ns->reject_insert));
    else
        ns->acceptance_rate_insert = 0;

    if(ns->accept_remove + ns->reject_remove)
        ns->acceptance_rate_remove = ((double)ns->accept_remove) / ((double)(ns->accept_remove + ns->reject_remove));
    else
        ns->acceptance_rate_remove = 0;

    if(ns->accept_displace + ns->reject_displace)
        ns->acceptance_rate_displace = ((double)ns->accept_displace) / ((double)(ns->accept_displace + ns->reject_displace));
    else
        ns->acceptance_rate_displace = 0;

    if(ns->accept_adiabatic + ns->reject_adiabatic)
        ns->acceptance_rate_adiabatic = ((double)ns->accept_adiabatic) / ((double)(ns->accept_adiabatic + ns->reject_adiabatic));
    else
        ns->acceptance_rate_adiabatic = 0;
}

/* update node statistics related to the processing */
void update_nodestats(nodestats_t *nodestats, avg_nodestats_t *avg_nodestats) {

    static int counter = 0;
    double factor;

    ++counter;
    factor = ((double)(counter - 1)) / ((double)(counter));

    avg_nodestats->boltzmann_factor = factor * avg_nodestats->boltzmann_factor + nodestats->boltzmann_factor / ((double)counter);
    avg_nodestats->boltzmann_factor_sq = factor * avg_nodestats->boltzmann_factor_sq +
    nodestats->boltzmann_factor * nodestats->boltzmann_factor / ((double)counter);

    /* these really aren't averages, but accumulative values */
    avg_nodestats->acceptance_rate = nodestats->acceptance_rate;
    avg_nodestats->acceptance_rate_insert = nodestats->acceptance_rate_insert;
    avg_nodestats->acceptance_rate_remove = nodestats->acceptance_rate_remove;
    avg_nodestats->acceptance_rate_displace = nodestats->acceptance_rate_displace;
    avg_nodestats->acceptance_rate_adiabatic = nodestats->acceptance_rate_adiabatic;

    avg_nodestats->cavity_bias_probability = factor * avg_nodestats->cavity_bias_probability + nodestats->cavity_bias_probability /
}

```

```

((double)counter);
    avg_nodestats->cavity_bias_probability_sq = factor*avg_nodestats->cavity_bias_probability_sq +
nodestats->cavity_bias_probability*nodestats->cavity_bias_probability / ((double)counter);

    avg_nodestats->polarization_iterations = factor*avg_nodestats->polarization_iterations + nodestats->polarization_iterations /
((double)counter);
    avg_nodestats->polarization_iterations_sq = factor*avg_nodestats->polarization_iterations_sq +
nodestats->polarization_iterations*nodestats->polarization_iterations / ((double)counter);

}

void update_root_averages(system_t *system, observables_t *observables, avg_nodestats_t *avg_nodestats, avg_observables_t *avg_observables)
{
    double particle_mass, frozen_mass;

    molecule_t *molecule_ptr;
    static int counter = 0;
    double factor;

    ++counter;
    factor = ((double)(counter - 1))/((double)(counter));

    /* the physical observables */
    avg_observables->energy = factor*avg_observables->energy + observables->energy / ((double)counter);
    avg_observables->energy_sq = factor*avg_observables->energy_sq + (observables->energy*observables->energy) / ((double)counter);
    avg_observables->energy_error = 0.5*sqrt(avg_observables->energy_sq - avg_observables->energy*avg_observables->energy);

    avg_observables->energy_sq_sq = factor*avg_observables->energy_sq_sq + pow(observables->energy, 4.0) / ((double)counter);
    avg_observables->energy_sq_error = 0.5*sqrt(avg_observables->energy_sq_sq - pow(observables->energy, 4.0));

    avg_observables->coulombic_energy = factor*avg_observables->coulombic_energy + observables->coulombic_energy / ((double)counter);
    avg_observables->coulombic_energy_sq = factor*avg_observables->coulombic_energy_sq +
observables->coulombic_energy*observables->coulombic_energy) / ((double)counter);
    avg_observables->coulombic_energy_error = 0.5*sqrt(avg_observables->coulombic_energy_sq -
avg_observables->coulombic_energy*avg_observables->coulombic_energy);

    avg_observables->rd_energy = factor*avg_observables->rd_energy + observables->rd_energy / ((double)counter);
    avg_observables->rd_energy_sq = factor*avg_observables->rd_energy_sq + (observables->rd_energy*observables->rd_energy) /
((double)counter);
    avg_observables->rd_energy_error = 0.5*sqrt(avg_observables->rd_energy_sq - avg_observables->rd_energy*avg_observables->rd_energy);

    avg_observables->polarization_energy = factor*avg_observables->polarization_energy + observables->polarization_energy / ((double)counter);
    avg_observables->polarization_energy_sq = factor*avg_observables->polarization_energy_sq +
(observables->polarization_energy*observables->polarization_energy) / ((double)counter);
    avg_observables->polarization_energy_error = 0.5*sqrt(avg_observables->polarization_energy_sq -
avg_observables->polarization_energy*avg_observables->polarization_energy);

    avg_observables->dipole_rrms = factor*avg_observables->dipole_rrms + observables->dipole_rrms / ((double)counter);
    avg_observables->dipole_rrms_sq = factor*avg_observables->dipole_rrms_sq + (observables->dipole_rrms*observables->dipole_rrms) /
((double)counter);
    avg_observables->dipole_rrms_error = 0.5*sqrt(avg_observables->dipole_rrms_sq -
avg_observables->dipole_rrms*avg_observables->dipole_rrms);

    avg_observables->kinetic_energy = factor*avg_observables->kinetic_energy + observables->kinetic_energy / ((double)counter);
    avg_observables->kinetic_energy_sq = factor*avg_observables->kinetic_energy_sq +
(observables->kinetic_energy*observables->kinetic_energy) / ((double)counter);
    avg_observables->kinetic_energy_error = 0.5*sqrt(avg_observables->kinetic_energy_sq -
avg_observables->kinetic_energy*avg_observables->kinetic_energy);

    avg_observables->temperature = factor*avg_observables->temperature + observables->temperature / ((double)counter);
    avg_observables->temperature_sq = factor*avg_observables->temperature_sq + (observables->temperature*observables->temperature) /
((double)counter);
    avg_observables->temperature_error = 0.5*sqrt(avg_observables->temperature_sq -
avg_observables->temperature*avg_observables->temperature);

    avg_observables->N = factor*avg_observables->N + observables->N / ((double)counter);
    avg_observables->N_sq = factor*avg_observables->N_sq + (observables->N*observables->N) / ((double)counter);
    avg_observables->N_error = 0.5*sqrt(avg_observables->N_sq - avg_observables->N*avg_observables->N);

    avg_observables->NU = factor*avg_observables->NU + observables->NU / ((double)counter);

    /* avg in nodestats */
    avg_observables->boltzmann_factor = factor*avg_observables->boltzmann_factor + avg_nodestats->boltzmann_factor / ((double)counter);
    avg_observables->boltzmann_factor_sq = factor*avg_observables->boltzmann_factor_sq + avg_nodestats->boltzmann_factor_sq /
((double)counter);
    avg_observables->boltzmann_factor_error = 0.5*sqrt(avg_observables->boltzmann_factor_sq -
avg_observables->boltzmann_factor*avg_observables->boltzmann_factor);

    avg_observables->acceptance_rate = factor*avg_observables->acceptance_rate + avg_nodestats->acceptance_rate / ((double)counter);
    avg_observables->acceptance_rate_insert = factor*avg_observables->acceptance_rate_insert + avg_nodestats->acceptance_rate_insert /
((double)counter);
    avg_observables->acceptance_rate_remove = factor*avg_observables->acceptance_rate_remove + avg_nodestats->acceptance_rate_remove /
((double)counter);
    avg_observables->acceptance_rate_displace = factor*avg_observables->acceptance_rate_displace +
avg_nodestats->acceptance_rate_displace / ((double)counter);
    avg_observables->acceptance_rate_adiabatic = factor*avg_observables->acceptance_rate_adiabatic +
avg_nodestats->acceptance_rate_adiabatic / ((double)counter);

    avg_observables->cavity_bias_probability = factor*avg_observables->cavity_bias_probability + avg_nodestats->cavity_bias_probability /
((double)counter);
    avg_observables->cavity_bias_probability_sq = factor*avg_observables->cavity_bias_probability_sq +
avg_nodestats->cavity_bias_probability_sq / ((double)counter);
    avg_observables->cavity_bias_probability_error = 0.5*sqrt(avg_observables->cavity_bias_probability_sq -
avg_observables->cavity_bias_probability*avg_observables->cavity_bias_probability);

    avg_observables->polarization_iterations = factor*avg_observables->polarization_iterations + avg_nodestats->polarization_iterations /
((double)counter);
    avg_observables->polarization_iterations_sq = factor*avg_observables->polarization_iterations_sq +
avg_nodestats->polarization_iterations_sq / ((double)counter);

```

```

avg_observables->polarization_iterations_error = 0.5*sqrt(avg_observables->polarization_iterations_sq -
avg_observables->polarization_iterations*avg_observables->polarization_iterations);

/* get the mass of the two phases */
for(molecule_ptr = system->molecules, particle_mass = 0, frozen_mass = 0; molecule_ptr; molecule_ptr = molecule_ptr->next) {
    if(molecule_ptr->frozen || molecule_ptr->adiabatic)
        frozen_mass += molecule_ptr->mass;
    else
        particle_mass = molecule_ptr->mass;
}

/* density in g/cm^3 */
avg_observables->density = avg_observables->N*particle_mass/(system->pb->volume*NA*A32CM3);
avg_observables->density_error = avg_observables->N_error*particle_mass/(system->pb->volume*NA*A32CM3);

/* heat capacity in kJ/mol K */
avg_observables->heat_capacity = (KB*NA/1000.0)*(avg_observables->energy_sq -
avg_observables->energy*avg_observables->energy)/(system->temperature*system->temperature);
avg_observables->heat_capacity_sq = factor*avg_observables->heat_capacity_sq +
(avg_observables->heat_capacity*avg_observables->heat_capacity) / ((double)counter);
avg_observables->heat_capacity_error = 0.5*sqrt(avg_observables->heat_capacity_sq -
avg_observables->heat_capacity*avg_observables->heat_capacity);

/* compressibility */
avg_observables->compressibility = ATM2PASCALS*(system->pb->volume/pow(METER2ANGSTROM, 3.0))*(avg_observables->N_sq -
avg_observables->N)*avg_observables->N)/(KB*system->temperature*avg_observables->N*avg_observables->N);
avg_observables->compressibility_sq = factor*avg_observables->compressibility_sq +
(avg_observables->compressibility*avg_observables->compressibility) / ((double)counter);
avg_observables->compressibility_error = 0.5*sqrt(avg_observables->compressibility_sq -
avg_observables->compressibility*avg_observables->compressibility);

/* we have a solid phase */
if(frozen_mass > 0.0) {

    /* percent weight */
    avg_observables->percent_wt = 100.0*avg_observables->N*particle_mass/(frozen_mass + avg_observables->N*particle_mass);
    avg_observables->percent_wt_error = 100.0*avg_observables->N_error*particle_mass/(frozen_mass +
avg_observables->N_error*particle_mass);

    /* percent weight like ME*/
    avg_observables->percent_wt_me = 100.0*avg_observables->N*particle_mass/frozen_mass;
    avg_observables->percent_wt_me_error = 100.0*avg_observables->N_error*particle_mass/frozen_mass;

    /* excess weight mg/g */
    if(system->free_volume > 0.0) {

        if(system->fugacity > 0.0)
            avg_observables->excess_ratio = 1000.0*(avg_observables->N*particle_mass -
(particle_mass*system->free_volume*system->fugacity*ATM2REDUCED)/system->temperature)/frozen_mass;
        else
            avg_observables->excess_ratio = 1000.0*(avg_observables->N*particle_mass -
(particle_mass*system->free_volume*system->pressure*ATM2REDUCED)/system->temperature)/frozen_mass;
        avg_observables->excess_ratio_error = 1000.0*avg_observables->N_error*particle_mass/frozen_mass;

        /* pore density */
        avg_observables->pore_density = avg_observables->N*particle_mass/(system->free_volume*NA*A32CM3);
        avg_observables->pore_density_error = avg_observables->N_error*particle_mass/(system->free_volume*NA*A32CM3);
    }
}

/* calculate the isosteric heat */
avg_observables->qst = -(avg_observables->NU - avg_observables->N*avg_observables->energy);
avg_observables->qst /= (avg_observables->N_sq - avg_observables->N*avg_observables->N);
avg_observables->qst += system->temperature;
avg_observables->qst *= KB*NA/1000.0; /* convert to kJ/mol */

avg_observables->qst_sq = factor*avg_observables->qst_sq + (avg_observables->qst*avg_observables->qst) / ((double)counter);
avg_observables->qst_error = 0.5*sqrt(avg_observables->qst_sq - avg_observables->qst*avg_observables->qst);
}

}

#include <mc.h>

#define COLOR_H "0.2 0.2 0.2"
#define COLOR_C "0.1 0.5 0.1"
#define COLOR_N "0.2 0.2 1.0"
#define COLOR_O "1.0 0.0 0.0"
#define COLOR_XXX "0.1 0.1 0.1"

void print_frozen_coords(FILE *fp_frozen, system_t *system)
{
    molecule_t *mol;
    atom_t *atom;

    for(mol = system->molecules; mol; mol=mol->next){
        if(mol->frozen){
            for(atom = mol->atoms; atom; atom=atom->next){
                fprintf(fp_frozen,"%f %f %f\n",atom->pos[0],atom->pos[1],atom->pos[2]);
            }
        }
    }
}

int count_frozen(system_t *system)
{
    int count=0;
    molecule_t *mol;
    atom_t *atom;

    for(mol = system->molecules; mol; mol=mol->next){
        if(mol->frozen){

```

```

        for(atom = mol->atoms; atom; atom=atom->next) count++;
    }
}
return count;
}

int bondlength_check(atom_t *atom1, atom_t *atom2, system_t *system)
{
    double distance;
    double gm_mass; /* geometric mean of mass */
    int is_bonded;
    double slope=0.0234; /* tune to meet bond length expectations */
    double yint=0.603; /* tune to meet bond length expectations */

    gm_mass=sqrt(atom1->mass * atom2->mass);
    distance=sqrt( pow(atom1->pos[0]-atom2->pos[0],2) + pow(atom1->pos[1] - atom2->pos[1],2) + pow(atom1->pos[2] - atom2->pos[2],2));
    if( distance < (gm_mass * slope + yint) * system->max_bondlength ) is_bonded=1;
    else is_bonded=0;
    return is_bonded;
}

int calculate_bonds(system_t *system)
{
    int inner_index=0,outer_index=0;
    int bonds=0;
    double bondlength;
    molecule_t *mol;
    atom_t *atom;
    atom_t *atom2;
    double *pos1, *pos2;

    for(mol=system->molecules; mol; mol=mol->next){
        if(mol->frozen){
            for(atom=mol->atoms; atom; atom=atom->next, inner_index++){
                pos1=atom->pos;
                if(atom->next){
                    for(atom2=atom->next, outer_index=inner_index+1; atom2; atom2=atom2->next, outer_index++){
                        pos2=atom2->pos;
                        if(bondlength_check(atom,atom2,system)){
                            bonds++;
                        }
                    }
                }
            }
        }
    }
    return bonds;
}

void print_frozen_bonds(FILE *fp_frozen, system_t *system)
{
    int inner_index=0,outer_index=0;
    int bonds=0;
    double bondlength;
    molecule_t *mol;
    atom_t *atom;
    atom_t *atom2;
    double *pos1, *pos2;

    for(mol=system->molecules; mol; mol=mol->next){
        if(mol->frozen){
            for(atom=mol->atoms; atom; atom=atom->next, inner_index++){
                pos1=atom->pos;
                if(atom->next){
                    for(atom2=atom->next, outer_index=inner_index+1; atom2; atom2=atom2->next, outer_index++){
                        pos2=atom2->pos;
                        if(bondlength_check(atom,atom2,system)){
                            bonds++;
                            fprintf(fp_frozen,"%d %d\n",inner_index,outer_index);
                        }
                    }
                }
            }
        }
    }
}

void print_frozen_masses(FILE *fp_frozen, system_t *system)
{
    molecule_t *mol;
    atom_t *atom;

    for(mol=system->molecules; mol; mol=mol->next){
        if(mol->frozen){
            for(atom=mol->atoms; atom; atom=atom->next){
                fprintf(fp_frozen,"%f\n",atom->mass);
            }
        }
    }
}

void print_frozen_colors(FILE *fp_frozen, system_t *system)
{
    double mass;
    molecule_t *mol;
    atom_t *atom;

    for(mol=system->molecules; mol; mol=mol->next){
        if(mol->frozen){
            for(atom=mol->atoms; atom; atom=atom->next){
                mass=atom->mass;
                if(mass < 1.1) fprintf(fp_frozen,"%s\n", COLOR_H);
                else if(mass < 12.2) fprintf(fp_frozen,"%s\n", COLOR_C);
                else if(mass < 14.1) fprintf(fp_frozen,"%s\n", COLOR_N);
            }
        }
    }
}

```

```

        else if(mass < 16.1) fprintf(fp_frozen,"%s\n", COLOR_O);
        else fprintf(fp_frozen,"%s\n", COLOR_XXX);
    }
}

void write_frozen(FILE *fp_frozen, system_t *system)
{
    int i,j,k;
    int numatoms;
    int numbonds;

    numatoms=count_frozen(system);
    numbonds=calculate_bonds(system);

    rewind(fp_frozen);
    fprintf(fp_frozen,"# OpenDX format coordinate file for frozen atoms\n");
    fprintf(fp_frozen,"object 1 class array type float rank 1 shape 3 items %d data follows\n",numatoms);
    print_frozen_coords(fp_frozen,system);
    fprintf(fp_frozen,"object 2 class array type int rank 1 shape 2 items %d data follows\n",numbonds);
    print_frozen_bonds(fp_frozen,system);
    fprintf(fp_frozen,"attribute \"element type\" string \"lines\"\n");
    fprintf(fp_frozen,"attribute \"ref\" string \"positions\"\n");
    fprintf(fp_frozen,"object 3 class array type float rank 0 items %d data follows\n",numatoms);
    print_frozen_masses(fp_frozen,system);
    fprintf(fp_frozen,"attribute \"dep\" string \"positions\"\n");
    fprintf(fp_frozen,"object 4 class array type float rank 1 shape 3 items %d data follows\n",numatoms);
    print_frozen_colors(fp_frozen,system);
    fprintf(fp_frozen,"object \"irregular positions irregular connections\" class field\n");
    fprintf(fp_frozen,"component \"positions\" value 1\n");
    fprintf(fp_frozen,"component \"connections\" value 2\n");
    fprintf(fp_frozen,"component \"data\" value 3\n");
    fprintf(fp_frozen,"component \"colors\" value 4\n");
    fprintf(fp_frozen,"end\n");
}

/*
@2007, Jonathan Belof
Space Research Group
Department of Chemistry
University of South Florida
*/
#include <mc.h>

system_t *read_config(char *input_file) {
    system_t *system;
    char linebuffer[MAXLINE], *n;
    char token1[MAXLINE], token2[MAXLINE], token3[MAXLINE], token4[MAXLINE];
    FILE *fp;

    system = calloc(1, sizeof(system_t));
    if(!system) {
        error("INPUT: couldn't allocate system data structure\n");
        return(NULL);
    }

    system->pbc = calloc(1, sizeof(pbc_t));
    if(!system->pbc) {
        error("INPUT: couldn't allocate the PBC data structure\n");
        return(NULL);
    }

    /* open the config file or error */
    fp = fopen(input_file, "r");
    if(!fp) {
        error("INPUT: couldn't open config file\n");
        return(NULL);
    }

    /* set the default scaling to 1 */
    system->scale_charge = 1.0;
    system->scale_rd = 1.0;
    system->rot_probability = 1.0;

    /* set histogram flag default: off */
    system->calc_hist=0;
    system->hist_resolution=0.0;
    system->histogram_output=NULL;

    /* default ewald parameters */
    system->ewald_alpha = EWALD_ALPHA;
    system->ewald_kmax = EWALD_KMAX;

    /* default polarization parameters */
    system->polar_gamma = 1.0;

    /* default rd LRC flag */
    system->rd_lrc = 1;

#ifndef QM_ROTATION
    /* default QR parameters */
    system->quantum_rotation_level_max = QUANTUM_ROTATION_LEVEL_MAX;
    system->quantum_rotation_l_max = QUANTUM_ROTATION_L_MAX;
    system->quantum_rotation_theta_max = QUANTUM_ROTATION_THETA_MAX;
    system->quantum_rotation_phi_max = QUANTUM_ROTATION_PHI_MAX;
#endif /* QM_ROTATION */

    /* loop over each line */
    memset(linebuffer, 0, MAXLINE);
    n = fgets(linebuffer, MAXLINE, fp);
}

```

```

while(n) {

    /* grab a line and parse it out */
    memset(token1, 0, MAXLINE);
    memset(token2, 0, MAXLINE);
    memset(token3, 0, MAXLINE);
    memset(token4, 0, MAXLINE);
    sscanf(linebuffer, "%s %s %s %s", token1, token2, token3, token4);

    if(!strcasecmp(token1, "ensemble")) {
        if(!strcasecmp(token2, "nvt"))
            system->ensemble = ENSEMBLE_NVT;
        else if(!strcasecmp(token2, "uvt"))
            system->ensemble = ENSEMBLE_UVT;
        else if(!strcasecmp(token2, "surf"))
            system->ensemble = ENSEMBLE_SURF;
        else if(!strcasecmp(token2, "surf_fit"))
            system->ensemble = ENSEMBLE_SURF_FIT;
        else if(!strcasecmp(token2, "nve"))
            system->ensemble = ENSEMBLE_NVE;
    }

    /* for NVE only */
    if(!strcasecmp(token1, "total_energy"))
        system->total_energy = atof(token2);

    if(!strcasecmp(token1, "surf_decomp")) {
        if(!strcasecmp(token2, "on"))
            system->surf_decomp = 1;
        else
            system->surf_decomp = 0;
    }

    if(!strcasecmp(token1, "surf_min"))
        system->surf_min = atof(token2);

    if(!strcasecmp(token1, "surf_max"))
        system->surf_max = atof(token2);

    if(!strcasecmp(token1, "surf_inc"))
        system->surf_inc = atof(token2);

    if(!strcasecmp(token1, "surf_ang"))
        system->surf_ang = atof(token2);

    if(!strcasecmp(token1, "surf_preserve")) {
        if(!strcasecmp(token2, "on"))
            system->surf_preserve = 1;
        else
            system->surf_preserve = 0;
    }

    if(!strcasecmp(token1, "seed"))
        system->seed = atol(token2);

    if(!strcasecmp(token1, "numsteps"))
        system->numsteps = atoi(token2);

    if(!strcasecmp(token1, "corrtime"))
        system->corrtime = atoi(token2);

    if(!strcasecmp(token1, "move_probability"))
        system->move_probability = atof(token2);

    if(!strcasecmp(token1, "rot_probability"))
        system->rot_probability = atof(token2);

    if(!strcasecmp(token1, "adiabatic_probability"))
        system->adiabatic_probability = atof(token2);

    if(!strcasecmp(token1, "insert_probability"))
        system->insert_probability = atof(token2);

    if(!strcasecmp(token1, "cavity_bias")) {
        if(!strcasecmp(token2, "on"))
            system->cavity_bias = 1;
        else
            system->cavity_bias = 0;
    }

    if(!strcasecmp(token1, "cavity_grid"))
        system->cavity_grid_size = atoi(token2);

    if(!strcasecmp(token1, "cavity_radius"))
        system->cavity_radius = atof(token2);

    if(!strcasecmp(token1, "temperature"))
        system->temperature = atof(token2);

    if(!strcasecmp(token1, "simulated_annealing")) {
        if(!strcasecmp(token2, "on"))
            system->simulated_annealing = 1;
        else
            system->simulated_annealing = 0;
    }

    if(!strcasecmp(token1, "simulated_annealing_schedule"))
        system->simulated_annealing_schedule = atof(token2);

    if(!strcasecmp(token1, "pressure"))
        system->pressure = atof(token2);

    if(!strcasecmp(token1, "h2_fugacity")) {
        if(!strcasecmp(token2, "on"))
            system->h2_fugacity = 1;
        else

```

```

        system->h2_fugacity = 0;
    }

    if(!strcasecmp(token1, "co2_fugacity")) {
        if(!strcasecmp(token2, "on"))
            system->co2_fugacity = 1;
        else
            system->co2_fugacity = 0;
    }
    if(!strcasecmp(token1, "free_volume"))
        system->free_volume = atof(token2);

    if(!strcasecmp(token1, "rd_only")) {
        if(!strcasecmp(token2, "on"))
            system->rd_only = 1;
        else
            system->rd_only = 0;
    }

    if(!strcasecmp(token1, "rd_lrc")) {
        if(!strcasecmp(token2, "on"))
            system->rd_lrc = 1;
        else
            system->rd_lrc = 0;
    }

    if(!strcasecmp(token1, "rd_anharmonic")) {
        if(!strcasecmp(token2, "on"))
            system->rd_anharmonic = 1;
        else
            system->rd_anharmonic = 0;
    }

    if(!strcasecmp(token1, "rd_anharmonic_k"))
        system->rd_anharmonic_k = atof(token2);

    if(!strcasecmp(token1, "rd_anharmonic_g"))
        system->rd_anharmonic_g = atof(token2);

    if(!strcasecmp(token1, "feynman_hibbs")) {
        if(!strcasecmp(token2, "on"))
            system->feynman_hibbs = 1;
        else
            system->feynman_hibbs = 0;
    }

    if(!strcasecmp(token1, "feynman_kleinert")) {
        if(!strcasecmp(token2, "on"))
            system->feynman_kleinert = 1;
        else
            system->feynman_kleinert = 0;
    }

    if(!strcasecmp(token1, "feynman_hibbs_order")) {
        system->feynman_hibbs_order = atoi(token2);
    }

    if(!strcasecmp(token1, "wpi")) {
        if(!strcasecmp(token2, "on"))
            system->wpi = 1;
        else
            system->wpi = 0;
    }

    if(!strcasecmp(token1, "wpi_grid"))
        system->wpi_grid = atoi(token2);

    if(!strcasecmp(token1, "fvm")) {
        if(!strcasecmp(token2, "on"))
            system->fvm = 1;
        else
            system->wpi = 0;
    }

    if(!strcasecmp(token1, "sg")) {
        if(!strcasecmp(token2, "on"))
            system->sg = 1;
        else if(!strcasecmp(token2, "off"))
            system->sg = 0;
    }

    if(!strcasecmp(token1, "wrapall")) {
        if(!strcasecmp(token2, "on"))
            system->wrapall = 1;
        else
            system->wrapall = 0;
    }

    if(!strcasecmp(token1, "scale_charge")) {
        system->scale_charge = atof(token2);
    }

    if(!strcasecmp(token1, "scale_rd")) {
        system->scale_rd = atof(token2);
    }

    if(!strcasecmp(token1, "ewald_alpha")) {
        system->ewald_alpha = atof(token2);
    }

    if(!strcasecmp(token1, "ewald_kmax")) {
        system->ewald_kmax = atoi(token2);
    }

    if(!strcasecmp(token1, "polarization")) {
        if(!strcasecmp(token2, "on"))
            system->polarization = 1;
    }

```

```

        else
            system->polarization = 0;
    }

    if(!strcasecmp(token1, "cavity_autoreject")) {
        if(!strcasecmp(token2, "on"))
            system->cavity_autoreject = 1;
        else
            system->cavity_autoreject = 0;
    }

    if(!strcasecmp(token1, "cavity_autoreject_scale")) {
        system->cavity_autoreject_scale = atof(token2);
    }

    if(!strcasecmp(token1, "polar_ewald")) {
        if(!strcasecmp(token2, "on"))
            system->polar_ewald = 1;
        else
            system->polar_ewald = 0;
    }

    if(!strcasecmp(token1, "polarizability_tensor")) {
        if(!strcasecmp(token2, "on"))
            system->polarizability_tensor = 1;
        else
            system->polarizability_tensor = 0;
    }

    if(!strcasecmp(token1, "polar_zodid")) {
        if(!strcasecmp(token2, "on"))
            system->polar_zodid = 1;
        else
            system->polar_zodid = 0;
    }

    if(!strcasecmp(token1, "polar_iterative")) {
        if(!strcasecmp(token2, "on"))
            system->polar_iterative = 1;
        else
            system->polar_iterative = 0;
    }

    if(!strcasecmp(token1, "polar_palmo")) {
        if(!strcasecmp(token2, "on"))
            system->polar_palmo = 1;
        else
            system->polar_palmo = 0;
    }

    if(!strcasecmp(token1, "polar_gs")) {
        if(!strcasecmp(token2, "on"))
            system->polar_gs = 1;
        else
            system->polar_gs = 0;
    }

    if(!strcasecmp(token1, "polar_gs_ranked")) {
        if(!strcasecmp(token2, "on"))
            system->polar_gs_ranked = 1;
        else
            system->polar_gs_ranked = 0;
    }

    if(!strcasecmp(token1, "polar_sor")) {
        if(!strcasecmp(token2, "on"))
            system->polar_sor = 1;
        else
            system->polar_sor = 0;
    }

    if(!strcasecmp(token1, "polar_esor")) {
        if(!strcasecmp(token2, "on"))
            system->polar_esor = 1;
        else
            system->polar_esor = 0;
    }

    if(!strcasecmp(token1, "polar_gamma")) {
        system->polar_gamma = atof(token2);
    }

    if(!strcasecmp(token1, "polar_damp")) {
        system->polar_damp = atof(token2);
    }

    if(!strcasecmp(token1, "field_damp")) {
        system->field_damp = atof(token2);
    }

    if(!strcasecmp(token1, "polar_precision")) {
        system->polar_precision = atof(token2);
    }

    if(!strcasecmp(token1, "polar_max_iter")) {
        system->polar_max_iter = atoi(token2);
    }

    if(!strcasecmp(token1, "polar_damp_type")) {
        if(!strcasecmp(token2, "linear"))
            system->damp_type = DAMPING_LINEAR;
        else if(!strcasecmp(token2, "exponential"))
            system->damp_type = DAMPING_EXPONENTIAL;
        else
            system->damp_type = 0;
    }
}

```

```

if(!strcasecmp(token1, "polar_self")) {
    if(!strcasecmp(token2, "on"))
        system->polar_self = 1;
    else
        system->polar_self = 0;
}

#ifndef QM_ROTATION
if(!strcasecmp(token1, "quantum_rotation")) {
    if(!strcasecmp(token2, "on"))
        system->quantum_rotation = 1;
    else
        system->quantum_rotation = 0;
}

if(!strcasecmp(token1, "quantum_rotation_hindered")) {
    if(!strcasecmp(token2, "on"))
        system->quantum_rotation_hindered = 1;
    else
        system->quantum_rotation_hindered = 0;
}

if(!strcasecmp(token1, "quantum_rotation_hindered_barrier")) {
    system->quantum_rotation_hindered_barrier = atof(token2);
}

if(!strcasecmp(token1, "quantum_rotation_B")) {
    system->quantum_rotation_B = atof(token2);
}

if(!strcasecmp(token1, "quantum_rotation_level_max")) {
    system->quantum_rotation_level_max = atof(token2);
}

if(!strcasecmp(token1, "quantum_rotation_l_max")) {
    system->quantum_rotation_l_max = atof(token2);
}
#endif /* QM_ROTATION */

#ifndef XXX
if(!strcasecmp(token1, "quantum_vibration")) {
    if(!strcasecmp(token2, "on"))
        system->quantum_vibration = 1;
    else
        system->quantum_vibration = 0;
}
#endif /* XXX */

if(!strcasecmp(token1, "pdb_input")) {
    if(!system->pdb_input) {
        system->pdb_input = calloc(MAXLINE, sizeof(char));
        strcpy(system->pdb_input, token2);
    }
}

if(!strcasecmp(token1, "pdb_output")) {
    if(!system->pdb_output) {
        system->pdb_output = calloc(MAXLINE, sizeof(char));
        strcpy(system->pdb_output, token2);
    }
}

if(!strcasecmp(token1, "pdb_restart")) {
    if(!system->pdb_restart) {
        system->pdb_restart = calloc(MAXLINE, sizeof(char));
        strcpy(system->pdb_restart, token2);
    }
}

if(!strcasecmp(token1, "traj_output")) {
    if(!system->traj_output) {
        system->traj_output = calloc(MAXLINE, sizeof(char));
        strcpy(system->traj_output, token2);
    }
}

if(!strcasecmp(token1, "energy_output")) {
    if(!system->energy_output) {
        system->energy_output = calloc(MAXLINE, sizeof(char));
        strcpy(system->energy_output, token2);
    }
}

if(!strcasecmp(token1, "pop_histogram_output")) {
    if(!system->histogram_output) {
        system->histogram_output = calloc(MAXLINE, sizeof(char));
        strcpy(system->histogram_output, token2);
    }
}

if(system->polarization && !strcasecmp(token1, "dipole_output")) {
    if(!system->dipole_output) {
        system->dipole_output = calloc(MAXLINE, sizeof(char));
        strcpy(system->dipole_output, token2);
    }
}

if(system->polarization && !strcasecmp(token1, "field_output")) {
    if(!system->field_output) {
        system->field_output = calloc(MAXLINE, sizeof(char));
        strcpy(system->field_output, token2);
    }
}

if(!strcasecmp(token1, "frozen_output")) {
    if(!system->frozen_output) {

```

```

        system->frozen_output = calloc(MAXLINE, sizeof(char));
        strcpy(system->frozen_output, token2);
    }

    if(!strcasecmp(token1, "fit_input")) {
        if(!system->fit_input) {
            system->fit_input = calloc(MAXLINE, sizeof(char));
            strcpy(system->fit_input, token2);
        }
    }

    if(!strcasecmp(token1, "basis1")) {
        system->pbc->basis[0][0] = atof(token2);
        system->pbc->basis[0][1] = atof(token3);
        system->pbc->basis[0][2] = atof(token4);
    }

    if(!strcasecmp(token1, "basis2")) {
        system->pbc->basis[1][0] = atof(token2);
        system->pbc->basis[1][1] = atof(token3);
        system->pbc->basis[1][2] = atof(token4);
    }

    if(!strcasecmp(token1, "basis3")) {
        system->pbc->basis[2][0] = atof(token2);
        system->pbc->basis[2][1] = atof(token3);
        system->pbc->basis[2][2] = atof(token4);
    }

    if(!strcasecmp(token1, "max_bondlength")) {
        system->max_bondlength = atof(token2);
    }

    if(!strcasecmp(token1, "pop_histogram")) {
        if(!strcasecmp(token2, "on"))
            system->calc_hist=1;
        else
            system->calc_hist=0;
    }

    if(!strcasecmp(token1, "pop_hist_resolution"))
        system->hist_resolution=atof(token2);

    memset(linebuffer, 0, MAXLINE);
    n = fgets(linebuffer, MAXLINE, fp);
}

/* close the config file */
fclose(fp);

/* calculate things related to the periodic boundary conditions */
pbc(system->pbc);

return(system);
}

double h2_fugacity(double temperature, double pressure) {
    if((temperature == 77.0) && (pressure <= 200.0)) {
        output("INPUT: fugacity calculation using Zhou function\n");
        return(h2_fugacity_zhou(temperature, pressure));
    } else if(temperature >= 273.15) {
        output("INPUT: fugacity calculation using Shaw function\n");
        return(h2_fugacity_shaw(temperature, pressure));
    } else {
        output("INPUT: fugacity calculation using BACK EoS\n");
        return(h2_fugacity_back(temperature, pressure));
    }
    return(0); /* NOT REACHED */
}

/* use the semi-empirical BACK equation of state */
/* Tomas Boublík, "The BACK equation of state for hydrogen and related compounds", Fluid Phase Equilibria, 240, 96-100 (2005) */
double h2_fugacity_back(double temperature, double pressure) {
    double fugacity_coefficient, fugacity;
    double comp_factor;
    double P, dP;

    /* integrate (z-1)/P from 0 to P */
    fugacity_coefficient = 0;
    for(P = 0.001, dP = 0.001; P <= pressure; P += dP) {
        comp_factor = h2_comp_back(temperature, P);
        fugacity_coefficient += dP*(comp_factor - 1.0)/P;
    }
    fugacity_coefficient = exp(fugacity_coefficient);

    comp_factor = h2_comp_back(temperature, pressure);
    printf("INPUT: BACK compressibility factor at %.3f atm is %.3f\n", pressure, comp_factor);

    fugacity = pressure*fugacity_coefficient;
    return(fugacity);
}

```

```

}

#define BACK_H2_ALPHA 1.033
#define BACK_H2_U0 38.488
#define BACK_H2_V00 9.746
#define BACK_H2_N 0.00
#define BACK_C 0.12

#define BACK_MAX_M 9
#define BACK_MAX_N 4

double h2_comp_back(double temperature, double pressure) {

    double alpha, y; /* repulsive part of the compressibility factor */
    double V, V0, u, D[BACK_MAX_M][BACK_MAX_N]; /* attractive part */
    int n, m; /* indices for double sum of attractive part */
    double comp_factor;
    double comp_factor_repulsive;
    double comp_factor_attractive;
    double fugacity;

    /* setup the BACK universal D constants */
    D[0][0] = -8.8043; D[0][1] = 2.9396; D[0][2] = -2.8225; D[0][3] = 0.34;
    D[1][0] = 4.164627; D[1][1] = -6.0865383; D[1][2] = 4.7600148; D[1][3] = -3.1875014;
    D[2][0] = -48.203555; D[2][1] = 40.137956; D[2][2] = 11.257177; D[2][3] = 12.231796;
    D[3][0] = 140.4362; D[3][1] = -76.230797; D[3][2] = -66.382743; D[3][3] = -12.110681;
    D[4][0] = -195.23339; D[4][1] = -133.70055; D[4][2] = 69.248785; D[4][3] = 0.0;
    D[5][0] = 113.515; D[5][1] = 860.25349; D[5][2] = 0.0; D[5][3] = 0.0;
    D[6][0] = 0.0; D[6][1] = -1538.3224; D[6][2] = 0.0; D[6][3] = 0.0;
    D[7][0] = 0.0; D[7][1] = 1221.4261; D[7][2] = 0.0; D[7][3] = 0.0;
    D[8][0] = 0.0; D[8][1] = -409.10539; D[8][2] = 0.0; D[8][3] = 0.0;

    /* calculate attractive part */
    V0 = BACK_H2_V00*(1.0 - BACK_C*exp(-3.0*BACK_H2_U0/temperature));
    V = NA*KB*temperature/(pressure*ATM2PASCALS*1.0e-6);
    u = BACK_H2_U0*(1.0 + BACK_H2_N/temperature);

    comp_factor_attractive = 0;
    for(n = 0; n < BACK_MAX_N; n++)
        for(m = 0; m < BACK_MAX_M; m++)
            comp_factor_attractive += ((double)(m+1))*D[m][n]*pow(u/temperature, ((double)(n+1)))*pow(V0/V, ((double)(m+1)));

    /* calculate repulsive part */
    alpha = BACK_H2_ALPHA;
    y = (M_PI*sqrt(2.0)/6.0)*(pressure*ATM2PASCALS*1.0e-6)/(NA*KB*temperature)*V0;
    comp_factor_repulsive = 1.0 + (3.0*alpha - 2.0)*y;
    comp_factor_repulsive += (3.0*pow(alpha, 2.0) - 3.0*alpha + 1.0)*pow(y, 2.0);
    comp_factor_repulsive -= pow(alpha, 2.0)*pow(y, 3.0);
    comp_factor_repulsive /= pow((1.0 - y), 3.0);

    comp_factor = comp_factor_repulsive + comp_factor_attractive;
    return(comp_factor);
}

/* calculate the fugacity correction for H2 for 0 C and higher */
/* this empirical relation follows from: */
/* H.R. Shaw, D.F. Wones, American Journal of Science, 262, 918-929 (1964) */
double h2_fugacity_shaw(double temperature, double pressure) {

    double C1, C2, C3;
    double fugacity, fugacity_coefficient;

    C1 = -3.8402*pow(temperature, 1.0/8.0) + 0.5410;
    C1 = exp(C1);

    C2 = -0.1263*pow(temperature, 1.0/2.0) - 15.980;
    C2 = exp(C2);

    C3 = -0.11901*temperature - 5.941;
    C3 = exp(C3);
    C3 *= 300.0;

    fugacity_coefficient = C1*pressure - C2*pow(pressure, 2.0) + C3*exp(-pressure/300.0 - 1.0);
    fugacity_coefficient = exp(fugacity_coefficient);
    fugacity = fugacity_coefficient*pressure;

    return(fugacity);
}

/* fugacity for low temperature and up to 200 atm */
/* Zhou, Zhou, Int. J. Hydrogen Energy, 26, 597-601 (2001) */
double h2_fugacity_zhou(double temperature, double pressure) {

    double fugacity, fugacity_coefficient;
    pressure *= ATM2PSI;

    fugacity_coefficient = -1.38130e-4*pressure;
    fugacity_coefficient += 4.67096e-8*pow(pressure, 2.0)/2;
    fugacity_coefficient += 5.93690e-12*pow(pressure, 3.0)/3;
    fugacity_coefficient += -3.24527e-15*pow(pressure, 4.0)/4;
    fugacity_coefficient += 3.54211e-19*pow(pressure, 5.0)/5;

    pressure /= ATM2PSI;

    fugacity_coefficient = exp(fugacity_coefficient);
    fugacity = pressure*fugacity_coefficient;

    return(fugacity);
}

```

```

#define MWCO2 44.0
#define W 0.225 /* acentric factor for CO2 */
#define Tc 304.12 /* critical temp (K) */
#define Pc 72.775 /* critical press (atm) */
#define Rpr 82.06 /* atm cm^3/(mol K) */

/* reads in temperature in K, and pressure (of the ideal gas in the reservoir) in atm */
/* return the CO2 fugacity via the Peng-Robinson equation of state */
/* else return 0.0 on error - I don't have an error statement*/
/* units are atm, K, cm^3, g Even for the constants above*/
double co2_fugacity(double temperature, double pressure) {

    double fugacity, lnfoverp,pressurePR;
    double density_ig,a,b,Tr,alpha,Vm,Z,Af,Bf;
    double p1,p2,f1, f2, f3,f4;

    /* calculate ideal gas density from input temp and press */
    density_ig= pressure*MWCO2/ (temperature *Rpr);

    /* calculate a,b,Tr,alpha,Vm,pressurePR */
    a=0.45724*Rpr*Rpr*Tc*Tc/Pc;
    b=0.07780*Rpr*Tc*Tc/Pc;
    Tr=temperature/Tc;
    alpi=(1.0+(0.37464+1.54226*W-0.26992*W*W)*(1.0-sqrt(Tr)));
    alpha=alpi*alpi;
    Vm=MWCO2*density_ig;
    p1=Rpr*temperature/(Vm-b);
    p2=a*alpha/(Vm*Vm +2*b*Vm -b*b);
    pressurePR= p1-p2;

    /* calculate each term of fugacity with Peng-Robinson EOS */
    Z=pressurePR*Vm/(Rpr*temperature);
    Af=a*pressurePR/(Rpr*Rpr*temperature*temperature);
    Bf=b*pressurePR/(Rpr*temperature);
    f1=(Z-1.0)-log(Z-Bf);
    f2=af/(2*sqrt(2)*Bf);
    f3=Z+(1.0+sqrt(2))*Bf;
    f4=Z-(1.0-sqrt(2))*Bf;
    lnfoverp=f1-f2*log(f3/f4);
    fugacity=exp(lnfoverp)*pressurePR;

    return(fugacity);
}

int check_system(system_t *system) {
    char linebuf[MAXLINE];
    switch(system->ensemble) {
        case ENSEMBLE_UVT:
            output("INPUT: Grand canonical ensemble\n");
            break;
        case ENSEMBLE_NVT:
            output("INPUT: Canonical ensemble\n");
            break;
        case ENSEMBLE_SURF:
            output("INPUT: Potential energy surface\n");
            break;
        case ENSEMBLE_SURF_FIT:
            output("INPUT: Potential energy surface fitting\n");
            break;
        case ENSEMBLE_NVE:
            output("INPUT: Microcanonical ensemble\n");
            break;
        case '?':
            error("INPUT: improper ensemble specified\n");
            return(-1);
    }

    if(system->ensemble == ENSEMBLE_SURF_FIT) {
        if(!system->fit_input) {
            error("INPUT: surface fitting requires a fit_input file\n");
            return(-1);
        }
    }

    if(system->ensemble == ENSEMBLE_SURF) {
        output("INPUT: surface module activated\n");
        if(system->surf_max < system->surf_min) {
            error("INPUT: surf_max is greater than surf_min\n");
            return(-1);
        } else {
            sprintf(linebuf, "INPUT: minimum surface coordinate is %.3f\n", system->surf_min);
            output(linebuf);
            sprintf(linebuf, "INPUT: maximum surface coordinate is %.3f\n", system->surf_max);
            output(linebuf);
        }

        if(system->surf_inc <= 0.0) {
            error("INPUT: surf_inc is less than or equal to 0\n");
            return(-1);
        } else {
            sprintf(linebuf, "INPUT: incremental surface displacement coordinate is %.3f\n", system->surf_inc);
            output(linebuf);
        }

        if(!system->surf_preserve && (system->surf_ang <= 0.0)) {
            error("INPUT: surf_ang is less than or equal to 0\n");
        }
    }
}

```

```

        return(-1);
    } else {
        sprintf(linebuf, "INPUT: incremental surface angle coordinate is %.3f\n", system->surf_ang);
        output(linebuf);
    }

}

if(system->rd_only) output("INPUT: calculating repulsion/dispersion only\n");
if(system->rd_lrc)
    output("INPUT: rd long-range corrections are ON\n");
else
    output("INPUT: rd long-range corrections are OFF\n");
if(system->sg) output("INPUT: Molecular potential is Silvera-Goldman\n");
sprintf(linebuf, "INPUT: frozen atom charges scaled by %.2f\n", system->scale_charge);
output(linebuf);

sprintf(linebuf, "INPUT: frozen atom rd scaled by %.2f\n", system->scale_rd);
output(linebuf);

if(system->feynman_hibbs) {
    output("INPUT: Feynman-Hibbs effective potential activated\n");
    if(system->feynman_kleinert) {
        output("INPUT: Feynman-Kleinert iteration method activated\n");
        if(!system->rd_anharmonic) {
            error("INPUT: Feynman-Kleinert iteration only implemented for anharmonic oscillator\n");
            return(-1);
        }
    } else {
        switch(system->feynman_hibbs_order) {
            case 2:
                sprintf(linebuf, "INPUT: Feynman-Hibbs second-order quantum correction activated\n");
                output(linebuf);
                break;
            case 4:
                sprintf(linebuf, "INPUT: Feynman-Hibbs fourth-order quantum correction activated\n");
                output(linebuf);
                break;
            default:
                output("INPUT: Feynman-Hibbs order unspecified - defaulting to h^2\n");
                system->feynman_hibbs_order = 2;
                break;
        }
    }
}

#endif QM_ROTATION
if(system->quantum_rotation){
    output("INPUT: Quantum rotational eigenspectrum calculation enabled\n");
    if(system->quantum_rotation_B <= 0.0) {
        error("INPUT: invalid quantum rotational constant B specified\n");
        return(-1);
    } else {
        sprintf(linebuf, "INPUT: Quantum rotational constant B = %.3f K (%.3f cm^-1)\n", system->quantum_rotation_B,
system->quantum_rotation_B*KB/(100.0*H*C));
        output(linebuf);
    }

    if(system->quantum_rotation_level_max <= 0) {
        error("INPUT: invalid quantum rotation level max\n");
        return(-1);
    } else {
        sprintf(linebuf, "INPUT: Quantum rotation level max = %d\n", system->quantum_rotation_level_max);
        output(linebuf);
    }

    if(system->quantum_rotation_l_max <= 0) {
        error("INPUT: invalid quantum rotation l_max\n");
        return(-1);
    } else {
        sprintf(linebuf, "INPUT: Quantum rotation l_max = %d\n", system->quantum_rotation_l_max);
        output(linebuf);
    }

    if(system->quantum_rotation_level_max > (system->quantum_rotation_l_max+1)*(system->quantum_rotation_l_max+1)) {
        error("INPUT: quantum rotational levels cannot exceed l_max + 1 X l_max +1\n");
        return(-1);
    }
}

#endif /* QM_ROTATION */

#endif XXX
if(system->quantum_vibration) output("INPUT: Quantum vibrational eigenspectrum calculation enabled\n");
#endif /* XXX */

if(system->simulated_annealing) {
    if(system->ensemble != ENSEMBLE_NVT) {
        error("INPUT: Simulated annealing only valid for canonical ensemble\n");
        return(-1);
    } else {
        output("INPUT: Simulated annealing active\n");
    }
}

```

```

if((system->simulated_annealing_schedule < 0.0) || (system->simulated_annealing_schedule > 1.0)) {
    error("INPUT: invalid simulated annealing temperature schedule specified\n");
    return(-1);
} else {
    sprintf(linebuf, "INPUT: Simulated annealing temperature schedule = %.3f\n", system->simulated_annealing_schedule);
    output(linebuf);
}
}

if(!system->pdb_input) {
    error("INPUT: must specify an input PDB\n");
    return(-1);
} else {
    sprintf(linebuf, "INPUT: molecular coordinates are in %s\n", system->pdb_input);
    output(linebuf);
}

if(system->polarization) {
    output("INPUT: Thole polarization activated\n");
    if(system->polar_iterative && system->polarizability_tensor) {
        error("INPUT: iterative polarizability tensor method not implemented\n");
        return(-1);
    }
    if(!system->polar_iterative && system->polar_zodid) {
        error("INPUT: ZODID and matrix inversion cannot both be set!\n");
        return(-1);
    }
    if(system->damp_type == DAMPING_LINEAR)
        output("INPUT: Thole linear damping activated\n");
    else if(system->damp_type == DAMPING_EXPONENTIAL)
        output("INPUT: Thole exponential damping activated\n");
    else {
        error("INPUT: Thole damping method not specified\n");
        return(-1);
    }
    if(system->polar_damp <= 0.0) {
        error("INPUT: damping factor must be specified\n");
        return(-1);
    } else {
        sprintf(linebuf, "INPUT: Thole damping parameter is %.4f\n", system->polar_damp);
        output(linebuf);
        sprintf(linebuf, "INPUT: Field damping parameter is %.4f\n", system->field_damp);
        output(linebuf);
    }
}

if(system->polar_iterative) {
    output("INPUT: Thole iterative solver activated\n");
    if(system->polar_zodid) {
        output("INPUT: ZODID polarization enabled\n");
    }
    if((system->polar_precision > 0.0) && (system->polar_max_iter > 0)) {
        error("INPUT: cannot specify both polar_precision and polar_max_iter, must pick one\n");
        return(-1);
    }
    if(system->polar_precision < 0.0) {
        error("INPUT: invalid polarization iterative precision specified\n");
        return(-1);
    } else if(system->polar_precision > 0.0) {
        sprintf(linebuf, "INPUT: Thole iterative precision is %e A*sqrt(KA) (%e D)\n", system->polar_precision,
            system->polar_precision/DEBYE25KA);
        output(linebuf);
    } else {
        sprintf(linebuf, "INPUT: using polar max SCF iterations = %d\n", system->polar_max_iter);
        output(linebuf);
    }
    if(system->polar_sor && system->polar_esor) {
        error("INPUT: cannot specify both SOR and ESOR SCF methods\n");
        return(-1);
    }
    if(system->polar_sor) output("INPUT: SOR SCF scheme active\n");
    else if(system->polar_esor) output("INPUT: ESOR SCF scheme active\n");
    if(system->polar_gamma < 0.0) {
        error("INPUT: invalid Pre-cond/SOR/ESOR gamma set\n");
        return(-1);
    } else {
        sprintf(linebuf, "INPUT: Pre-cond/SOR/ESOR gamma = %.3f\n", system->polar_gamma);
        output(linebuf);
    }
}

if(system->polar_gs && system->polar_gs_ranked) {
    error("INPUT: both polar_gs and polar_gs_ranked cannot be set\n");
    return(-1);
}

if(system->polar_gs)
    output("INPUT: Gauss-Seidel iteration scheme active\n");
else if(system->polar_gs_ranked)
    output("INPUT: Gauss-Seidel Ranked iteration scheme active\n");

if(system->polar_palmo) output("INPUT: Polarization energy of Palmo and Krimm enabled\n");

```

```

    } else {
        output("INPUT: Matrix polarization activated\n");
        if(system->polarizability_tensor)
            output("INPUT: Polarizability tensor calculation activated\n");
    }

    if(system->polar_self) output("INPUT: Polarization self-induction is active\n");
}

if((system->pb->volume < 0.0) || (system->pb->cutoff < 0.0))
    error("INPUT: something is wrong with the periodic boundary conditions\n");
else {
    sprintf(linebuf, "INPUT: basis vector 1 = %.5f %.5f %.5f\n", system->pb->basis[0][0], system->pb->basis[0][1],
    system->pb->basis[0][2]);
    output(linebuf);
    sprintf(linebuf, "INPUT: basis vector 2 = %.5f %.5f %.5f\n", system->pb->basis[1][0], system->pb->basis[1][1],
    system->pb->basis[1][2]);
    output(linebuf);
    sprintf(linebuf, "INPUT: basis vector 3 = %.5f %.5f %.5f\n", system->pb->basis[2][0], system->pb->basis[2][1],
    system->pb->basis[2][2]);
    output(linebuf);
    sprintf(linebuf, "INPUT: unit cell volume = %.3f A^3 (cutoff = %.3f A)\n", system->pb->volume, system->pb->cutoff);
    output(linebuf);
    sprintf(linebuf, "INPUT: recip basis vector 1 = %.5f %.5f %.5f\n", system->pb->reciprocal_basis[0][0],
    system->pb->reciprocal_basis[0][1], system->pb->reciprocal_basis[0][2]);
    output(linebuf);
    sprintf(linebuf, "INPUT: recip basis vector 2 = %.5f %.5f %.5f\n", system->pb->reciprocal_basis[1][0],
    system->pb->reciprocal_basis[1][1], system->pb->reciprocal_basis[1][2]);
    output(linebuf);
    sprintf(linebuf, "INPUT: recip basis vector 3 = %.5f %.5f %.5f\n", system->pb->reciprocal_basis[2][0],
    system->pb->reciprocal_basis[2][1], system->pb->reciprocal_basis[2][2]);
    output(linebuf);
}

if(system->rd_anharmonic) {

    if(!system->rd_only) {
        error("INPUT: rd_anharmonic being set requires rd_only\n");
        return(-1);
    } else {
        sprintf(linebuf, "INPUT: rd_anharmonic_k = %.3f K/A^2\n", system->rd_anharmonic_k);
        output(linebuf);
        sprintf(linebuf, "INPUT: rd_anharmonic_g = %.3f K/A^4\n", system->rd_anharmonic_g);
        output(linebuf);
    }
}

if((system->ensemble != ENSEMBLE_SURF) && (system->ensemble != ENSEMBLE_SURF_FIT)) {

    if(!system->seed) {
        error("INPUT: no seed specified\n");
        return(-1);
    } else {
        sprintf(linebuf, "INPUT: rng seed is %ld\n", system->seed);
        output(linebuf);
    }

    if(system->numsteps < 1) {
        error("INPUT: improper numsteps specified\n");
        return(-1);
    } else {
        sprintf(linebuf, "INPUT: each core performing %d simulation steps\n", system->numsteps);
        output(linebuf);
    }

    if(system->corrtime < 1) {
        error("INPUT: improper corrtime specified\n");
        return(-1);
    } else {
        sprintf(linebuf, "INPUT: system correlation time is %d steps\n", system->corrtime);
        output(linebuf);
    }

    if(system->ensemble != ENSEMBLE_NVE) {
        if(system->temperature <= 0.0) {
            error("INPUT: invalid temperature specified\n");
            return(-1);
        } else {
            sprintf(linebuf, "INPUT: system temperature is %.3f K\n", system->temperature);
            output(linebuf);
        }
    } else {
        sprintf(linebuf, "INPUT: NVE energy is %.3f K\n", system->total_energy);
        output(linebuf);
    }

    if(system->free_volume > 0.0) {
        sprintf(linebuf, "INPUT: system free_volume is %.3f A^3\n", system->free_volume);
        output(linebuf);
    }

    if((system->ensemble == ENSEMBLE_UVT) && (system->pressure <= 0.0)) {
        error("INPUT: invalid pressure set for GCMC\n");
        return(-1);
    } else {
        if(system->ensemble == ENSEMBLE_UVT) {
            sprintf(linebuf, "INPUT: reservoir pressure is %.3f atm\n", system->pressure);
            output(linebuf);
        }
    }

    if(system->h2_fugacity) {

        system->fugacity = h2_fugacity(system->temperature, system->pressure);
        if(system->h2_fugacity == 0.0) {

```

```

        error("INPUT: error in H2 fugacity assignment\n");
        return(-1);
    }

    sprintf(linebuf, "INPUT: H2 fugacity = %.3f atm\n", system->fugacity);
    output(linebuf);

if(system->co2_fugacity) {
    system->fugacity = co2_fugacity(system->temperature, system->pressure);
    if(system->co2_fugacity == 0.0) {
        error("INPUT: error in CO2 fugacity assignment\n");
        return(-1);
    }

    sprintf(linebuf, "INPUT: CO2 fugacity = %.3f atm\n", system->fugacity);
    output(linebuf);
}

if((system->insert_probability < 0.0) || (system->insert_probability > 1.0)) {
    error("INPUT: insert scale invalid, should be (0,1)\n");
    return(-1);
} else {
    sprintf(linebuf, "INPUT: insert probability is %.3f\n", system->insert_probability);
    output(linebuf);

    sprintf(linebuf, "INPUT: move probability is %.3f\n", system->move_probability);
    output(linebuf);

    sprintf(linebuf, "INPUT: rotation probability is %.3f\n", system->rot_probability);
    output(linebuf);
}

/* autoreject insertions closer than some scaling factor of sigma */
if(system->cavity_autoreject) {
    output("INPUT: cavity autorejection activated\n");

    if((system->cavity_autoreject_scale <= 0.0) || (system->cavity_autoreject_scale > 1.0))
        error("INPUT: cavity_autoreject_scale either not set or out of range\n");
}

if(system->cavity_bias) {
    if((system->cavity_grid_size <= 0.0) || (system->cavity_radius <= 0.0)) {
        error("INPUT: invalid cavity grid or radius specified\n");
    } else {
        output("INPUT: cavity-biased umbrella sampling activated\n");
        sprintf(linebuf, "INPUT: cavity grid size is %dx%dx%d points with a sphere radius of %.3f A\n",
            system->cavity_grid_size, system->cavity_grid_size, system->cavity_grid_size, system->cavity_radius);
        output(linebuf);
    }
}

if(!system->pdb_output) {
    error("INPUT: must specify an output PDB\n");
    return(-1);
} else {
    sprintf(linebuf, "INPUT: will be writing final configuration to %s\n", system->pdb_output);
    output(linebuf);
}

if(!system->pdb_restart) {
    error("INPUT: must specify a restart PDB\n");
    return(-1);
} else {
    sprintf(linebuf, "INPUT: will be writing restart configuration to %s\n", system->pdb_restart);
    output(linebuf);
}

if(!system->traj_output) {
    error("INPUT: must specify a trajectory PDB\n");
    return(-1);
} else {
    sprintf(linebuf, "INPUT: will be writing trajectory to %s\n", system->traj_output);
    output(linebuf);
}

if(system->polarization && (!system->dipole_output || !system->field_output)) {
    error("INPUT: must specify a dipole and field output file\n");
    return(-1);
} else if(system->polarization) {
    sprintf(linebuf, "INPUT: dipole field will be written to %s\n", system->dipole_output);
    output(linebuf);
    sprintf(linebuf, "INPUT: electric field will be written to %s\n", system->field_output);
    output(linebuf);
}

if(system->wpi) {
    output("INPUT: Widom Particle Insertion is enabled\n");
}

}

if(system->calc_hist){
    output("INPUT: histogram calculation will be performed\n");
    if(!system->hist_resolution){
        output("INPUT: no histogram resolution set but histogram calculation requested\n");
        output("INPUT: setting hist_resolution to default value of 0.7A\n");
    }
}

```

```

        system->hist_resolution=0.7;
    }
    else if(system->hist_resolution<0.01 || system->hist_resolution>5.0){
        output("INPUT: histogram resolution out of bounds\n");
        output("INPUT: setting hist_resolution to default value of 0.7A\n");
        system->hist_resolution=0.7;
    }
    else if(!system->histogram_output){
        output("INPUT: no histogram outputfile selected, defaulting to histogram.dat\n");
        system->histogram_output=calloc(MAXLINE,sizeof(char));
        sprintf(system->histogram_output,"histogram.dat");
    }
    else{
        sprintf(linebuf,"INPUT: histogram resolution set to %.3f A\n",system->hist_resolution);
        output(linebuf);
    }

    if(system->max_bondlength < .5){
        output("INPUT: max_bondlength either not set or out of bounds\n");
        output("INPUT: setting max_bondlength to default value of 1.8A\n");
        system->max_bondlength=1.8;
    }

    if(!system->frozen_output){
        output("INPUT: no frozen_output set! setting frozen coordinate output file to frozen.dx\n");
        system->frozen_output = calloc(MAXLINE, sizeof(char));
        sprintf(system->frozen_output,"frozen.dx");
    } else {
        sprintf(linebuf, "INPUT: will be writing frozen coordinates to %s\n", system->frozen_output);
        output(linebuf);
    }

}

return(0);
}

molecule_t *read_molecules(system_t *system) {

    int i;
    molecule_t *molecules, *molecule_ptr;
    atom_t *atom_ptr, *prev_atom_ptr;
    char linebuf[MAXLINE], *n;
    FILE *fp;
    char token_atom[MAXLINE], token_atomid[MAXLINE], token_atomtype[MAXLINE], token_moleculetype[MAXLINE];
    char token_frozen[MAXLINE], token_moleculeid[MAXLINE], token_x[MAXLINE], token_y[MAXLINE], token_z[MAXLINE];
    char token_mass[MAXLINE], token_charge[MAXLINE], token_alpha[MAXLINE], token_epsilon[MAXLINE], token_sigma[MAXLINE];
    int current_frozen, current_adiabatic, current_moleculeid, current_atomid;
    double current_x, current_y, current_z;
    double current_mass, current_charge, current_alpha, current_epsilon, current_sigma;
    double current_molecule_mass;
    int moveable;
    int atom_counter;

    /* allocate the start of the list */
    molecules = calloc(i, sizeof(molecule_t));
    molecule_ptr = molecules;
    molecule_ptr->id = 1;
    molecule_ptr->atoms = calloc(1, sizeof(atom_t));
    atom_ptr = molecule_ptr->atoms;
    prev_atom_ptr = atom_ptr;

    /* open the molecule input file */
    fp = fopen(system->pdb_input, "r");
    if(!fp) {
        sprintf(linebuf, "INPUT: couldn't open PDB input file %s\n", system->pdb_input);
        error(linebuf);
        return(NULL);
    }

    /* clear the linebuffer and read the tokens in */
    atom_counter = 0;
    memset(linebuf, 0, MAXLINE);
    n = fgets(linebuf, MAXLINE, fp);
    while(n) {

        /* clear the tokens */
        memset(token_atom, 0, MAXLINE);
        memset(token_atomid, 0, MAXLINE);
        memset(token_atomtype, 0, MAXLINE);
        memset(token_moleculetype, 0, MAXLINE);
        memset(token_frozen, 0, MAXLINE);
        memset(token_moleculeid, 0, MAXLINE);
        memset(token_x, 0, MAXLINE);
        memset(token_y, 0, MAXLINE);
        memset(token_z, 0, MAXLINE);
        memset(token_mass, 0, MAXLINE);
        memset(token_charge, 0, MAXLINE);
        memset(token_alpha, 0, MAXLINE);
        memset(token_epsilon, 0, MAXLINE);
        memset(token_sigma, 0, MAXLINE);

        /* parse the line */
        sscanf(linebuf, "%s %s %s\n", token_atom, token_atomid, token_atomtype, token_moleculetype,
        token_frozen, token_moleculeid, token_x, token_y, token_z, token_mass, token_charge, token_alpha, token_epsilon, token_sigma);

        if(!strcasecmp(token_atom, "ATOM") && strcasecmp(token_moleculetype, "BOX")) {

            current_frozen = 0; current_adiabatic = 0;

```

```

if(!strcasecmp(token_frozen, "F"))
    current_frozen = 1;
if(!strcasecmp(token_frozen, "A"))
    current_adiabatic = 1;

current_moleculeid = atoi(token_moleculeid);
current_atomid = atoi(token_atomid);
current_x = atof(token_x);
current_y = atof(token_y);
current_z = atof(token_z);
current_mass = atof(token_mass); /* mass in amu */
current_charge = atof(token_charge);
current_charge *= E2REDUCED; /* convert charge into reduced units */
current_alpha = atof(token_alpha);
current_epsilon = atof(token_epsilon);
current_sigma = atof(token_sigma);
if(current_frozen) current_charge *= system->scale_charge;

if(molecule_ptr->id != current_moleculeid) {
    molecule_ptr->next = calloc(1, sizeof(molecule_t));
    molecule_ptr = molecule_ptr->next;
    molecule_ptr->atoms = calloc(1, sizeof(atom_t));
    prev_atom_ptr->next = NULL;
    free(atom_ptr);
    atom_ptr = molecule_ptr->atoms;
}
strcpy(molecule_ptr->moleculetype, token_moleculetype);

molecule_ptr->id = current_moleculeid;
molecule_ptr->frozen = current_frozen;
molecule_ptr->adiabatic = current_adiabatic;
molecule_ptr->mass += current_mass;

#ifndef QM_ROTATION
/* if quantum rot calc. enabled, allocate the necessary structures */
if(system->quantum_rotation && !molecule_ptr->frozen) {

    molecule_ptr->quantum_rotational_energies = calloc(system->quantum_rotation_level_max, sizeof(double));
    molecule_ptr->quantum_rotational_eigenvectors = calloc(system->quantum_rotation_level_max, sizeof(complex_t *));
    for(i = 0; i < system->quantum_rotation_level_max; i++)
        molecule_ptr->quantum_rotational_eigenvectors[i] = calloc((system->quantum_rotation_l_max +
1)*(system->quantum_rotation_l_max + 1), sizeof(complex_t));
    molecule_ptr->quantum_rotational_eigensymmetry = calloc(system->quantum_rotation_level_max, sizeof(int));
}
#endif /* QM_ROTATION */

#ifndef XXX
/* if quantum vib calc. enabled, allocate the necessary structures */
if(system->quantum_vibration && !molecule_ptr->frozen) {

    molecule_ptr->quantum_vibrational_energies = calloc(system->quantum_vibration_level_max, sizeof(double));
    molecule_ptr->quantum_vibrational_eigenvectors = calloc(system->quantum_vibration_level_max,
sizeof(complex_t *));
    for(i = 0; i < system->quantum_vibration_level_max; i++)
        molecule_ptr->quantum_vibrational_eigenvectors[i] = calloc((system->quantum_vibration_l_max +
1)*(system->quantum_vibration_l_max + 1), sizeof(complex_t));
    molecule_ptr->quantum_vibrational_eigensymmetry = calloc(system->quantum_vibration_level_max, sizeof(int));
}
#endif /* XXX */

++atom_counter;
atom_ptr->id = atom_counter;
memset(atom_ptr->atomtype, 0, MAXLINE);
strcpy(atom_ptr->atomtype, token_atomtype);
atom_ptr->frozen = current_frozen;
atom_ptr->adiabatic = current_adiabatic;
atom_ptr->pos[0] = current_x;
atom_ptr->pos[1] = current_y;
atom_ptr->pos[2] = current_z;
atom_ptr->mass = current_mass;
atom_ptr->charge = current_charge;
atom_ptr->polarizability = current_alpha;
atom_ptr->epsilon = current_epsilon;
atom_ptr->sigma = current_sigma;

atom_ptr->next = calloc(1, sizeof(atom_t));
prev_atom_ptr = atom_ptr;
atom_ptr = atom_ptr->next;
}

memset(linebuf, 0, MAXLINE);
n = fgets(linebuf, MAXLINE, fp);
}

/* terminate the atom list */
prev_atom_ptr->next = NULL;
free(atom_ptr);

/* scan the list, make sure that there is at least one moveable molecule */
for(molecule_ptr = molecules, moveable = 0; molecule_ptr; molecule_ptr = molecule_ptr->next) {
    if(!molecule_ptr->frozen) ++moveable;
}

if(!moveable) {
    error("INPUT: no moveable molecules found, there must be at least one in your PDB file\n");
    return(NULL);
} else
    return(molecules);
}

system_t *setup_system(char *input_file) {

```

```

system_t *system;
char linebuf[MAXLINE];

/* read in all of the tokens and parameters from the config file */
system = read_config(input_file);
if(!system) {
    error("INPUT: error reading config file\n");
    return(NULL);
} else
    output("INPUT: finished reading config file\n");

/* validate configuration parameters */
if(check_system(system) < 0) {
    error("INPUT: invalid config parameters specified\n");
    return(NULL);
} else
    output("INPUT: config file validated\n");

/* read in the input pdb and setup the data structures */
system->molecules = read_molecules(system);
if(!system->molecules) {
    error("INPUT: error reading in molecules\n");
    return(NULL);
} else
    output("INPUT: finished reading in molecules\n");

/* allocate the necessary pairs */
setup_pairs(system->molecules);
output("INPUT: finished allocating pair lists\n");

/* calculate the periodic boundary conditions */
pbc(system->pbc);
output("INPUT: finished calculating periodic boundary conditions\n");

/* get all of the pairwise interactions, exclusions, etc. */
if(system->cavity_bias) setup_cavity_grid(system);
pairs(system);

/* set all pairs to initially have their energies calculated */
flag_all_pairs(system);
output("INPUT: finished calculating pairwise interactions\n");

/* set the ewald gaussian width appropriately */
system->ewald_alpha = 3.5/system->pbc->cutoff;
if(!(system->sg || system->rd_only)) {
    sprintf(linebuf, "INPUT: Ewald gaussian width = %f A\n", system->ewald_alpha);
    output(linebuf);
    sprintf(linebuf, "INPUT: Ewald kmax = %d\n", system->ewald_kmax);
    output(linebuf);
}

/* write the frozen lattice configuration */
if(system->file_pointers.fp_frozen) write_frozen(system->file_pointers.fp_frozen,system);

return(system);
}

#ifndef DEBUG
void test_list(molecule_t *molecules) {

molecule_t *molecule_ptr;
atom_t *atom_ptr;
pair_t *pair_ptr;

for(molecule_ptr = molecules; molecule_ptr; molecule_ptr = molecule_ptr->next) {
    printf("moleculeid = %d\n", molecule_ptr->id);
    printf("moleculename = %s\n", molecule_ptr->moleculename);
    printf("molecule_frozen = %d\n", molecule_ptr->frozen);
    printf("molecule_mass = %f\n", molecule_ptr->mass);
    for(atom_ptr = molecule_ptr->atoms; atom_ptr; atom_ptr = atom_ptr->next) {
        printf("atomtype = %s x = %f y = %f z = %f\n", atom_ptr->atomtype, atom_ptr->pos[0], atom_ptr->pos[1],
atom_ptr->pos[2]);
        printf("atom frozen = %d mass = %f, charge = %f, alpha = %f, eps = %f, sig = %f\n", atom_ptr->frozen,
atom_ptr->mass, atom_ptr->charge, atom_ptr->polarizability, atom_ptr->epsilon, atom_ptr->sigma);
        for(pair_ptr = atom_ptr->pairs; pair_ptr; pair_ptr = pair_ptr->next)
            if((pair_ptr->rd_excluded || pair_ptr->es_excluded || pair_ptr->frozen)) printf("pair = 0x%lx eps = %f sig
= %f\n", pair_ptr, pair_ptr->epsilon, pair_ptr->sigma);
    }
}
fflush(stdout);
}

void test_molecule(molecule_t *molecule) {

atom_t *atom_ptr;
pair_t *pair_ptr;

printf("moleculeid = %d\n", molecule->id);
printf("moleculename = %s\n", molecule->moleculename);
printf("molecule_frozen = %d\n", molecule->frozen);
printf("molecule_mass = %f\n", molecule->mass);
for(atom_ptr = molecule->atoms; atom_ptr; atom_ptr = atom_ptr->next) {
    printf("atomtype = %s x = %f y = %f z = %f\n", atom_ptr->atomtype, atom_ptr->pos[0], atom_ptr->pos[1], atom_ptr->pos[2]);
    printf("atom frozen = %d mass = %f, charge = %f, alpha = %f, eps = %f, sig = %f\n", atom_ptr->frozen, atom_ptr->mass,
atom_ptr->charge, atom_ptr->polarizability, atom_ptr->epsilon, atom_ptr->sigma);
    for(pair_ptr = atom_ptr->pairs; pair_ptr; pair_ptr = pair_ptr->next) {
        printf("pair at 0x%lx\n", pair_ptr);fflush(stdout);
    }
}
printf("...finished\n");fflush(stdout);
}

```

```

}

#endif /* DEBUG */

/*
@2007, Jonathan Belof
Space Research Group
Department of Chemistry
University of South Florida
*/
#include <mc.h>

void mpi_copy_histogram_to_sendbuffer(char *snd, int ***grid, system_t *system){
    int i,j,k;
    int x_dim=system->grids->histogram->x_dim;
    int y_dim=system->grids->histogram->y_dim;
    int z_dim=system->grids->histogram->z_dim;
    int *sndcast=(int *)snd; /* cast the send buffer address to an (int *) */

    /* histogram */
    for(k=0;k<z_dim;k++){
        for(j=0;j<y_dim;j++){
            for(i=0;i<x_dim;i++){
                sndcast[i + j*x_dim + k*x_dim*y_dim] = system->grids->histogram->grid[i][j][k];
            }
        }
    }
}

void mpi_copy_rcv_histogram_to_data(char *rcv, int ***histogram, system_t *system)
{
    int i,j,k;
    int x_dim=system->grids->histogram->x_dim;
    int y_dim=system->grids->histogram->y_dim;
    int z_dim=system->grids->histogram->z_dim;
    int *rcvcast=(int *)rcv;

    /* histogram */
    for(k=0;k<z_dim; k++){
        for(j=0; j<y_dim; j++){
            for(i=0; i<x_dim; i++){
                histogram[i][j][k]=rcvcast[i + j*x_dim + k*x_dim*y_dim];
            }
        }
    }
}

/*
@2007, Jonathan Belof
Space Research Group
Department of Chemistry
University of South Florida
*/
#include <mc.h>

void error(char *msg) {
    if(!rank) fprintf(stderr, "%s", msg);
    fflush(stderr);
}

void output(char *msg) {
    if(!rank) printf("%s", msg);
    fflush(stdout);
}

int open_files(system_t *system) {
    if(system->energy_output) {
        system->file_pointers.fp_energy = fopen(system->energy_output, "w");
        if(!system->file_pointers.fp_energy) {
            error("MC: could not open energy file for writing\n");
            return(-1);
        }
    }

    if(system->traj_output) {
        system->file_pointers.fp_traj = fopen(system->traj_output, "w");
        if(!system->file_pointers.fp_traj) {
            error("MC: could not open trajectory file for writing\n");
            return(-1);
        }
    }

    if(system->dipole_output) {
        system->file_pointers.fp_dipole = fopen(system->dipole_output, "w");
        if(!system->file_pointers.fp_dipole) {
            error("MC: could not open dipole file for writing\n");
            return(-1);
        }
    }
}

```

```

    }

    if(system->field_output) {
        system->file_pointers.fp_field = fopen(system->field_output, "w");
        if(!system->file_pointers.fp_field) {
            error("MC: could not open field file for writing\n");
            return(-1);
        }
    }

    if(system->histogram_output) {
        system->file_pointers.fp_histogram = fopen(system->histogram_output, "w");
        if(!system->file_pointers.fp_histogram) {
            error("MC: could not open energy file for writing\n");
            return(-1);
        }
    }

    if(system->frozen_output) {
        system->file_pointers.fp_frozen = fopen(system->frozen_output, "w");
        if(!system->file_pointers.fp_frozen) {
            error("MC: could not open frozen dx file for writing\n");
            return(-1);
        }
    }

    return(0);
}

void close_files(system_t *system) {

    if(system->file_pointers.fp_energy) fclose(system->file_pointers.fp_energy);
    if(system->file_pointers.fp_traj) fclose(system->file_pointers.fp_traj);
    if(system->file_pointers.fp_dipole) fclose(system->file_pointers.fp_dipole);
    if(system->file_pointers.fp_field) fclose(system->file_pointers.fp_field);
    if(system->file_pointers.fp_histogram) fclose(system->file_pointers.fp_histogram);
    if(system->file_pointers.fp_frozen) fclose(system->file_pointers.fp_frozen);

}

/* enforce molecule wrapping around the periodic boundaries on output - keep the atoms together */
int wrapall(molecule_t *molecules, pbc_t *pbc) {

    int i, j;
    molecule_t *molecule_ptr;
    atom_t *atom_ptr;
    double d[3], dimg[3];

    for(molecule_ptr = molecules; molecule_ptr; molecule_ptr = molecule_ptr->next) {

        if(!molecule_ptr->frozen) {
            /* get the minimum imaging distance for the com */
            for(i = 0; i < 3; i++) {
                for(j = 0, d[i] = 0; j < 3; j++) {
                    d[i] += pbc->reciprocal_basis[i][j]*molecule_ptr->com[j];
                }
                d[i] = rint(d[i]);
            }

            for(i = 0; i < 3; i++) {
                for(j = 0, dimg[i] = 0; j < 3; j++)
                    dimg[i] += pbc->basis[i][j]*d[j];

                /* store the wrapped com coordinate */
                molecule_ptr->wrapped_com[i] = dimg[i];
            }

            /* apply the distance to all of the atoms of this molecule */
            for(atom_ptr = molecule_ptr->atoms; atom_ptr; atom_ptr = atom_ptr->next) {
                for(i = 0; i < 3; i++)
                    atom_ptr->wrapped_pos[i] = atom_ptr->pos[i] - dimg[i];
            }
        } else {
            /* don't wrap frozen */
            for(atom_ptr = molecule_ptr->atoms; atom_ptr; atom_ptr = atom_ptr->next) {
                for(i = 0; i < 3; i++)
                    atom_ptr->wrapped_pos[i] = atom_ptr->pos[i];
            }
        }
    } /* molecule */

    return(0);
}

/* write out the final system state as a PDB file */
int write_molecules(system_t *system, char *filename) {

    molecule_t *molecule_ptr;
    atom_t *atom_ptr;
    char linebuf[MAXLINE];
    FILE *fp;
    int i, j, k;
    int atom_box, molecule_box, p, q;
    double box_pos[3], box_occuancy[3];
    int l, m, n, box_labels[2][2][2], diff;

    fp = fopen(filename, "w");
    if(!fp) {

```

```

        sprintf(linebuf, "OUTPUT: could not write pdb to file %s\n", filename);
        error(linebuf);
        return(-1);
    }

/* write pdb */
for(molecule_ptr = system->molecules, i = 1, j = 1; molecule_ptr; molecule_ptr = molecule_ptr->next, j++) {
    for(atom_ptr = molecule_ptr->atoms; atom_ptr; atom_ptr = atom_ptr->next, i++) {

        fprintf(fp, "ATOM ");
        fprintf(fp, "%5d", i); /* give each one a unique id */
        fprintf(fp, " %-4.45s", atom_ptr->atomtype);
        fprintf(fp, " %-3.3s ", molecule_ptr->moleculename);
        if(atom_ptr->adiabatic)
            fprintf(fp, "%-1.1s", "A");
        else if(atom_ptr->frozen)
            fprintf(fp, " %-1.1s", "F");
        else
            fprintf(fp, " %-1.1s", "M");
        fprintf(fp, "%4d", j); /* give each molecule a unique id */
        if(system->wrapall) {
            fprintf(fp, "%8.3f", atom_ptr->wrapped_pos[0]);
            fprintf(fp, "%8.3f", atom_ptr->wrapped_pos[1]);
            fprintf(fp, "%8.3f", atom_ptr->wrapped_pos[2]);
        } else {
            fprintf(fp, "%8.3f", atom_ptr->pos[0]);
            fprintf(fp, "%8.3f", atom_ptr->pos[1]);
            fprintf(fp, "%8.3f", atom_ptr->pos[2]);
        }
        fprintf(fp, " %8.4f", atom_ptr->mass);
        fprintf(fp, " %8.4f", atom_ptr->charge/E2REDUCED); /* convert charge back to real units */
        fprintf(fp, " %8.5f", atom_ptr->polarizability);
        fprintf(fp, " %8.5f", atom_ptr->epsilon);
        fprintf(fp, " %8.5f", atom_ptr->sigma);
        fprintf(fp, "\n");
    }
}

if(system->wrapall) {

/* output the box coords as virtual particles for visualization */
atom_box = i;
molecule_box = j;
for(i = 0; i < 2; i++) {
    for(j = 0; j < 2; j++) {
        for(k = 0; k < 2; k++) {

/* make this frozen */
            fprintf(fp, "ATOM ");
            fprintf(fp, "%5d", atom_box);
            fprintf(fp, " %-4.45s", "X");
            fprintf(fp, " %-3.3s ", "BOX");
            fprintf(fp, "%-1.1s", "F");
            fprintf(fp, "%4d", molecule_box);

/* box coords */
            box_occupancy[0] = ((double)i) - 0.5;
            box_occupancy[1] = ((double)j) - 0.5;
            box_occupancy[2] = ((double)k) - 0.5;

            for(p = 0; p < 3; p++)
                for(q = 0, box_pos[p] = 0; q < 3; q++)
                    box_pos[p] += system->pbcb->basis[p][q]*box_occupancy[q];

            for(p = 0; p < 3; p++)
                fprintf(fp, " %8.3f", box_pos[p]);

/* null interactions */
            fprintf(fp, " %8.4f", 0.0);
            fprintf(fp, " %8.4f", 0.0);
            fprintf(fp, " %8.5f", 0.0);
            fprintf(fp, " %8.5f", 0.0);
            fprintf(fp, " %8.5f", 0.0);
            fprintf(fp, "\n");
        }
        box_labels[i][j][k] = atom_box;
        ++atom_box;
    }
}

/* output the connectivity information */
for(i = 0; i < 2; i++) {
    for(j = 0; j < 2; j++) {
        for(k = 0; k < 2; k++) {

            for(l = 0; l < 2; l++) {
                for(m = 0; m < 2; m++) {
                    for(n = 0; n < 2; n++) {

                        diff = fabs(i - l) + fabs(j - m) + fabs(k - n);
                        if(diff == 1)
                            fprintf(fp, "CONNECT %4d %4d\n", box_labels[i][j][k],
box_labels[l][m][n]);
                    }
                }
            }
        }
    }
}

/* if wrapall */
}

```

```

fprintf(fp, "END\n");
fflush(fp);
fclose(fp);
return(0);
}

void write_states(FILE *fp, molecule_t *molecules) {
    molecule_t *molecule_ptr;
    atom_t *atom_ptr;
    char linebuf[MAXLINE];
    int i, j, k;
    int atom_box, molecule_box, p, q;
    double box_pos[3], box_occupancy[3];
    int l, m, n, box_labels[2][2][2], diff;
    int num_frozen_molecules, num_moveable_molecules;
    int num_frozen_atoms, num_moveable_atoms;

    /* count the number of molecules, atoms, etc. */
    num_frozen_molecules = 0, num_moveable_molecules = 0;
    num_frozen_atoms = 0, num_moveable_atoms = 0;
    for(molecule_ptr = molecules; molecule_ptr; molecule_ptr = molecule_ptr->next) {
        if(molecule_ptr->frozen) {
            ++num_frozen_molecules;
            for(atom_ptr = molecule_ptr->atoms; atom_ptr; atom_ptr = atom_ptr->next)
                ++num_frozen_atoms;
        } else {
            ++num_moveable_molecules;
            for(atom_ptr = molecule_ptr->atoms; atom_ptr; atom_ptr = atom_ptr->next)
                ++num_moveable_atoms;
        }
    }
    fprintf(fp, "REMARK total_molecules=%d, total_atoms=%d\n", (num_frozen_molecules + num_moveable_molecules), (num_frozen_atoms + num_moveable_atoms));
    fprintf(fp, "REMARK frozen_molecules=%d, moveable_molecules=%d\n", num_frozen_molecules, num_moveable_molecules);
    fprintf(fp, "REMARK frozen_atoms=%d, moveable_atoms=%d\n", num_frozen_atoms, num_moveable_atoms);

    /* write pdb formatted states */
    for(molecule_ptr = molecules, i = 1, j = 1; molecule_ptr; molecule_ptr = molecule_ptr->next, j++) {
        for(atom_ptr = molecule_ptr->atoms; atom_ptr; atom_ptr = atom_ptr->next, i++) {
            fprintf(fp, "ATOM ");
            fprintf(fp, "%5d", i); /* give each one a unique id */
            fprintf(fp, " %-4.4fs", atom_ptr->atomtype);
            fprintf(fp, " %-3.3s", molecule_ptr->moleculename);
            if(atom_ptr->adiabatic)
                fprintf(fp, "%-1.1s", "A");
            else if(atom_ptr->frozen)
                fprintf(fp, "%-1.1s", "F");
            else
                fprintf(fp, "%-1.1s", "M");
            fprintf(fp, "%4d ", j); /* give each molecule a unique id */
            fprintf(fp, "%8.3f", atom_ptr->wrapped_pos[0]);
            fprintf(fp, "%8.3f", atom_ptr->wrapped_pos[1]);
            fprintf(fp, "%8.3f", atom_ptr->wrapped_pos[2]);
            fprintf(fp, " %8.4f", atom_ptr->mass);
            fprintf(fp, " %8.4f", atom_ptr->charge/E2REDUCED); /* convert charge back to real units */
            fprintf(fp, " %8.5f", atom_ptr->polarizability);
            fprintf(fp, " %8.5f", atom_ptr->epsilon);
            fprintf(fp, " %8.5f", atom_ptr->sigma);
            fprintf(fp, "\n");
        }
    }
    fprintf(fp, "ENDMDL\n");
    fflush(fp);
}

int write_performance(int i, system_t *system) {
    char linebuf[MAXLINE];
    time_t current_time;
    double sec_step;
    static time_t last_time;
    static int last_step;

    current_time = time(NULL);
    if(i > system->corrtime) {
        sec_step = difftime(current_time, last_time)/((double)(i - last_step));

        if(system->ensemble == ENSEMBLE_UVT) {
            sprintf(linebuf, "OUTPUT: Grand Canonical Monte Carlo simulation running on %d cores\n", size);
            output(linebuf);
        } else {
            sprintf(linebuf, "OUTPUT: Canonical Monte Carlo simulation running on %d cores\n", size);
            output(linebuf);
        }
        sprintf(linebuf, "OUTPUT: Root collecting statistics at %s", ctime(&current_time));
        output(linebuf);
        sprintf(linebuf, "OUTPUT: Completed step %d/%d\n", i, system->numsteps);
        output(linebuf);
        sprintf(linebuf, "OUTPUT: %f sec/step, ETA = %.3f hrs\n", sec_step, sec_step*(system->numsteps - i)/3600.0);
        output(linebuf);
    }

    last_step = i;
    last_time = current_time;
}

```

```

        return(0);
    }

    void write_observables(FILE *fp_energy, observables_t *observables) {
        fprintf(fp_energy, "%f %f %f %f %f %f\n", observables->energy, observables->coulombic_energy, observables->rd_energy,
observables->polarization_energy, observables->kinetic_energy, observables->temperature, observables->N);
fflush(fp_energy);

}

int write_averages(avg_observables_t *averages) {
    if(averages->boltzmann_factor > 0.0)
        printf("OUTPUT: BF = %.3f +- %.3f\n", averages->boltzmann_factor, 0.5*averages->boltzmann_factor_error);

    if(averages->acceptance_rate > 0.0) {
        printf("OUTPUT: AR = %.3f (%.3f I/ %.3f R/ %.3f D", averages->acceptance_rate, averages->acceptance_rate_insert,
averages->acceptance_rate_remove, averages->acceptance_rate_displace);
        if(averages->acceptance_rate_adiabatic > 0.0) printf("/ %.3f A", averages->acceptance_rate_adiabatic);
        printf("\n");
    }

    if(averages->cavity_bias_probability > 0.0)
        printf("OUTPUT: Cavity bias probability = %.5f +- %.5f\n", averages->cavity_bias_probability,
0.5*averages->cavity_bias_probability_error);

    printf("OUTPUT: potential energy = %.3f +- %.3f K\n", averages->energy, 0.5*averages->energy_error);

    if(averages->coulombic_energy != 0.0)
        printf("OUTPUT: electrostatic energy = %.3f +- %.3f K\n", averages->coulombic_energy, 0.5*averages->coulombic_energy_error);

    printf("OUTPUT: repulsion/dispersion energy = %.3f +- %.3f K\n", averages->rd_energy, 0.5*averages->rd_energy_error);

    if(averages->polarization_energy != 0.0) {
        printf("OUTPUT: polarization energy = %.5f +- %.5f K", averages->polarization_energy,
0.5*averages->polarization_energy_error);
        if(averages->polarization_iterations != 0.0)
            printf(" iterations = %.if +- %.if rrms = %e +- %e)", averages->polarization_iterations,
0.5*averages->polarization_iterations_error, averages->dipole_rrms, 0.5*averages->dipole_rrms_error);
        printf("\n");
    }

    if(averages->kinetic_energy > 0.0) {
        printf("OUTPUT: kinetic energy = %.3f +- %.3f K\n", averages->kinetic_energy, 0.5*averages->kinetic_energy_error);
        printf("OUTPUT: temperature = %.3f +- %.3f K\n", averages->temperature, 0.5*averages->temperature);
    }

    printf("OUTPUT: N = %.3f +- %.3f molecules\n", averages->N, 0.5*averages->N_error);
    printf("OUTPUT: density = %.5f +- %.5f g/cm^3\n", averages->density, 0.5*averages->density_error);
    if(averages->pore_density != 0.0)
        printf("OUTPUT: pore density = %.5f +- %.5f g/cm^3\n", averages->pore_density, 0.5*averages->pore_density_error);

    if(averages->percent_wt > 0.0) {
        printf("OUTPUT: wt %% = %.5f +- %.5f %%\n", averages->percent_wt, 0.5*averages->percent_wt_error);
        printf("OUTPUT: wt %% (ME) = %.5f +- %.5f %%\n", averages->percent_wt_me, 0.5*averages->percent_wt_me_error);
    }

    if(averages->excess_ratio > 0.0)
        printf("OUTPUT: excess adsorption ratio = %.5f +- %.5f mg/g\n", averages->excess_ratio, 0.5*averages->excess_ratio_error);

    if(averages->qst > 0.0) {
        if(averages->qst_error > MAXDOUBLE) /* lack of error bar, avoid nan from sqrt(-small) */
            printf("OUTPUT: qst = %.3f +- N/A kJ/mol\n", averages->qst);
        else
            printf("OUTPUT: qst = %.3f +- %.3f kJ/mol\n", averages->qst, 0.5*averages->qst_error);
    }

    if(averages->heat_capacity > 0.0) {
        if(averages->heat_capacity_error > MAXDOUBLE)
            printf("OUTPUT: heat capacity = %.3f +- N/A kJ/mol K\n", averages->heat_capacity);
        else
            printf("OUTPUT: heat capacity = %.3f +- %.3f kJ/mol K\n", averages->heat_capacity,
0.5*averages->heat_capacity_error);
    }

    if(averages->compressibility > 0.0) {
        if(averages->compressibility_error > MAXDOUBLE) {
            printf("OUTPUT: compressibility = %.6f +- N/A atm^-1\n", averages->compressibility);
            printf("OUTPUT: bulk modulus = %.6f +- N/A GPa\n", ATM2PASCALS*1.0e-9/averages->compressibility);
        } else {
            printf("OUTPUT: compressibility = %.6f +- %.6f atm^-1\n", averages->compressibility,
            printf("OUTPUT: bulk modulus = %.6f +- %.6f GPa\n", ATM2PASCALS*1.0e-9/averages->compressibility,
ATM2PASCALS*1.0e-9/(0.5*averages->compressibility_error));
        }
    }

    printf("\n");
fflush(stdout);

    return(0);
}

/*
@2007, Jonathan Belof
Space Research Group
Department of Chemistry
University of South Florida
*/

```

```

#include <mc.h>

/* free a molecule and all of it's associated stuff */
void free_molecule(system_t *system, molecule_t *molecule) {
    int i;
    molecule->next = NULL;

#ifdef QM_ROTATION
    if(system->quantum_rotation && !molecule->frozen) {
        free(molecule->quantum_rotational_energies);
        free(molecule->quantum_rotational_eigensymmetry);
        for(i = 0; i < system->quantum_rotation_level_max; i++)
            free(molecule->quantum_rotational_eigenvectors[i]);
        free(molecule->quantum_rotational_eigenvectors);
    }
#endif /* QM_ROTATION */

    free_pairs(molecule);
    free_atoms(molecule);
    free(molecule);
}

void free_pairs(molecule_t *molecules) {
    int i;
    pair_t **ptr_array;
    molecule_t *molecule_ptr;
    atom_t *atom_ptr;
    pair_t *pair_ptr;

    /* build an array of ptrs to be freed */
    for(molecule_ptr = molecules, i = 0, ptr_array = NULL; molecule_ptr; molecule_ptr = molecule_ptr->next) {
        for(atom_ptr = molecule_ptr->atoms; atom_ptr; atom_ptr = atom_ptr->next) {
            for(pair_ptr = atom_ptr->pairs; pair_ptr; pair_ptr = pair_ptr->next) {
                ptr_array = realloc(ptr_array, sizeof(pair_t *)*(i + 1));
                ptr_array[i] = pair_ptr;
                ++i;
            }
        }
    }

    /* free the whole array of ptrs */
    for(--i; i >= 0; i--) free(ptr_array[i]);

    /* zero out the heads */
    for(molecule_ptr = molecules; molecule_ptr; molecule_ptr = molecule_ptr->next)
        for(atom_ptr = molecule_ptr->atoms; atom_ptr; atom_ptr = atom_ptr->next)
            atom_ptr->pairs = NULL;

    /* free our temporary array */
    if(ptr_array) free(ptr_array);
}

void free_atoms(molecule_t *molecules) {
    int i, n;
    atom_t **ptr_array;
    molecule_t *molecule_ptr;
    atom_t *atom_ptr;

    /* build the ptr array */
    for(molecule_ptr = molecules, i = 0, n = 0, ptr_array = NULL; molecule_ptr; molecule_ptr = molecule_ptr->next) {
        for(atom_ptr = molecule_ptr->atoms; atom_ptr; atom_ptr = atom_ptr->next) {
            ptr_array = realloc(ptr_array, sizeof(atom_t *)*(i + 1));
            ptr_array[i] = atom_ptr;
            ++i, ++n;
        }
    }

    /* free the whole array of ptrs */
    for(i = 0; i < n; i++) free(ptr_array[i]);

    /* free our temporary array */
    free(ptr_array);
}

void free_molecules(molecule_t *molecules) {
    int i;
    molecule_t **ptr_array = NULL;
    molecule_t *molecule_ptr;

    /* build the ptr array */
    for(molecule_ptr = molecules, i = 0; molecule_ptr; molecule_ptr = molecule_ptr->next) {
        ptr_array = realloc(ptr_array, sizeof(molecule_t *)*(i + 1));
        ptr_array[i] = molecule_ptr;
        ++i;
    }

    /* free the whole array of ptrs */
    for(--i; i >= 0; i--) free(ptr_array[i]);

    /* free our temporary array */
    free(ptr_array);
}

```

```

/* free the polarization matrices */
void free_matrices(system_t *system) {
    int i, N;
    N = 3*system->checkpoint->N_atom;
    for(i = 0; i < N; i++) {
        free(system->A_matrix[i]);
        if(!system->polar_iterative)
            free(system->B_matrix[i]);
    }
    free(system->A_matrix);
    free(system->B_matrix);
}

/* free the cavity bias insertion grid */
void free_cavity_grid(system_t *system) {
    int i, j, N;
    N = system->cavity_grid_size;
    for(i = 0; i < N; i++) {
        for(j = 0; j < N; j++)
            free(system->cavity_grid[i][j]);
        free(system->cavity_grid[i]);
    }
    free(system->cavity_grid);
}

#endif /* QM_ROTATION */

/* free structures associated with quantum rotations */
void free_rotational(system_t *system) {
    int i;
    molecule_t *molecule_ptr;
    for(molecule_ptr = system->molecules; molecule_ptr; molecule_ptr = molecule_ptr->next) {
        if(!molecule_ptr->frozen) {
            free(molecule_ptr->quantum_rotational_energies);
            free(molecule_ptr->quantum_rotational_eigensymmetry);
            for(i = 0; i < system->quantum_rotation_level_max; i++)
                free(molecule_ptr->quantum_rotational_eigenvalues[i]);
            free(molecule_ptr->quantum_rotational_eigenvectors);
        }
    }
}
#endif /* QM_ROTATION */

/* free all of our data structures */
void cleanup(system_t *system) {
#ifdef QM_ROTATION
    if(system->quantum_rotation) free_rotational(system);
#endif /* QM_ROTATION */
    free_pairs(system->molecules);
    free_atoms(system->molecules);
    free_molecules(system->molecules);

    free(system->pdb_input);
    free(system->pdb_output);
    free(system->energy_output);

    if(system->traj_output) free(system->traj_output);
    if(system->dipole_output) free(system->dipole_output);
    if(system->field_output) free(system->field_output);

    if(system->polarization) free_matrices(system);
    if(system->cavity_bias) free_cavity_grid(system);
    free(system);
}
#endif /* on SIGTERM, cleanup and exit */
void terminate_handler(int sigtype, system_t *sys_ptr) {
    static system_t *system;
    switch(sigtype) {
        case SIGTERM:
            output("CLEANUP: ***** SIGTERM received, exiting *****\n");
            close_files(sys_ptr);
            cleanup(system);
#ifdef MPI
            if(!rank) close_files(sys_ptr);
            MPI_Finalize();
#else
            exit(1);
#endif /* MPI */
            break;
    }
}

```

```

        case SIGUSR1:
            output("CLEANUP: ***** SIGUSR1 received, exiting *****\n");
            close_files(sys_ptr);
            cleanup(system);
#endif MPI
            if(!rank) close_files(sys_ptr);
            MPI_Finalize();
#else
            exit(1);
#endif /* MPI */
            break;

        case SIGUSR2:
            output("CLEANUP: ***** SIGUSR2 received, exiting *****\n");
            close_files(sys_ptr);
            cleanup(system);
#endif MPI
            if(!rank) close_files(sys_ptr);
            MPI_Finalize();
#else
            exit(1);
#endif /* MPI */
            break;
        case -1: /* install the static ptr */
            system = sys_ptr;
            break;
    }
}

/*
@2007, Jonathan Belof
Space Research Group
Department of Chemistry
University of South Florida
*/
int rank, size;
#include <mc.h>

void usage(char *progname) {
    if(!rank) fprintf(stderr, "usage: %s <config>\n", progname);
#ifndef MPI
    MPI_Finalize();
#else
    exit(1);
#endif /* MPI */
}

void seed_rng(long int seed) {
    rule30_rng(seed);
}

double get_rand(void) {
    return(rule30_rng(0));
}

int main(int argc, char **argv) {
    int i, j, N;
    molecule_t *molecule_ptr;
    atom_t *atom_ptr;
    char linebuf[MAXLINE];
    char input_file[MAXLINE];
    system_t *system;

    /* set the default rank */
    rank = 0; size = 1;

    /* check args */
    if(argc < 2) usage(argv[0]);

    if(!argv[1]) {
        error("MAIN: invalid config file specified");
        exit(1);
    }

    /* start up the MPI chain */
#ifndef MPI
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    sprintf(linebuf, "MAIN: processes started on %d cores\n", size);
    output(linebuf);
#endif /* MPI */

    /* get the config file arg */
    strcpy(input_file, argv[1]);
    sprintf(linebuf, "MAIN: running parameters found in %s\n", input_file);
    output(linebuf);

    /* read the input files and setup the simulation */
    system = setup_system(input_file);
    if(!system) {
        error("MAIN: could not initialize the simulation\n");
#ifndef MPI

```

```

        MPI_Finalize();
#else
    exit(1);
#endif /* MPI */
} else {
    output("MAIN: the simulation has been initialized\n");
}

/* install the signal handler to catch SIGTERM cleanly */
terminate_handler(-1, system);
signal(SIGTERM, ((void *)(terminate_handler)));
signal(SIGUSR1, ((void *)(terminate_handler)));
signal(SIGUSR2, ((void *)(terminate_handler)));
output("MAIN: signal handler installed\n");

/* allocate space for the statistics */
system->nodestats = calloc(1, sizeof(nodestats_t));
system->avg_nodestats = calloc(1, sizeof(avg_nodestats_t));
system->observables = calloc(1, sizeof(observables_t));
system->avg_observables = calloc(1, sizeof(avg_observables_t));
system->checkpoint = calloc(1, sizeof(checkpoint_t));
system->checkpoint->observables = calloc(1, sizeof(observables_t));
system->grids = calloc(1, sizeof(grid_t));
system->grids->histogram = calloc(1, sizeof(histogram_t));
system->grids->avg_histogram = calloc(1, sizeof(histogram_t));

if(!system->observables || !system->avg_observables || !system->checkpoint || !system->checkpoint->observables) {
    error("MAIN: couldn't allocate space for statistics and/or checkpoint\n");
    exit(1);
}

/* if polarization active, allocate the necessary matrices */
if(system->polarization) {

    /* count the number of atoms initially in the system */
    for(molecule_ptr = system->molecules, N = 0; molecule_ptr; molecule_ptr = molecule_ptr->next)
        for(atom_ptr = molecule_ptr->atoms; atom_ptr; atom_ptr = atom_ptr->next)
            ++N;

    system->A_matrix = calloc(3*N, sizeof(double *));
    for(i = 0; i < 3*N; i++) system->A_matrix[i] = calloc(3*N, sizeof(double));

    if(!system->polar_iterative) {
        system->B_matrix = calloc(3*N, sizeof(double *));
        for(i = 0; i < 3*N; i++) system->B_matrix[i] = calloc(3*N, sizeof(double));
    }
}

/* if histogram calculation flag is set, allocate grid */
if(system->calc_hist){
    setup_histogram(system);
    allocate_histogram_grid(system);
}

/* seed the rng */
seed_rng(system->seed + rank);
output("MAIN: the random seeds on the compute cores are all set\n");

#endif /* MPI */

/* start the MC simulation */
if(system->ensemble == ENSEMBLE_UVT) {
    output("MAIN: ****\n");
    output("MAIN: *** starting Grand Canonical Monte Carlo simulation ***\n");
    output("MAIN: ****\n");
} else if(system->ensemble == ENSEMBLE_NVT) {
    output("MAIN: ****\n");
    output("MAIN: *** starting Canonical Monte Carlo simulation ***\n");
    output("MAIN: ****\n");
} else if(system->ensemble == ENSEMBLE_NVE) {
    output("MAIN: ****\n");
    output("MAIN: *** starting Microcanonical Monte Carlo simulation ***\n");
    output("MAIN: ****\n");
} else if(system->ensemble == ENSEMBLE_SURF) { /* surface run */
    output("MAIN: ****\n");
    output("MAIN: *** starting potential energy surface calculation ***\n");
    output("MAIN: ****\n");
} else if(system->ensemble == ENSEMBLE_SURF_FIT) { /* surface fitting run */
    output("MAIN: ****\n");
    output("MAIN: *** starting potential energy surface fitting calculation ***\n");
    output("MAIN: ****\n");
}

if(!(system->ensemble == ENSEMBLE_SURF) || (system->ensemble == ENSEMBLE_SURF_FIT)) {

    if(mc(system) < 0) {
        error("MAIN: MC failed on error, exiting\n");
    #ifdef MPI
        MPI_Finalize();
    #else
        exit(1);
    #endif /* MPI */
    } else {
        if(system->ensemble == ENSEMBLE_UVT) {
            output("MAIN: ****\n");
            output("MAIN: *** finishing Grand Canonical Monte Carlo simulation ***\n");
            output("MAIN: ****\n");
        } else if(system->ensemble == ENSEMBLE_NVT) {
            output("MAIN: ****\n");
            output("MAIN: *** finishing Canonical Monte Carlo simulation ***\n");
            output("MAIN: ****\n");
        }
    }
}

```



```

in_reg6 >= RHS_ONE;
in_reg6 |= ((in_reg5 & RHS_ONE) << (WORDSIZE - 1));
in_reg5 >= RHS_ONE;
in_reg5 |= ((in_reg4 & RHS_ONE) << (WORDSIZE - 1));
in_reg4 >= RHS_ONE;
in_reg3 |= ((in_reg3 & RHS_ONE) << (WORDSIZE - 1));
in_reg2 >= RHS_ONE;
in_reg1 |= ((in_reg1 & RHS_ONE) << (WORDSIZE - 1));
in_reg1 >= RHS_ONE;
in_reg1 |= mp & LHS_ONE;

}

/* now must rotate output registers one bit to the left */
mp &= LHS_ZERO; /* clear the carry bit */
mp |= out_reg1 & LHS_ONE; /* set the carry bit if needed */
out_reg1 <<= RHS_ONE;
out_reg1 |= ((out_reg2 & LHS_ONE) >> (WORDSIZE - 1));
out_reg2 <<= RHS_ONE;
out_reg2 |= ((out_reg3 & LHS_ONE) >> (WORDSIZE - 1));
out_reg3 <<= RHS_ONE;
out_reg3 |= ((out_reg4 & LHS_ONE) >> (WORDSIZE - 1));
out_reg4 <<= RHS_ONE;
out_reg4 |= ((out_reg5 & LHS_ONE) >> (WORDSIZE - 1));
out_reg5 <<= RHS_ONE;
out_reg5 |= ((out_reg6 & LHS_ONE) >> (WORDSIZE - 1));
out_reg6 <<= RHS_ONE;
out_reg6 |= ((out_reg7 & LHS_ONE) >> (WORDSIZE - 1));
out_reg7 <<= RHS_ONE;
out_reg7 |= ((mp & LHS_ONE) >> (WORDSIZE - 1));

/* set output bits of random number */
random_result_int |= ((out_reg4 & CENTER_MASK) >> DELTA_CENTER) << ((DELTA_MANTISSA - 1) - ((mp & OUTER_COUNT) >> DELTA_COUNT));

/* swap the input and output registers */
in_reg1 = out_reg1;
in_reg2 = out_reg2;
in_reg3 = out_reg3;
in_reg4 = out_reg4;
in_reg5 = out_reg5;
in_reg6 = out_reg6;
in_reg7 = out_reg7;

/* clear output registers */
out_reg1 = out_reg2 = out_reg3 = out_reg4 = out_reg5 = out_reg6 = out_reg7 = 0;

}

/* save last state point to static memory */
last_reg1 = in_reg1;
last_reg2 = in_reg2;
last_reg3 = in_reg3;
last_reg4 = in_reg4;
last_reg5 = in_reg5;
last_reg6 = in_reg6;
last_reg7 = in_reg7;

random_result = (double)random_result_int;
random_result /= (double)MAX_MANTISSA; /* ensure that result is normalized from 0 to 1 */
return(random_result);
}

#else /* 32-bit */

double rule30_rng(unsigned long int seed) {

register unsigned long int rule = RULE30; /* the rule to enforce */
register unsigned long int in_reg1 = 0, /* input registers */
    in_reg2 = 0,
    in_reg3 = 0;
register unsigned long int out_reg1 = 0, /* output registers */
    out_reg2 = 0,
    out_reg3 = 0;
register unsigned long int mp = 0; /* multi-purpose register:
    * - the right-most 8 bits are for the inner loop counter
    * - the next 16 bits are for the outer loop counter
    * - the left-most bit is for carries
*/
static unsigned long int last_reg1, /* static memory addrs to store results from the current run */
    last_reg2,
    last_reg3;

double random_result = 0; /* return a double from 0.0 to 1.0 */
unsigned long long int random_result_int = 0; /* integer version of the above for boolean ops */

/* start with initial config */
if(seed) {
    in_reg1 = in_reg2 = in_reg3 = seed;
} else {
    /* already seeded - restore state from memory */
    in_reg1 = last_reg1;
    in_reg2 = last_reg2;
    in_reg3 = last_reg3;
}

for((mp &= OUTER_ZERO); ((mp & OUTER_COUNT) >> DELTA_COUNT) < DELTA_MANTISSA; mp += OUTER_ONE) {/* -- notice the fact that the
increment here */
    for((mp &= INNER_ZERO); (mp & INNER_COUNT) < WORDSIZE; mp += INNER_ONE) { /* will blow away the low-order bits
doesn't matter */

        /* mask off first three bits and compare with rule */
        /* set the output register bit appropriately */
        out_reg1 |= ((rule >> (in_reg1 & CELL_MASK)) & RHS_ONE) << (mp & INNER_COUNT);
        out_reg2 |= ((rule >> (in_reg2 & CELL_MASK)) & RHS_ONE) << (mp & INNER_COUNT);

    }
}
}

```

```

out_reg3 |= ((rule >> (in_reg3 & CELL_MASK)) & RHS_ONE) << (mp & INNER_COUNT);

/* rotate all input registers one bit to the right, preserve carry */
mp &= LHS_ZERO; /* clear the carry bit */
mp |= ((in_reg3 & RHS_ONE) << (WORDSIZE - 1)); /* set carry bit if needed */
in_reg3 >>= RHS_ONE;
in_reg3 |= ((in_reg2 & RHS_ONE) << (WORDSIZE - 1));
in_reg2 >>= RHS_ONE;
in_reg2 |= ((in_reg1 & RHS_ONE) << (WORDSIZE - 1));
in_reg1 >>= RHS_ONE;
in_reg1 |= mp & LHS_ONE;

}

/* now must rotate output registers one bit to the left */
mp &= LHS_ZERO; /* clear the carry bit */
mp |= out_reg1 & RHS_ONE; /* set the carry bit if needed */
out_reg1 <<= RHS_ONE;
out_reg1 |= ((out_reg2 & RHS_ONE) >> (WORDSIZE - 1));
out_reg2 <<= RHS_ONE;
out_reg2 |= ((out_reg3 & RHS_ONE) >> (WORDSIZE - 1));
out_reg3 <<= RHS_ONE;
out_reg3 |= ((mp & RHS_ONE) >> (WORDSIZE - 1));

/* set output bits of random sequence */
random_result_int |= ((out_reg2 & CENTER_MASK) >> DELTA_CENTER) << ((DELTA_MANTISSA - 1) - ((mp & OUTER_COUNT) >> DELTA_COUNT));

/* swap the input and output registers */
in_reg1 = out_reg1;
in_reg2 = out_reg2;
in_reg3 = out_reg3;

/* clear the output registers */
out_reg1 = out_reg2 = out_reg3 = 0;

}

/* save the last state point in static memory */
last_reg1 = in_reg1;
last_reg2 = in_reg2;
last_reg3 = in_reg3;

random_result = (double)random_result_int;
random_result /= (double)MAX_MANTISSA; /* ensure that result is normalized from 0 to 1 */
return(random_result);

}

#endif /* WORDSIZE == 64 */

/*
* 2007, Jonathan Belof
Space Research Group
Department of Chemistry
University of South Florida
*/
#include <mc.h>

/* check whether a point (x,y,z) lies within an empty cavity */
/* if so, return 1 */
int is_point_empty(system_t *system, double x, double y, double z) {

    int i, j, k;
    int incavity;
    double r;

    incavity = 0;
    for(i = 0; i < system->cavity_grid_size; i++) {
        for(j = 0; j < system->cavity_grid_size; j++) {
            for(k = 0; k < system->cavity_grid_size; k++) {
                if(!system->cavity_grid[i][j][k].occupancy) {
                    r = pow((x - system->cavity_grid[i][j][k].pos[0]), 2.0);
                    r += pow((y - system->cavity_grid[i][j][k].pos[1]), 2.0);
                    r += pow((z - system->cavity_grid[i][j][k].pos[2]), 2.0);
                    r = sqrt(r);
                    if(r < system->cavity_radius)
                        incavity = 1;
                }
            }
        }
    }
    return(incavity);
}

/* total volume of accessible cavities via Monte Carlo integration */
void cavity_volume(system_t *system) {

    int throw, hits, num_darts;
    int p, q;
    double pos_vec[3], grid_vec[3];
    double fraction_hits;

    /* good rule of thumb is 1 per 10 A^3 */
    num_darts = system->pbc->volume*DARTSCALE;

    /* throw random darts and count the number of hits */
}

```

```

for(throw = 0, hits = 0; throw < num_darts; throw++) {
    /* generate a random grid position */
    for(p = 0; p < 3; p++) grid_vec[p] = -0.5 + get_rand();

    /* zero the coordinate vector */
    for(p = 0; p < 3; p++) pos_vec[p] = 0;

    /* linear transform vector into real coordinates */
    for(p = 0; p < 3; p++)
        for(q = 0; q < 3; q++)
            pos_vec[p] += system->pbc->basis[p][q]*grid_vec[q];

    /* check if the random point lies within an empty cavity */
    if(is_point_empty(system, pos_vec[0], pos_vec[1], pos_vec[2])) ++hits;
}

/* determine the percentage of free cavity space */
fraction_hits = ((double)hits)/((double)num_darts);
system->cavity_volume = fraction_hits*system->pbc->volume; /* normalize w.r.t. the cell volume */
}

/* probability of finding an empty cavity on the grid */
void cavity_probability(system_t *system) {

    int i, j, k;
    double probability;
    int total_points;

    /* total number of potential cavities */
    total_points = system->cavity_grid_size*system->cavity_grid_size*system->cavity_grid_size;

    /* find the number of open cavities */
    system->cavities_open = 0;
    for(i = 0; i < system->cavity_grid_size; i++)
        for(j = 0; j < system->cavity_grid_size; j++)
            for(k = 0; k < system->cavity_grid_size; k++)
                if(system->cavity_grid[i][j][k].occupancy) ++system->cavities_open;

    /* the overall probability ratio */
    probability = ((double)system->cavities_open)/((double)total_points);

    /* update the observable */
    system->nodestats->cavity_bias_probability = probability;
}

/* allocate the grid */
void setup_cavity_grid(system_t *system) {

    int i, j;

    system->cavity_grid = calloc(system->cavity_grid_size, sizeof(cavity_t *));
    for(i = 0; i < system->cavity_grid_size; i++) {
        system->cavity_grid[i] = calloc(system->cavity_grid_size, sizeof(cavity_t *));
        for(j = 0; j < system->cavity_grid_size; j++)
            system->cavity_grid[i][j] = calloc(system->cavity_grid_size, sizeof(cavity_t));
    }
}

/* create a 3D histogram of atoms lying within a sphere centered at each grid point */
void cavity_update_grid(system_t *system) {

    int i, j, k, G;
    int p, q;
    molecule_t *molecule_ptr;
    atom_t *atom_ptr;
    double grid_component[3];
    double grid_vector[3];
    double r;

    G = system->cavity_grid_size;

    /* clear the grid */
    for(i = 0; i < G; i++) {
        for(j = 0; j < G; j++) {
            for(k = 0; k < G; k++) {
                system->cavity_grid[i][j][k].occupancy = 0;
                for(p = 0; p < 3; p++)
                    system->cavity_grid[i][j][k].pos[p] = 0;
            }
        }
    }

    /* loop over the grid, bin a sphere when needed */
    for(i = 0; i < G; i++) {
        for(j = 0; j < G; j++) {
            for(k = 0; k < G; k++) {

                /* divide up each grid component */
                grid_component[0] = ((double)(i + 1))/((double)(G + 1));
                grid_component[1] = ((double)(j + 1))/((double)(G + 1));
                grid_component[2] = ((double)(k + 1))/((double)(G + 1));

                /* project the grid point onto our actual basis */
                for(p = 0; p < 3; p++)
                    for(q = 0, grid_vector[p] = 0; q < 3; q++)
                        grid_vector[p] += system->pbc->basis[p][q]*grid_component[q];

                /* put into real coordinates */
                for(p = 0; p < 3; p++)

```

```

        for(q = 0; q < 3; q++)
            grid_vector[p] -= 0.5*system->pbc->basis[p][q];

        /* if an atomic coordinate lies within a sphere centered on the grid point, then bin it */
        for(molecule_ptr = system->molecules; molecule_ptr != molecule_ptr->next) {
            for(atom_ptr = molecule_ptr->atoms; atom_ptr != atom_ptr->next) {

                /* get the displacement from the grid point */
                for(p = 0, r = 0; p < 3; p++)
                    r += (grid_vector[p] - atom_ptr->wrapped_pos[p])*(grid_vector[p] -
atom_ptr->wrapped_pos[p]);
                r = sqrt(r);

                /* inside the sphere? */
                if(r < system->cavity_radius) ++system->cavity_grid[i][j][k].occupancy;

            } /* for atom */
        } /* for molecule */

        /* store the location of this grid point */
        for(p = 0; p < 3; p++)
            system->cavity_grid[i][j][k].pos[p] = grid_vector[p];

    } /* for k */
} /* for j */
} /* for i */

/* update the cavity insertion probability estimate */
cavity_probability(system);
/* update the accessible insertion volume */
cavity_volume(system);

}

#ifndef DEBUG
void test_cavity_grid(system_t *system) {

    int i, j, k, G;
    G = system->cavity_grid_size;

    printf("\n");
    for(i = 0; i < G; i++) {
        for(j = 0; j < G; j++) {
            for(k = 0; k < G; k++)
                printf("%d ", system->cavity_grid[i][j][k].occupancy);
            printf("\n");
        }
        printf("\n");
        fflush(stdout);
    }
    printf("\n");
    #endif /* DEBUG */
}

/*
@2007, Jonathan Belof
Space Research Group
Department of Chemistry
University of South Florida
*/
#include <mc.h>

/* the prime quantity of interest */
void boltzmann_factor(system_t *system, double initial_energy, double final_energy) {

    double delta_energy;
    double fugacity;

    delta_energy = final_energy - initial_energy;

    if(system->h2_fugacity || system->co2_fugacity)
        fugacity = system->fugacity;
    else
        fugacity = system->pressure;

    if(system->ensemble == ENSEMBLE_UVT) {

        /* if biased_move not set, no cavity available so do normal evaluation below */
        if(system->cavity_bias && system->checkpoint->biased_move) {

            /* modified metropolis function */
            if(system->checkpoint->movetype == MOVETYPE_INSERT) { /* INSERT */
                system->nodestats->boltzmann_factor =
(system->cavity_volume*system->avg_nodestats->cavity_bias_probability*fugacity*ATM2REDUCED/(system->temperature*system->observables->N))*exp(-delta_energy/system->temperature);
            } else if(system->checkpoint->movetype == MOVETYPE_REMOVE) { /* REMOVE */
                system->nodestats->boltzmann_factor = (system->temperature*(system->observables->N +
1))/(system->cavity_volume*system->avg_nodestats->cavity_bias_probability*fugacity*ATM2REDUCED)*exp(-delta_energy/system->temperature);
            } else { /* DISPLACE */
                system->nodestats->boltzmann_factor = exp(-delta_energy/system->temperature);
            }
        } else {

            if(system->checkpoint->movetype == MOVETYPE_INSERT) { /* INSERT */
                system->nodestats->boltzmann_factor =
(system->pbc->volume*fugacity*ATM2REDUCED/(system->temperature*system->observables->N))*exp(-delta_energy/system->temperature);
            } else if(system->checkpoint->movetype == MOVETYPE_REMOVE) { /* REMOVE */
                system->nodestats->boltzmann_factor = (system->temperature*(system->observables->N +
1))/(system->pbc->volume*fugacity*ATM2REDUCED)*exp(-delta_energy/system->temperature);
            }
        }
    }
}

```

```

    } else { /* DISPLACE */
        system->nodestats->boltzmann_factor = exp(-delta_energy/system->temperature);
    }
}

} else if(system->ensemble == ENSEMBLE_NVT) {
    system->nodestats->boltzmann_factor = exp(-delta_energy/system->temperature);
} else if(system->ensemble == ENSEMBLE_NVE) {
    system->nodestats->boltzmann_factor = pow((system->total_energy - final_energy), 3.0*system->N/2.0);
    system->nodestats->boltzmann_factor /= pow((system->total_energy - initial_energy), 3.0*system->N/2.0);
}

/* keep track of which specific moves were accepted */
void register_accept(system_t *system) {
    ++system->nodestats->accept;
    switch(system->checkpoint->movetype) {
        case MOVE_TYPE_INSERT:
            ++system->nodestats->accept_insert;
            break;
        case MOVE_TYPE_REMOVE:
            ++system->nodestats->accept_remove;
            break;
        case MOVE_TYPE_DISPLACE:
            ++system->nodestats->accept_displace;
            break;
        case MOVE_TYPE_ADIABATIC:
            ++system->nodestats->accept_adiabatic;
            break;
    }
}

/* keep track of which specific moves were rejected */
void register_reject(system_t *system) {
    ++system->nodestats->reject;
    switch(system->checkpoint->movetype) {
        case MOVE_TYPE_INSERT:
            ++system->nodestats->reject_insert;
            break;
        case MOVE_TYPE_REMOVE:
            ++system->nodestats->reject_remove;
            break;
        case MOVE_TYPE_DISPLACE:
            ++system->nodestats->reject_displace;
            break;
        case MOVE_TYPE_ADIABATIC:
            ++system->nodestats->reject_adiabatic;
            break;
    }
}

/* implements the Markov chain */
int mc(system_t *system) {
    int i, j, msgsize;
    double initial_energy, final_energy, delta_energy;
    observables_t *observables_mpi;
    avg_nodestats_t *avg_nodestats_mpi;
    char *snd_strct, *rcv_strct;
#ifndef MPI
    MPI_Datatype msgtype;
#endif /* MPI */

    /* allocate the statistics structures */
    observables_mpi = calloc(1, sizeof(observables_t));
    avg_nodestats_mpi = calloc(1, sizeof(avg_nodestats_t));
    if(!(observables_mpi && avg_nodestats_mpi)) {
        error("MC: couldn't allocate space for statistics\n");
        return(-1);
    }

    /* if MPI not defined, then compute message size like dis */
    msgsize = sizeof(observables_t) + sizeof(avg_nodestats_t);
    if(system->calc_hist)
        msgsize += system->n_histogram_bins*sizeof(int);
#ifndef MPI
    MPI_Type_contiguous(msgsize, MPI_BYTE, &msgtype);
    MPI_Type_commit(&msgtype);
#endif /* MPI */

    /* allocate MPI structures */
    snd_strct = calloc(msgsize, 1);
    if(!snd_strct) {
        error("MC: couldn't allocate MPI message send space!\n");
        return(-1);
    }

    if(!rank) {
        rcv_strct = calloc(size, msgsize);
        if(!rcv_strct) {
            error("MC: root couldn't allocate MPI message receive space!\n");
            return(-1);
        }
    }

    /* clear our statistics */
    clear_nodestats(system->nodestats);
}

```

```

clear_node_averages(system->avg_nodestats);
clear_observables(system->observables);
clear_root_averages(system->avg_observables);

/* update the grid for the first time */
if(system->cavity_bias) cavity_update_grid(system);

/* determine the initial number of atoms in the simulation */
system->checkpoint->N_atom = num_atoms(system);
system->checkpoint->N_atom_prev = system->checkpoint->N_atom;

/* get the initial energy of the system */
initial_energy = energy(system);

/* be a bit forgiving of the initial state */
if(!finite(initial_energy)) {
    initial_energy = MAXVALUE;
    system->observables->energy = MAXVALUE;
}

/* set the initial values */
track_ar(system->nodestats);
update_nodestats(system->nodestats, system->avg_nodestats);
update_root_averages(system, system->observables, system->avg_nodestats, system->avg_observables);

/* if root, open necessary output files */
if(!rank) {
    output("MC: initial values:\n");
    write_averages(system->avg_observables);
    if(open_files(system) < 0) {
        error("MC: could not open files\n");
        return(-1);
    }
}

/* save the initial state */
checkpoint(system);

/* main MC loop */
for(i = 1; i <= system->numsteps; i++) {

    /* restore the last accepted energy */
    initial_energy = system->observables->energy;

    /* perturb the system */
    make_move(system);

    /* calculate the energy change */
    final_energy = energy(system);
    delta_energy = final_energy - initial_energy;
    /* treat a bad contact as a reject */
    if(!finite(final_energy)) {
        system->observables->energy = MAXVALUE;
        system->nodestats->boltzmann_factor = 0;
    } else {
        boltzmann_factor(system, initial_energy, final_energy);
    }

    /* Metropolis function */
    if(get_rand() < system->nodestats->boltzmann_factor) { /* ACCEPT */

        /* checkpoint */
        checkpoint(system);
        register_accept(system);

        /* SA */
        if(system->simulated_annealing) system->temperature *= system->simulated_annealing_schedule;

    } else { /* REJECT */

        /* restore from last checkpoint */
        restore(system);
        register_reject(system);

    }

    /* track the acceptance_rate */
    track_ar(system->nodestats);

    /* each node calculates it's stats */
    update_nodestats(system->nodestats, system->avg_nodestats);

    /* do this every correlation time, and at the very end */
    if(!(i % system->corrtime) || (i == system->numsteps)) {

#endif /* QM_ROTATION
/* solve for the rotational energy levels */
if(system->quantum_rotation) quantum_system_rotational_energies(system);
#endif /* QM_ROTATION */

/* copy observables and avgs to the mpi send buffer */
/* histogram array is at the end of the message */
if(system->calc_hist) {
    zero_grid(system->grids->histogram->grid, system);
    population_histogram(system);
}

/* zero the send buffer */
memset(snd_strct, 0, msgsize);
memcpy((snd_strct + sizeof(observables_t)), system->avg_nodestats, sizeof(avg_nodestats_t));
if(system->calc_hist)
    memcpy((snd_strct + sizeof(observables_t)), system->avg_nodestats, sizeof(avg_nodestats_t));
    mpi_copy_histogram_to_sendbuffer(snd_strct + sizeof(observables_t) + sizeof(avg_nodestats_t),
system->grids->histogram->grid, system);
}

```

```

        if(!rank) memset(rcv_strct, 0, size*msgsize);

#ifdef MPI
        MPI_Gather(snd_strct, 1, msgtype, rcv_strct, 1, msgtype, 0, MPI_COMM_WORLD);
#else
        memcpy(rcv_strct, snd_strct, msgsize);
#endif /* MPI */
/* head node collects all observables and averages */
if(!rank) {
    for(j = 0; j < size; j++) {
        /* copy from the mpi buffer */
        memcpy(observables_mpi, rcv_strct + j*msgsize, sizeof(observables_t));
        memcpy(avg_nodestats_mpi, rcv_strct + j*msgsize + sizeof(observables_t), sizeof(avg_nodestats_t));
        if(system->calc_hist)
            mpi_copy_rcv_histogram_to_data(rcv_strct + j*msgsize + sizeof(observables_t) +
                sizeof(avg_nodestats_t), system->grids->histogram->grid, system);

        /* collect the averages */
        update_root_averages(system, observables_mpi, avg_nodestats_mpi, system->avg_observables);
        if(system->calc_hist) update_root_histogram(system);
        if(system->file_pointers.fp_energy) write_observables(system->file_pointers.fp_energy,
observables_mpi);
    }
    /* XXX - this needs to be fixed, currently only writing the root node's states */
    if(system->file_pointers.fp_traj) write_states(system->file_pointers.fp_traj, system->molecules);

    /* write the averages to stdout */
    if(system->file_pointers.fp_histogram)
        write_histogram(system->file_pointers.fp_histogram, system->grids->avg_histogram->grid, system);

    if(write_performance(i, system) < 0) {
        error("MC: could not write performance data to stdout\n");
        return(-1);
    }
    if(write_averages(system->avg_observables) < 0) {
        error("MC: could not write statistics to stdout\n");
        return(-1);
    }
    if(write_molecules(system, system->pdb_restart) < 0) {
        error("MC: could not write restart state to disk\n");
        return(-1);
    }
}
/* !rank */
} /* corrtime */
} /* main loop */

/* write output, close any open files */
free(snd_strct);
if(!rank) {
    if(write_molecules(system, system->pdb_output) < 0) {
        error("MC: could not write final state to disk\n");
        return(-1);
    }
    close_files(system);
    free(observables_mpi);
    free(avg_nodestats_mpi);
    free(rcv_strct);
}
return(0);
}

/*
@2007, Jonathan Belof
Space Research Group
Department of Chemistry
University of South Florida
*/
/* herein lie sleeping demons */
/* i truly hate this fucking file, but dont see a better way JB */

#include <mc.h>

/* make an exact copy of src */
molecule_t *copy_molecule(system_t *system, molecule_t *src) {
    int i, j;
    molecule_t *dst;
    atom_t *atom_dst_ptr, *prev_atom_dst_ptr, *atom_src_ptr;
    pair_t *pair_dst_ptr, *prev_pair_dst_ptr, *pair_src_ptr;

    /* allocate the start of the new lists */
    dst = calloc(1, sizeof(molecule_t));
    /* copy molecule attributes */
    dst->id = src->id;
    strcpy(dst->moleculename, src->moleculename);
    dst->mass = src->mass;
    dst->frozen = src->frozen;
    dst->adiabatic = src->adiabatic;
    memcpy(dst->com, src->com, 3*sizeof(double));
    memcpy(dst->wrapped_com, src->wrapped_com, 3*sizeof(double));

#ifdef QM_ROTATION
    if(system->quantum_rotation) {
        dst->quantum_rotational_energies = calloc(system->quantum_rotation_level_max, sizeof(double));

```

```

dst->quantum_rotational_eigenvectors = calloc(system->quantum_rotation_level_max, sizeof(complex_t *));
for(i = 0; i < system->quantum_rotation_level_max; i++)
    dst->quantum_rotational_eigenvectors[i] = calloc((system->quantum_rotation_l_max +
1)*(system->quantum_rotation_l_max + 1), sizeof(complex_t));
    dst->quantum_rotational_eigensymmetry = calloc(system->quantum_rotation_level_max, sizeof(int));

    memcpy(dst->quantum_rotational_energies, src->quantum_rotational_energies,
system->quantum_rotation_level_max*sizeof(double));
    for(i = 0; i < system->quantum_rotation_level_max; i++) {
        for(j = 0; j < (system->quantum_rotation_l_max + 1)*(system->quantum_rotation_l_max + 1); j++) {
            dst->quantum_rotational_eigenvectors[i][j].real = src->quantum_rotational_eigenvectors[i][j].real;
            dst->quantum_rotational_eigenvectors[i][j].imaginary = src->quantum_rotational_eigenvectors[i][j].imaginary;
        }
    }
    memcpy(dst->quantum_rotational_eigensymmetry, src->quantum_rotational_eigensymmetry,
system->quantum_rotation_level_max*sizeof(int));
}
#endif /* QM_ROTATION */

dst->next = NULL;

/* new atoms list */
dst->atoms = calloc(1, sizeof(atom_t));
prev_atom_dst_ptr = dst->atoms;

for(atom_dst_ptr = dst->atoms, atom_src_ptr = src->atoms; atom_src_ptr; atom_dst_ptr = atom_dst_ptr->next, atom_src_ptr =
atom_src_ptr->next) {

    atom_dst_ptr->id = atom_src_ptr->id;
    strcpy(atom_dst_ptr->atomtype, atom_src_ptr->atomtype);
    atom_dst_ptr->frozen = atom_src_ptr->frozen;
    atom_dst_ptr->adiabatic = atom_src_ptr->adiabatic;
    atom_dst_ptr->mass = atom_src_ptr->mass;
    atom_dst_ptr->charge = atom_src_ptr->charge;
    atom_dst_ptr->polarizability = atom_src_ptr->polarizability;
    atom_dst_ptr->epsilon = atom_src_ptr->epsilon;
    atom_dst_ptr->sigma = atom_src_ptr->sigma;

    memcpy(atom_dst_ptr->pos, atom_src_ptr->pos, 3*sizeof(double));
    memcpy(atom_dst_ptr->wrapped_pos, atom_src_ptr->wrapped_pos, 3*sizeof(double));
    memcpy(atom_dst_ptr->ef_static, atom_src_ptr->ef_static, 3*sizeof(double));
    memcpy(atom_dst_ptr->ef_induced, atom_src_ptr->ef_induced, 3*sizeof(double));
    memcpy(atom_dst_ptr->mu, atom_src_ptr->mu, 3*sizeof(double));
    memcpy(atom_dst_ptr->old_mu, atom_src_ptr->old_mu, 3*sizeof(double));
    memcpy(atom_dst_ptr->new_mu, atom_src_ptr->new_mu, 3*sizeof(double));

    atom_dst_ptr->pairs = calloc(1, sizeof(pair_t));
    pair_dst_ptr = atom_dst_ptr->pairs;
    prev_pair_dst_ptr = pair_dst_ptr;
    for(pair_src_ptr = atom_src_ptr->pairs; pair_src_ptr; pair_src_ptr = pair_src_ptr->next) {

        pair_dst_ptr->rd_energy = pair_src_ptr->rd_energy;
        pair_dst_ptr->es_real_energy = pair_src_ptr->es_real_energy;
        pair_dst_ptr->es_self_intra_energy = pair_src_ptr->es_self_intra_energy;

        pair_dst_ptr->frozen = pair_src_ptr->frozen;
        pair_dst_ptr->rd_excluded = pair_src_ptr->rd_excluded;
        pair_dst_ptr->es_excluded = pair_src_ptr->es_excluded;
        pair_dst_ptr->charge = pair_src_ptr->charge;
        pair_dst_ptr->epsilon = pair_src_ptr->epsilon;
        pair_dst_ptr->lrc = pair_src_ptr->lrc;
        pair_dst_ptr->sigma = pair_src_ptr->sigma;
        pair_dst_ptr->r = pair_src_ptr->r;
        pair_dst_ptr->rimg = pair_src_ptr->rimg;

        pair_dst_ptr->next = calloc(1, sizeof(pair_t));
        prev_pair_dst_ptr = pair_dst_ptr;
        pair_dst_ptr = pair_dst_ptr->next;
    }
    prev_pair_dst_ptr->next = NULL;
    free(pair_dst_ptr);
    /* handle an empty list */
    if(!atom_src_ptr->pairs) atom_dst_ptr->pairs = NULL;
    prev_atom_dst_ptr = atom_dst_ptr;
    atom_dst_ptr->next = calloc(1, sizeof(atom_t));
}
prev_atom_dst_ptr->next = NULL;
free(atom_dst_ptr);

return(dst);
}

/* perform a general random translation */
void translate(molecule_t *molecule, pbc_t *pbc, double scale) {

    atom_t *atom_ptr;
    double trans_x, trans_y, trans_z;
    double boxlength;

    trans_x = scale*get_rand()*pbc->cutoff;
    trans_y = scale*get_rand()*pbc->cutoff;
    trans_z = scale*get_rand()*pbc->cutoff;
    if(get_rand() < 0.5) trans_x *= -1.0;
    if(get_rand() < 0.5) trans_y *= -1.0;
    if(get_rand() < 0.5) trans_z *= -1.0;

    molecule->com[0] += trans_x;
    molecule->com[1] += trans_y;
    molecule->com[2] += trans_z;
}

```

```

        for(atom_ptr = molecule->atoms; atom_ptr; atom_ptr = atom_ptr->next) {
            atom_ptr->pos[0] += trans_x;
            atom_ptr->pos[1] += trans_y;
            atom_ptr->pos[2] += trans_z;
        }
    }

/* perform a general random rotation */
void rotate(molecule_t *molecule, pbc_t *pbc, double scale) {
    atom_t *atom_ptr;
    double alpha, beta, gamma; /* Euler angles */
    double rotation_matrix[3][3];
    double com[3];
    int i, ii, n;
    double *new_coord_array;

    /* get the randomized Euler angles */
    alpha = scale*get_rand()*2.0*M_PI; if(get_rand() < 0.5) alpha *= -1.0;
    beta = scale*get_rand()*M_PI; if(get_rand() < 0.5) beta *= -1.0;
    gamma = scale*get_rand()*2.0*M_PI; if(get_rand() < 0.5) gamma *= -1.0;

    /* count the number of atoms in a molecule, and allocate new coords array */
    for(atom_ptr = molecule->atoms, n = 0; atom_ptr; atom_ptr = atom_ptr->next)
        ++n;
    new_coord_array = calloc(n*3, sizeof(double));

    /* save the com coordinate */
    com[0] = molecule->com[0];
    com[1] = molecule->com[1];
    com[2] = molecule->com[2];

    /* translate the molecule to the origin */
    for(atom_ptr = molecule->atoms; atom_ptr; atom_ptr = atom_ptr->next) {
        atom_ptr->pos[0] -= com[0];
        atom_ptr->pos[1] -= com[1];
        atom_ptr->pos[2] -= com[2];
    }

    /* construct the 3D rotation matrix */
    rotation_matrix[0][0] = cos(alpha)*cos(beta)*cos(gamma) - sin(alpha)*sin(gamma);
    rotation_matrix[0][1] = sin(alpha)*cos(beta)*cos(gamma) + cos(alpha)*sin(gamma);
    rotation_matrix[0][2] = -sin(beta)*cos(gamma);
    rotation_matrix[1][0] = -cos(alpha)*cos(beta)*sin(gamma) - sin(alpha)*cos(gamma);
    rotation_matrix[1][1] = -sin(alpha)*cos(beta)*sin(gamma) + cos(alpha)*cos(gamma);
    rotation_matrix[1][2] = sin(beta)*sin(gamma);
    rotation_matrix[2][0] = cos(alpha)*sin(beta);
    rotation_matrix[2][1] = sin(alpha)*sin(beta);
    rotation_matrix[2][2] = cos(beta);

    /* matrix multiply */
    for(atom_ptr = molecule->atoms, i = 0; atom_ptr; atom_ptr = atom_ptr->next, i++) {
        ii = i*3;
        new_coord_array[ii+0] = rotation_matrix[0][0]*atom_ptr->pos[0] + rotation_matrix[0][1]*atom_ptr->pos[1] +
        rotation_matrix[0][2]*atom_ptr->pos[2];
        new_coord_array[ii+1] = rotation_matrix[1][0]*atom_ptr->pos[0] + rotation_matrix[1][1]*atom_ptr->pos[1] +
        rotation_matrix[1][2]*atom_ptr->pos[2];
        new_coord_array[ii+2] = rotation_matrix[2][0]*atom_ptr->pos[0] + rotation_matrix[2][1]*atom_ptr->pos[1] +
        rotation_matrix[2][2]*atom_ptr->pos[2];
    }

    /* set the new coordinates and then translate back from the origin */
    for(atom_ptr = molecule->atoms, i = 0; atom_ptr; atom_ptr = atom_ptr->next, i++) {
        ii = i*3;
        atom_ptr->pos[0] = new_coord_array[ii+0];
        atom_ptr->pos[1] = new_coord_array[ii+1];
        atom_ptr->pos[2] = new_coord_array[ii+2];

        atom_ptr->pos[0] += com[0];
        atom_ptr->pos[1] += com[1];
        atom_ptr->pos[2] += com[2];
    }

    /* free our temporary array */
    free(new_coord_array);
}

/* perform a 1D translation without periodic boundaries */
void displace_1D(system_t *system, molecule_t *molecule, double scale) {

    atom_t *atom_ptr;
    double trans;

    trans = scale*get_rand();
    if(get_rand() < 0.5) trans *= -1.0;
    for(atom_ptr = molecule->atoms; atom_ptr; atom_ptr = atom_ptr->next)
        atom_ptr->pos[0] += trans;

    molecule->com[0] += trans;
}

/* perform a random translation/rotation of a molecule */
void displace(molecule_t *molecule, pbc_t *pbc, double trans_scale, double rot_scale) {
    translate(molecule, pbc, trans_scale);
    rotate(molecule, pbc, rot_scale);
}

```

```

/* apply what was already determined in checkpointing */
void make_move(system_t *system) {

    int i, j, k, p, q;
    cavity_t *cavities_array;
    int cavities_array_counter, random_index;
    double com[3], rand[3];
    molecule_t *molecule_ptr;
    atom_t *atom_ptr;
    pair_t *pair_ptr;

    /* update the cavity grid prior to making a move */
    if(system->cavity_bias) {
        cavity_update_grid(system);
        system->checkpoint->biased_move = 0;
    }

    if(system->checkpoint->movetype == MOVETYPE_INSERT) { /* insert a molecule at a random position and orientation */

        /* umbrella sampling */
        if(system->cavity_bias && system->cavities_open) {

            /* doing a biased move - this flag let's mc.c know about it */
            system->checkpoint->biased_move = 1;

            /* make an array of possible insertion points */
            cavities_array = calloc(system->cavities_open, sizeof(cavity_t));
            for(i = 0, cavities_array_counter = 0; i < system->cavity_grid_size; i++) {
                for(j = 0; j < system->cavity_grid_size; j++) {
                    for(k = 0; k < system->cavity_grid_size; k++) {

                        if(!system->cavity_grid[i][j][k].occupancy) {

                            for(p = 0; p < 3; p++)
                                cavities_array[cavities_array_counter].pos[p] =
                            system->cavity_grid[i][j][k].pos[p];

                            ++cavities_array_counter;
                        }
                    }
                }
            }
        } /* end i */
    } /* end j */
} /* end k */

/* insert randomly at one of the free cavity points */
random_index = (system->cavities_open - 1) - (int)rint(((double)(system->cavities_open - 1))*get_rand());
for(p = 0; p < 3; p++)
    com[p] = cavities_array[random_index].pos[p];

/* free the insertion array */
free(cavities_array);

} else {

    /* insert the molecule to a random location within the unit cell */
    for(p = 0; p < 3; p++)
        rand[p] = 0.5 - get_rand();

    for(p = 0; p < 3; p++)
        for(q = 0, com[p] = 0; q < 3; q++)
            com[p] += system->pbc->basis[p][q]*rand[q];
}

/* process the inserted molecule */
for(atom_ptr = system->checkpoint->molecule_backup->atoms; atom_ptr; atom_ptr = atom_ptr->next) {

    /* move the molecule back to the origin and the assign it to com */
    for(p = 0; p < 3; p++)
        atom_ptr->pos[p] += com[p] - system->checkpoint->molecule_backup->com[p];
}

/* update the molecular com */
for(p = 0; p < 3; p++)
    system->checkpoint->molecule_backup->com[p] = com[p];

/* give it a random orientation */
rotate(system->checkpoint->molecule_backup, system->pbc, 1.0);

/* insert into the list */
if(!system->checkpoint->head) { /* if we're at the start of the list */
    system->molecules = system->checkpoint->molecule_backup;
}
else {
    system->checkpoint->head->next = system->checkpoint->molecule_backup;
}
system->checkpoint->molecule_backup->next = system->checkpoint->molecule_altered;

/* set new altered and tail to reflect the insertion */
system->checkpoint->molecule_altered = system->checkpoint->molecule_backup;
system->checkpoint->tail = system->checkpoint->molecule_altered->next;
system->checkpoint->molecule_backup = NULL;
update_pairs_insert(system);

} else if(system->checkpoint->movetype == MOVETYPE_REMOVE) { /* remove a randomly chosen molecule */

    if(system->cavity_bias) {

        if(get_rand() < pow((1.0 - system->avg_observables->cavity_bias_probability),
        ((double)system->cavity_grid_size*system->cavity_grid_size*system->cavity_grid_size)))
            system->checkpoint->biased_move = 0;
        else
    }
}

```

```

        system->checkpoint->biased_move = 1;
    }

    /* remove 'altered' from the list */
    if(!system->checkpoint->head) { /* handle the case where we're removing from the start of the list */
        system->checkpoint->molecule_altered = system->molecules;
        system->molecules = system->molecules->next;
    } else {
        system->checkpoint->head->next = system->checkpoint->tail;
    }
    free_molecule(system, system->checkpoint->molecule_altered);
    update_pairs_remove(system);

} else if(system->checkpoint->movetype == MOVETYPE_DISPLACE) {

    /* change coords of 'altered' */
    if(system->rd_anharmonic)
        displace_1D(system, system->checkpoint->molecule_altered, system->move_probability);
    else
        displace(system->checkpoint->molecule_altered, system->pbc, system->move_probability, system->rot_probability);

} else if(system->checkpoint->movetype == MOVETYPE_ADIABATIC) {

    /* change coords of 'altered' */
    displace(system->checkpoint->molecule_altered, system->pbc, system->adiabatic_probability, 1.0);

}

/* this function (a) determines what move will be made next time make_move() is called and (b) backups up the state to be restore upon rejection */
void checkpoint(system_t *system) {

    int i_exchange, i_adiabatic;
    int num_molecules_exchange, num_molecules_adiabatic, altered;
    double num_molecules_exchange_double, num_molecules_adiabatic_double;
    molecule_t **ptr_array_exchange, **ptr_array_adiabatic;
    molecule_t *molecule_ptr, *prev_molecule_ptr;

    /* save the current observables */
    memcpy(system->checkpoint->observables, system->observables, sizeof(observables_t));

    /* count the number of exchangeable molecules */
    for(molecule_ptr = system->molecules, num_molecules_exchange = 0; molecule_ptr; molecule_ptr = molecule_ptr->next) {
        if(!(molecule_ptr->frozen || molecule_ptr->adiabatic)) ++num_molecules_exchange;
    }

    /* count the number of adiabatic molecules */
    for(molecule_ptr = system->molecules, num_molecules_adiabatic = 0; molecule_ptr; molecule_ptr = molecule_ptr->next) {
        if(molecule_ptr->adiabatic) ++num_molecules_adiabatic;
    }

    /* go thru again, make an array of all eligible molecules */
    ptr_array_exchange = calloc(num_molecules_exchange, sizeof(molecule_t *));
    ptr_array_adiabatic = calloc(num_molecules_adiabatic, sizeof(molecule_t *));
    for(molecule_ptr = system->molecules, i_exchange = 0, i_adiabatic = 0; molecule_ptr; molecule_ptr = molecule_ptr->next) {

        if(!(molecule_ptr->frozen || molecule_ptr->adiabatic)) {
            ptr_array_exchange[i_exchange] = molecule_ptr;
            ++i_exchange;
        }

        if(molecule_ptr->adiabatic) {
            ptr_array_adiabatic[i_adiabatic] = molecule_ptr;
            ++i_adiabatic;
        }
    }

    /* determine what kind of move to do */
    if(system->ensemble == ENSEMBLE_UVT) { /* UVT */

        if(get_rand() < system->insert_probability) { /* do a particle insertion/deletion move */

            if(get_rand() < 0.5) { /* INSERT */
                system->checkpoint->movetype = MOVETYPE_INSERT;
            } else { /* REMOVE */
                system->checkpoint->movetype = MOVETYPE_REMOVE;
            }

        } else { /* DISPLACE */
            if(num_molecules_adiabatic && (get_rand() < 0.5))
                system->checkpoint->movetype = MOVETYPE_ADIABATIC; /* for the adiabatic mole fraction */
            else
                system->checkpoint->movetype = MOVETYPE_DISPLACE;
        }

    } else if((system->ensemble == ENSEMBLE_NVT) || (system->ensemble == ENSEMBLE_NVE)){
        system->checkpoint->movetype = MOVETYPE_DISPLACE;
    }

    /* if we have any adiabatic molecules, then we have to do some special stuff */
    /* randomly pick a (moveable) atom */
    if(system->checkpoint->movetype == MOVETYPE_ADIABATIC) {

        --num_molecules_adiabatic;
        num_molecules_adiabatic_double = (double)num_molecules_adiabatic;
        altered = num_molecules_adiabatic - (int)rint(num_molecules_adiabatic_double*get_rand());
        system->checkpoint->molecule_altered = ptr_array_adiabatic[altered];

    } else {

        --num_molecules_exchange;
    }
}

```

```

num_molecules_exchange_double = (double)num_molecules_exchange;
altered = num_molecules_exchange - (int)rint(num_molecules_exchange_double*get_rand());
system->checkpoint->molecule_altered = ptr_array_exchange[altered];
}

/* free our temporary eligibility lists */
free(ptr_array_exchange);
if(ptr_array_adiabatic) free(ptr_array_adiabatic);

/* never completely empty the list */
if(!num_molecules_exchange && system->checkpoint->movetype == MOVETYPE_REMOVE)
    system->checkpoint->movetype = MOVETYPE_DISPLACE;

/* determine the head and tail of the selected molecule */
for(molecule_ptr = system->molecules, prev_molecule_ptr = NULL; molecule_ptr; molecule_ptr = molecule_ptr->next) {

    if(molecule_ptr == system->checkpoint->molecule_altered) {
        system->checkpoint->head = prev_molecule_ptr;
        system->checkpoint->tail = molecule_ptr->next;
    }
    prev_molecule_ptr = molecule_ptr;
}

/* if we have a molecule already backed up (from a previous accept), go ahead and free it */
if(system->checkpoint->molecule_backup) free_molecule(system, system->checkpoint->molecule_backup);
/* backup the state that will be altered */
system->checkpoint->molecule_backup = copy_molecule(system, system->checkpoint->molecule_altered);

}

/* this function (a) undoes what make_move() did and (b) determines the next move sequence by calling checkpoint() */
void restore(system_t *system) {

molecule_t *molecule_ptr;
atom_t *atom_ptr;
pair_t *prev_pair_ptr, *pair_ptr;

/* restore the remaining observables */
memcpy(system->observables, system->checkpoint->observables, sizeof(observables_t));

/* restore state by undoing the steps of make_move() */
if(system->checkpoint->movetype == MOVETYPE_INSERT) {

    /* take altered out of the list */
    if(!system->checkpoint->head) { /* handle the case where we inserted at the head of the list */
        system->molecules = system->molecules->next;
    } else {
        system->checkpoint->head->next = system->checkpoint->tail;
    }
    unupdate_pairs_insert(system);
    free_molecule(system, system->checkpoint->molecule_altered);
}

else if(system->checkpoint->movetype == MOVETYPE_REMOVE) {

    /* put backup back into the list */
    if(!system->checkpoint->head) {
        system->molecules = system->checkpoint->molecule_backup;
    } else {
        system->checkpoint->head->next = system->checkpoint->molecule_backup;
    }
    system->checkpoint->molecule_backup->next = system->checkpoint->tail;
    unupdate_pairs_remove(system);
    system->checkpoint->molecule_backup = NULL;
}

else if((system->checkpoint->movetype == MOVETYPE_DISPLACE) || (system->checkpoint->movetype == MOVETYPE_ADIABATIC)) {

    /* link the backup into the working list again */
    if(!system->checkpoint->head)
        system->molecules = system->checkpoint->molecule_backup;
    else
        system->checkpoint->head->next = system->checkpoint->molecule_backup;
    system->checkpoint->molecule_backup->next = system->checkpoint->tail;
    free_molecule(system, system->checkpoint->molecule_altered);
    system->checkpoint->molecule_backup = NULL;
}

/* establish the previous checkpoint again */
checkpoint(system);
}

/*
@2007, Jonathan Belof
Space Research Group
Department of Chemistry
University of South Florida
*/
#include <mc.h>

int pimc(system_t *system) {
    return(0);
}

#endif XXX

/*****************************************/

```

```

/*
 * Path integral Monte Carlo calculation for singular atomic systems
 */
/*
 * This code calculates the path integral for a 1s electron around an external
 * potential generated by Z protons at the cartesian origin. The calculated
 * observables are the total energy and the coordinates of the Trotter beads.
 */
/*
 * compilation:
 * gcc -o atomic_pimc atomic_pimc.c -lm
 */
/*
 * usage: ./pimc <Z> <trot> <temp> <move> <scale> <steps> <corr>
 */
/* <Z> = atomic number */
/* <trot> = Trotter number of beads */
/* <temp> = temperature of the system (K) */
/* <scale> = scale for the bead move */
/* <steps> = number of MC steps to perform */
/* <corr> = sampling interval */
/*
 * The pseudo-potential allows the use of ~100 Trotter beads per fermion, but
 * the sampling efficiency could be much better by anticipating the true
 * wavefunction nodes a la Ceperley (Recent Adv. in QM Methods II, 2002) and
 * focusing the sampling there.
 */
/* @2007 Jonathan Belof */
/* Space Research Group */
/* Department of Chemistry */
/* University of South Florida */
/***************************************************************/

/* set the nucleus */

#define COORDS_OUTPUT_FILENAME "output.coords" /* xmov formatted output of coordinates */
#define ENERGY_OUTPUT_FILENAME "output.energy" /* energy output for plotting */
#define XMOM_SET_FILENAME "output.set" /* for xmov viewing */
#define XMOM_PROTON_TOP_FILENAME "proton.top" /* for xmov viewing */
#define XMOM_ELECTRON_TOP_FILENAME "electron.top" /* for xmov viewing */

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <math.h>
#include <time.h>

/* conversions */
#define KPSA 1.202717 /* number of amu's in 1 KPSA mass unit */
#define H_OVER_K 47.9923862 /* Planck's constant divided by boltzmann's constant (K ps) */
#define H_OVER_K_SQUARED 2303.26913 /* above constant, squared */
#define HBAR_OVER_K 7.63822547 /* Planck's bar divided by boltzmann's constant (K ps) */
#define HBAR_OVER_K_SQUARED 58.3424884 /* the above constant, squared */
#define BOLTZMANNS_CONSTANT 1.38650e-23 /* Boltzmann's constant */
#define K_TO_EV 8.65385313e-5 /* Conversion factor for K -> eV */

/* physical constants */
#define BOHR_RADIUS 0.529177210818 /* Bohr radius in angstroms */
#define ELECTRON_CHARGE 408.781604283 /* in reduced units of sqrt(K*A) */
#define ELECTRON_MASS 6.597860309e-4 /* electron mass in KPSA (0.00054857961673 g/mol) */
#define PROTON_MASS 1.211466984 /* proton mass in KPSA (1.007275181 g/mol) */

/* pseudo-potential switching distance */
#define SWITCHING 0.5*BOHR_RADIUS

/* data structure for a quantum particle */
struct particle_t {
    double mass; /* particle mass */
    int trotter; /* number of beads in the ring polymer */
    struct bead_t *bead_list; /* circular linked list of beads */
};

/* data structure for the Trotter beads on the cyclic polymer */
struct bead_t {
    double x, y, z; /* coordinates */
    struct bead_t *next; /* ptr to the next bead */
};

/* global copy of the particle state for saving/restoring across functions */
struct particle_t *saved_particle_array;

/* initialize the pseudo RNG */
void seed_random_number(void) {
    srand48(time(NULL));
}

/* returns a double from 0.0 to 1.0 */
double get_random_number() {
    return(drand48());
}

/* takes system parameters as input, returns an array of initialized particles */
struct particle_t *initialize_system(int trotter) {
    int i;
    struct particle_t *particle_array = NULL; /* particle array pointer */
    struct bead_t *current_bead_ptr; /* current working bead pointer */
    double random_theta, random_phi; /* random angles */

    /* allocate the particle array */
    particle_array = malloc(sizeof(struct particle_t), 1); /* a single particle */
    if(!particle_array) {
        fprintf(stderr, "initialize_system: couldn't allocate particle array\n");
    }

    /* initialize the particle array */
    for(i=0; i<trotter; i++) {
        /* calculate random angles */
        random_theta = 2.0 * M_PI * (double) i / (double) trotter;
        random_phi = M_PI * (double) i / (double) trotter;
        /* calculate position */
        /* ... */
    }
}

```

```

        return(NULL);
    }
particle_array[0].mass = ELECTRON_MASS; /* set the particle mass */
particle_array[0].trotter = trotter; /* set the trotter number for the ring */

/* allocate the beads */
particle_array[0].bead_list = calloc(sizeof(struct bead_t), 1); /* start with the first one */
current_bead_ptr = particle_array[0].bead_list;
if(!current_bead_ptr) {
    fprintf(stderr, "initialize_system: couldn't allocate first bead element\n");
    return(NULL);
}
current_bead_ptr->x = BOHR_RADIUS;
current_bead_ptr->y = 0.0;
current_bead_ptr->z = 0.0;
for(i = 0; i < (trotter - 1); i++) { /* allocate the rest of the beads */
    current_bead_ptr->next = calloc(sizeof(struct bead_t), 1);
    current_bead_ptr = current_bead_ptr->next;
if(!current_bead_ptr) {
    fprintf(stderr, "initialize_system: couldn't allocate bead number %d\n", (i + 1));
    return(NULL);
}
current_bead_ptr->x = BOHR_RADIUS;
current_bead_ptr->y = 0.0;
current_bead_ptr->z = 0.0;
}
current_bead_ptr->next = particle_array[0].bead_list; /* connect the circle */
/* done setting up the particles */

/* setup the space for saving/restoring the system state */
saved_particle_array = calloc(sizeof(struct particle_t), 1);
if(!saved_particle_array) {
    fprintf(stderr, "initialize_system: couldn't allocate saved particle array\n");
    return(NULL);
}
saved_particle_array[0].bead_list = calloc(sizeof(struct bead_t), 1);
current_bead_ptr = saved_particle_array[0].bead_list;
if(!current_bead_ptr) {
    fprintf(stderr, "initialize_system: couldn't allocate first saved bead element\n");
    return(NULL);
}
for(i = 0; i < (trotter - 1); i++) {
    current_bead_ptr->next = calloc(sizeof(struct bead_t), 1);
    current_bead_ptr = current_bead_ptr->next;
if(!current_bead_ptr) {
    fprintf(stderr, "initialize_system: couldn't allocate bead number %d\n", (i + 1));
    return(NULL);
}
current_bead_ptr->next = saved_particle_array[0].bead_list;

/* seed the RNG */
seed_random_number();

return(particle_array);
}

/* calculate the coulombic energy due to the proton at the origin, include a pseudo-potential */
double get_external_potential(struct particle_t *particle_array, double Z, double switching) {

int i;
double r, external_energy;
struct bead_t *cur_ptr;

cur_ptr = particle_array[0].bead_list;
for(i = 0, external_energy = 0.0; i < particle_array[0].trotter; i++) {
    r = sqrt(cur_ptr->x*cur_ptr->x + cur_ptr->y*cur_ptr->y + cur_ptr->z*cur_ptr->z);
    if(r < switching) /* impose the pseudo-potential */
        external_energy += -(Z*ELECTRON_CHARGE*ELECTRON_CHARGE*((r*r/(2.0*switching*switching)) - 3.0/2.0))/switching;
    else
        external_energy += -Z*ELECTRON_CHARGE*ELECTRON_CHARGE/r;
    cur_ptr = cur_ptr->next;
}
external_energy /= ((double)particle_array[0].trotter);

return(external_energy);
}

/* calculate the path integral harmonic energy for the ring polymer */
double get_internal_potential(struct particle_t *particle_array, double temperature) {

int i;
struct bead_t *cur_ptr, *nxt_ptr;
double internal_energy;
double dx, dy, dz, displacement_squared;

/* get the ring polymer energy */
cur_ptr = particle_array[0].bead_list;
nxt_ptr = cur_ptr->next;
for(i = 0, internal_energy = 0.0; i < particle_array[0].trotter; i++) {
    dx = cur_ptr->x - nxt_ptr->x;
    dy = cur_ptr->y - nxt_ptr->y;
    dz = cur_ptr->z - nxt_ptr->z;
    displacement_squared = dx*dx + dy*dy + dz*dz;
    internal_energy += displacement_squared;
    cur_ptr = cur_ptr->next;
    nxt_ptr = cur_ptr->next;
}
internal_energy *= (((double)particle_array[0].trotter)*particle_array[0].mass*temperature*temperature)/(2.0*HBAR_OVER_K_SQUARED);

return(internal_energy);
}

/* get the total energy for the system */

```

```

double get_energy(struct particle_t *particle_array, double Z, double temperature) {
    double energy;
    energy = get_external_potential(particle_array, Z, SWITCHING);
    energy += get_internal_potential(particle_array, temperature);
    return(energy);
}

/* make a random trial move */
void make_move(struct particle_t *particle_array, double scale) {
    int i;
    struct bead_t *cur_ptr;

    /* move each bead by a random amount */
    cur_ptr = particle_array[0].bead_list;
    for(i = 0; i < particle_array[0].trotter; i++) {
        cur_ptr->x += scale*(0.5 - get_random_number());
        cur_ptr->y += scale*(0.5 - get_random_number());
        cur_ptr->z += scale*(0.5 - get_random_number());

        cur_ptr = cur_ptr->next;
    }
}

/* MC accept routine, stores the accepted coordinates */
void mc_accept(struct particle_t *particle_array) {
    int i;
    struct bead_t *saved_ptr, *current_ptr;

    /* save the accepted configuration */
    saved_particle_array[0].trotter = particle_array[0].trotter;
    saved_particle_array[0].mass = particle_array[0].mass;

    current_ptr = particle_array[0].bead_list;
    saved_ptr = saved_particle_array[0].bead_list;
    for(i = 0; i < particle_array[0].trotter; i++) {
        saved_ptr->x = current_ptr->x;
        saved_ptr->y = current_ptr->y;
        saved_ptr->z = current_ptr->z;
        current_ptr = current_ptr->next;
        saved_ptr = saved_ptr->next;
    }
}

/* MC reject routine, restore the last accepted coordinates */
void mc_reject(struct particle_t *particle_array) {
    int i;
    struct bead_t *saved_ptr, *current_ptr;

    /* restore the last accepted configuration */
    particle_array[0].trotter = saved_particle_array[0].trotter;
    particle_array[0].mass = saved_particle_array[0].mass;

    current_ptr = particle_array[0].bead_list;
    saved_ptr = saved_particle_array[0].bead_list;
    for(i = 0; i < saved_particle_array[0].trotter; i++) {
        current_ptr->x = saved_ptr->x;
        current_ptr->y = saved_ptr->y;
        current_ptr->z = saved_ptr->z;
        current_ptr = current_ptr->next;
        saved_ptr = saved_ptr->next;
    }
}

/* write the system coordinates to a file */
void write_coords(struct particle_t *particle_array, FILE *fp_coords) {
    int i;
    struct bead_t *cur_ptr;

    /* print the proton coords */
    fprintf(fp_coords, "0.0 0.0 0.0\n");

    /* print the electron's beads */
    cur_ptr = particle_array[0].bead_list;
    for(i = 0; i < particle_array[0].trotter; i++) {
        fprintf(fp_coords, "%lg %lg %lg\n", cur_ptr->x, cur_ptr->y, cur_ptr->z); /* output coords */
        cur_ptr = cur_ptr->next;
    }
    /* print artificial periodic boundary */
    fprintf(fp_coords, "%lg 0.0 0.0 0.0 %lg 0.0 0.0 0.0 %lg\n", 4*BOHR_RADIUS, 4*BOHR_RADIUS, 4*BOHR_RADIUS);
}

/* write the system energy to a file */
void write_energy(double energy, FILE *fp_energy) {
    fprintf(fp_energy, "%lg\n", energy);
}

/* perform the path integral monte carlo calculation - the meat 'n potatos */
int do_pimc(struct particle_t *particle_array, double Z, double temperature, double scale, int num_steps, int corr_time) {
    int i, j; /* MC step counter */
    double initial_energy; /* energy before making MC move */
}

```

```

double final_energy;           /* energy after making MC move */
double delta_energy;          /* energy change due to the MC move */
double boltzmann;             /* boltzmann factor */
double accepted_energy;        /* accepted energy to be averaged */
double accepted_energy_squared; /* accepted energy squared */
double radius;                /* bead radius from the proton */
double accepted_radius;       /* accepted value for averaging */
double average_radius;        /* simulation average radius */
double average_energy = 0;     /* average accepted energy */
double average_energy_squared = 0; /* average square of the accepted energy */
double average_boltzmann = 0;   /* average boltzmann factor */
double average_factor = 0;      /* numerical factor used for the running averages */
int accepts = 0, rejects = 0;  /* keep track of acceptance rate */
FILE *fp_coords, *fp_energy;   /* file pointers for output */
FILE *fp_set, *fp_proton_top, *fp_electron_top; /* used for xmov movie output */
struct bead_t *cur_ptr;        /* used for radius calculation */

if(!particle_array || (num_steps <= 0)) {
    fprintf(stderr, "do_pimc: invalid input parameters passed\n");
    return(-1);
}

/* set the temperature */
if(temperature < 0.0) {
    fprintf(stderr, "do_pimc: invalid temperature specified\n");
    return(-1);
}

/* open the output files for writing */
fp_coords = fopen(COORDS_OUTPUT_FILENAME, "w");
fp_energy = fopen(ENERGY_OUTPUT_FILENAME, "w");
if(!fp_coords || !fp_energy) {
    fprintf(stderr, "do_pimc: couldn't open output files for writing\n");
    return(-1);
}

/* write xmov header to coords file */
fprintf(fp_coords, "# %d 1 0.001\n", (particle_array[0].trotter + 1));
/* output the xmov set and top files */
fp_set = fopen(XMOV_SET_FILENAME, "w");
fprintf(fp_set, "'mol_def\\\"%mol\\_name{proton.top}\\\"\\mol_therm_opt{none}\\\"\\nmol{1}\\\"\\mol_index{1}\\]\\n\"");
fprintf(fp_set, "'mol_def\\\"%mol\\_parm_file{electron.top}\\\"\\mol_therm_opt{none}\\\"\\nmol{2}\\\"\\mol_index{2}\\]\\n",
particle_array[0].trotter);
fp_proton_top = fopen(XMOV_PROTON_TOP_FILENAME, "w");
fprintf(fp_proton_top, "'mol_name{proton}\\\"\\natom{1}\\\"\\nbond{0}\\\"\\nbondx{0}\\]\\n\"");
fprintf(fp_proton_top, "'atom_def\\\"%atom\\_typ{P}\\\"\\atom\\_ind{1}\\\"\\mass{1.0}\\\"\\charge{0.0}\\\"\\alpha{0.000}\\]\\n\"");
fp_electron_top = fopen(XMOV_ELECTRON_TOP_FILENAME, "w");
fprintf(fp_electron_top, "'mol_name{electron}\\\"\\natom{1}\\\"\\nbond{0}\\\"\\nbondx{0}\\]\\n\"");
fprintf(fp_electron_top, "'atom_def\\\"%atom\\_typ{E}\\\"\\atom\\_ind{1}\\\"\\mass{0.0001}\\\"\\charge{0.0}\\\"\\alpha{0.000}\\]\\n\"");
fclose(fp_set); fclose(fp_proton_top); fclose(fp_electron_top);

/* get the first energy and automatically accept it */
initial_energy = get_energy(particle_array, Z, temperature);
mc_accept(particle_array);
accepted_energy = get_external_potential(particle_array, Z, SWITCHING);
accepted_energy_squared = accepted_energy*accepted_energy;

/* get the initial radius */
cur_ptr = particle_array[0].bead_list;
for(j = 0, accepted_radius = 0.0; j < particle_array[0].trotter; j++) {
    radius = sqrt(cur_ptr->x*cur_ptr->x + cur_ptr->y*cur_ptr->y + cur_ptr->z*cur_ptr->z);
    accepted_radius += radius;
    cur_ptr = cur_ptr->next;
}
accepted_radius /= ((double)particle_array[0].trotter);

/* this is the main loop */
for(i = 0; i < num_steps; i++) {

    /* make a trial move */
    make_move(particle_array, scale);

    /* get the post-move energy, energy change and boltzmann factor */
    final_energy = get_energy(particle_array, Z, temperature);
    delta_energy = final_energy - initial_energy;
    boltzmann = exp(-delta_energy/temperature);

    /* perform the metropolis evaluation */
    if(get_random_number() < boltzmann) { /****** ACCEPT *****/

        mc_accept(particle_array);
        ++accepts;
        initial_energy = final_energy; /* save for the next MC step */
        accepted_energy = get_external_potential(particle_array, Z, SWITCHING);
        accepted_energy_squared = accepted_energy*accepted_energy;

        /* get the average bead radius */
        cur_ptr = particle_array[0].bead_list;
        for(j = 0, accepted_radius = 0.0; j < particle_array[0].trotter; j++) {
            radius = sqrt(cur_ptr->x*cur_ptr->x + cur_ptr->y*cur_ptr->y + cur_ptr->z*cur_ptr->z);
            accepted_radius += radius;
            cur_ptr = cur_ptr->next;
        }
        accepted_radius /= ((double)particle_array[0].trotter);

    } else { /****** REJECT *****/
        mc_reject(particle_array);
        ++rejects;
    }

    if(!(i % corr_time)) {
        /* write out the coordinates */
        write_coords(particle_array, fp_coords);
        /* write out the total energy */
        write_energy(accepted_energy*K_TO_EV, fp_energy);
    }
}

```

```

    }

    /* keep a running average of the energy */
    average_factor = ((double)i)/((double)(i + 1));
    average_energy = average_energy*average_factor + (accepted_energy / ((double)(i + 1)));
    average_energy_squared = average_energy_squared*average_factor + (accepted_energy_squared / ((double)(i + 1)));
    average_boltzmann = average_boltzmann*average_factor + (boltzmann / ((double)(i + 1)));
    average_radius = average_radius*average_factor + (accepted_radius / ((double)(i + 1)));

}

/* output useful stats to stdout */
printf("\n\n*****\n", temperature);
printf("***** temperature = %.2f (K)\n", temperature);
printf("***** acceptance rate = %.3f\n", ((double)accepts)/((double)i));
printf("***** average boltzmann factor = %.5f\n", average_boltzmann);
printf("***** average energy = %.3f (eV)\n", average_energy*K_TO_EV);
printf("***** average electron radius = %.3f (A)\n", average_radius);
printf("***** energy sigma = %.3f (eV)\n", K_TO_EV*sqrt(average_energy_squared - average_energy*average_energy));
printf("*****\n\n");

/* gracefully close our file pointer streams */
fclose(fp_coords);
fclose(fp_energy);

return(0);
}

/* cleans up the particle array */
void free_particle(struct particle_t *particle_array) {

int i;
struct bead_t *current_bead_ptr, **free_array;

free_array = calloc(sizeof(struct bead_t *), particle_array[0].trotter);
current_bead_ptr = particle_array[0].bead_list;

for(i = 0; i < particle_array[0].trotter; i++) {
    free_array[i] = current_bead_ptr;
    current_bead_ptr = current_bead_ptr->next;
}

for(i = 0; i < particle_array[0].trotter; i++)
    free(free_array[i]);

free(particle_array);
free(free_array);
}

/* provide usage information for the user */
void usage(char *progname) {

fprintf(stderr, "usage: %s <Z> <trot> <temp> <scale> <steps> <corr>\n", progname);
fprintf(stderr, "\t<Z>\t\t= atomic number\n");
fprintf(stderr, "\t<trot>\t\t= Trotter number of beads\n");
fprintf(stderr, "\t<temp>\t\t= temperature of the system (K)\n");
fprintf(stderr, "\t<scale>\t\t= bead move scaling parameter\n");
fprintf(stderr, "\t<steps>\t\t= number of MC steps to perform\n");
fprintf(stderr, "\t<corr>\t\t= interval of configuration sampling\n\n");
exit(1);
}

int main(int argc, char **argv) {

int i; /* counter */
int num_steps, corr_time; /* number of MC steps to do and correlation time */
int trotter; /* Trotter number of beads */
double temperature; /* temperature of the system (K) */
double scale; /* amount to scale the bead moves by */
struct particle_t *particle_array; /* particle array, currently just a single particle */
double Z; /* atomic number */

/* read cmd line args */
if(argc != 7)
    usage(argv[0]);

printf("%s: input invocation = ", argv[0]);
for(i = 0; i < argc; i++)
    printf("%s ", argv[i]);
putchar('\n');

/* get the atomic number */
Z = atof(argv[1]);
if(Z < 0) {
    fprintf(stderr, "%s: ERROR: invalid Z specified\n", argv[0]);
    usage(argv[0]);
}

/* get the Trotter number */
trotter = atoi(argv[2]);
if(trotter < 0) {
    fprintf(stderr, "%s: ERROR: invalid Trotter number specified\n", argv[0]);
    usage(argv[0]);
}

/* get the temperature */
temperature = atof(argv[3]);
if(temperature < 0.0) {
    fprintf(stderr, "%s: ERROR: invalid temperature specified\n", argv[0]);
    usage(argv[0]);
}

/* get the amount to scale the beads by */
scale = atof(argv[4]);
if(scale <= 0.0) {
}
}

```

```

    fprintf(stderr, "%s: ERROR: invalid bead scaling specified\n", argv[0]);
    exit(1);
}

/* get the number of MC steps to perform */
num_steps = atoi(argv[5]);
if(num_steps <= 0) {
    fprintf(stderr, "%s: ERROR: invalid number of MC steps specified\n", argv[0]);
    exit(1);
}

/* get the correlation interval for sampling */
corr_time = atoi(argv[6]);
if(corr_time <= 0) {
    fprintf(stderr, "%s: ERROR: invalid correlation time specified\n", argv[0]);
    exit(1);
}

particle_array = initialize_system(trotter);
if(!particle_array) {
    fprintf(stderr, "%s: ERROR: couldn't initialize the system\n", argv[0]);
    exit(1);
}

/* start the path integral monte carlo */
if(do_pimc(particle_array, Z, temperature, scale, num_steps, corr_time) < 0) {
    fprintf(stderr, "%s: ERROR: path integral MC loop died\n", argv[0]);
    exit(1);
}

free_particle(particle_array);
exit(0);
}

#endif /* XXX */

/*
@2007, Jonathan Belof
Space Research Group
Department of Chemistry
University of South Florida
*/
/* this file is rather sloppy, but got the job done */

#include <mc.h>

#define SCALE_CHARGE 0.0
#define SCALE_EPSILON 0.1
#define SCALE_SIGMA 0.01
#define SCALE_R 0.001

#define TEMPERATURE 500.0
#define MIN_TEMPERATURE 4.0
#define SCHEDULE 0.9
#define EQUIL 1000

#define ENERGY_TOTAL 0
#define ENERGY_ES 1
#define ENERGY_RD 2
#define ENERGY_POLAR 3

/* calculate the energy for the surface scan - same as function energy() but without observables */
double surface_energy(system_t *system, int energy_type) {

    molecule_t *molecule_ptr;
    double potential_energy, rd_energy, coulombic_energy, polar_energy;

    /* zero the initial values */
    potential_energy = 0;
    rd_energy = 0;
    coulombic_energy = 0;
    polar_energy = 0;

    /* get the pairwise terms necessary for the energy calculation */
    pairs(system);

    switch(energy_type) {
        case ENERGY_TOTAL:
            if(!(system->sg || system->rd_only)) coulombic_energy = coulombic_nopbc(system->molecules);
            if(system->sg) {
                rd_energy = sg_nopbc(system->molecules);
            } else {
                rd_energy = lj_nopbc(system->molecules);
            }
            if(system->polarization) polar_energy = polar(system);
            break;
        case ENERGY_ES:
            if(!(system->sg || system->rd_only)) coulombic_energy = coulombic_nopbc(system->molecules);
            break;
        case ENERGY_RD:
            if(system->sg) {
                rd_energy = sg_nopbc(system->molecules);
            } else {
                rd_energy = lj_nopbc(system->molecules);
            }
            break;
        case ENERGY_POLAR:
            if(system->polarization) polar_energy = polar(system);
            break;
    }
}

```

```

/* sum the total potential energy */
potential_energy = rd_energy + coulombic_energy + polar_energy;
return(potential_energy);
}

/* rotate a molecule about three Euler angles - same as normal function but for a single mol and without random angles */
void molecule_rotate(molecule_t *molecule, double alpha, double beta, double gamma) {

atom_t *atom_ptr;
double rotation_matrix[3][3];
double com[3];
int i, ii, n;
double *new_coord_array;

/* count the number of atoms in a molecule, and allocate new coords array */
for(atom_ptr = molecule->atoms, n = 0; atom_ptr; atom_ptr = atom_ptr->next)
    ++n;
new_coord_array = calloc(n*3, sizeof(double));

/* save the com coordinate */
com[0] = molecule->com[0];
com[1] = molecule->com[1];
com[2] = molecule->com[2];

/* translate the molecule to the origin */
for(atom_ptr = molecule->atoms; atom_ptr; atom_ptr = atom_ptr->next) {
    atom_ptr->pos[0] -= com[0];
    atom_ptr->pos[1] -= com[1];
    atom_ptr->pos[2] -= com[2];
}

/* construct the 3D rotation matrix */
rotation_matrix[0][0] = cos(gamma)*cos(beta)*cos(alpha) - sin(gamma)*sin(alpha);
rotation_matrix[0][1] = sin(gamma)*cos(beta)*cos(alpha) + cos(gamma)*sin(alpha);
rotation_matrix[0][2] = -sin(beta)*cos(alpha);
rotation_matrix[1][0] = -cos(gamma)*cos(beta)*sin(alpha) - sin(gamma)*cos(alpha);
rotation_matrix[1][1] = -sin(gamma)*cos(beta)*sin(alpha) + cos(gamma)*cos(alpha);
rotation_matrix[1][2] = sin(beta)*sin(alpha);
rotation_matrix[2][0] = cos(gamma)*sin(beta);
rotation_matrix[2][1] = sin(gamma)*sin(beta);
rotation_matrix[2][2] = cos(beta);

/* matrix multiply */
for(atom_ptr = molecule->atoms, i = 0; atom_ptr; atom_ptr = atom_ptr->next, i++) {
    ii = i*3;
    new_coord_array[ii+0] = rotation_matrix[0][0]*atom_ptr->pos[0] + rotation_matrix[0][1]*atom_ptr->pos[1] +
    rotation_matrix[0][2]*atom_ptr->pos[2];
    new_coord_array[ii+1] = rotation_matrix[1][0]*atom_ptr->pos[0] + rotation_matrix[1][1]*atom_ptr->pos[1] +
    rotation_matrix[1][2]*atom_ptr->pos[2];
    new_coord_array[ii+2] = rotation_matrix[2][0]*atom_ptr->pos[0] + rotation_matrix[2][1]*atom_ptr->pos[1] +
    rotation_matrix[2][2]*atom_ptr->pos[2];
}

/* set the new coordinates and then translate back from the origin */
for(atom_ptr = molecule->atoms, i = 0; atom_ptr; atom_ptr = atom_ptr->next, i++) {
    ii = i*3;
    atom_ptr->pos[0] = new_coord_array[ii+0];
    atom_ptr->pos[1] = new_coord_array[ii+1];
    atom_ptr->pos[2] = new_coord_array[ii+2];

    atom_ptr->pos[0] += com[0];
    atom_ptr->pos[1] += com[1];
    atom_ptr->pos[2] += com[2];
}

/* free our temporary array */
free(new_coord_array);
}

/* calculate the isotropic potential energy surface of a molecule */
int surface(system_t *system) {

int i;
int avg_counter;
double avg_factor;
double r, pe_es, pe_rd, pe_polar, pe_total;
double pe_total_avg, pe_es_avg, pe_rd_avg, pe_polar_avg;
double alpha_origin, beta_origin, gamma_origin;
double alpha_move, beta_move, gamma_move;

/* output the potential energy curve */
if(system->surf_preserve) { /* preserve the orientation and only calculate based on displacement */
    for(r = system->surf_min; r <= system->surf_max; r += system->surf_inc) {
        /* calculate the energy */
        surface_dimer_geometry(system, r, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0);

        if(system->surf_decomp) {
            pe_es = surface_energy(system, ENERGY_ES);
            pe_rd = surface_energy(system, ENERGY_RD);
            pe_polar = surface_energy(system, ENERGY_POLAR);
            printf("%.5f %.5f %.5f\n", r, pe_es, pe_rd, pe_polar);
            fflush(stdout);
        } else {
            pe_total = surface_energy(system, ENERGY_TOTAL);
            printf("%.5f %.5f\n", r, pe_total);
            fflush(stdout);
        }
    }
}
}

```

```

}
}

} else { /* default is to do isotropic averaging */

for(r = system->surf_min; r <= system->surf_max; r+= system->surf_inc) {

    /* zero the averages for this r value */
    avg_counter = 0;
    pe_total_avg = 0;
    pe_es_avg = 0;
    pe_rd_avg = 0;
    pe_polar_avg = 0;

    /* average over the angles */
    for(alpha_origin = 0; alpha_origin <= 2.0*M_PI; alpha_origin += system->surf_ang) {
        for(beta_origin = 0; beta_origin <= M_PI; beta_origin += system->surf_ang) {
            for(gamma_origin = 0; gamma_origin <= 2.0*M_PI; gamma_origin += system->surf_ang) {
                for(alpha_move = 0; alpha_move <= 2.0*M_PI; alpha_move += system->surf_ang) {
                    for(beta_move = 0; beta_move <= M_PI; beta_move += system->surf_ang) {
                        for(gamma_move = 0; gamma_move <= 2.0*M_PI; gamma_move += system->surf_ang)

{
                            ++avg_counter;
                            avg_factor = ((double)(avg_counter - 1))/((double)(avg_counter));

                            surface_dimer_geometry(system, r, alpha_origin, beta_origin,
gamma_origin, alpha_move, beta_move, gamma_move);

                            if(system->srf_decomp) {

                                pe_es = surface_energy(system, ENERGY_ES);
                                pe_rd = surface_energy(system, ENERGY_RD);
                                pe_polar = surface_energy(system, ENERGY_POLAR);

                                pe_es_avg = pe_es_avg*avg_factor +
(pe_es/((double)avg_counter));
                                pe_rd_avg = pe_rd_avg*avg_factor +
(pe_rd/((double)avg_counter));
                                pe_polar_avg = pe_polar_avg*avg_factor +
(pe_polar/((double)avg_counter));

                            } else {

                                pe_total = surface_energy(system, ENERGY_TOTAL);
                                pe_total_avg = pe_total_avg*avg_factor +
(pe_total/((double)avg_counter));
                            }
                        }

                        } /* end gamma_move */
                    } /* end beta_move */
                } /* end alpha_move */
            } /* end beta_origin */
        } /* end alpha_origin */
    } /* end alpha_origin */

    if(system->srf_decomp) {

        printf("%.5f %.5f %.5f %.5f\n", r, pe_es_avg, pe_rd_avg, pe_polar_avg);
        fflush(stdout);
    } else {

        printf("%.5f %.5f\n", r, pe_total_avg);
        fflush(stdout);
    }
}

} /* end r */

}

return(0);
}

/* set the distance and angles for a particular dimer orientation */
int surface_dimer_geometry(system_t *system, double r, double alpha_origin, double beta_origin, double gamma_origin, double alpha_move,
double beta_move, double gamma_move) {

int i;
molecule_t *molecule_origin, *molecule_move;
atom_t *atom_ptr;

/* make sure that there are only two molecules */
molecule_origin = system->molecules;
molecule_move = molecule_origin->next;
if(!molecule_move) {
    error("SURFACE: the input PDB has only a single molecule\n");
    return(-1);
}
if(molecule_move->next) {
    error("SURFACE: the input PDB must contain exactly two molecules\n");
    return(-1);
}

/* relocate both molecules to the origin */
for(atom_ptr = molecule_origin->atoms; atom_ptr; atom_ptr = atom_ptr->next)
    for(i = 0; i < 3; i++)
        atom_ptr->pos[i] -= molecule_origin->com[i];
    for(i = 0; i < 3; i++)
        molecule_origin->com[i] = 0;
}

```

```

for(atom_ptr = molecule_move->atoms; atom_ptr; atom_ptr = atom_ptr->next)
    for(i = 0; i < 3; i++)
        atom_ptr->pos[i] -= molecule_move->com[i];
    for(i = 0; i < 3; i++)
        molecule_move->com[i] = 0;

/* relocate the moveable molecule r distance away along the x axis */
molecule_move->com[0] = r;
molecule_move->com[1] = 0;
molecule_move->com[2] = 0;
for(atom_ptr = molecule_move->atoms; atom_ptr; atom_ptr = atom_ptr->next)
    for(i = 0; i < 3; i++)
        atom_ptr->pos[i] += molecule_move->com[i];

/* apply the rotation to the second molecule */
molecule_rotate(molecule_origin, alpha_origin, beta_origin, gamma_origin);
molecule_rotate(molecule_move, alpha_move, beta_move, gamma_move);

return(0);
}

int surface_dimer_parameters(system_t *system, param_t *params) {
molecule_t *molecule_ptr;
atom_t *atom_ptr;
param_t *param_ptr;

for(molecule_ptr = system->molecules; molecule_ptr; molecule_ptr = molecule_ptr->next) {
    for(atom_ptr = molecule_ptr->atoms; atom_ptr; atom_ptr = atom_ptr->next) {
        for(param_ptr = params; param_ptr; param_ptr = param_ptr->next) {
            if(!strcasecmp(atom_ptr->atomtype, param_ptr->atomtype)) {
                atom_ptr->charge = param_ptr->charge;
                atom_ptr->epsilon = param_ptr->epsilon;
                atom_ptr->sigma = param_ptr->sigma;
                if(atom_ptr->pos[param_ptr->axis] > 0.0)
                    atom_ptr->pos[param_ptr->axis] += param_ptr->dr;
                else
                    atom_ptr->pos[param_ptr->axis] -= param_ptr->dr;
            }
        }
    }
}
return(0);
}

void surface_curve(system_t *system, double r_min, double r_max, double r_inc, double *curve) {
int i;
double r;

for(r = r_min, i = 0; r <= r_max; r += r_inc, i++) {
    surface_dimer_geometry(system, r, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0);
    curve[i] = surface_energy(system, ENERGY_TOTAL);
}
}

/* fit potential energy parameters via simulated annealing */
/* XXX - require that the random seeds be set, error on memory alloc */
int surface_fit(system_t *system) {
int i, N, nsteps;
FILE *fp_fit;
char linebuf[MAXLINE];
double *r_input, *EOE_input, *PAR_input, *T_input, *X_input;
double *EOE_output, *PAR_output, *T_output, *X_output;
double *EOE_global, *PAR_global, *T_global, *X_global;
atom_t *atom_ptr;
param_t *param_ptr, *params, *prev_param_ptr;
double r, r_min, r_max, r_inc;
double current_error, last_error, global_minimum;
double temperature = TEMPERATURE;
char last_atomtype[MAXLINE];
double ghost_charge;

/* read in the number of lines */
fp_fit = fopen(system->fit_input, "r");
for(N = 0; fgets(linebuf, MAXLINE, fp_fit); N++);
fclose(fp_fit);

/* allocate our functions */
r_input = calloc(N, sizeof(double));
EOE_input = calloc(N, sizeof(double));
PAR_input = calloc(N, sizeof(double));
T_input = calloc(N, sizeof(double));
X_input = calloc(N, sizeof(double));
EOE_output = calloc(N, sizeof(double));
PAR_output = calloc(N, sizeof(double));
T_output = calloc(N, sizeof(double));
X_output = calloc(N, sizeof(double));
EOE_global = calloc(N, sizeof(double));
PAR_global = calloc(N, sizeof(double));
T_global = calloc(N, sizeof(double));
X_global = calloc(N, sizeof(double));
}

```

```

/* read in the 4 curves */
fp_fit = fopen(system->fit_input, "r");
for(i = 0; i < N; i++)
    fscanf(fp_fit, "%lg %lg %lg %lg\n", &r_input[i], &EOE_input[i], &PAR_input[i], &T_input[i], &X_input[i]);
fclose(fp_fit);

/* determine independent domain */
r_min = r_input[0];
r_max = r_input[N-1];
r_inc = r_input[1] - r_input[0];

/* get the initial error */
/* EOE */
surface_dimer_geometry(system, r_min, 0.0, 0.0, 0.0, 0.0, 0.0);
surface_curve(system, r_min, r_max, r_inc, EOE_output);

/* PAR */
surface_dimer_geometry(system, r_min, 0.5*M_PI, 0.0, 0.0, 0.5*M_PI, 0.0, 0.0);
surface_curve(system, r_min, r_max, r_inc, PAR_output);

/* T */
surface_dimer_geometry(system, r_min, 0.0, 0.0, 0.0, 0.5*M_PI, 0.0, 0.0);
surface_curve(system, r_min, r_max, r_inc, T_output);

/* X */
surface_dimer_geometry(system, r_min, 0.0, 0.0, 0.0, 0.0, 0.5*M_PI, 0.0);
surface_curve(system, r_min, r_max, r_inc, X_output);

/* accumulate the squared error */
for(i = 0, current_error = 0; i < N; i++) {
    current_error += pow((EOE_input[i] - EOE_output[i]), 2.0);
    current_error += pow((PAR_input[i] - PAR_output[i]), 2.0);
    current_error += pow((T_input[i] - T_output[i]), 2.0);
    current_error += pow((X_input[i] - X_output[i]), 2.0);
}

/* reset to the origin */
surface_dimer_geometry(system, 0.0, 1.5*M_PI, 0.0, 0.0, 0.0, 0.5*M_PI, 0.0);

params = calloc(1, sizeof(param_t));
param_ptr = params;
for(atom_ptr = system->molecules->atoms; atom_ptr; atom_ptr = atom_ptr->next) {

    /* XXX - uncomment for quadrupole fitting */
    /* if(strcasecmp(atom_ptr->atomtype, last_atomtype)) { */
    if(strcasecmp(atom_ptr->atomtype, last_atomtype) && ((atom_ptr->epsilon != 0.0) && (atom_ptr->sigma != 0.0))) {

        strcpy(param_ptr->atomtype, atom_ptr->atomtype);

        param_ptr->charge = atom_ptr->charge;
        param_ptr->epsilon = atom_ptr->epsilon;
        param_ptr->sigma = atom_ptr->sigma;

        for(i = 0; i < 3; i++) {
            if(fabs(atom_ptr->pos[i]) >= 0.001) {
                param_ptr->axis = i;
            }
        }

        param_ptr->last_charge = param_ptr->charge;
        param_ptr->last_epsilon = param_ptr->epsilon;
        param_ptr->last_sigma = param_ptr->sigma;

        param_ptr->next = calloc(1, sizeof(param_t));
        prev_param_ptr = param_ptr;
        param_ptr = param_ptr->next;
        strcpy(last_atomtype, atom_ptr->atomtype);

    }
}

prev_param_ptr->next = NULL;
free(param_ptr);

for(param_ptr = params; param_ptr; param_ptr = param_ptr->next) {

    for(atom_ptr = system->molecules->atoms; atom_ptr; atom_ptr = atom_ptr->next)
        if(!strcasecmp(param_ptr->atomtype, atom_ptr->atomtype))
            ++param_ptr->nTypes;

}

printf("temperature = %f, sq_error = %f\n", temperature, current_error);
for(param_ptr = params; param_ptr; param_ptr = param_ptr->next) {
    printf("\tatomtype %s q = %f eps = %f sig = %f dr = %f\n", param_ptr->atomtype, param_ptr->charge/E2REDUCED,
    param_ptr->epsilon, param_ptr->sigma, param_ptr->dr);
}
fflush(stdout);

global_minimum = current_error;
last_error = current_error;
for(i = 0; i < N; i++) {
    EOE_global[i] = EOE_output[i];
    PAR_global[i] = PAR_output[i];
    T_global[i] = T_output[i];
    X_global[i] = X_output[i];
}

for(nsteps = 0; temperature > MIN_TEMPERATURE; ++nsteps) {

    /* randomly perturb the parameters */
}

```

```

for(param_ptr = params; param_ptr; param_ptr = param_ptr->next) {
    param_ptr->epsilon += SCALE_EPSILON*(0.5 - get_rand());
    if(param_ptr->epsilon < 0.0) param_ptr->epsilon = param_ptr->last_epsilon;

    param_ptr->sigma += SCALE_SIGMA*(0.5 - get_rand());
    if(param_ptr->sigma < 0.0) param_ptr->sigma = param_ptr->last_sigma;

    if((param_ptr->natypes > 1) && strcasecmp(param_ptr->atomtype, "H2E"))
        param_ptr->dr = SCALE_R*(0.5 - get_rand());
    else
        param_ptr->dr = 0;

    if(!strcasecmp(param_ptr->atomtype, "H2G")) {
        ghost_charge = param_ptr->charge + SCALE_CHARGE*(0.5 - get_rand());
        param_ptr->charge = ghost_charge;
    } else if(!strcasecmp(param_ptr->atomtype, "H2E")) {
        param_ptr->charge = -0.5*ghost_charge;
        /* XXX - single LJ site only */
        param_ptr->epsilon = 0.0;
        param_ptr->sigma = 0.0;
    }
}

/* apply the trial parameters */
surface_dimer_parameters(system, params);

/* EOE */
surface_dimer_geometry(system, r_min, 0.0, 0.0, 0.0, 0.0, 0.0);
surface_curve(system, r_min, r_max, r_inc, EOE_output);

/* PAR */
surface_dimer_geometry(system, r_min, 0.5*M_PI, 0.0, 0.0, 0.5*M_PI, 0.0, 0.0);
surface_curve(system, r_min, r_max, r_inc, PAR_output);

/* T */
surface_dimer_geometry(system, r_min, 0.0, 0.0, 0.0, 0.5*M_PI, 0.0, 0.0);
surface_curve(system, r_min, r_max, r_inc, T_output);

/* X */
surface_dimer_geometry(system, r_min, 0.0, 0.0, 0.0, 0.0, 0.5*M_PI, 0.0);
surface_curve(system, r_min, r_max, r_inc, X_output);

/* reset to the origin */
surface_dimer_geometry(system, 0.0, 1.5*M_PI, 0.0, 0.0, 0.0, 0.5*M_PI, 0.0);

/* accumulate the squared error */
for(i = 0; current_error = 0; i < N; i++) {
    current_error += pow((EOE_input[i] - EOE_output[i]), 2.0);
    current_error += pow((PAR_input[i] - PAR_output[i]), 2.0);
    current_error += pow((T_input[i] - T_output[i]), 2.0);
    current_error += pow((X_input[i] - X_output[i]), 2.0);
}

/* do MC at this 'temperature' */
if(get_rand() < exp(-(current_error - last_error)/temperature)) {

    for(param_ptr = params; param_ptr; param_ptr = param_ptr->next) {
        param_ptr->last_charge = param_ptr->charge;
        param_ptr->last_epsilon = param_ptr->epsilon;
        param_ptr->last_sigma = param_ptr->sigma;
        param_ptr->last_dr = param_ptr->dr;
    }
    last_error = current_error;

    if(nsteps >= EQUIL) {
        temperature = temperature*SCHEDULE; /* schedule */
        nsteps = 0;
        printf("temperature = %f, sq_error = %f\n", temperature, current_error);
        for(param_ptr = params; param_ptr; param_ptr = param_ptr->next) {
            printf("(atomtype %s q = %f eps = %f sig = %f dr = %f)\n", param_ptr->atomtype,
param_ptr->charge/E2REDUCED, param_ptr->epsilon, param_ptr->sigma, param_ptr->dr);
        }
        fflush(stdout);
    }
    /* output the new global minimum */
    if(current_error < global_minimum) {

        global_minimum = current_error;
        /* output the final geometry */
        write_molecules(system, "fit_geometry.pdb");

        for(i = 0; i < N; i++) {
            EOE_global[i] = EOE_output[i];
            PAR_global[i] = PAR_output[i];
            T_global[i] = T_output[i];
            X_global[i] = X_output[i];
        }
    }
}
}

} else {
    for(param_ptr = params; param_ptr; param_ptr = param_ptr->next) {
        param_ptr->charge = param_ptr->last_charge;
}
}

```

```

        param_ptr->epsilon = param_ptr->last_epsilon;
        param_ptr->sigma = param_ptr->last_sigma;
        param_ptr->dr = param_ptr->last_dr;
    }
}

}

/* output the fit curves */
for(i = 0; i < N; i++) {
    printf("%f %f %f %f\n", r_input[i], EOE_global[i], PAR_global[i], T_global[i], X_global[i]);
    fflush(stdout);
}
return(0);
}

/*
@2007, Jonathan Belof
Space Research Group
Department of Chemistry
University of South Florida
*/
#include <mc.h>

/* calculate the field with periodic boundaries */
void thole_field(system_t *system) {

    int p;
    molecule_t *molecule_ptr;
    atom_t *atom_ptr;

    /* zero the field vectors */
    for(molecule_ptr = system->molecules; molecule_ptr; molecule_ptr = molecule_ptr->next)
        for(atom_ptr = molecule_ptr->atoms; atom_ptr; atom_ptr = atom_ptr->next)
            for(p = 0; p < 3; p++) atom_ptr->ef_static[p] = 0;

    /* calculate the electrostatic field */
    if(system->polar_ewald) {
        thole_field_real(system);
        thole_field_recip(system);
        thole_field_self(system);
    } else
        thole_field_nopbc(system);

}

/* calculate the field without ewald summation */
void thole_field_nopbc(system_t *system) {

    molecule_t *molecule_ptr;
    atom_t *atom_ptr, **atom_array;
    pair_t *pair_ptr;
    int p;
    double r, damping;

    for(molecule_ptr = system->molecules; molecule_ptr; molecule_ptr = molecule_ptr->next) {
        for(atom_ptr = molecule_ptr->atoms; atom_ptr; atom_ptr = atom_ptr->next) {
            for(pair_ptr = atom_ptr->pairs; pair_ptr; pair_ptr = pair_ptr->next) {
                if(!pair_ptr->frozen) {
                    r = pair_ptr->rimg;
                    /* exponential field damping */
                    if(system->field_damp == 0.0)
                        damping = 1.0;
                    else
                        damping = 1.0 - exp(-pow(r/system->field_damp, 3.0));
                    /* include self-induction if keyword polar_self is set */
                    if((!pair_ptr->es_excluded || system->polar_self) && (r < system->pbc->cutoff)) {
                        for(p = 0; p < 3; p++) {
                            atom_ptr->ef_static[p] += damping*pair_ptr->charge*pair_ptr->dimg[p]/(r*r*r);
                            pair_ptr->atom->ef_static[p] -= damping*atom_ptr->charge*pair_ptr->dimg[p]/(r*r*r);
                        }
                    }
                }
            }
        }
    }
}

/* real space sum */
void thole_field_real(system_t *system) {

    molecule_t *molecule_ptr;
    atom_t *atom_ptr;
    pair_t *pair_ptr;
    int p;
    double alpha;
    double field_value;
}

```

```

alpha = system->ewald_alpha;

for(molecule_ptr = system->molecules; molecule_ptr = molecule_ptr->next) {
    for(atom_ptr = molecule_ptr->atoms; atom_ptr; atom_ptr = atom_ptr->next) {
        for(pair_ptr = atom_ptr->pairs; pair_ptr; pair_ptr = pair_ptr->next) {
            if(!pair_ptr->frozen) {
                if(!pair_ptr->es_excluded) {
                    field_value = erfc(alpha*pair_ptr->rimg) +
                    2.0*alpha*pair_ptr->rimg*exp(-alpha*alpha*pair_ptr->rimg*pair_ptr->rimg)/sqrt(M_PI);
                    for(p = 0; p < 3; p++) {
                        atom_ptr->ef_static[p] +=
                        pair_ptr->charge*field_value*pair_ptr->dimg[p]/(pair_ptr->rimg*pair_ptr->rimg*pair_ptr->rimg);
                        pair_ptr->atom->ef_static[p] =
                        atom_ptr->charge*field_value*pair_ptr->dimg[p]/(pair_ptr->rimg*pair_ptr->rimg*pair_ptr->rimg);
                    }
                }
            } /* !frozen */
        } /* pair */
    } /* atom */
} /* molecule */

}

/* fourier space sum */
void thole_field_recip(system_t *system) {

int i, N, r, p, q;
int kmax;
double alpha, gaussian;
double vector_product_i, vector_product_j;
double SF_sin, SF_cos;
int norm, l[3];
int k_squared, k[3];
molecule_t *molecule_ptr;
atom_t *atom_ptr, **atom_array;
pair_t *pair_ptr;

/* our convergence parameters */
alpha = system->ewald_alpha;
kmax = system->ewald_kmax;

/* generate an array of atom ptrs */
for(molecule_ptr = system->molecules, N = 0, atom_array = NULL; molecule_ptr = molecule_ptr->next) {
    for(atom_ptr = molecule_ptr->atoms; atom_ptr; atom_ptr = atom_ptr->next, N++) {
        atom_array = realloc(atom_array, sizeof(atom_t *)*(N + 1));
        atom_array[N] = atom_ptr;
    }
}

/* calculate the field vector for each nuclear coordinate */
for(i = 0; i < N; i++) {
    /* perform the fourier sum over the reciprocal lattice */
    for(l[0] = 0; l[0] <= kmax; l[0]++) {
        for(l[1] = (!l[0]) ? 0 : -kmax; l[1] <= kmax; l[1]++) {
            for(l[2] = ((!l[0] && !l[1]) ? 1 : -kmax); l[2] <= kmax; l[2]++) {
                /* k-vector norm */
                for(p = 0, norm = 0; p < 3; p++)
                    norm += l[p]*l[p];
                /* get the reciprocal lattice vectors */
                for(p = 0, k_squared = 0; p < 3; p++) {
                    for(q = 0, k[p] = 0; q < 3; q++)
                        k[p] += system->pbc->reciprocal_basis[p][q]*2.0*M_PI*((double)l[q]);
                    k_squared += k[p]*k[p];
                }
                /* compare the norm */
                if((norm <= kmax*kmax) && (k_squared > 0.0)) {
                    /* our gaussian centered on the charge */
                    gaussian = exp(-k_squared/(4.0*alpha*alpha))/k_squared;
                    /* the structure factor */
                    SF_sin = 0; SF_cos = 0;
                    for(molecule_ptr = system->molecules; molecule_ptr = molecule_ptr->next) {
                        for(atom_ptr = molecule_ptr->atoms; atom_ptr; atom_ptr = atom_ptr->next) {
                            /* make sure it's not a frozen-frozen interaction */
                            if(!(atom_ptr->frozen && atom_array[i]->frozen)) {
                                /* inner product of j-th position vector and k-vector */
                                for(p = 0, vector_product_j = 0; p < 3; p++)
                                    vector_product_j += k[p]*atom_ptr->pos[p];
                                SF_sin += atom_ptr->charge*sin(vector_product_j);
                                SF_cos += -atom_ptr->charge*cos(vector_product_j);
                            }
                        } /* !frozen */
                    } /* atom */
                } /* molecule */
                /* inner product of i-th position vector and k-vector */
                for(p = 0, vector_product_i = 0; p < 3; p++)

```

```

    vector_product_i += k[p]*atom_array[i]->pos[p];
    /* the i-th terms outside of the SF sum */
    SF_sin *= cos(vector_product_i);
    SF_cos *= sin(vector_product_i);
    /* project the field components */
    for(r = 0; r < 3; r++)
        atom_array[i]->ef_static[r] += -4.0*M_PI*k[r]*gaussian*(SF_sin +
    SF_cos)/system->pbc->volume;

    } /* end if norm */

} /* end for n */
} /* end for m */
} /* end for l */

} /* end i */

free(atom_array);

}

/* self interaction term */
void thole_field_self(system_t *system) {

molecule_t *molecule_ptr;
atom_t *atom_ptr;
pair_t *pair_ptr;
int p;
double alpha;
double field_value;

alpha = system->ewald_alpha;

for(molecule_ptr = system->molecules; molecule_ptr; molecule_ptr = molecule_ptr->next) {
    for(atom_ptr = molecule_ptr->atoms; atom_ptr; atom_ptr = atom_ptr->next) {
        for(pair_ptr = atom_ptr->pairs; pair_ptr; pair_ptr = pair_ptr->next) {
            if(!pair_ptr->frozen) {
                if(pair_ptr->es_excluded) {
                    field_value = erf(alpha*pair_ptr->r) -
2.0*alpha*pair_ptr->r*exp(-alpha*alpha*pair_ptr->r*pair_ptr->r)/sqrt(M_PI);
                    for(p = 0; p < 3; p++) {
                        atom_ptr->ef_static[p] -=
pair_ptr->charge*field_value*pair_ptr->d[p]/(pair_ptr->r*pair_ptr->r*pair_ptr->r);
                        pair_ptr->atom->ef_static[p] +=
atom_ptr->charge*field_value*pair_ptr->d[p]/(pair_ptr->r*pair_ptr->r*pair_ptr->r);
                    }
                }
            } /* !frozen */
        } /* pair */
    } /* atom */
} /* molecule */

}

/*
@2007, Jonathan Belof
Space Research Group
Department of Chemistry
University of South Florida
*/
#include <mc.h>

/* iterative solver of the dipole field tensor */
/* returns the number of iterations required */
int thole_iterative(system_t *system) {

int i, j, ii, jj, N, p, q, sorted;
int iteration_counter, keep_iterating;
double error;
molecule_t *molecule_ptr;
atom_t *atom_ptr, **atom_array;
int *ranked_array, ranked, tmp, index;

/* generate an array of atom ptrs */
for(molecule_ptr = system->molecules, N = 0, atom_array = NULL; molecule_ptr; molecule_ptr = molecule_ptr->next) {
    for(atom_ptr = molecule_ptr->atoms; atom_ptr; atom_ptr = atom_ptr->next, N++) {
        atom_array = realloc(atom_array, sizeof(atom_t *)*(N + 1));
        atom_array[N] = atom_ptr;
    }
}

/* array for ranking */
ranked_array = calloc(N, sizeof(int));
for(i = 0; i < N; i++) ranked_array[i] = i;

/* initialize things */
for(i = 0; i < N; i++) {

```

```

    for(p = 0; p < 3; p++) {
        /* set the first guess to alpha*E */
        atom_array[i]->mu[p] = atom_array[i]->polarizability*atom_array[i]->ef_static[p];
        if(!system->polar_sor) atom_array[i]->mu[p] *= system->polar_gamma;
    }
}

/* if ZODID is enabled, then stop here and just return the alpha*E dipoles */
if(system->polar_zodid) return(0);

/* iterative solver of the dipole field equations */
for(iteration_counter = 0, keep_iterating = 1; keep_iterating; iteration_counter++) {

    /* divergence detection */
    /* if we fail to converge, then return dipoles as alpha*E */
    if(iteration_counter > MAX_ITERATION_COUNT) {

        for(i = 0; i < N; i++)
            for(p = 0; p < 3; p++) atom_array[i]->mu[p] = atom_array[i]->polarizability*atom_array[i]->ef_static[p];

        free(atom_array);
        return(iteration_counter);
    }

    /* save the current dipole set and clear the induced field vectors */
    for(i = 0; i < N; i++) {
        for(p = 0; p < 3; p++) {
            atom_array[i]->old_mu[p] = atom_array[i]->mu[p];
            atom_array[i]->ef_induced[p] = 0;
            atom_array[i]->ef_induced_change[p] = 0;
        }
    }

    /* contract the dipoles with the field tensor */
    for(i = 0; i < N; i++) {
        index = ranked_array[i];
        ii = index*3;

        for(j = 0; j < N; j++) {
            jj = j*3;

            if(index != j) {
                for(p = 0; p < 3; p++)
                    for(q = 0; q < 3; q++)
                        atom_array[index]->ef_induced[p] -=
                            system->A_matrix[ii+p][jj+q]*atom_array[j]->mu[q];
            }
        }
    } /* end j */

    /* dipole is the sum of the static and induced parts */
    for(p = 0; p < 3; p++) {
        atom_array[index]->new_mu[p] = atom_array[index]->polarizability*(atom_array[index]->ef_static[p] +
            atom_array[index]->ef_induced[p]);
        /* Gauss-Seidel */
        if(system->polar_gs || system->polar_gs_ranked)
            atom_array[index]->mu[p] = atom_array[index]->new_mu[p];
    }
} /* end i */

/* get the dipole RRMS */
for(i = 0; i < N; i++) {
    /* mean square difference */
    atom_array[i]->dipole_rrms = 0;
    for(p = 0; p < 3; p++)
        atom_array[i]->dipole_rrms += pow((atom_array[i]->new_mu[p] - atom_array[i]->old_mu[p]), 2.0);

    /* normalize */
    atom_array[i]->dipole_rrms /= pow(atom_array[i]->new_mu[0], 2.0) + pow(atom_array[i]->new_mu[1], 2.0) +
        pow(atom_array[i]->new_mu[2], 2.0);
    atom_array[i]->dipole_rrms = sqrt(atom_array[i]->dipole_rrms);
}

/* determine if we are done... */
if(system->polar_precision == 0.0) { /* ... by fixed iteration ... */

    if(iteration_counter == system->polar_max_iter)
        keep_iterating = 0;
    else
        keep_iterating = 1;
} else { /* ... or by dipole precision */

    /* immediately reiterate if any component broke tolerance, otherwise we are done */
    for(i = 0, keep_iterating = 0; i < N; i++) {

        /* get the change of dipole between iterations */
        for(p = 0; p < 3; p++) {
            error = pow((atom_array[i]->new_mu[p] - atom_array[i]->old_mu[p]), 2.0);
            if(error > pow(system->polar_precision*DEBYE2SKA, 2.0)) keep_iterating = 1;
        }
    }
}

```

```

        }
    }

/* contract once more for the next induced field */
if(system->polar_paldo) {
    /* calculate change in induced field due to this iteration */
    for(i = 0; i < N; i++) {
        index = ranked_array[i];
        ii = index*3;

            for(j = 0; j < N; j++) {
                if(index != j) {

                    for(p = 0; p < 3; p++)
                        for(q = 0; q < 3; q++)
                            atom_array[index]->ef_induced_change[p] -=
system->A_matrix[ii+p][jj+q]*atom_array[j]->mu[q];
                }
                for(p = 0; p < 3; p++)
                    atom_array[index]->ef_induced_change[p] -= atom_array[index]->ef_induced[p];
            } /* end i */
    } /* palmo */

/* rank the dipoles by bubble sort */
if(system->polar_gs_ranked) {
    for(i = 0; i < N; i++) {
        for(j = 0, sorted = 1; j < (N-i); j++) {
            if(atom_array[ranked_array[j]]->rank_metric < atom_array[ranked_array[j+1]]->rank_metric) {
                sorted = 0;
                tmp = ranked_array[j];
                ranked_array[j] = ranked_array[j+1];
                ranked_array[j+1] = tmp;
            }
            if(sorted) break;
        }
    }
}

/* save the dipoles for the next pass */
for(i = 0; i < N; i++) {
    for(p = 0; p < 3; p++) {

        /* allow for different successive over-relaxation schemes */
        if(system->polar_sor)
            atom_array[i]->mu[p] = system->polar_gamma*atom_array[i]->new_mu[p] + (1.0 -
system->polar_gamma)*atom_array[i]->old_mu[p];
        else if(system->polar_esor)
            atom_array[i]->mu[p] = (1.0 - exp(-system->polar_gamma*iteration_counter))*atom_array[i]->new_mu[p]
+ exp(-system->polar_gamma*iteration_counter)*atom_array[i]->old_mu[p];
        else
            atom_array[i]->mu[p] = atom_array[i]->new_mu[p];
    }
}
}

/* end keep iterating */
free(atom_array);
free(ranked_array);

/* return the iteration count */
return(iteration_counter);
}

/*
02007, Jonathan Belof
Space Research Group
Department of Chemistry
University of South Florida
*/
#include <mc.h>

void print_matrix(int N, double **matrix) {

int i, j;

printf("\n");
for(i = 0; i < N; i++) {
    for(j = 0; j < N; j++) {
        printf("%.3f ", matrix[i][j]);
    }
    printf("\n");
}
printf("\n");

}

/* calculate the dipole field tensor */
void thole_amatrix(system_t *system) {

int i, j, ii, jj, N, p, q;
molecule_t *molecule_ptr;
atom_t *atom_ptr, **atom_array;
pair_t *pair_ptr;
double r, damping_term1, damping_term2;
double r3, r5;

```

```

double v, s;

/* generate an array of atom ptrs */
for(molecule_ptr = system->molecules, N = 0, atom_array = NULL; molecule_ptr; molecule_ptr = molecule_ptr->next) {
    for(atom_ptr = molecule_ptr->atoms; atom_ptr; atom_ptr = atom_ptr->next, N++) {
        atom_array = realloc(atom_array, sizeof(atom_t)*(N + 1));
        atom_array[N] = atom_ptr;
    }
}

/* zero out the matrix */
for(i = 0; i < 3*N; i++)
    for(j = 0; j < 3*N; j++)
        system->A_matrix[i][j] = 0;

/* set the diagonal blocks */
for(i = 0; i < N; i++) {
    ii = i*3;

    for(p = 0; p < 3; p++) {
        if(atom_array[i]->polarizability != 0.0)
            system->A_matrix[ii+p][ii+p] = 1.0/atom_array[i]->polarizability;
        else
            system->A_matrix[ii+p][ii+p] = MAXVALUE;
    }
}

/* calculate each Tij tensor component for each dipole pair */
for(i = 0; i < (N - 1); i++) {
    ii = i*3;

    for(j = (i + 1), pair_ptr = atom_array[i]->pairs; j < N; j++, pair_ptr = pair_ptr->next) {
        jj = j*3;

        /* inverse displacements */
        r3 = pow(pair_ptr->rimg, -3.0);
        r5 = pow(pair_ptr->rimg, -5.0);

        /* set the damping function */
        if(system->damp_type == DAMPING_LINEAR) { /* linear damping */

            s = (system->polar_damp)*pow((atom_array[i]->polarizability*atom_array[j]->polarizability), (1.0/6.0));
            v = pair_ptr->rimg/s;

            if(pair_ptr->rimg < s) {
                damping_term1 = (4.0*v*v*v - 3.0*v*v*v*v);
                damping_term2 = v*v*v*v;
            } else {
                damping_term1 = 1.0;
                damping_term2 = 1.0;
            }
        }

        /* else if(system->damp_type == DAMPING_EXPONENTIAL) { /* exponential damping */

            damping_term1 = 1.0 -
                exp(-system->polar_damp*pair_ptr->rimg)*(0.5*system->polar_damp*system->polar_damp*pair_ptr->r2img + system->polar_damp*pair_ptr->rimg + 1.0);
            damping_term2 = 1.0 -
                exp(-system->polar_damp*pair_ptr->rimg)*(system->polar_damp*system->polar_damp*system->polar_damp*pair_ptr->r3img/6.0 +
                0.5*system->polar_damp*system->polar_damp*pair_ptr->r2img + system->polar_damp*pair_ptr->rimg + 1.0);
        }

        /* build the tensor */
        for(p = 0; p < 3; p++) {
            for(q = 0; q < 3; q++) {
                system->A_matrix[ii+p][jj+q] = -3.0*pair_ptr->dimg[p]*pair_ptr->dimg[q]*damping_term2*r5;
                /* additional diagonal term */
                if(p == q)
                    system->A_matrix[ii+p][jj+q] += damping_term1*r3;
            }
        }

        /* set the lower half of the tensor component */
        for(p = 0; p < 3; p++)
            for(q = 0; q < 3; q++)
                system->A_matrix[jj+p][ii+q] = system->A_matrix[ii+p][jj+q];
    }
}
}

/* returns the current number of atoms in the system */
int num_atoms(system_t *system) {

    int N_atoms;
    molecule_t *molecule_ptr;
    atom_t *atom_ptr;

    /* count the number of atoms in the system initially */
    for(molecule_ptr = system->molecules, N_atoms = 0; molecule_ptr; molecule_ptr = molecule_ptr->next) {
        for(atom_ptr = molecule_ptr->atoms; atom_ptr; atom_ptr = atom_ptr->next) {

```

```

        ++N_atoms;
    }
}

return(N_atoms);
}

/* for uvt runs, resize the A (and B) matrices */
void thole_resize_matrices(system_t *system) {

    int i, N, dN;

    /* determine how the number of atoms has changed and realloc matrices */
    system->checkpoint->N_atom_prev = system->checkpoint->N_atom;
    system->checkpoint->N_atom = num_atoms(system);
    N = 3*system->checkpoint->N_atom;
    dN = 3*(system->checkpoint->N_atom - system->checkpoint->N_atom_prev);

    if(!dN) return;
    if(dN < 0) {

        for(i = N; i < (N - dN); i++) free(system->A_matrix[i]);
        system->A_matrix = realloc(system->A_matrix, N*sizeof(double *));
        for(i = 0; i < N; i++) system->A_matrix[i] = realloc(system->A_matrix[i], N*sizeof(double));

        if(!system->polar_iterative) {
            for(i = N; i < (N - dN); i++) free(system->B_matrix[i]);
            system->B_matrix = realloc(system->B_matrix, N*sizeof(double *));
            for(i = 0; i < N; i++) system->B_matrix[i] = realloc(system->B_matrix[i], N*sizeof(double));
        }
    } else if (dN > 0) {

        system->A_matrix = realloc(system->A_matrix, N*sizeof(double *));
        for(i = (N - dN); i < N; i++) system->A_matrix[i] = calloc(N, sizeof(double));
        for(i = 0; i < (N - dN); i++) system->A_matrix[i] = realloc(system->A_matrix[i], N*sizeof(double));

        if(!system->polar_iterative) {
            system->B_matrix = realloc(system->B_matrix, N*sizeof(double *));
            for(i = (N - dN); i < N; i++) system->B_matrix[i] = calloc(N, sizeof(double));
            for(i = 0; i < (N - dN); i++) system->B_matrix[i] = realloc(system->B_matrix[i], N*sizeof(double));
        }
    }
}

/* invert the A matrix */
void thole_bmatrix(system_t *system) {

    int N;
    molecule_t *molecule_ptr;
    atom_t *atom_ptr;

    /* count the number of atoms */
    N = 0;
    for(molecule_ptr = system->molecules; molecule_ptr; molecule_ptr = molecule_ptr->next)
        for(atom_ptr = molecule_ptr->atoms; atom_ptr; atom_ptr = atom_ptr->next)
            ++N;

    invert_matrix(3*N, system->A_matrix, system->B_matrix);
}

/* get the dipoles by vector matrix multiply */
void thole_bmatrix_dipoles(system_t *system) {

    int i, j, ii, p, N;
    molecule_t *molecule_ptr;
    atom_t *atom_ptr, **atom_array;
    double *mu_array, *field_array;

    /* generate an array of atom ptrs */
    for(molecule_ptr = system->molecules; N == 0, atom_array = NULL; molecule_ptr; molecule_ptr = molecule_ptr->next) {
        for(atom_ptr = molecule_ptr->atoms; atom_ptr; atom_ptr = atom_ptr->next, N++) {
            atom_array = realloc(atom_array, sizeof(atom_t *)*(N + 1));
            atom_array[N] = atom_ptr;
        }
    }

    /* allocate working arrays */
    mu_array = calloc(3*N, sizeof(double));
    field_array = calloc(3*N, sizeof(double));

    /* copy the field in */
    for(i = 0; i < N; i++) {
        ii = i*3;
        for(p = 0; p < 3; p++)
            field_array[ii+p] = atom_array[i]->ef_static[p];
    }

    /* multiply the supervector with the B matrix */
    for(i = 0; i < 3*N; i++)
        for(j = 0; j < 3*N; j++)
            mu_array[i] += system->B_matrix[i][j]*field_array[j];

    /* copy the dipoles out */
    for(i = 0; i < N; i++) {
}

```

```

    ii = i*3;

    for(p = 0; p < 3; p++)
        atom_array[i]->mu[p] = mu_array[ii+p];
}

/* free the working arrays */
free(atom_array);
free(mu_array);
free(field_array);

}

/* numerical recipes routines for inverting a general matrix */

#define TINY 1.0e-20

void LU_decomp(double **a,int n,int *indx,double *d)
{
    int i,j,k,imax=0;
    double big,dum,sum,temp,*vv;

    vv=(double *)malloc(n*sizeof(double));
    *d=1.0;
    for (i=0;i<n;i++) {
        big=0.0;
        for (j=0;j<n;j++) {
            if ((temp=fabs(a[i][j])) > big) big=temp;
        /* note big cannot be zero */
            if(big == 0.0) {
                printf("LU_decomp: matrix to invert can't be singular!\n");
                exit(0);
            }
            vv[i]=1.0/big;
        }
        for (j=0;j<n;j++) {
            for (i=0;i<j;i++) {
                sum=a[i][j];
                for (k=0;k<i;k++) sum -= a[i][k]*a[k][j];
                a[i][j]=sum;
            }
            big=0.0;
            for (i=j;i<n;i++) {
                sum=a[i][j];
                for (k=0;k<j;k++) sum -= a[i][k]*a[k][j];
                a[i][j]=sum;
                if ( (dum=vv[i]*fabs(sum)) >= big) {
                    big=dum;
                    imax=i;
                }
            }
            if (j != imax) {
                for (k=0;k<n;k++) {
                    dum=a[imax][k];
                    a[imax][k]=a[j][k];
                    a[j][k]=dum;
                }
                *d = -(*d);
                vv[imax]=vv[j];
            }
            indx[j]=imax;
            if (a[j][j] == 0.0) a[j][j]=TINY;
            if (j != n) {
                dum=1.0/(a[j][j]);
                for (i=j+1;i<n;i++) a[i][j] *= dum;
            }
            free(vv);
        }
    }

    void LU_bksb(double **a,int n,int *indx,double b[])
    {
        int i,j,ii=-1,ip;
        double sum;

        for (i=0;i<n;i++) {
            ip=indx[i];
            sum=b[ip];
            b[ip]=b[i];
            if (ii != -1) for (j=ii;j<=i-1;j++) sum -= a[i][j]*b[j];
            else if (sum) ii=i;
            b[i]=sum;
        }

        for (i=n-1;i>=0;i--) {
            sum=b[i];
            for (j=i+1;j<n;j++) sum -= a[i][j]*b[j];
            b[i]=sum/a[i][i];
        }
    }

    /* invert a NxN matrix using routines from numerical recipes */
    void invert_matrix(int n,double **a,double **ai) {

        int i,j,*indx;
        double *col,d;

        col = (double *)malloc(n*sizeof(double));
        indx = (int *)malloc(n*sizeof(int));

        LU_decomp(a,n,indx,&d);

        for(i=0;i<n;i++){

```

```

    for(j=0;j<n;j++) col[j] = 0.;
    col[i] = 1.;
    LU_bksb(a,n,indx,col);
    for(j=0;j<n;j++) ai[j][i] = col[j];
}

free(col);free(indx);

}

/*
@2007, Jonathan Belof
Space Research Group
Department of Chemistry
University of South Florida
*/
#include <mc.h>

/* calculate the molecular polarizability tensor from the B matrix */

void thole_polarizability_tensor(system_t *system) {

    int i, j, ii, jj, N;
    int p, q;
    double isotropic;

    N = system->checkpoint->N_atom;

    /* clear the polarizability tensor */
    for(p = 0; p < 3; p++)
        for(q = 0; q < 3; q++)
            system->C_matrix[p][q] = 0;

    /* sum the block terms for the 3x3 molecular tensor */
    for(p = 0; p < 3; p++) {
        for(q = 0; q < 3; q++) {
            for(i = 0; i < N; i++) {
                ii = i*3;
                for(j = 0; j < N; j++) {
                    jj = j*3;
                    system->C_matrix[p][q] += system->B_matrix[ii+p][jj+q];
                }
            }
        }
    }

    /* get the isotropic term */
    for(p = 0, isotropic = 0; p < 3; p++)
        isotropic += system->C_matrix[p][p];
    isotropic /= 3.0;

    printf("POLARIZATION: polarizability tensor (A^3):\n");
    printf("#####\n");
    for(p = 0; p < 3; p++) {
        for(q = 0; q < 3; q++)
            printf("%.3f ", system->C_matrix[p][q]);
        printf("\n");
    }
    printf("#####\n");
    printf("isotropic = %.3f\n", isotropic);
}

/*
@2007, Jonathan Belof
Space Research Group
Department of Chemistry
University of South Florida
*/
#include <mc.h>

/* support the first 5 spherical harmonics, they are hard-coded but necessary for gaussian quadrature integration */
/* (doing something slick like a func ptr array [l][m] would be just as ugly) */
double rotational_basis(int type, int l, int m, double theta, double phi) {

    int m_abs;
    double legendre, Ylm;

    /* calculate the prefactor*legendre part */
    m_abs = abs(m);

    if(l == 0) {
        legendre = (1.0/2.0)*sqrt(1.0/M_PI);
    } else if(l == 1) {
        if(m_abs == 0)
            legendre = (1.0/2.0)*sqrt(3.0/M_PI)*cos(theta);
        else if(m_abs == 1)
            legendre = -1.0*(1.0/2.0)*sqrt(3.0/(2.0*M_PI))*sin(theta);
    } else if(l == 2) {
        if(m_abs == 0)
            legendre = (1.0/4.0)*sqrt(5.0/M_PI)*(3.0*pow(cos(theta), 2.0) - 1.0);
        else if(m_abs == 1)
            legendre = -1.0*(1.0/2.0)*sqrt(15.0/(2.0*M_PI))*(sin(theta)*cos(theta));
        else if(m_abs == 2)
            legendre = (1.0/8.0)*sqrt(35.0/M_PI)*(5.0*pow(cos(theta), 2.0) - 3.0*cos(theta));
    }
}

```

```

legendre = (1.0/4.0)*sqrt(15.0/(2.0*M_PI))*pow(sin(theta), 2.0);
} else if(l == 3) {
    if(m_abs == 0)
        legendre = (1.0/4.0)*sqrt(7.0/M_PI)*(5.0*pow(cos(theta), 3.0) - 3.0*cos(theta));
    else if(m_abs == 1)
        legendre = -1.0*(1.0/8.0)*sqrt(21.0/M_PI)*sin(theta)*(5.0*pow(cos(theta), 2.0) - 1.0);
    else if(m_abs == 2)
        legendre = (1.0/4.0)*sqrt(105.0/(2.0*M_PI))*pow(sin(theta), 2.0)*cos(theta);
    else if(m_abs == 3)
        legendre = -1.0*(1.0/8.0)*sqrt(35.0/M_PI)*pow(sin(theta), 3.0);
} else if(l == 4) {
    if(m_abs == 0)
        legendre = (3.0/16.0)*sqrt(1.0/M_PI)*(35.0*pow(cos(theta), 4.0) - 30.0*pow(cos(theta), 2.0) + 3.0);
    else if(m_abs == 1)
        legendre = -1.0*(3.0/8.0)*sqrt(5.0/M_PI)*sin(theta)*(7.0*pow(cos(theta), 3.0) - 3.0*cos(theta));
    else if(m_abs == 2)
        legendre = (3.0/8.0)*sqrt(5.0/(2.0*M_PI))*pow(sin(theta), 2.0)*(7.0*pow(cos(theta), 2.0) - 1.0);
    else if(m_abs == 3)
        legendre = -1.0*(3.0/8.0)*sqrt(35.0/M_PI)*pow(sin(theta), 3.0)*cos(theta);
    else if(m_abs == 4)
        legendre = (3.0/16.0)*sqrt(35.0/(2.0*M_PI))*pow(sin(theta), 4.0);
} else if(l == 5) {
    if(m_abs == 0)
        legendre = (1.0/16.0)*sqrt(11.0/M_PI)*(63.0*pow(cos(theta), 5.0) - 70.0*pow(cos(theta), 3.0) + 15.0*cos(theta));
    else if(m_abs == 1)
        legendre = -1.0*(1.0/16.0)*sqrt(165.0/(2.0*M_PI))*sin(theta)*(21.0*pow(cos(theta), 4.0) - 14.0*pow(cos(theta), 2.0) +
1.0);
    else if(m_abs == 2)
        legendre = (1.0/8.0)*sqrt(1155.0/(2.0*M_PI))*pow(sin(theta), 2.0)*(3.0*pow(cos(theta), 3.0) - cos(theta));
    else if(m_abs == 3)
        legendre = -1.0*(1.0/32.0)*sqrt(385.0/M_PI)*pow(sin(theta), 3.0)*(9.0*pow(cos(theta), 2.0) - 1.0);
    else if(m_abs == 4)
        legendre = (3.0/16.0)*sqrt(385.0/(2.0*M_PI))*pow(sin(theta), 4.0)*cos(theta);
    else if(m_abs == 5)
        legendre = -1.0*(3.0/32.0)*sqrt(77.0/M_PI)*pow(sin(theta), 5.0);
} else if(l == 6) {
    if(m_abs == 0)
        legendre = (1.0/32.0)*sqrt(13.0/M_PI)*(231.0*pow(cos(theta), 6.0) - 315.0*pow(cos(theta), 4.0) +
105.0*pow(cos(theta), 2.0) - 5.0);
    else if(m_abs == 1)
        legendre = -1.0*(1.0/16.0)*sqrt(273.0/(2.0*M_PI))*(33.0*pow(cos(theta), 5.0) - 30.0*pow(cos(theta), 3.0) +
5.0*cos(theta));
    else if(m_abs == 2)
        legendre = (1.0/64.0)*sqrt(1365.0/M_PI)*pow(sin(theta), 2.0)*(33.0*pow(cos(theta), 4.0) - 18.0*pow(cos(theta), 2.0) +
1.0);
    else if(m_abs == 3)
        legendre = -1.0*(1.0/32.0)*sqrt(1365.0/M_PI)*pow(sin(theta), 3.0)*(11.0*pow(cos(theta), 3.0) - 3.0*cos(theta));
    else if(m_abs == 4)
        legendre = (3.0/32.0)*sqrt(91.0/(2.0*M_PI))*pow(sin(theta), 4.0)*(11.0*pow(cos(theta), 2.0) - 1.0);
    else if(m_abs == 5)
        legendre = -1.0*(3.0/32.0)*sqrt(1001.0/M_PI)*pow(sin(theta), 5.0)*cos(theta);
    else if(m_abs == 6)
        legendre = (1.0/64.0)*sqrt(3003.0/M_PI)*pow(sin(theta), 6.0);
} else if(l == 7) {
    if(m_abs == 0)
        legendre = (1.0/32.0)*sqrt(15.0/M_PI)*(429.0*pow(cos(theta), 7.0) - 693.0*pow(cos(theta), 5.0) +
315.0*pow(cos(theta), 3.0) - 35.0*cos(theta));
    else if(m_abs == 1)
        legendre = -1.0*(1.0/64.0)*sqrt(105.0/(2.0*M_PI))*sin(theta)*(429.0*pow(cos(theta), 6.0) - 495.0*pow(cos(theta),
4.0) + 135.0*pow(cos(theta), 2.0) - 5.0);
    else if(m_abs == 2)
        legendre = (3.0/64.0)*sqrt(35.0/M_PI)*pow(sin(theta), 2.0)*(143.0*pow(cos(theta), 5.0) - 110.0*pow(cos(theta), 3.0) +
15.0*cos(theta));
    else if(m_abs == 3)
        legendre = -1.0*(3.0/64.0)*sqrt(35.0/(2.0*M_PI))*pow(sin(theta), 3.0)*(143.0*pow(cos(theta), 4.0) -
66.0*pow(cos(theta), 2.0) + 3.0);
    else if(m_abs == 4)
        legendre = (3.0/32.0)*sqrt(385.0/(2.0*M_PI))*pow(sin(theta), 4.0)*(13.0*pow(cos(theta), 3.0) - 3.0*cos(theta));
    else if(m_abs == 5)
        legendre = -1.0*(3.0/64.0)*sqrt(385.0/(2.0*M_PI))*pow(sin(theta), 5.0)*(13.0*pow(cos(theta), 2.0) - 1.0);
    else if(m_abs == 6)
        legendre = (3.0/64.0)*sqrt(5005.0/M_PI)*pow(sin(theta), 6.0)*cos(theta);
    else if(m_abs == 7)
        legendre = -1.0*(3.0/64.0)*sqrt(715.0/(2.0*M_PI))*pow(sin(theta), 6.0);
} else if(l == 8) {
    if(m_abs == 0)
        legendre = (1.0/256.0)*sqrt(17.0/M_PI)*(6435.0*pow(cos(theta), 8.0) - 12012.0*pow(cos(theta), 6.0) +
6930.0*pow(cos(theta), 4.0) - 1280.0*pow(cos(theta), 2.0) + 35.0);
    else if(m_abs == 1)
        legendre = -1.0*(3.0/64.0)*sqrt(17.0/(2.0*M_PI))*sin(theta)*(715.0*pow(cos(theta), 7.0) - 1001.0*pow(cos(theta),
5.0) + 385.0*pow(cos(theta), 3.0) - 35.0*cos(theta));
    else if(m_abs == 2)
        legendre = (3.0/128.0)*sqrt(595.0/M_PI)*pow(sin(theta), 2.0)*(143.0*pow(cos(theta), 6.0) - 143.0*pow(cos(theta),
4.0) + 33.0*pow(cos(theta), 2.0) - 1.0);
    else if(m_abs == 3)
        legendre = -1.0*(1.0/64.0)*sqrt(19635.0/(2.0*M_PI))*pow(sin(theta), 3.0)*(39.0*pow(cos(theta), 5.0) -
26.0*pow(cos(theta), 3.0) + 3.0*cos(theta));
    else if(m_abs == 4)
        legendre = (3.0/128.0)*sqrt(1309.0/(2.0*M_PI))*pow(sin(theta), 4.0)*(65.0*pow(cos(theta), 4.0) -
26.0*pow(cos(theta), 2.0) + 1.0);
    else if(m_abs == 5)
        legendre = -1.0*(3.0/64.0)*sqrt(17017.0/(2.0*M_PI))*pow(sin(theta), 5.0)*(5.0*pow(cos(theta), 3.0) - cos(theta));
    else if(m_abs == 6)
        legendre = (1.0/128.0)*sqrt(7293.0/M_PI)*pow(sin(theta), 6.0)*(15.0*pow(cos(theta), 2.0) - 1.0);
    else if(m_abs == 7)
}

```

```

    legendre = -1.0*(3.0/64.0)*sqrt(12155.0/(2.0*M_PI))*pow(sin(theta), 7.0)*cos(theta);
    else if(m_abs == 8)
        legendre = (3.0/256.0)*sqrt(12155.0/(2.0*M_PI))*pow(sin(theta), 8.0);
}

/* tack on the e^{i*m*phi} */
if(m < 0) { /* use the fact that Y_l(-m) = (-1)^m*cc(Y_lm) */
    if(type == REAL)
        Ylm = legendre*cos(fabs((double)m)*phi);
    else if(type == IMAGINARY)
        Ylm = -1.0*legendre*sin(fabs((double)m)*phi);
    Ylm *= pow(-1.0, fabs((double)m));
} else {
    if(type == REAL)
        Ylm = legendre*cos(((double)m)*phi);
    else if(type == IMAGINARY)
        Ylm = legendre*sin(((double)m)*phi);
}

return(Ylm);
}

/*
@2007, Jonathan Belof
Space Research Group
Department of Chemistry
University of South Florida
*/
#include <mc.h>

/* returns the hamiltonian matrix for the potential integrated over the Ylm basis */
complex_t **rotational_hamiltonian(system_t *system, molecule_t *molecule, int level_max, int l_max) {

int dim;
int i, j;
int li, mi, lj, mj;
complex_t **hamiltonian;

/* determine the size of the symmetric matrix */
dim = (l_max + 1)*(l_max + 1);

/* allocate our hamiltonian matrix */
hamiltonian = calloc(dim, sizeof(complex_t *));
for(i = 0; i < dim; i++)
    hamiltonian[i] = calloc(dim, sizeof(complex_t));

/* construct the hamiltonian matrix, integrate to get each element */
for(i = 0, li = 0; i < dim; i++) {
    for(j = i, lj = li; j < dim; j++) {

        /* determine m values */
        mi = i - li*(li + 1);
        mj = j - lj*(lj + 1);

        /* calculate the dirac braket of <li,mi|V|lj,mj> */
        hamiltonian[i][j].real = rotational_integrate(system, molecule, REAL, li, mi, lj, mj);
        hamiltonian[i][j].imaginary = rotational_integrate(system, molecule, IMAGINARY, li, mi, lj, mj);

        /* store the lower half */
        if(i != j) {
            hamiltonian[j][i].real = hamiltonian[i][j].real;
            hamiltonian[j][i].imaginary = -1.0*hamiltonian[i][j].imaginary;
        } else {
            /* add the kinetic part to the diagonal elements */
            hamiltonian[i][j].real += system->quantum_rotation_B*((double)(li*(li + 1)));
        }

        /* output the matrix */
        printf("rotmat %d %d %.16lg %.16lg K (%.16lg %.16lg cm^-1) <%d %d | V | %d %d>\n", (i+1), (j+1),
        hamiltonian[i][j].real, hamiltonian[i][j].imaginary, hamiltonian[i][j].real*K2WN, hamiltonian[i][j].imaginary*K2WN, li, mi, lj, mj);
        fflush(stdout);

        /* advance the lj when mj hits the top */
        if(mj == lj) lj++;

    } /* for j */
    /* advance the li when mi hits the top */
    if(mi == li) li++;
}

/* for i */
printf("\n");fflush(stdout);

return(hamiltonian);
}

/* check the wavefunction for g or u symmetry */
int determine_rotational_eigensymmetry(molecule_t *molecule, int level, int l_max) {

int symmetry;

```

```

double theta, phi;
int l, m, index;
complex_t wavefunction[QUANTUM_ROTATION_SYMMETRY_POINTS];
double sqmod[QUANTUM_ROTATION_SYMMETRY_POINTS];
double max_sqmod, max_theta, max_phi;
complex_t max_wavefunction, inv_wavefunction;

/* scan a few random points, pick the one with the largest square of the real part */
for(i = 0, max_sqmod = 0; i < QUANTUM_ROTATION_SYMMETRY_POINTS; i++) {

    /* get random theta and phi */
    theta = get_rand()*M_PI;
    phi = get_rand()*2.0*M_PI;

    /* project the eigenvector into the basis */
    wavefunction[i].real = 0;
    for(l = 0, index = 0; l <= l_max; l++)
        for(m = -l; m <= l; m++, index++)
            wavefunction[i].real += molecule->quantum_rotational_eigenvectors[level][index].real*rotational_basis(REAL,
l, m, theta, phi);

    /* get the square modulus */
    sqmod[i] = wavefunction[i].real*wavefunction[i].real;

    /* if we have a new max, save the domain point */
    if(sqmod[i] > max_sqmod) {
        max_sqmod = sqmod[i];
        max_theta = theta;
        max_phi = phi;
    }
}

/* check the symmetry at the maximum */
max_wavefunction.real = 0;
inv_wavefunction.real = 0;
for(l = 0, index = 0; l <= l_max; l++) {
    for(m = -l; m <= l; m++, index++) {

        /* get the value of the wavefunction at the maximum */
        max_wavefunction.real += molecule->quantum_rotational_eigenvectors[level][index].real*rotational_basis(REAL, l, m,
max_theta, max_phi);

        /* get the value of it's inversion */
        inv_wavefunction.real += molecule->quantum_rotational_eigenvectors[level][index].real*rotational_basis(REAL, l, m,
(max_theta + M_PI), (max_phi + M_PI));
    }
}

/* did we change sign? */
if(max_wavefunction.real*inv_wavefunction.real < 0.0)
    symmetry = QUANTUM_ROTATION_ANTISYMMETRIC;
else
    symmetry = QUANTUM_ROTATION_SYMMETRIC;

return(symmetry);
}

/* get the rotational energy levels of a single rotor in an external potential */
void quantum_rotational_energies(system_t *system, molecule_t *molecule, int level_max, int l_max) {

complex_t **hamiltonian;
int i, j, dim;
/* variables needed for zhpevx() */
char jobz, range, uplo;
int vl, vu, il, iu, m, ldz, *iwork, *ifail, info;
double *hamiltonian_packed, *eigenvalues, abstol, *z, *work, *rwork;

/* determine the size of the symmetric matrix */
dim = (l_max + 1)*(l_max + 1);

/* setup the lapack arguments */
hamiltonian_packed = calloc(dim*(dim + 1)/2, sizeof(complex_t));
eigenvalues = calloc(dim, sizeof(double));
z = calloc(dim*dim, sizeof(complex_t));
work = calloc((2*dim), sizeof(complex_t));
rwork = calloc((7*dim), sizeof(double));
iwork = calloc((5*dim), sizeof(int));
ifail = calloc(dim, sizeof(int));

jobz = 'V'; range = 'A'; uplo = 'U';
vl = 0; vu = 0; il = 0; iu = 0;
abstol = 0;
m = 0; ldz = dim; info = 0;

/* build the rotational hamiltonian */
hamiltonian = rotational_hamiltonian(system, molecule, level_max, l_max);

/* pack the lapack array */
for(j = 0, p = 0; j < dim; j++) {
    for(i = 0; i <= j ; i++, p += 2) {
        hamiltonian_packed[p+0] = hamiltonian[i][j].real;
        hamiltonian_packed[p+1] = hamiltonian[i][j].imaginary;
    }
}

/* diagonalize the hamiltonian */
zhpevx_(&jobz, &range, &uplo, &dim, hamiltonian_packed, &vl, &vu, &il, &iu, &abstol, &m, eigenvalues, z, &ldz, work, rwork, iwork,
ifail, &info);

/* store the energy levels */
for(i = 0; i < level_max; i++) molecule->quantum_rotational_energies[i] = eigenvalues[i];
}

```

```

/* store the eigenvectors */
for(i = 0, p = 0; i < level_max; i++) {
    for(j = 0; j < dim; j++, p += 2) {
        molecule->quantum_rotational_eigenvectors[i][j].real = z[p];
        molecule->quantum_rotational_eigenvectors[i][j].imaginary = z[p+1];
    }
}

/* get the symmetry of each eigenvector */
for(i = 0; i < level_max; i++)
    molecule->quantum_rotational_eigensymmetry[i] = determine_rotational_eigensymmetry(molecule, i, l_max);

/* free our arrays */
for(i = 0; i < dim; i++) free(hamiltonian[i]);
free(hamiltonian);
free(hamiltonian_packed);
free(eigenvalues);
free(z);
free(work);
free(rwork);
free(iwork);
free(ifail);
}

/* generate the potential over the quadrature grid */
/* Gauss-Legendre abscissas+weights courtesy of Abramowitz & Stegun, "Handbook of Mathematical Functions", 9th Ed., p.916 */
void quantum_rotational_grid(system_t *system, molecule_t *molecule) {

    int t, p;
    double theta, phi;
    double potential;
    /* N = 16 */
    double roots[QUANTUM_ROTATION_GRID] = { -0.989400934991649932596,
                                              -0.944575023073232576078,
                                              -0.865631202387831743380,
                                              -0.755404408355003033895,
                                              -0.617876244402643748447,
                                              -0.458016777657227386342,
                                              -0.281603550779258913230,
                                              -0.095012509837637440185,
                                              0.095012509837637440185,
                                              0.281603550779258913230,
                                              0.458016777657227386342,
                                              0.617876244402643748447,
                                              0.755404408355003033895,
                                              0.865631202387831743380,
                                              0.944575023073232576078,
                                              0.989400934991649932596 };

    #ifdef XXX
    /* N = 8 */
    double roots[QUANTUM_ROTATION_GRID] = { -0.960289856497536, -0.796666477413627, -0.525532409916329, -0.183434642495650,
                                              0.183434642495650, 0.525532409916329, 0.796666477413627, 0.960289856497536 };
    /* N = 32 */
    double roots[QUANTUM_ROTATION_GRID] = { -0.997263861849481563545, -0.985611511545268335400, -0.964762255587506430774,
                                              -0.934906075937739689171, -0.896321155766052123965, -0.849367613732569970134, -0.794483795967942406963,
                                              -0.732182118740289680387, -0.663044266930215200975, -0.587715757240762329041, -0.506899908932229390024,
                                              -0.421351276130635345364, -0.331868602282127649780, -0.239287362252137074545, -0.144471961582796493485,
                                              -0.048307665687738316235, -0.048307665687738316235, 0.144471961582796493485, 0.239287362252137074545,
                                              0.331868602282127649780, 0.421351276130635345364, 0.506899908932229390024, 0.587715757240762329041,
                                              0.663044266930215200975, 0.732182118740289680387, 0.794483795967942406963, 0.849367613732569970134,
                                              0.896321155766052123965, 0.934906075937739689171, 0.964762255587506430774, 0.985611511545268335400,
                                              0.997263861849481563545 };
    #endif /* XXX */

    /* get the potential on an angular grid */
    for(p = 0; p < QUANTUM_ROTATION_GRID; p++) {
        phi = M_PI*roots[p] + M_PI;
        for(t = 0; t < QUANTUM_ROTATION_GRID; t++) {
            theta = 0.5*M_PI*roots[t] + 0.5*M_PI;
            molecule->quantum_rotational_potential_grid[t][p] = sin(theta)*rotational_potential(system, molecule, theta, phi);
        }
    }
}

/* find the rotational 1-body energies for each molecule in the system */
void quantum_system_rotational_energies(system_t *system) {

    int i, j;
    molecule_t *molecule_ptr;

    /* get the rotational eigenspectrum for each moveable molecule */
    for(molecule_ptr = system->molecules; molecule_ptr; molecule_ptr = molecule_ptr->next) {
        if(!(molecule_ptr->frozen || molecule_ptr->adiabatic)) {
            /* generate the necessary grids */
            quantum_rotational_grid(system, molecule_ptr);

            /* get the energy levels */
            quantum_rotational_energies(system, molecule_ptr, system->quantum_rotation_level_max,

```

```

system->quantum_rotation_l_max);
}

}

for(molecule_ptr = system->molecules; molecule_ptr; molecule_ptr = molecule_ptr->next) {
    if(!(molecule_ptr->frozen || molecule_ptr->adiabatic)) {
        for(i = 0; i < system->quantum_rotation_level_max; i++) {
            printf("molecule #%d (%s) rotational level %d = %f K (%f cm^-1 or %f / B) ", molecule_ptr->id,
molecule_ptr->moleculename, i, molecule_ptr->quantum_rotational_energies[i], molecule_ptr->quantum_rotational_energies[i]*KB/(100.0*H*C),
molecule_ptr->quantum_rotational_energies[i]/system->quantum_rotation_B);

            if(molecule_ptr->quantum_rotational_eigensymmetry[i] == QUANTUM_ROTATION_SYMMETRIC)
                printf("*** symmetric ***");
            else if(molecule_ptr->quantum_rotational_eigensymmetry[i] == QUANTUM_ROTATION_ANTISYMMETRIC)
                printf("*** antisymmetric ***");

            printf("\n");
        }

        for(i = 0; i < system->quantum_rotation_level_max; i++) {
            printf("molecule #%d (%s) eigenvec rot level %d\n", molecule_ptr->id, molecule_ptr->moleculename, i);
            for(j = 0; j < (system->quantum_rotation_l_max+1)*(system->quantum_rotation_l_max+1); j++)
                printf("tj=%d %.16e %.16e\n", j, molecule_ptr->quantum_rotational_eigenvalues[i][j].real,
molecule_ptr->quantum_rotational_eigenvalues[i][j].imaginary);
            printf("\n");
        }
    }
}

/*
@2007, Jonathan Belof
Space Research Group
Department of Chemistry
University of South Florida
*/
#include <mc.h>

/* Gauss-Legendre integrator */
/* abscissas+weights courtesy of Abramowitz & Stegun, "Handbook of Mathematical Functions", 9th Ed., p.916 */
double rotational_integrate(system_t *system, molecule_t *molecule, int type, int li, int mi, int lj, int mj) {

    int t, p;
    double theta, phi, theta_weight, phi_weight;
    double Yi_real, Yj_real, Yi_img, Yj_img;
    double potential, integrand, integral;
    /* N = 16 */
    double roots[QUANTUM_ROTATION_GRID] = { -0.989400934991649932596,
                                           -0.944575023073232576078,
                                           -0.865631202387831743880,
                                           -0.755404408355003033895,
                                           -0.617876244402643748447,
                                           -0.458016777657227386342,
                                           -0.28160350779258913230,
                                           -0.095012509837637440185,
                                           0.095012509837637440185,
                                           0.28160350779258913230,
                                           0.458016777657227386342,
                                           0.617876244402643748447,
                                           0.755404408355003033895,
                                           0.865631202387831743880,
                                           0.944575023073232576078,
                                           0.989400934991649932596 };
    double weights[QUANTUM_ROTATION_GRID] = { 0.027152459411754094852,
                                              0.062253523938647892863,
                                              0.095158511682492784810,
                                              0.124628971255533872052,
                                              0.149595988816576732081,
                                              0.169156519395002538189,
                                              0.18260341504492358867,
                                              0.189450610455068496285,
                                              0.189450610455068496285,
                                              0.18260341504492358867,
                                              0.169156519395002538189,
                                              0.149595988816576732081,
                                              0.124628971255533872052,
                                              0.095158511682492784810,
                                              0.062253523938647892863,
                                              0.027152459411754094852 };

    #ifdef XXX
    /* N = 8 */
    double roots[QUANTUM_ROTATION_GRID] = { -0.960289856497536, -0.796666477413627, -0.525532409916329, -0.183434642495650,
                                             0.183434642495650, 0.525532409916329, 0.796666477413627, 0.960289856497536 };
    double weights[QUANTUM_ROTATION_GRID] = { 0.101228536290376, 0.222381034453374, 0.313706454877887, 0.362683783378362,
                                              0.362683783378362, 0.313706454877887, 0.222381034453374, 0.101228536290376, };

    /* N = 32 */
    double roots[QUANTUM_ROTATION_GRID] = { -0.997263861849481563545, -0.985611511545268335400, -0.964762255587506430774,
                                             -0.934906075937739689171, -0.896321155766052123965, -0.849367613732569970134, -0.794483795967942406963,

```

```

-0.732182118740289680387,
-0.663044266930215200975, -0.587715757240762329041, -0.50689908932229390024,
-0.421351276130635345364,
-0.331868602282127649780, -0.239287362252137074545, -0.144471961582796493485,
-0.048307665687738316235,
0.048307665687738316235, 0.144471961582796493485, 0.239287362252137074545,
0.331868602282127649780,
0.421351276130635345364, 0.50689908932229390024, 0.587715757240762329041,
0.663044266930215200975,
0.732182118740289680387, 0.794483795967942406963, 0.849367613732569970134,
0.896321155766052123965,
0.934906075937739689171, 0.964762255587506430774, 0.985611511545268335400,
0.997263861849481563545 };

double weights[QUANTUM_ROTATION_GRID] = { 0.007018610009470096600, 0.016274394730905670605, 0.025392065309262059456,
0.034273862913021433103,
0.042835898022226680657, 0.050998059262376176196, 0.058684093478535547145,
0.065822222776361846838,
0.072345794108848506225, 0.078193895787070306472, 0.083311924226946755222,
0.087652093004403811143,
0.091173878695763884713, 0.093844399080804565639, 0.095638720079274859419,
0.096540088514727800567,
0.096540088514727800567, 0.095638720079274859419, 0.093844399080804565639,
0.091173878695763884713,
0.087652093004403811143, 0.083311924226946755222, 0.078193895787070306472,
0.072345794108848506225,
0.065822222776361846838, 0.058684093478535547145, 0.050998059262376176196,
0.042835898022226680657,
0.034273862913021433103, 0.025392065309262059456, 0.016274394730905670605,
0.007018610009470096600 };

#endif /* XXX */

integral = 0;
for(p = 0; p < QUANTUM_ROTATION_GRID; p++) {
    phi = M_PI*roots[p] + M_PI;
    phi_weight = weights[p];

    for(t = 0; t < QUANTUM_ROTATION_GRID; t++) {
        theta = 0.5*M_PI*roots[t] + 0.5*M_PI;
        theta_weight = weights[t];

        /* retrieve the basis set */
        Yi_real = rotational_basis(REAL, li, mi, theta, phi);
        Yj_real = rotational_basis(REAL, lj, mj, theta, phi);
        Yi_img = rotational_basis(IMAGINARY, li, mi, theta, phi);
        Yj_img = rotational_basis(IMAGINARY, lj, mj, theta, phi);

        /* form the square modulus for integration */
        if(type == REAL)
            integrand = (Yi_real*Yj_real + Yi_img*Yj_img);
        else if(type == IMAGINARY)
            integrand = (Yi_real*Yj_img - Yi_img*Yj_real);

        /* potential part of the integrand */
        potential = molecule->quantum_rotational_potential_grid[t][p];
        integrand *= potential;

        integral += phi_weight*theta_weight*integrand;
    } /* for t */
} /* for p */
integral *= 0.5*M_PI*M_PI;
return(integral);
}

/*
@2007, Jonathan Belof
Space Research Group
Department of Chemistry
University of South Florida
*/
#include <mc.h>

/* align the molecule with the x-axis for initialization */
void align_molecule(molecule_t *molecule) {

    atom_t *atom_ptr;
    double rotation_matrix[3][3];
    double com[3];
    int i, ii, n;
    double *new_coord_array;
    double theta, phi;

    /* count the number of atoms in the molecule, and allocate new coords array */
    for(atom_ptr = molecule->atoms; n = 0; atom_ptr = atom_ptr->next) ++n;
    new_coord_array = calloc(n*3, sizeof(double));

    /* save the com coordinate */
    com[0] = molecule->com[0];
    com[1] = molecule->com[1];
    com[2] = molecule->com[2];

    /* translate the molecule to the origin */
    for(atom_ptr = molecule->atoms; atom_ptr; atom_ptr = atom_ptr->next) {
        atom_ptr->pos[0] -= com[0];
        atom_ptr->pos[1] -= com[1];
        atom_ptr->pos[2] -= com[2];
    }
}

```

```

}

/* make sure that our atom chosen for finding the polar coords doesn't happen to be at the origin */
for(atom_ptr = molecule->atoms; atom_ptr; atom_ptr = atom_ptr->next)
    if(!(atom_ptr->pos[0] == 0.0) && (atom_ptr->pos[1] == 0.0) && (atom_ptr->pos[2] == 0.0))) break;

/* get the current polar coordinates */
if(atom_ptr->pos[2] == 0.0)
    theta = 0.0;
else
    theta = acos(atom_ptr->pos[2]/sqrt(atom_ptr->pos[0]*atom_ptr->pos[0] + atom_ptr->pos[1]*atom_ptr->pos[1] +
atom_ptr->pos[2]*atom_ptr->pos[2]));
if((atom_ptr->pos[1]/atom_ptr->pos[0]) > MAXDOUBLE)
    phi = atan(MAXDOUBLE);
else {
    phi = atan(fabs(atom_ptr->pos[1]/atom_ptr->pos[0]));
    if((atom_ptr->pos[1]/atom_ptr->pos[0]) < 0.0) phi *= -1.0;
}

/* form the inverse rotation matrix and multiply */
theta *= -1.0; phi *= -1.0;
rotation_matrix[0][0] = cos(phi)*cos(theta);
rotation_matrix[0][1] = -sin(phi);
rotation_matrix[0][2] = cos(phi)*(-sin(theta));
rotation_matrix[1][0] = sin(phi)*cos(theta);
rotation_matrix[1][1] = cos(phi);
rotation_matrix[1][2] = sin(phi)*(-sin(theta));
rotation_matrix[2][0] = sin(theta);
rotation_matrix[2][1] = 0;
rotation_matrix[2][2] = cos(theta);
for(atom_ptr = molecule->atoms, i = 0; atom_ptr; atom_ptr = atom_ptr->next, i++) {

    ii = i*3;
    new_coord_array[ii+0] = rotation_matrix[0][0]*atom_ptr->pos[0] + rotation_matrix[0][1]*atom_ptr->pos[1] +
rotation_matrix[0][2]*atom_ptr->pos[2];
    new_coord_array[ii+1] = rotation_matrix[1][0]*atom_ptr->pos[0] + rotation_matrix[1][1]*atom_ptr->pos[1] +
rotation_matrix[1][2]*atom_ptr->pos[2];
    new_coord_array[ii+2] = rotation_matrix[2][0]*atom_ptr->pos[0] + rotation_matrix[2][1]*atom_ptr->pos[1] +
rotation_matrix[2][2]*atom_ptr->pos[2];
}

/* set the new coordinates and then translate back from the origin */
for(atom_ptr = molecule->atoms, i = 0; atom_ptr; atom_ptr = atom_ptr->next, i++) {

    ii = i*3;
    atom_ptr->pos[0] = new_coord_array[ii+0];
    atom_ptr->pos[1] = new_coord_array[ii+1];
    atom_ptr->pos[2] = new_coord_array[ii+2];

    atom_ptr->pos[0] += com[0];
    atom_ptr->pos[1] += com[1];
    atom_ptr->pos[2] += com[2];
}

/* free our temporary array */
free(new_coord_array);
}

/* rotate the selected molecule to an absolute spherical polar position of (theta, phi) */
void rotate_spherical(molecule_t *molecule, double theta, double phi) {

    atom_t *atom_ptr;
    double rotation_matrix[3][3];
    double com[3];
    int i, ii, n;
    double *new_coord_array;

    /* count the number of atoms in the molecule, and allocate new coords array */
    for(atom_ptr = molecule->atoms, n = 0; atom_ptr; atom_ptr = atom_ptr->next) ++n;
    new_coord_array = calloc(n*3, sizeof(double));

    /* save the com coordinate */
    com[0] = molecule->com[0];
    com[1] = molecule->com[1];
    com[2] = molecule->com[2];

    /* translate the molecule to the origin */
    for(atom_ptr = molecule->atoms; atom_ptr; atom_ptr = atom_ptr->next) {
        atom_ptr->pos[0] -= com[0];
        atom_ptr->pos[1] -= com[1];
        atom_ptr->pos[2] -= com[2];
    }

    /* form the rotation matrix and multiply */
    rotation_matrix[0][0] = cos(phi)*cos(theta);
    rotation_matrix[0][1] = -sin(phi);
    rotation_matrix[0][2] = cos(phi)*(-sin(theta));
    rotation_matrix[1][0] = sin(phi)*cos(theta);
    rotation_matrix[1][1] = cos(phi);
    rotation_matrix[1][2] = sin(phi)*(-sin(theta));
    rotation_matrix[2][0] = sin(theta);
    rotation_matrix[2][1] = 0;
    rotation_matrix[2][2] = cos(theta);
    for(atom_ptr = molecule->atoms, i = 0; atom_ptr; atom_ptr = atom_ptr->next, i++) {

        ii = i*3;
        new_coord_array[ii+0] = rotation_matrix[0][0]*atom_ptr->pos[0] + rotation_matrix[0][1]*atom_ptr->pos[1] +
rotation_matrix[0][2]*atom_ptr->pos[2];
        new_coord_array[ii+1] = rotation_matrix[1][0]*atom_ptr->pos[0] + rotation_matrix[1][1]*atom_ptr->pos[1] +
rotation_matrix[1][2]*atom_ptr->pos[2];
        new_coord_array[ii+2] = rotation_matrix[2][0]*atom_ptr->pos[0] + rotation_matrix[2][1]*atom_ptr->pos[1] +

```

```

rotation_matrix[2][2]*atom_ptr->pos[2];
}

/* set the new coordinates and then translate back from the origin */
for(atom_ptr = molecule->atoms, i = 0; atom_ptr; atom_ptr = atom_ptr->next, i++) {
    ii = i*3;
    atom_ptr->pos[0] = new_coord_array[ii+0];
    atom_ptr->pos[1] = new_coord_array[ii+1];
    atom_ptr->pos[2] = new_coord_array[ii+2];
    atom_ptr->pos[0] += com[0];
    atom_ptr->pos[1] += com[1];
    atom_ptr->pos[2] += com[2];
}
/* free our temporary array */
free(new_coord_array);

}

/* hindered potential of sin(theta)^2 */
/* used for testing, compared with Curl, Hopkins, Pitzer, JCP (1968) 48:4064 */
double hindered_potential(double theta) {
    return(pow(sin(theta), 2.0));
}

/* put the molecule in a (theta,phi) orientation and return the energy */
double rotational_potential(system_t *system, molecule_t *molecule, double theta, double phi) {
    int q;
    molecule_t *molecule_backup;
    atom_t *atom_backup, *atom_ptr;
    double potential;

    if(system->quantum_rotation_hindered) {
        /* use a hindered rotor potential, useful for testing accuracy */
        potential = system->quantum_rotation_B*system->quantum_rotation_hindered_barrier*hindered_potential(theta);
    } else {
        /* backup the molecular coordinates */
        molecule_backup = copy_molecule(system, molecule);

        /* align the molecule with the x-axis initially */
        align_molecule(molecule);

        /* perform the rotation */
        rotate_spherical(molecule, theta, phi);

        /* get the energy for this configuration */
        potential = energy_no_observables(system);

        /* restore the original atomic coordinates */
        for(atom_backup = molecule_backup->atoms, atom_ptr = molecule->atoms; atom_backup; atom_backup = atom_backup->next, atom_ptr
= atom_ptr->next)
            for(q = 0; q < 3; q++)
                atom_ptr->pos[q] = atom_backup->pos[q];

        /* restore the c.o.m. coordinates */
        for(q = 0; q < 3; q++) molecule->com[q] = molecule_backup->com[q];
    }

    /* free the backup structure */
    free_molecule(system, molecule_backup);
}

return(potential);
}

```

About the Author

Jon Belof received his B.A. in Biochemistry, with a minor in Mathematics, from the University of South Florida and began doctoral work in Physical Chemistry with Professor Brian Space in 2005. His doctoral work having been funded by DOE, NSF and NASA, Jon has authored numerous papers in the areas of nanomaterials, molecular theory and computation. His research is focused on the application of statistical physics, quantum mechanics and the theory & simulation of condensed matter toward interesting problems in materials and biophysics. He has extensive experience in developing supercomputing applications, having developed for the Cray XT4, BlueGene/L, SGI Altix and HPC clusters. Prior to pursuing his doctorate, Jon worked as an information security consultant for numerous Fortune 500 companies and government agencies including the NSA, the U.S. Army, FCC, HUD and DOJ.