

EXPLORING CRITICAL CONFORMATIONS:
STATE SEARCHING AND SAMPLING IN BOTH
GERMANIUM CHAINS AND ICE INTERFACES
{EARLY DRAFT}

By
GENTRY H. SMITH
B.S.
Southern Nazarene University
Bethany, OK, USA
2016

Submitted to the Faculty of the
Graduate College of
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
Master of Science
December 2018

EXPLORING CRITICAL CONFORMATIONS:
STATE SEARCHING AND SAMPLING IN BOTH
GERMANIUM CHAINS AND ICE INTERFACES
{EARLY DRAFT}

Thesis Approved:

Advisor Name

Thesis Advisor

Reader 1 Name

Reader 2 Name

Reader 3 Name

ACKNOWLEDGMENTS

To Oklahoma State University, for providing the environment in which I have been able to study, teach, and research.

To the HPCC and the individuals who manage it for providing a powerful cluster for computations and continuous support for technical issues.

To my advisor, who instructed and assisted me in research.

To my parents, by blood and marriage, who have always encouraged me toward higher goals.

To my wife, Miranda, who has supported me for over five years.

Acknowledgments reflect the views of the author and are not endorsed by committee members or Oklahoma State University.

Name: GENTRY H SMITH

Date of Degree: December 2018

Title of Study: EXPLORING CRITICAL CONFORMATIONS

Major Field: COMPUTATIONAL CHEMISTRY

Abstract: Molecular conformation plays a critical role in the properties of systems in either the condensed or vapor states. The ensemble of conformations dictates structural properties, energies, heat capacities, and other thermodynamic and dynamic quantities. Here, we explore the role of conformation in proton ordering and orientational defect formation in ice as well as strategies for exhaustive conformer searching for molecules using Group IV element backbones. In the ice systems, we show algorithmic strategies for seeking optimized proton disordered crystals that satisfy the Bernal-Fowler ice rules. In the Group IV molecule investigations, we develop an automated strategy for seeking the optimal low energy conformer and uncover previously unreported deficiencies in common computational software used in investigating Germanium complex energies.

TABLE OF CONTENTS

Chapter	Page
1 Introduction	1
1.1 Computational Chemistry: Chemistry on the Computer	1
1.2 Relevant Computational Methods	1
1.2.1 Quantum Mechanical Methods and Basis Sets	2
1.2.1.1 Basis Sets	2
1.2.1.2 <i>Ab Initio</i> Methods	4
1.2.1.3 Density Functional Theory Methods	4
1.2.1.4 Semi-Empirical Methods	5
1.2.2 Monte Carlo Molecular Modeling	6
1.3 Hardware	7
1.4 Software	7
1.4.1 Programs	7
1.4.2 Programming Languages	8
2 On Algorithms for Building and Sampling Disordered Crystal States	10
2.1 States and Properties of Ice	10
2.1.1 Bernal-Fowler Ice Rules	10
2.1.2 Forms of Ice	10
2.1.3 Ice I _h	11
2.1.4 Efforts to Generate Ice I _h	11
2.1.5 Comparison between Ice XI and Ice I _h	12
2.2 Method Design	12
2.2.1 Overview	12
2.2.2 Selection of Software Tools	13
2.2.3 Generation of Source Ice XI	13
2.2.4 Source Ingestion	13
2.2.5 Identifying Neighboring Molecules	13
2.2.6 Defining Tetrahedral Positions	14
2.2.7 Pseudorandom Rearrangement of Water Molecules	15
2.2.8 Detecting Hydrogen Bond Defects	15
2.2.9 Correcting Hydrogen Bond Defects	16
2.3 Results of Method	17
2.4 Comparison with Other Methods	18
2.4.1 benefits of own method over others	18
2.4.2 benefits of other methods over this	18
2.5 Comments on Limitations and Proposed Improvements	18

3 Germanium Compounds and QM Concerns	20
3.1 Modeling Germanium Compounds	20
3.1.1 Computational Complexity of Germanium Compounds	20
3.2 The Initial Problem: Germanium Study	20
3.2.1 Parameters of Work and Previous Collaborator's Results	21
3.2.2 Design and Approach to Solution	23
3.2.2.1 Design 1: Occam's Smallest Razor	23
3.2.2.2 Design 2: A Blunt Effort	24
3.2.2.3 Design 3: Death by 1.59 Million Cuts	28
3.2.3 Scale Reduction Efforts	30
3.2.4 Efforts at Simplification	31
3.3 Discovery of a Consistent Inconsistency	35
3.4 Final Thoughts	37
4 Sampling Conformation Landscapes by Rotatable Bond Degrees of Freedom	40
4.1 A Brief History on Conformation Landscapes	40
4.1.1 Levinthal's Paradox	40
4.1.2 Levinthal Golf Courses	40
4.2 Purpose of Project	41
4.3 Design of System	43
4.3.1 Variation of Theory and Basis Set Usage by System Size and largest atom type	43
4.3.2 Computational Optimization by Varying Resolution	44
4.3.3 Inherent Complications	44
4.4 Results	45
4.4.1 Difficulties and Anticipated Future Approaches	45
References	48
A Ice Ih to Ice XI Conversion	48
A.1 Code: Crystal Disorganizer Tool	48
B Germanium Landscape	60
B.1 Sample Gaussian 09 Germanium File	60
B.2 Building Group 4 Chains	63
B.3 Collecting and Comparing Torsional Data	65
C Conformation Landscapes	72
C.1 Code: hexagermane-transall.pdb	72
C.2 Code: ge4h.pdb	81
C.3 Progress on Torsion Minimizer System	82

LIST OF TABLES

LIST OF FIGURES

Figure	Page
2.1 "Before" image of Ice XI	17
2.2 "After" image of generated ice I_h	18
3.1 Fully trans configuration of pentagermanium-based compound.	21
3.2 Fully trans configuration of hexagermanium-based compound.	22
3.3 Sample Newman projection of cis-butane.	24
3.4 Visualization of a trans-cis-trans hexagermane structure.	31
3.5 Sample torsion plot at reduced energy scale.	33
3.6 Visualization of a multiple pure group IV torsions at various theories and basis sets	34
3.7 A curious seemingly-inverted torsion plot of butagermane.	36
3.8 Hartree Fock energy minimization of butagermane torsion run at vary- ing basis sets.	37
3.9 Minimization of butagermane torsion run at varying theories and the 6-31G(d) basis set.	38
3.10 B3LYP energy minimization of butasilane torsion run at 6-31G(d) basis set.	38
4.1 Example Levinthal Golf Course taken from?	41
4.2 Flow of method design for variable resolution conformation landscape search.	42
4.3 Example variable resolution chart (hand-drafted, CGI pending) . . .	43

4.4	Highlighted torsions of the hexagermane molecule by type of bond, where green, red, and blue represent Ge-Ge, Ge-phenyl, and Ge-isopropyl torsion centers, respectively.	45
4.5	Highest energy conformer of o-nitrophenol, ignoring any ring strain conformations. Noticably was unable to rotate and form the hydrogen bond between ortho nitrogen and hydrogen.	46
4.6	Lowest energy conformer of o-nitrophenol. Formed the expected hydrogen bond between the ortho nitrogen and hydrogen.	47

CHAPTER 1

Introduction

1.1 Computational Chemistry: Chemistry on the Computer

For nearly a century, computational methods have greatly assisted chemists in their efforts of research and discovery and have relatively recently become their own field of focus within chemistry. As a standalone field, computational chemistry uses computer simulations to solve chemical problems. These simulations typically rely on theoretical methods adapted to run highly efficiently on computers. While initial computational methods were designed to solve wave functions and atomic orbitals, the scope quickly expanded into multiple fields of chemistry as more methods were developed to confirm or predict properties of molecules and systems.

With the introduction of *ab initio* and density functional methods, computational methods began to stand as a distinct field within chemistry. This introduction serves to introduce necessary background information generally relevant to the methods developed and utilized in the following chapters.

1.2 Relevant Computational Methods

Modern computational methods take one of several approaches in computing a system. Usually limited by the scale of molecule and scope of system, multiple methods exist to analytically solve or closely approximate a system by way of solving or approximating the quantum mechanical wave function or through a similarly consistent method of generalization by way of statistical solutions. Many other methods exist, but are not directly relevant to this work.

The first hurdle in any computational system is the likely impossibility of analytically solving the problem. In a system with more than two particles, this multi-body problem usually cannot be solved analytically excepting the dihydrogen cation due to the exorbitant amount of information contained in solving the wave function. This is largely due to the complexity introduced into the wave function of multiple interacting species that often lead to prohibitively large or impossible equations to solve.

1.2.1 Quantum Mechanical Methods and Basis Sets

In computational chemistry, quantum mechanical methods generally refer to computational methods that attempt to solve or closely approximate the electronic Schrödinger equation given nuclei and electron position information to determine properties of the system like energies or electron densities. Because the Schrödinger equation is impossible to solve for many-body systems, different methods use different approximations to balance between accuracy of the approximation and efficiency of computation.

1.2.1.1 Basis Sets

While running calculations, both *Ab initio* and DFT methods require basis sets to represent the electronic wave function as a system of algebraic equations that can be efficiently calculated. While basis sets can be designed of atomic orbitals or plane waves, this work focus primarily on basis sets derived by atomic orbitals. The two most often used types of orbitals are Gaussian-type and Slater-type orbitals. Slater-type orbitals (STOs), named after the physicist John Slater who introduced them in 1930,⁷ function as a linear combination of atomic orbitals (LCAO) adopted as a molecular orbital. STOs notably exhibit similar properties to Schrödinger-based orbitals, excepting that STOs have no radial nodes.

Gaussian-type orbitals (GTOs), introduced by S. Francis Boys in 1950,⁷ also function as orbitals in the LCAO method. GTOs are similar to STOs in premise,

but have further reduced realism when compared to Schrdinger-based orbitals. One example of this is the lack of accuracy of electron density near the nucleus. While exhibiting a lesser accuracy, GTOs excel in computational efficiency compared to STOs by roughly 4-5 orders of magnitude. This allows GTO-based calculations to compute more orbitals. Specifically, Boys designed GTOs as a method of approximating STOs.

Basis sets are often grouped by their sizes. The smallest sets, known as minimal basis sets, use a single basis function for each orbital. The most common minimal basis set, STO-nG where n is an integer usually between 2 and 6, was first proposed by John Pople in 1969.⁷ This method describes that a Slater-type orbital can be approximated using n-Gaussian orbitals. These STO-nG approximations end up fitting electron densities well at all radial distances except those close to the nucleus. The STO-3G basis set used in this work is a popular basis set as the 3 Gaussian-type orbitals works well for atoms in the [H-Xe] range.

The other basis sets used in the work fall under the category of split-valence basis sets. These basis sets represent valence electrons with more than one basis function, which allows for electron density to be more flexible in different molecular systems. The most common form of these basis sets were introduced by John Pople as the X-YZg form and are commonly referred to as Pople basis sets.⁷ These follow the form that each orbital basis function is comprised of X Gaussians. The Y and Z represent the two additional linear combination of Gaussian functions made of Y and Z Gaussians each, respectively, that compose the valence. These basis sets are not limited to two valence functions, referred to as a double-zeta, and can also be triple- or quadruple-zeta. Additional values, typically denoted by one or two stars, one or two plus signs, or explicitly-defined orbital combinations in parentheses can also be used to further complicate the basis set as desired. The star notation defines a polarization function for heavy atoms to account for d and f polarizations. The plus signs denote diffuse functions that more-accurately represent less common valence electrons like

carbanions that may diffuse further out from the nucleus.

1.2.1.2 *Ab Initio* Methods

Ab initio, or "from first principles," methods refer to calculation methods that rely solely on physical constants as external values. By design, *ab initio* methods avoid using any empirically-acquired data and rely on theoretically calculated values. The development of these methods allowed computational chemists to solve a new class of problems and resulted in John Pople and Walter Kohn receiving the Nobel Prize for their work. The *ab initio* method utilized in this work is the Hartree-Fock (HF) method used to determine the energy of a many-body system in a stationary state, which is to say time-independent.[?] Known initially as the self-consistent field method, the HF method utilizes approximations defined by the basis set to approximate the Schrödinger equation. The consistency of this self-consistent field method arose by the requirement that the final calculated field be self-consistent with the initial field. An additional property of HF is that electron-electron repulsion is not taken into account, requiring that a basis set account for this interaction. As larger basis sets are used, the overall energy of the wavefunction is increased toward a value known as the Hartree-Fock limit. This limit is approached as the larger basis sets approach the exact solution of the non-relativistic Schrödinger equation without spin orbital terms. The calculation of relativistic and spin terms require a further method known as Post-Hartree-Fock, which is not used considered further in this work.

1.2.1.3 Density Functional Theory Methods

Density Function Theory (DFT) Methods function very similarly to *ab initio* methods in how Slater-type orbitals are used to approximate the Schrödinger equation, but differ in that DFT utilizes some empirical data to speed up the calculation process.[?] These simplifications are stable able to model exchange and correlation interactions

very well, however the reliability of calculated properties, specifically intermolecular interactions, dispersion forces, and other internal properties are greatly reduced. Just as with *ab initio* methods, DFT methods require a basis set definition for the approximation calculations. Many DFT methods exist and even some so-called hybrid functional methods that exchange with HF terms for greater reliability in calculated values. One pure DFT method used in this work is BLYP, which utilizes the Becke exchange with the Lee-Yang-Parr correlation part. Some hybrid functional methods used are the B3LYP, M06L, and PBE methods. The B3LYP utilizes the BLYP but combined Becke's exchange with the exact energy from HF theory. M06L, known as the Minnesota functionals, depend on kinetic energy density values from databases. It specifically was designed to work well with transition metals, inorganics, and organometallics.⁷ The PBE method is another method with similar levels of accuracy to B3LYP that attempts to increase the number of HF-exchanged functionals.

1.2.1.4 Semi-Empirical Methods

Like DFT, Semi-empirical methods also pull somewhat from Hartree-Fock methods, but rely even more on approximations and empirical data to nearly completely substitute out any proper calculation of the Schrödinger equation. These data can produce fairly accurate results to experimental data, but rely heavily on a similarity between the subject molecule and the database molecules. Due to its restrictive scope, semi-empirical methods excel in organic chemistry calculations where relatively few elements are used with moderately sized molecules.⁷ Additionally, various semi-empirical methods have been designed to produce results with close accuracies to specific properties of experimental data. Two methods used in this work, AM1⁷ and PM3,⁷ serve particularly well at calculating data to fit heats of formation, dipole moments, ionization potentials, and structural geometries. Unlike the other meth-

ods described so far, basis sets are not used at all in the calculation of energies and properties.

1.2.2 Monte Carlo Molecular Modeling

A major problem in sampling comes with the scale of many molecules of one or more type in a single or mixed phase. A popular method of working around this problem is known as Monte Carlo methods, or MC. While not named until the 1950s, MC methods were first seen in the 18th century thought experiment Buffon's needle.[?] In his work, Buffon proposed dropping n needles of length l onto a plane with parallel lines spaced t units apart. Buffon worked out that the probability, P , of a needle crossing one of the lines to be $P = \frac{2l}{t\pi}$. Solving for π , the probability can be rearranged as $\pi = \frac{2l}{tP}$ to approximate π . Since P can also be approximated by dividing the number of needles crossing one of the lines, h , by the n needles as $P = \frac{h}{n}$, the approximation can be expressed as $\pi = \frac{2l*n}{t*h}$.

This method of randomness was improved upon by Stanislaw Ulam while working at Los Alamos National Laboratory in the late 1940s by introducing markov chains to favor the probability of events occurring. Ulam shared this work with John von Neumann and together they created a program to run on the ENIAC computer capable of computing this favored version of random sampling. As the project was secretive due to being used as a part of the Manhattan Project, a collaborator named Nikolas Metropolis suggested the name Monte Carlo due to Ulam's uncle's propensity to gambling at a casino in Monaco of the same name.[?] Later dubbed Markov Chain Monte Carlo (MCMC) sampling, this allowed for random sampling to instead become a virtual statistically-appropriate sampling method. Eventually published in 1949 by Metropolis and Ulam, this laid the groundwork for modern MC methods used in modern chemical simulation packages.

1.3 Hardware

Since computation methods were developed slightly before and during the rise of modern computers, early calculations were performed by hand with minimal assistance by machines. Over time, these methods were increasingly assisted by early computers and further development eventually led to the first computational programs. These first computers, like the ENIAC and EDSAC offered computation power in the order of a few dozen to a few thousand operations per second.

For this work, the majority of calculations were computed on the Oklahoma State University Cowboy Cluster. Available since 2014, this cluster collectively offers the computing power of 3048 cores and 8576 GB of RAM, totalling 48.8 trillion FLoating Point Operations Per Second (Tera FLOPS or TFLOPS).

1.4 Software

If hardware denotes the realm of study of a computational chemist, software denotes the tools. By utilizing preexisting packages of and developing new and more advanced tools, computational chemists are able to simulate a wide variety of chemical systems.

1.4.1 Programs

While computational programs have existed for nearly 50 years, additional programs have relatively recently developed to aid in the visualization and depiction of chemical systems. Gaussian, developed by John Pople and his team is one of the earliest *ab initio* computation programs developed. Released as Gaussian 70 in 1970, it has received regular updates and capability expansions is still in use today and is one of the most widely-used computational chemistry tools available in its latest iteration, Gaussian 16. Gaussian tends to carry a lot of clout in the computational community as founder John Pople won a Nobel Prize in 1998 with Walter Kohn for his work in *ab initio* quantum mechanical systems and for being one of the oldest packages around.

In addition to Gaussian, many other packages exist with a large set of available tools for investigators. Two additional packages used in this work are GAMESS,⁷ a package also in active development since the 1970s led by Mark Gordon, and NWChem,⁷ a popular open source package developed by Pacific Northwest National Laboratory since the late 2000s.

Once the rounds computation has completed, investigators often report the calculated system graphically through visualization tools. These tools are also popular among any investigator wishing to represent a compound or system as more than its molecular formula. Two visualization tools used in this work are Avogadro and UCSF Chimera. Avogadro, in development since 2008, is a relatively simple molecular visualization tool designed to work across multiple operating systems across multiple languages and with many extensions.⁷ UCSF Chimera, developed by the Resource for Biocomputing, Visualization, and Informatics (RBVI) at the University of California, San Francisco, focuses on more advanced representations of compounds and systems. It allows for multi-structure files to generate videos of simulations and also provides a powerful Application Program Interface for programmatically creating or altering molecules and systems.

1.4.2 Programming Languages

A final note should be made about programming languages and their usage in general and in this work. Programming languages have existed for as long as computers. From original punch cards and bitwise commands to modern interpreted languages, programming languages allows investigators to control computers to enact explicit commands. In a way, the job command files in computational tools like those in Gaussian and GAMESS are programmatically used as a programming language to tell a system to enact a calculation of type X on system Y with Z parameters. Even these tools utilize code to enact their commands, usually in older and highly efficient

languages like C and Fortran. Because these tools directly interact with hardware to calculate an immense number of calculations, efficiency is key.

One language almost exclusively used in this work is Python.[?] The Python programming language has recently become one of the most used programming languages for scientific analysis. This is possibly due to Python’s initial development focus of data analysis, support for extensions by the development team, and ease of use. As a scripted type languages, Python does not directly interface with hardware in assembly like the more efficient C and Fortran languages, but certain packages and extensions can take advantage of those efficiency boosts to improve Python’s effectiveness. Math and science packages like NumPy[?] and SciPy[?] interface with C code to rapidly speed up complex mathematic evaluations like matrix manipulations while retaining the usability expected in Python. Additional packages like Cython[?] will take a completed Python script and compile much of it in C code to greatly improve efficiency and reduce the computational strain on the system.

As will be seen in this work, code can be used to generate and run these sets of code, effectively creating a train of code that can operate as a tool within a tool. One aspect of this is abstracting out method and basis sets to that of a computational requirement and level of accuracy. Effectively, an investigator could remove any necessary knowledge of which basis set or method is necessary for a specific system, although a tool with sufficient awareness to automatically enact this has not been published.

CHAPTER 2

On Algorithms for Building and Sampling Disordered Crystal States

2.1 States and Properties of Ice

Ice has many forms, each with unique environments and structures that give rise to similar and distinct properties.

2.1.1 Bernal-Fowler Ice Rules

Based on their paper published in 1933, the Bernal-Fowler Ice Rules are the foundational observations of how water molecules interact in an ice structure and have remained the standard for ice bonding rules for nearly a century. Essentially, water's tetrahedral structure allows for four interactions on each molecule. The two protons allow for a hydrogen bond with a lone pair from a neighboring oxygen atom. Similarly, the oxygen atom's two lone pairs allow for a hydrogen bond with a neighboring proton. These rules are fairly rigid in the sense that every water molecule can interact with two oxygen atoms and two protons from four surrounding water molecules. These are also relatively relaxed in the sense that each of the four attached water molecules can occupy one of three rotational positions, allowing for 81 possible microstates when including rotational duplicates.

2.1.2 Forms of Ice

While ubiquitous in the ' I_h ' form, ice water has many known forms. As of the writing of this work, there are 17 established forms of ice. These forms usually occur in cubic, hexagonal, and orthorhombic crystal structures. The relationship between external

pressure and temperature are the primary defining characteristics of which form will form in a given system, although doping may be required to produce some structures. The subject of this work will be on the proton-ordered orthorhombic ice XI and its proton-disordered isomer, ice I_h .

2.1.3 Ice I_h

Ice I_h naturally forms in pure solution at temperatures in the 72 K to 273.15 K range at pressures in the 1 Pa to 100 MPa range, with some temperature curving off into the vapour and liquid phases at very high and very low pressures. As the most commonly found form on earth, ice I_h is a highly desired form for computational studies involving ice systems for investigators investigating ice simulations in normal earth conditions.

As famously discussed by Linus Pauling, hexagonal ice water contains a residual entropy at very low temperatures.⁷ This residual entropy goes according to Boltzmann's entropy equation $K = \ln(W)$ where $W = (\frac{3}{2})^N$ for N molecules in the crystal. He additionally predicted that a ice may exist at sufficiently low temperatures with zero entropy.

2.1.4 Efforts to Generate Ice I_h

The most commonly cited effort to create an ice I_h crystal comes from⁷. Her work involved a monte carlo effort to generate ice I_h by initializing a deprotonated oxygen structure, randomly placing the appropriate number of protons, and then using MC methods to adjust the structure until every oxygen atom was bonded to exactly two hydrogen bonds.

2.1.5 Comparison between Ice XI and Ice I_h

While ice I_h is known as the most common form of ice found on the planet, it is much more difficult to computationally generate than an ice XI crystal. The ease of generation of an ice XI structure stems from the repetition of a unit cell with consistent layering and orientation throughout the crystal lattice.

With ice I_h crystals, the proton-disordered form introduces entropy by way of rotational disorder. The disordered protons allow for a greater number of microstates in the organization of the crystal, increasing the multiplicity and, by its very definition, entropy. As the protons and lone pairs are no longer consistently ordered, hydrogen bonds may no longer form properly at all interaction sites. The interaction of proton with proton or lone pair with lone pair are not hydrogen bonds and are considered defects in the lattice. An ice structure of randomly oriented molecules without consideration of hydrogen bonds will likely produce defects at many interaction sites across the lattice and weaken the integrity of the system, leading to stability problems while running simulations. In generating the crystal, the cause of these defects must be considered and countered effectively.

2.2 Method Design

2.2.1 Overview

The big idea is to convert an easy-to-make ice XI crystal into an ice I_h crystal. Because the key difference in structure is the proton-orderedness, it might be possible to rearrange the water molecule orientations in a pseudorandom way to create an ice I_h crystal. This section walks through the method developed to convert ice XI into ice I_h, the results of initial testing, and imperfections discovered in the design.

2.2.2 Selection of Software Tools

Python was chosen as the language of the tool due to the versatility of the language and the ease of development due to the "pseudocode" written style of the language and the availability of scientific packages including SciPy and NumPy. Python version 2.7 was specifically chosen due to familiarity with the language. Crystal files were defined and saved as Protein Data Bank (.pdb) files as this format allows for defining multiple molecules within a larger structure with a simple X, Y, Z grid position format.

2.2.3 Generation of Source Ice XI

This is Dr. Fennell's method to create an ice XI pdb file. Basically, the ice XI unit cell of eight water molecules is repeated as desired to create a sufficiently large crystal. The primarily used crystal consists of a 3 x 3 x 6 unit cell repetition totaling 432 water molecules.

2.2.4 Source Ingestion

It is important that the crystal be read and stored in an efficient method to keep relevant information about each molecule easily accessible. As the file is read in, each molecule is stored as an entry in a multidimensional array where the first index is the molecule number. Further, the second index defines the molecule number where 0 is oxygen and 1 and 2 are the protons. The third, fourth, and fifth indices define the X, Y, and Z position coordinates.

2.2.5 Identifying Neighboring Molecules

Identifying the neighboring molecules proved computationally difficult. The most effective method is to find the closest four molecules by computing a distance calculation between every two oxygen atoms. This ensures every molecule is considered, but

also presents significant hurdles. First, a distance calculation utilizes an extremely computationally-inefficient square root calculation, which can be ignored by instead calculating the squared-distance between molecules and finding the lowest values.

Second, molecules on the walls and edges of the molecule will not have four neighbors in the non-periodic crystal. This is accounted for by shifting all six sides to make a pseudo-periodicity for these edge cases. Those periodically-neighboring molecules are flagged with a shifting value in the neighboring atom array by specifying a translation in the x, y, or z axis values. Unfortunately, the necessary code to implement the periodically-neighboring molecule detections requires a major rewrite of the entire tool and has not yet been implemented.

Once these closest neighboring oxygen atoms have been discovered, the appropriate interacting tetrahedral position is identified by finding the closest of the four tetrahedral positions using the same squared-distance calculation with the four defined tetrahedral positions detailed in the next subsection.

2.2.6 Defining Tetrahedral Positions

An important aspect of pseudorandom selection is the existence of a bank of options. Utilizing the ingestion portion of the tool to calculate and store all tetrahedral possibilities proves effective for tracking position options. For each water molecule, the first two tetrahedral positions are known by the positions of the two hydrogen atoms. The other two positions are found by rotating one hydrogen atom 120° twice about the vector from the oxygen atom through the other hydrogen atom and storing the resulting positions as tetrahedral positions three and four.

This does not produce an exactly correct tetrahedral position of potential hydrogen atoms due to the slight acuteness of the H-O-H bond created by the variance in repulsive forces between the two lone pairs of electrons and two hydrogen atoms. Fortunately, this difference is sufficiently small for visualization programs like Avogadro

to still recognize hydrogen bonds between a rotated hydrogen atom and corresponding neighboring lone pair. Currently, the method does not correct for these minor angle variations and relies on the user to anneal the crystal by way of simulation to fully adjust the angles. Future versions of this method may account for the variations.

2.2.7 Pseudorandom Rearrangement of Water Molecules

Once the tetrahedral positions have been defined, each water molecule is ready to rotate. What may seem the most crucial step in this methods ends up being the most simple. As designed, the rotation of water molecules is as simple as using a stepwise iterator to pseudorandomly select two tetrahedral positions for the hydrogen bonds and store the new positions in a new crystal array. An extremely important note is that this rearrangement does not consider the orientations of neighboring molecules and likely introduces defects of hydrogen - hydrogen and lone pair - lone pair interactions. The likelihood of a defect-free interaction lattice forming is nearly zero and is assumed to have a great deal of defects within the lattice.

2.2.8 Detecting Hydrogen Bond Defects

After all water molecules have been rearranged, defects between incorrectly-interacting hydrogen bonds must be found and corrected. Discovering the defects relies on the detection of neighboring molecules and the appropriate interacting hydrogen atom or electron lone pair. As previously discussed, the initial data ingest records and detects the nearest water molecules and determines the tetrahedral position containing the interacting space, be it electron lone pair or hydrogen atom. From that data, the detection of a valid hydrogen bond is as simple as checking both all interacting tetrahedral positions and confirming that they both do not contain or lack a hydrogen atom. Additionally, each water molecule keeps a count of how many defects are present among the four positions. This allows for contextual changes during the

correction step.

2.2.9 Correcting Hydrogen Bond Defects

Once the hydrogen bond defects have been discovered and marked, each needs to be corrected. The most direct approach to this is to sequentially walk through each defect and repeat the pseudorandom rotation until the number of defective regions is zero or a user-specified value. The current implementation sorts the defect list by the number of defects and attempts to fix the most defective molecules first. The most defective molecules may include defects impossible to solve by simple rotation, specifically when neighboring molecules have collectively directed three or four hydrogen atoms or electron lone pairs at the target water. These can only be solved by adjusting one or more of the neighboring molecules until the number of hydrogen atoms and electron lone pairs have balanced. Unfortunately, this high-defect problem can quickly escalate if the neighboring molecules contain the same problem of unbalanced hydrogen atoms and electron lone pairs. The current solution is to recursively check for and fix these impossible interactions first, but has not yet yielded a defect-free crystal in testing.

The current design of the method allows for the user to specify a threshold of defects as an average per molecule. For example, a threshold of 2.5 will allow a maximum of 3 defects on any given molecule and will continue to correct defects until the average number of defects per molecule is equal to or below 2.5. Because each of these defects will be counted twice, once for each molecule, the total number of defects in a crystal can be determined by multiplying the average defect value by the number of molecules and dividing by two. As of the current implementation, the method cannot reliably produce a crystal with a threshold below 2 as it will continue to recursively search until the system runs out of available memory and crashes without finalizing the structure.

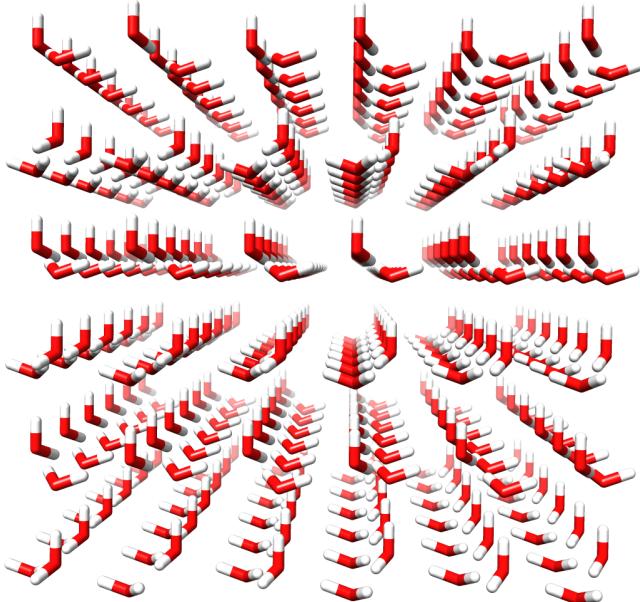


Figure 2.1: "Before" image of Ice XI

2.3 Results of Method

When supplied with an input ice XI crystal, an output structure with rotated water molecule orientations strictly consistent with ice I_h describes a success at the most basic level. An example before and after of the method is given in figures 2.1 and 2.2. As can be seen, the "after" image has experienced rotation and can no longer be classified as ice XI. However, as ice I_h also has a standard shape, the generated crystal can not be considered ice I_h . Instead, it can be considered a proton-disordered orthorhombic ice crystal similar to ice I_h .

Unfortunately, the result is not without defect. When following the subsequent layers in the crystal, patterns emerge. Inconsistently, some rows of waters remain consistent. Some of these are a uniform rotation of both hydrogen atoms, while others are just one consistently placed hydrogen atom. Multiple trials yield internally unique results, yet all contain these strange consistencies. This may be due to some accidental pattern in the method's implementation.

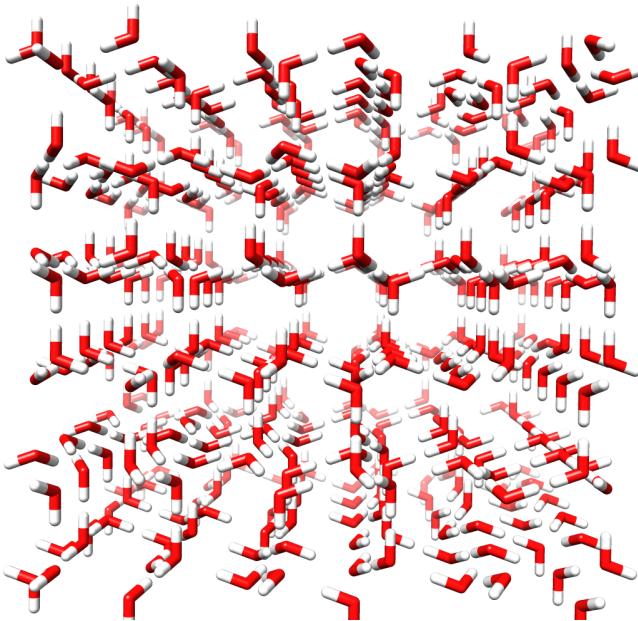


Figure 2.2: "After" image of generated ice I_h

2.4 Comparison with Other Methods

Currently in progress, this section will primarily compare the results of this method with[?].

2.4.1 benefits of own method over others

2.4.2 benefits of other methods over this

2.5 Comments on Limitations and Proposed Improvements

During the hydrogen bond defect correction step, a weakness in the design is that any clustering or regions of high defect density will not be noticed. This allows the existence of a highly-defective region within the larger structure that could potentially cause problems when the crystal is used in simulations. The prevalence and occurrence of these defects have not been studied in this work, but seem a natural inevitability of statistics. A potential solution with partial development will score regions based on the number of defects as a weighted function expanding out from a central molecule for N connections. For example, consider a given molecule defined as level 1. The

neighboring four molecules are defined as level 2, and continued onward excepting already-defined molecules out to an N^{th} level. The number of defects in each level can be counted and averaged. Then a depressive factor along the lines of $\frac{1}{level}$ can be used to diminish the value of defects further away from the first-level molecule. This would create a value for each molecule that shows the relative density of defects centered about that specific molecule and could even be plotted as a gradient change within the crystal. The general approach to a scoring mechanism may take a form similar to equation 2.1.

$$Value = \sum_{l=1}^{N_{levels}} \left[\frac{1}{l} * \frac{1}{N_{molecules}} * \sum_{m=1}^{N_{molecules}} [N_{defects,m}] \right] \quad (2.1)$$

CHAPTER 3

Germanium Compounds and QM Concerns

3.1 Modeling Germanium Compounds

While primarily used in optical applications including fiber optic cables and solar cell systems, germanium-based compounds are also used as polymerization catalysts. Relative to many other elements on the periodic table, computational reports on germanium are not common. Recent works have shown germanium's potential to polarize light.

3.1.1 Computational Complexity of Germanium Compounds

Publications on germanium computational efforts are not as common as many other main group elements. Of those extant publications, the majority of final published data involve a Density Functional Theory (DFT) with either the 6-31G(d), 6-31G(d,p), or 6-311G(2d) basis set.[?] As with most other lighter elements calculated with Pople basis sets, the 6-31G(d,p) basis set is most commonly used for the final energy calculation.

3.2 The Initial Problem: Germanium Study

During Fall 2017, Dr. Christopher Fennell was approached by Dr. Charles Weinert of OSU to continue a collaborative effort in sampling conformation energies of two germanium-based compounds of interest to Dr. Weinert's work. Seen as an opportunity to train a new graduate student in conformational calculations, this project was delegated to me. The initial focus was to create the two compounds in a 3D

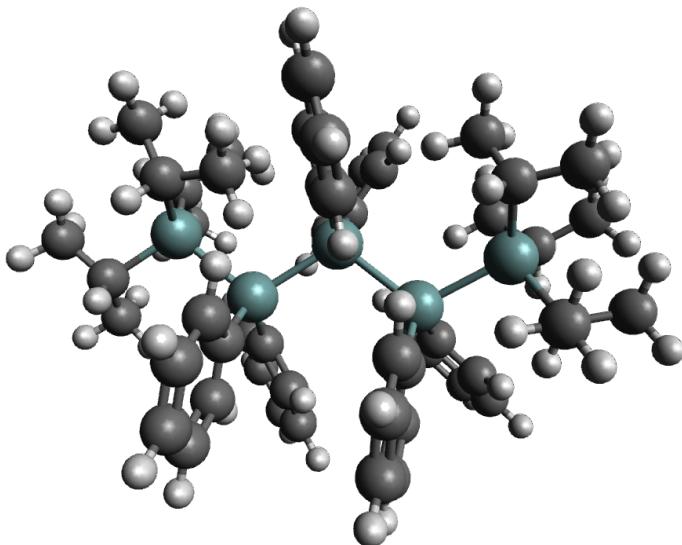


Figure 3.1: Fully trans configuration of pentagermanium-based compound.

modeling program, save a file of each, run a conformation optimization program on a supercomputer, and read the output to report the findings. As detailed below, this work led to impossibilities, curiosities, and inconsistencies that resulted in a general solution and a discovery of a flaw in a popular computational program.

3.2.1 Parameters of Work and Previous Collaborator's Results

The two subject germanium-based compounds are very similar: a germanium backbone with terminal isopropyl groups and internal phenyl rings. One compound constituted a pentagermanium chain while the other a hexagermanium backbone. The molecular formula for both is $Pr_3^iGe(GePh_2)_nGePr_3^i$ where n equals 3 for the pentagermanium or 4 for the hexagermanium compounds, respectively. An example image of both compounds in their fully-trans configurations are provided in figures 3.1 and 3.2.

Dr. Weinert had worked previously with a collaborator who provided conformation data supplied in table 3.1. While the basis set was not explicitly provided, it is likely that the most common 6-31G(d,p) basis set was used. Unfortunately, the collaborator is no longer active in research and was inaccessible for clarification.

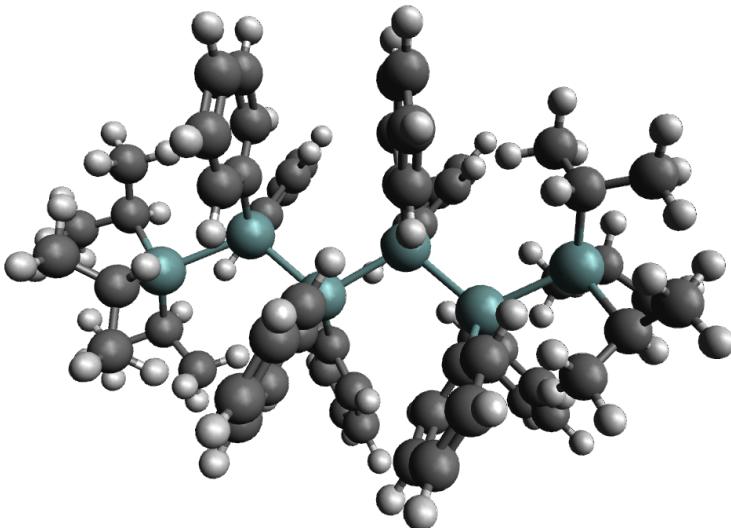


Figure 3.2: Fully trans configuration of hexagermanium-based compound.

Conformation	Energy (E_h)	Δ Energy (E_h)	Δ Energy ($\frac{KJ}{mol}$)
Trans-coplanar	-15014.8403143	0.0066255	17.39525025
Cis-Trans-Cis	-15014.7983311	0.0486087	127.6221418
Trans-Cis-Trans	-15014.8469398	0.0000000	0.0000000
Cis-Trans-Trans	-15014.8246918	0.0222480	58.412124

Table 3.1: Collaborator's Hexagermanium Energies by Conformation
(density functional theory, unknown basis set, energy in Hartrees and KJ/mol)

The approach of labeling the conformation shape of each compound, given the many points of torsion, focuses on the backbone structure. As the raw data from the collaborator was not available, the general dihedral angles of cis and trans proved a vexing focus for initial efforts at conformer design. Using Newman projections like in figure 3.3 as a visual guide, each Ge-Ge bond was defined as cis or trans based on the relative angle produced by the two adjacent bonded Ge atoms to each subject Ge. Specifically, the bonds are marked cis if the most acute angle is 90° or fewer, and likewise trans if greater than 90° up to the maximum 180° . Effectively the cis and trans angles coincide with gauche and anti in organic structure nomenclature. Terminal germanium atoms are not considered as a part of the conformation state. This is partly due to the definition in labeling where the terminal germanium does not have an adjacent germanium for the measured relative angle, in addition to the assumed C_3 symmetry of the terminal Ge with three isopropyl groups reducing the relative effects of terminal germanium rotation. Effectively, only dihedrals formed by four consecutive Ge are given a cis or trans label.

3.2.2 Design and Approach to Solution

The initial approach involved an attempt at basic replication of the collaborative results. As detailed below, the design gradually grew in complexity as a learning process. Eventually, curiosities in results and a desire to automate an objective search algorithm developed into two unique investigations.

3.2.2.1 Design 1: Occam's Smallest Razor

With each non-terminal Ge-Ge dihedral initially labeled cis or trans for 0° or 180° , about 3 unique pentagermanium and 6 unique hexagermanium structures were built visually on a 3D visualization program (Avogadro). These were rotated without consideration for the phenyl rings populating the non-terminal Ge atoms. Each molecule

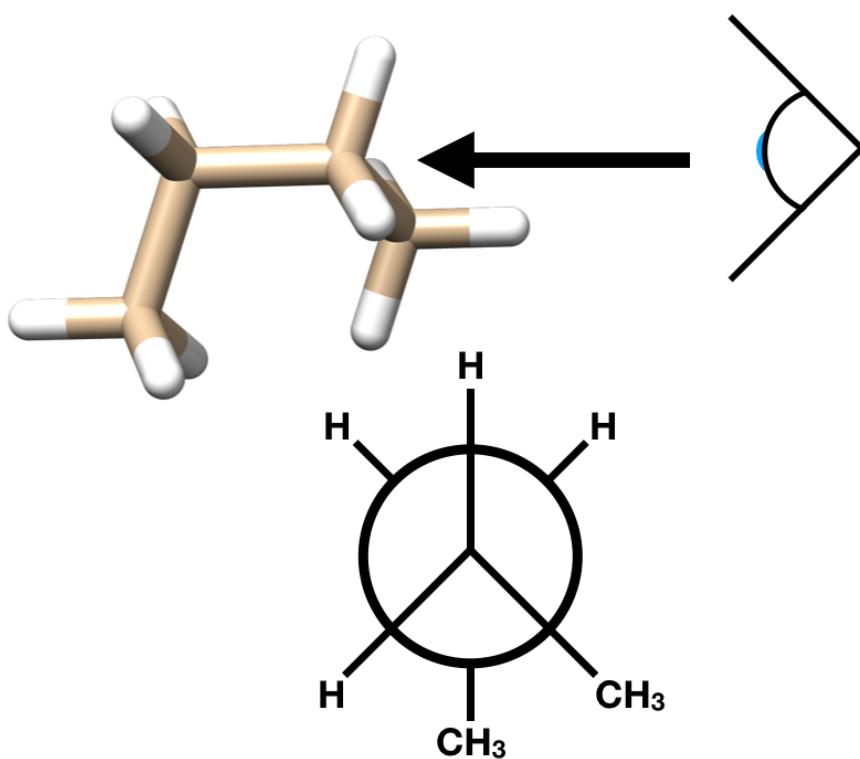


Figure 3.3: Sample Newman projection of cis-butane.

was subjected to an energy minimization in Gaussian 09 with the B3LYP hybrid function and STO-3G basis set as a single particle in a vacuum at otherwise default settings.

Unsurprisingly, only the fully trans conformers successfully converged (a 22% success rate) into a stable form. These troubles were likely caused by the poor design of the initial conformers. With initial results, the conformer design was altered into a more systematic approach with some consideration for the phenyl rings.

3.2.2.2 Design 2: A Blunt Effort

In the second iteration of the conformer design process, a greater number of backbone conformers were generated. Instead of the simple 180° opposition between the cis and trans conformers, more intentional initial angles seen in Newman projections were selected. Specifically, the anti and both gauche angles were chosen for the natural local minima in a non-bulky molecule, with both gauche angles (60 and 300) labeled

as cis and the anti angle (180) as trans. For initial conformer design, these backbone angles were limited to three positions: 60°, 180°, or 300°. For the hexagermanium compound, these structures were sequentially labeled trans-trans-trans, trans-trans-cis, trans-cis-trans, et cetera until all major unique conformers were produced. For clarity, each conformer was identified by the dihedral angles (60-60-60, 60-60-180) in increasing order (Ge 1-2-3-4, Ge 2-3-4-5, Ge 3-4-5-6 dihedral). The phenyl rings on the non-terminal Ge atoms were left untouched from an initial steepest-descent minimization available from Avogadro ran in the fully trans conformer.

To prevent potentially strong interactions between adjacent phenyl rings, an additional steepest-descent minimization from Avogadro was initially ran with the conformer-defining Ge-Ge dihedral angles locked in place. Additionally, a visual inspection of the phenyl rings and manual adjustments were utilized on Avogadro to reduce the chance of a relatively high energy local minima conformer. The phenyl rings usually were settled in a form of pi stacking or some kind of perpendicular ring interaction, based on relative energy stability according to the immediate simple minimization available.

To further avoid backbone rotation restrictions, variations of the bulky molecules were also produced. These included versions where the phenyl rings were replaced by methyl groups and also where the isopropyl ends were additionally replaced by methyl groups. The intention in these designs were to observe the shift in relative energy between the sets of conformers to determine how significant of a role the phenyl rings and isopropyl groups played. These variations, along with the original form structures, were subject to the same calculations as in the first design: Gaussian 09, B3LYP hybrid functional, STO-3G basis set, no angle restrictions, single particle in a vacuum, otherwise default parameters. The results of these calculations are tabulated in tables 3.2 and 3.3.

Immediately obvious in the table are the considerable number of nonconverged

Internal Species	Terminal Species	Conformer	Final Energy (Hartrees)	Energy (Hartrees)	Energy (KJ/mol)
methyl	methyl	60-60	-10738.91336	0.0000454	0.119
methyl	methyl	60-180	-10738.9134	0	0
methyl	methyl	60-300	-10738.91286	0.0005358	1.407
methyl	methyl	180-60	-10738.91325	0.0001533	0.402
methyl	methyl	180-180	-10738.91335	0.0000475	0.125
methyl	methyl	180-300	-10738.91336	0.0000451	0.118
methyl	methyl	300-60	-10738.91336	0.0000455	0.119
methyl	methyl	300-180	-10738.91287	0.0005357	1.406
methyl	methyl	300-300	-10738.9107	0.002703	7.097
phenyl	methyl	60-60	-11875.15183	0.0001451	0.381
phenyl	methyl	60-180	-11875.15144	0.0005304	1.393
phenyl	methyl	60-300	-11875.15197	0	0
phenyl	methyl	180-60	-11875.14282	0.0091505	24.025
phenyl	methyl	180-180	-11875.15004	0.0019354	5.081
phenyl	methyl	180-300	-11875.15064	0.0013353	3.506
phenyl	methyl	300-60	-11875.06665	0.0853257	224.023
phenyl	methyl	300-180	DNC	DNC	DNC
phenyl	methyl	300-300	-11875.1497	0.0022723	5.966
phenyl	isopropyl	60-60	DNC	DNC	DNC
phenyl	isopropyl	60-180	-12341.23176	0.0053028	13.923
phenyl	isopropyl	60-300	DNC	DNC	DNC
phenyl	isopropyl	180-60	DNC	DNC	DNC
phenyl	isopropyl	180-180	-12341.23513	0.001935	5.08
phenyl	isopropyl	180-300	DNC	DNC	DNC
phenyl	isopropyl	300-60	DNC	DNC	DNC
phenyl	isopropyl	300-180	-12341.23706	0	0
phenyl	isopropyl	300-300	DNC	DNC	DNC

Table 3.2: Data of B3LYP/STO-3G minimization of variations of pentagermane compound at various conformers. DNC denotes a failure to converge with the self-consistent field method.

Internal Species	Terminal Species	Conformer	Final Energy (Hartrees)	Energy (Hartrees)	Energy (KJ/mol)
methyl	methyl	60-60-60	-12870.91834	0.0009503	2.495
methyl	methyl	60-180-60	-12870.91929	0.0000004	0.001
methyl	methyl	60-180-180	-12870.91813	0.0011628	3.053
methyl	methyl	60-180-300	-12870.91869	0.0005972	1.568
methyl	methyl	60-300-300	DNC	DNC	DNC
methyl	methyl	180-60-60	-12870.91897	0.0003189	0.837
methyl	methyl	180-180-60	-12870.91833	0.0009585	2.517
methyl	methyl	180-180-180	-12870.91929	0.0000004	0.001
methyl	methyl	180-180-300	-12870.91929	0.0000003	0.001
methyl	methyl	180-300-60	-12870.91897	0.0003192	0.838
methyl	methyl	300-60-180	DNC	DNC	DNC
methyl	methyl	300-180-60	-12870.91929	0	0
methyl	methyl	300-180-180	DNC	DNC	DNC
methyl	methyl	300-180-300	-12870.91814	0.0011527	3.026
phenyl	methyl	60-60-60	DNC	DNC	DNC
phenyl	methyl	60-60-180	-14385.89674	0.0052183	13.701
phenyl	methyl	60-60-300	-14385.89487	0.0070829	18.596
phenyl	methyl	60-180-60	DNC	DNC	DNC
phenyl	methyl	180-60-60	DNC	DNC	DNC
phenyl	methyl	180-60-180	-14385.90195	0	0
phenyl	methyl	180-60-300	-14385.89855	0.0033998	8.926
phenyl	methyl	180-180-180	-14385.83838	0.0635763	166.92
phenyl	methyl	180-300-180	-14385.79233	0.1096251	287.821
phenyl	methyl	300-60-60	DNC	DNC	DNC
phenyl	methyl	300-60-180	-14385.89836	0.003597	9.444
phenyl	methyl	300-60-300	-14385.89836	0.0035979	9.446
phenyl	methyl	300-180-60	DNC	DNC	DNC
phenyl	methyl	300-300-300	DNC	DNC	DNC
phenyl	isopropyl	60-180-180	-14851.9865	0	0
phenyl	isopropyl	60-300-60	DNC	DNC	DNC
phenyl	isopropyl	60-300-180	DNC	DNC	DNC
phenyl	isopropyl	180-300-60	DNC	DNC	DNC
phenyl	isopropyl	180-300-180	DNC	DNC	DNC
phenyl	isopropyl	180-300-300	DNC	DNC	DNC
phenyl	isopropyl	300-300-60	DNC	DNC	DNC
phenyl	isopropyl	300-300-180	DNC	DNC	DNC
phenyl	isopropyl	300-300-300	DNC	DNC	DNC

Table 3.3: Data of B3LYP/STO-3G minimization of variations of hexagermane compound at various conformers. DNC denotes a failure to converge with the self-consistent field method.

results. An unexpected bulkiness trend followed that a fully methylated variation of the structure was most likely to converge to a stable state, while the fully internal phenyl structures with methyl ends slightly reduced convergence and the original fully internal phenyl structures with isopropyl ends drastically reduced convergence. The common-sense expectation that the addition of the phenyl ends would reduce stability was not parroted in these results. A deeper exploration into the change of stability is a promising avenue for future investigation, but was not further explored in this work. As can be seen in table 3.3, the lowest energy conformer for each structure varied greatly, but never included the fully trans conformer and only once the collaborator-reported trans-cis-trans conformer as the most stable. Still, given the considerable amount of nonconverged conformers, a new design was necessary to further improve the scope of the lowest energy conformation search.

3.2.2.3 Design 3: Death by 1.59 Million Cuts

In the final version of the conformer generation effort, additional creation efforts were focused on the individual phenyl rings. The unfavorable interactions between the phenyl rings were considerable hurdle in the previous designs and a potential explanation for the large number of nonconverged structures, including the possibility that the terminal isopropyl hexagermanium structures contained particularly unfavorable interactions among the phenyl rings. This third design sought to remove the uncertainty in phenyl ring bulkiness by applying the same approach as the backbone generation: create unique conformers of every backbone torsion and phenyl ring, limiting each torsion to one of three rotational positions following the Newman projection style. Unfortunately, this task proved prohibitively large.

As an explanation for the insurmountability of the problem, consider the hexagermanium structure. The germanium dihedrals represent three rotatable bonds each with three initial positions. To include the phenyl rings would require the inclusion of

eight new rotatable bonds each with three initial positions. Additionally, considering each terminal germanium's rotation while ignoring each isopropyl's rotatable bonds adds two initial positions each with three initial positions. Together, this creates a structure with 13 rotatable bonds each with three initial positions. The number of conformers follows as $3^{13} = 1,594,323$ initial conformers. Now we must consider the computational aspect of this many conformers. At 10 conformers rotated and generated per second and 16 KB per conformer, the initial conformers would require 44.3 hours and generate 25.49 GB of data just in the initial structures. At an average of 72 minutes per computation and 73.7 MB produced at B3LYP hybrid functional and STO-3G basis set and access to all 255 regular nodes of Oklahoma State University's Cowboy cluster running in parallel, the complete computation would generate 117.5 TB of data and require 312 days of continuous computation to determine a possible lowest energy conformer of this one molecule at a relatively low level basis set and theory. A request to utilize 100% of university supercomputer resources for nearly a year for the sake of determining the lowest energy conformer of one molecule would likely be rejected, so this task would likely require a time scale of years or even decades to produce with shared access to university resources. While conventionally considered a small molecule, the scale of conformers and computational requirements pushes this problem into the realm of Levinthal's paradox.

While this third design would have likely revealed the lowest energy conformer, or at least one considerably close the the exactly lowest energy conformer, the effort ultimate fails under its own weight. Even with efforts to truncate duplicate forms, the problem of scale remains. A reduction by 50% still requires a computation effort in the timescale of years or decades for the calculation of a single molecule. For an effective computational outlook, this system needs to be reduced by several orders of magnitude.

3.2.3 Scale Reduction Efforts

For a system with conformers on the millions scale and computations on the hour scale, a magnitude reduction in either aspect would improve the practicality of this design approach. For example, by simplifying the computational method from 72 minutes on average to 5 minutes on average, the overall computational requirement would be reduced by 92%, a full order of magnitude. Unfortunately, reducing the complexity of the method sacrifices the reliability of data. A potential solution here would be to create rounds of calculations at different complexities, where each sequential round restricts the pool of potential conformers. Ideally, the balance of the increasing computational complexity and the decreasing pool size would maintain a consistent computational requirement. For example, a new round using a higher functional theory and basis set at 5x computational requirement would ideally be paired with a reduction in conformer pool size by a factor of 5. This would produce a series of calculation sets with additive computational requirement instead of a magnitudinal expansion.

The natural next question lies within the reliability of basis sets and functional theories. It naturally follows that a less-accurate method should not be relied on while better methods exist. However, considering the scale of the conformer pool, it follows that a less accurate method would still produce energy values with a roughly similar internal consistency. For example, a 180-0-180 form of the hexagermanium compound with parallel phenyl rings as modeled in figure 3.4 will have intense syn interactions between some phenyl rings and will likely not yield a desirable energy value at any level of calculation while a fully trans form with perfect pi stacking phenyl rings will likely have a lower energy value at all levels of calculation. It follows that, at lower levels of accuracy, the extremely high energy conformers can be pruned from the pool early and drastically reduce overall computational requirements. A generic effort at producing a method in this style is detailed in chapter 4, while the remainder of this

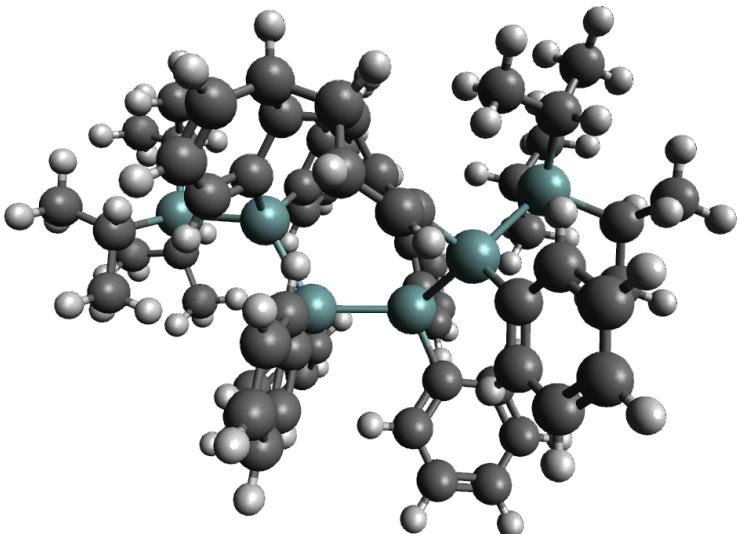


Figure 3.4: Visualization of a trans-cis-trans hexagermane structure.

chapter details additional efforts of calculating these germanium compounds.

3.2.4 Efforts at Simplification

One potential avenue of simplifying the process is computing the energy minimizations of lower-period atoms (e.g. a carbon backbone instead of germanium) and then applying a correction factor for a net reduction in computation time. As a period 4 element, germanium exhibits computational qualities similar to but more complicated than both carbon and silicon. Using tested samples, an energy minimization of a carbon-backbone molecule instead of the germanium represented a 92% reduction in computation speed. Assuming a nominal correction factor exists and can be applied, this represents an order of magnitude reduction in computation time with one simplification. Potentially, this would allow investigators to much more quickly eliminate high energy conformers and more rapidly reduce the scope of the search.

The approach to acquiring sufficient data for a possible correction factor involved running an extremely simplified form of the germanium compounds, specifically a butagermanium backbone with hydrogens occupying all terminal and internal bonds. This reduced the complication and complexity of bulkiness and allowed for quick full

torsion rotations about the single Ge-Ge-Ge-Ge dihedral. By operating at intervals of 5°, a full torsion drive provides a glimpse at relative energies of the molecule at 72 discrete states.

The extended round of torsion drive calculations included an alteration in representation of the data. As the focus had shifted from relative energies and intensities across multiple theories and basis sets to a focus on graph smoothness and internal relative energies, the energy axis of plots were reduced to a unitless scale ranging 0 to 1, where 0 represents the minimum energy and 1 represents the maximum energy in a given set of torsion drive data. This allowed for graphical representations of each torsion drive to emphasize the internal variation of torsions relative to the minimum and maximum values. This was accomplished by taking any set of data with absolute scale energy unit, identifying the minimum and maximum values, and scaling each data point according to equation 3.1. The script to collect and scale data points is detailed in B.

$$E_{i,red} = \frac{E_{i,abs} - E_{min,abs}}{E_{max,abs} - E_{min,abs}} \quad (3.1)$$

An example plot of this torsion drive is shown in figure 3.5 Once multiple torsion drives had completed in multiple group four elements (butyl C, Si, and Ge were all built and tested), the energies could be compared and analyzed for any relative or absolute scaling at the additive or multiplicative reference.

For a full comparative set, 3456 points of analyzed data were generated for each reference molecule's free energy in comparison with the others. Unsurprisingly, no simple correction factor arose by method of a simple additive or multiplicative term applied toward all torsion points. To expand on the comparative set, a set of butyl-group IV conformers were generated with every possible permutation of C, Si, and Ge, each then rotated about the torsion in 5° increments to produce a total 5832 conformers. These were then subject to the same data comparison method as before,

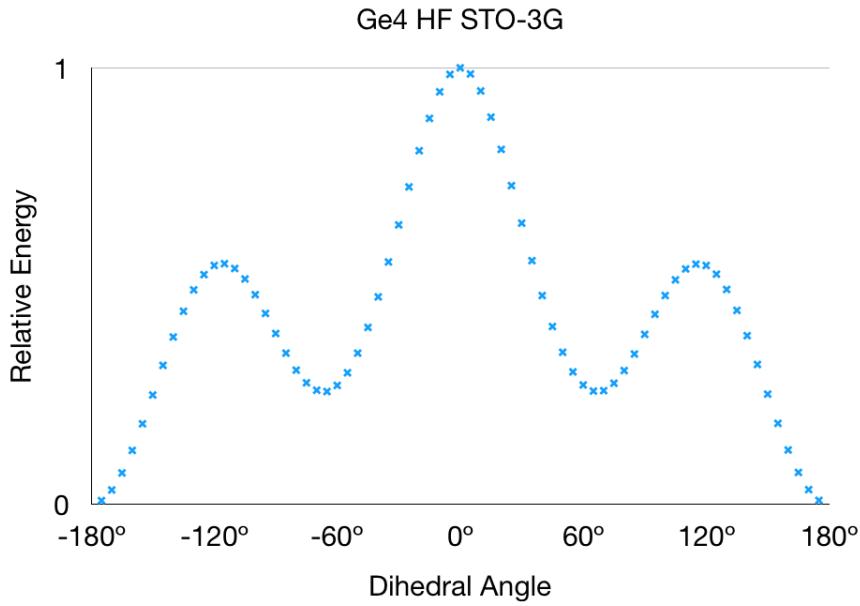
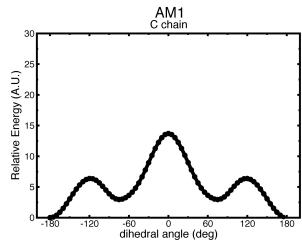
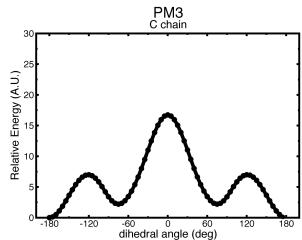
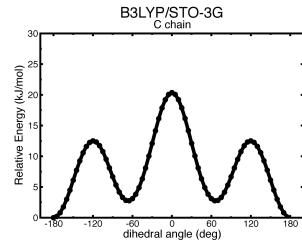
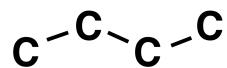
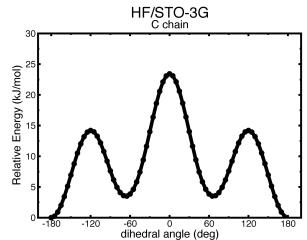


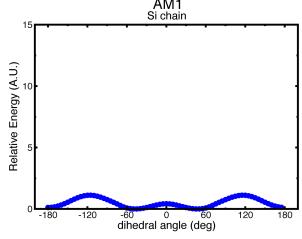
Figure 3.5: Sample torsion plot at reduced energy scale.

again to no noticeable trend. A future avenue of research could be to further explore this with depressive or polynomial terms to discover whether a simple corrective function might exist with specific molecules.

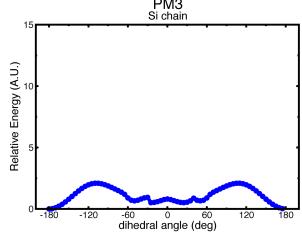
While this approach likewise did not find any simple correction factor, a graphical representation of multiple functionals across the butyl C, Si, and Ge show an interesting trend, as visualized by a graph provided by Dr. Christopher Fennell and shown in figure 3.6. A common theme of these graphs is that the relative energies follow the expected energetic barrier of a Newman projection, with local maxima at the 120° and 240° (or -120°) angles and local minima at the 60° and 300° (or -60°). The global maximum and minimum were consistently at 0° and 180° angles, respectively. As expected by different types of calculations, the torsion graphs hold different internal relative energies. For carbon, all four functionals produced a clean curve. The AM1 and PM3 functionals produced unexpected results for both Si and Ge graphs. In each, the expected highest energy 0° torsion angle was instead the most favorable of the three eclipsed angles. Additionally, the Si PM3 and the Ge AM1 and PM3 functionals showed strong spikes along the expectedly smooth curve, with the Ge PM3

AM1**PM3****B3LYP/STO-3G****HF/STO-3G**

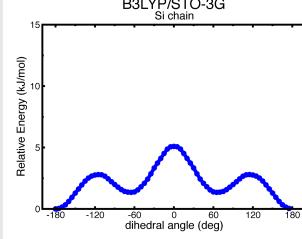
AM1



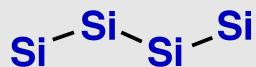
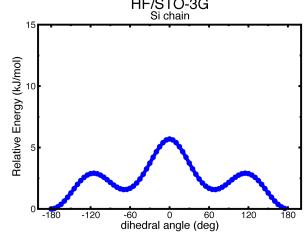
PM3



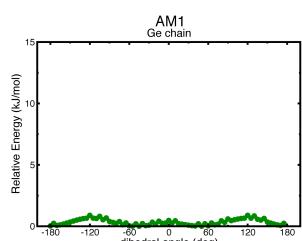
B3LYP/STO-3G



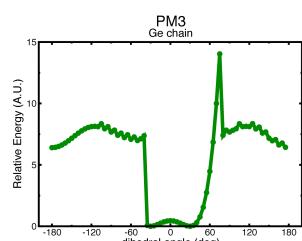
HF/STO-3G



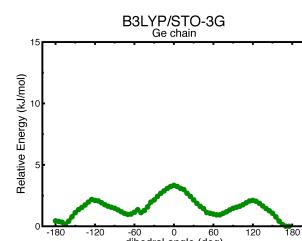
AM1



PM3



B3LYP/STO-3G



HF/STO-3G

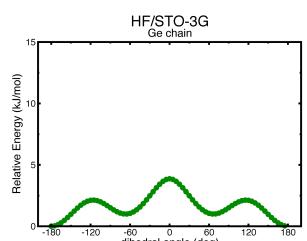


Figure 3.6: Visualization of a multiple pure group IV torsions at various theories and basis sets

being noticeably broken.

While the Si graphs smoothed out for the B3LYP and HF functionals at STO-3G basis set, the Ge B3LYP showed significant spikes and only the HF STO-3G exhibited a smooth curve. Effectively, this discovery of spikes along torsion drives led to the realization that the validity of a basis set could possibly be determined by the smoothness of a torsion drive. For example, any calculation of a germanium-containing molecule will likely not produce reliable results with a B3LYP hybrid functional and STO-3G basis set, while the Hartree Fock STO-3G calculation would at least be tentatively reliable for comparative energy levels at various conformations.

3.3 Discovery of a Consistent Inconsistency

The next natural step was to calculate and plot additional functional theories and basis sets with the butagermanium chain. While effectively a lightly guided meandering through the available calculation types, the first effort was to observe relative differences across multiple basis sets of the Hartree Fock theory and to examine the relative computational requirements of each. This plan was quickly redirected, however, when a curiosity within the data was revealed.

While running additional torsion drives of butagermane at differing basis sets and functional theories, an inverted energy was discovered. As can be seen in figure 3.7, the B3LYP theory with 6-31G(d) basis set appears flipped upon a cursory glance. After a more careful observation, the minima and maxima are at the "wrong" angles and cannot be a simple flip of the minima and maxima. Instead, the data is simply junk.

Naturally, the focus shifted toward discovering the source of the bad data. A repeat of the trial yielded the same data. A repeat of the system with a freshly created butagermane yielded the same data. A trial with data from a butagermane trial with the 6-31G(d,p) basis set yielded the same data. Each attempt at a 6-31G(d)

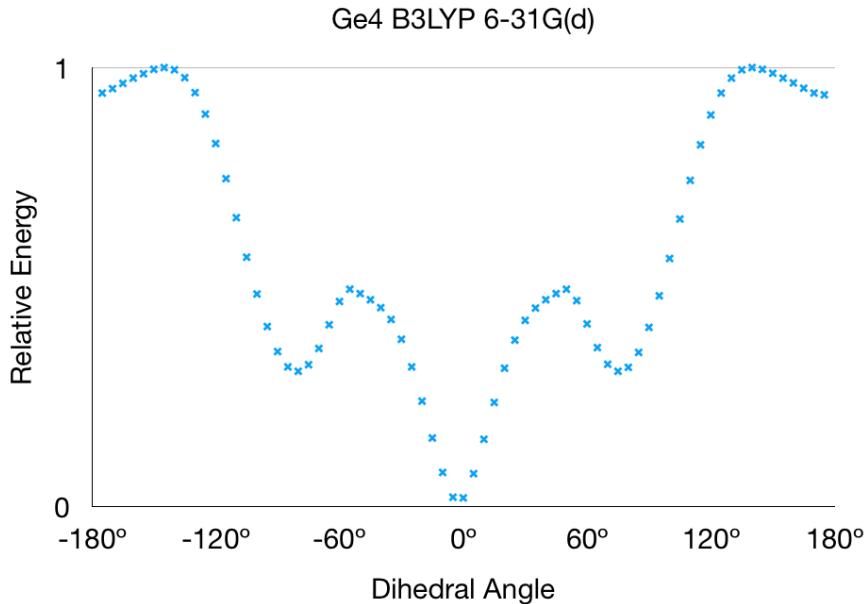


Figure 3.7: A curious seemingly-inverted torsion plot of butagermane.

basis set with the B3LYP theory yielded junk data, while other basis sets within the theory produced expected data. Next, the butagermane torsions were run with an identical basis set group with the Hartree-Fock theory, the results of which are shown in table 3.8.

Surprisingly, the 6-31G(d) result was also strangely inverted. This process was repeated for several more theories, with the 6-31G(d) basis set results plotted in figure 3.9. Curious to see if the germanium atom's basis set data or if the entire basis set method was the source, a similar run with butasilane was made and graphed in figure 3.10, to expected results. A quick run confirmed the problem to also exist on Gaussian 03 as well as Gaussian 09. The final effort was to check whether this error was isolated to Gaussian 09 or to all QM programs. A simplified test to calculate the energy of the expected global minimum (180°) and maximum (0°) of a Hartree Fock theory with the suspect 6-31G(d) basis set was prepared and executed, with the results tabulated in 3.4. As can be seen, critical energetic difference was negative for Gaussian 09 and positive for both GAMESS and NWChem. Since the expected conformations should yield a positive difference, it was concluded that both Gaussian

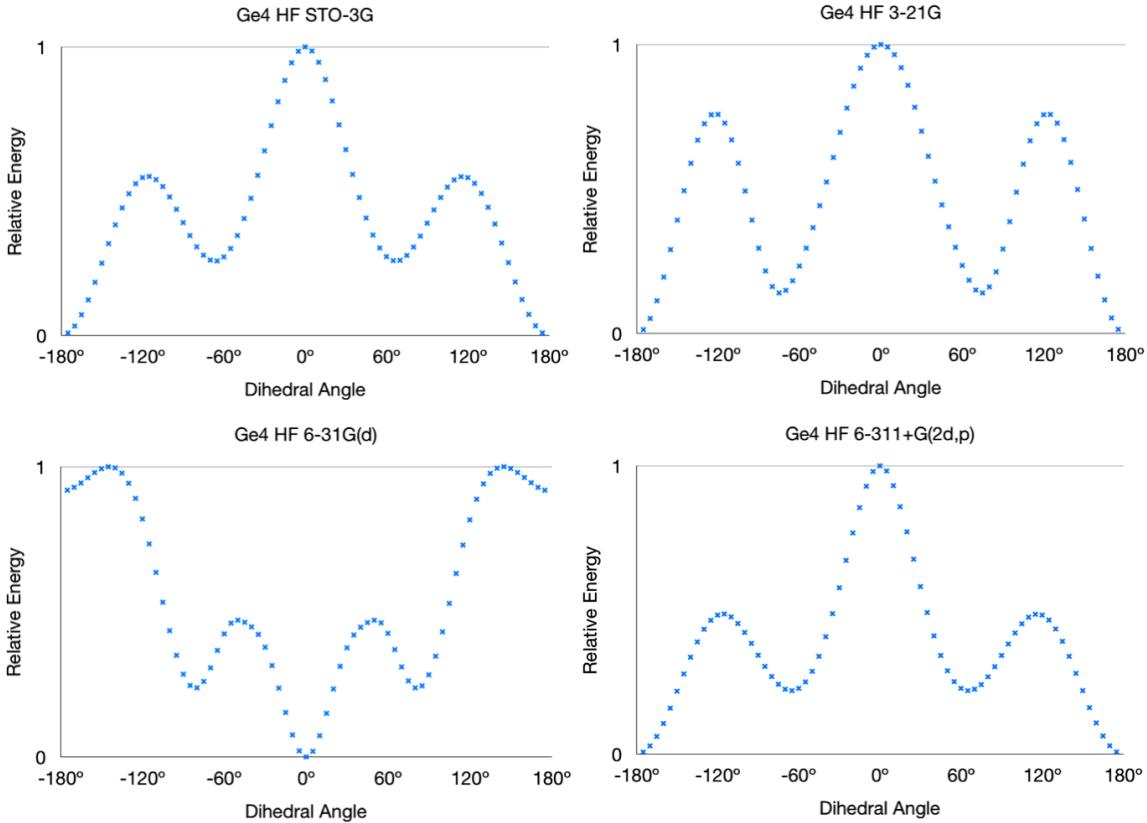


Figure 3.8: Hartree Fock energy minimization of butagermane torsion run at varying basis sets.

03 and 09 contain bad 6-31G(d) basis set data for germanium.

3.4 Final Thoughts

Unfortunately, a trend for simplifying the computation requirements of germanium was not discovered. While it may exist among the data as a more involved function or as some other representation, there also may very well be no simple trend for

Program	Trans Energy (Hartree)	Cis Energy (Hartree)	Energy trans - cis (Hartree)	Energy trans - cis (kcal / mol)
Gaussian	-8298.8259	-8298.8268	-0.0009	-0.5775
GAMESS	-8306.1290	-8306.1290	0.0040	2.4975
NWChem	-8306.1290	-8306.1250	0.0040	2.4974

Table 3.4: Energy comparison of HF theory with 6-31G(d) basis set across multiple computational programs. The expected E should be positive.

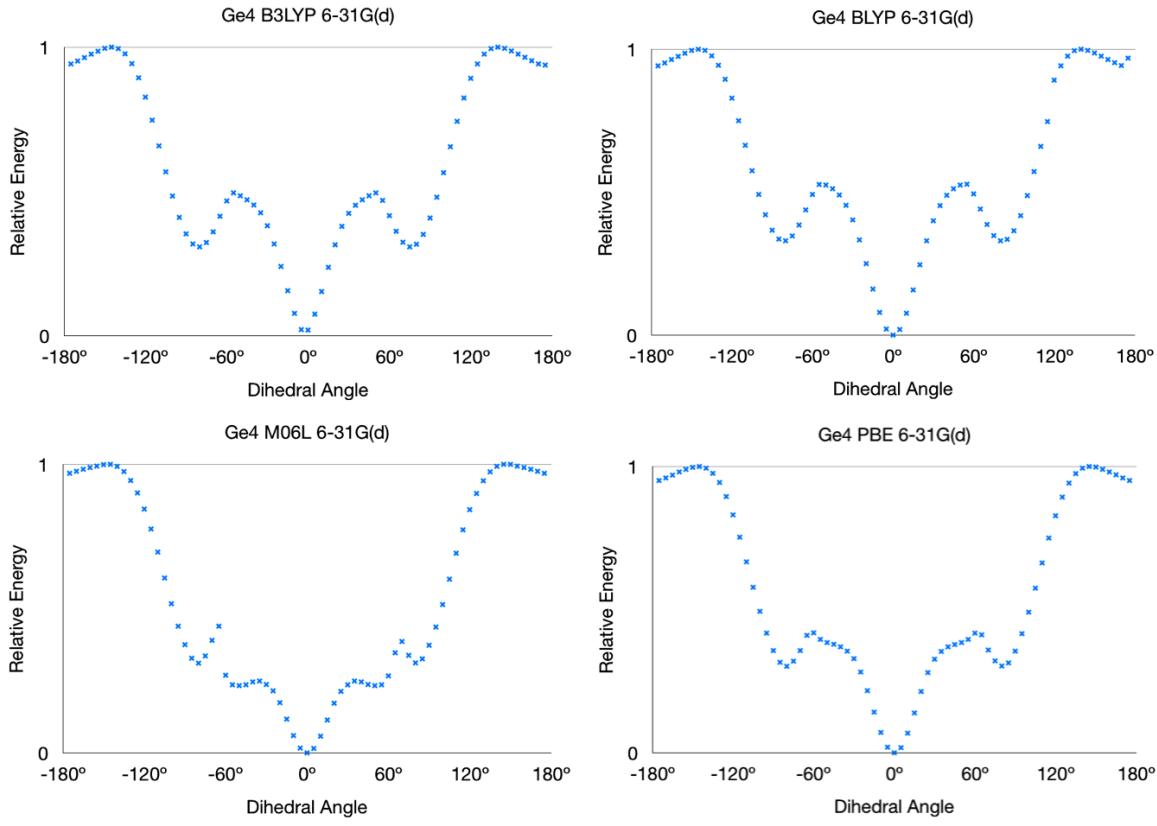


Figure 3.9: Minimization of butagermane torsion run at varying theories and the 6-31G(d) basis set.

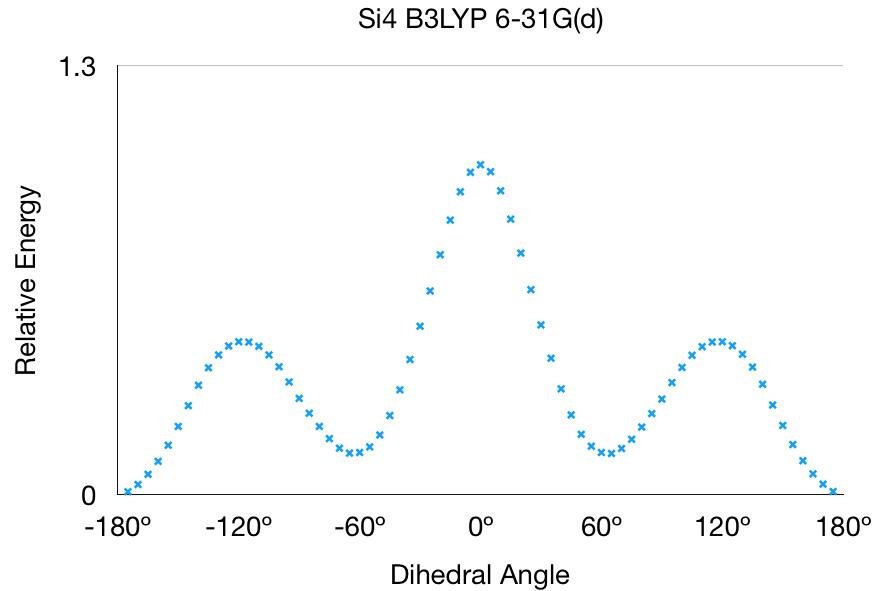


Figure 3.10: B3LYP energy minimization of butasilane torsion run at 6-31G(d) basis set.

switching between germanium and another group IV element.

On a much more interesting note, the results of the torsion drives revealed that Gaussian 03 and 09 contain some mistake within the 6-31G(d) basis set data for germanium. Considering the popularity of Gaussian software in computational chemistry, there are concerning implications about reliability of data for any germanium energy data with the 6-31G(d) basis set. Given that the torsion tests produced expected data for 6-31G(d) data subsequently run through a higher or lower basis set, only reported data with 6-31G(d) as the final calculated energy need be considered. It is recommended that any investigator into computational aspects of germanium either replace the basis set data, use another basis set, or instead use a program like GAMESS or NWChem for that final computation.

CHAPTER 4

Sampling Conformation Landscapes by Rotatable Bond Degrees of Freedom

4.1 A Brief History on Conformation Landscapes

4.1.1 Levinthal's Paradox

In 1969, a molecular biologist by the name of Cyrus Levinthal proposed a thought experiment regarding protein formation⁷ :

Consider a relatively small 150-residue peptide chain completely unfolded. This protein will have 149 peptide bonds and therefore 149 rotatable bonds, 149 phi angles, and 149 psi angles. Assuming three possible angle positions each, the number of possible folds of this protein follows as 3^{447} . How does this peptide chain fold into the appropriate secondary and tertiary structures? Even at attosecond rates of rotating and folding, this peptide chain would likely not fold into the correct structure for many times the age of the universe! Obviously, this is not the case, since proteins fold on the timescale of microseconds to milliseconds.⁷ How, then, do proteins fold so quickly and efficiently? The answer lies in energy cascades through a visualization tool called a golf course.

4.1.2 Levinthal Golf Courses

If one imagines the energy landscape of a peptide chain like a golf course, interesting similarities arise. For example, the lowest point could be considered "the hole" of the course with the lowest energy conformer. When starting at the "tee off" point, there

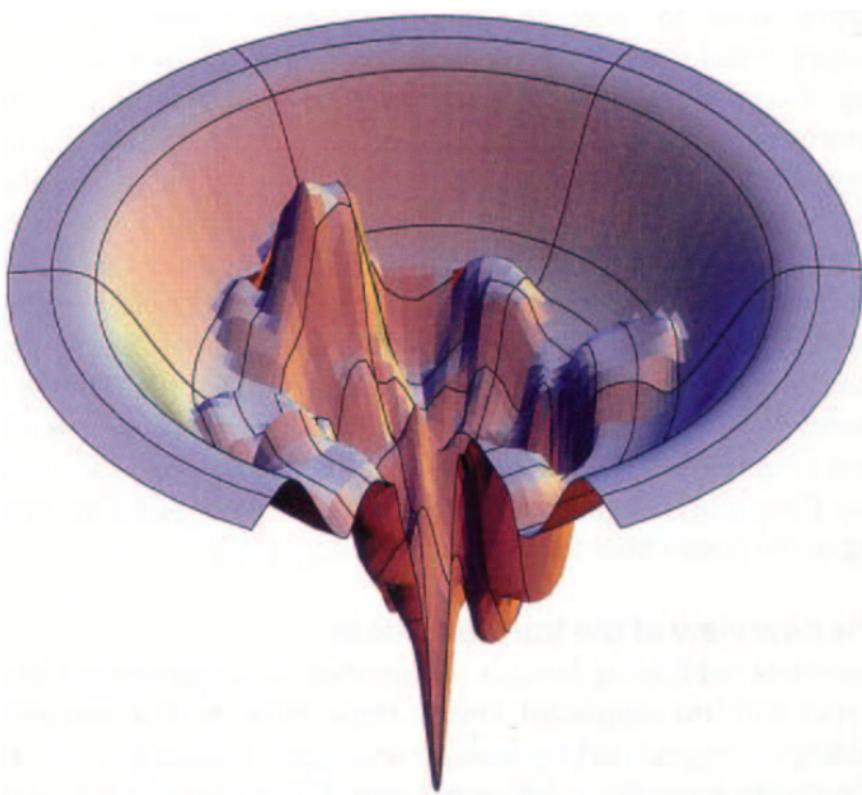


Figure 4.1: Example Levinthal Golf Course taken from⁷.

may not be a clean pathway of energetic difference for the ball to throw. Therefore, the ball must be "struck" toward the hole in a series of motions where the ball is removed from one local minima and placed in another hopefully closer to the hole. Like the image shown in figure 4.1, the course is not always an easy, natural cascade toward the global minima. Most often, investigators will initiate several searches in several locations of this conformation landscape in hopes that one will discover a clear minima that is hopefully the true global minima.

4.2 Purpose of Project

As introduced in chapter 3, there may be a generic solution toward discovering the tetrahedral lowest energy conformer by roughly sampling the full "golf course" and procedurally focusing in on hot spots using automated methods. Ideally, the tool would work through the seemingly infinite possibilities and quickly remove the im-

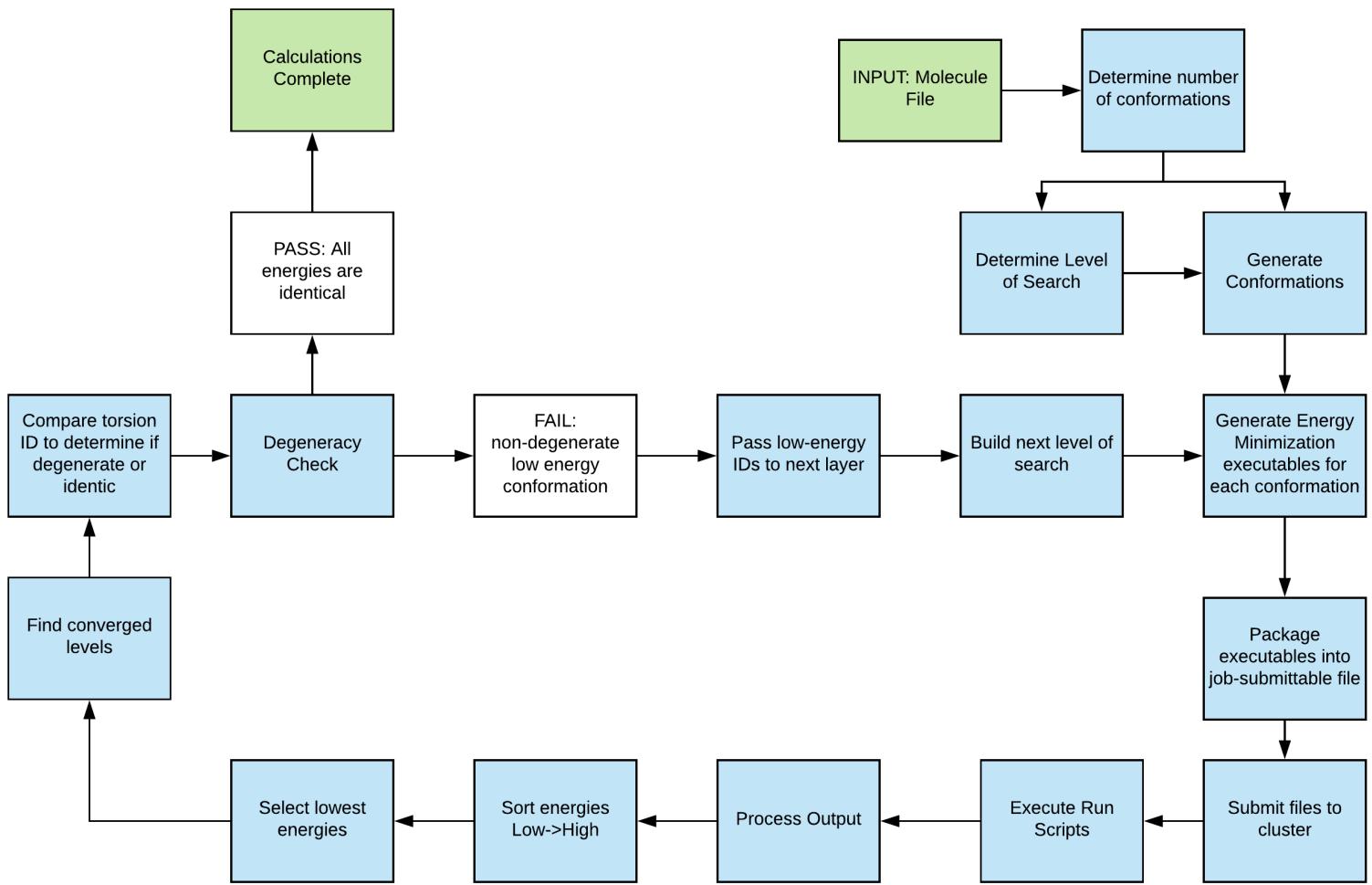


Figure 4.2: Flow of method design for variable resolution conformation landscape search.

possible or duplicate conformers. The tool would roughly take shape though the following pattern. Run through a set of dihedral positions at a constant interval. Selection of lowest-energy optimization organized on dihedral values. Quick determination of importance of dihedral based on how heavily it impacts internal energy. Splitting "best" dihedral(s) into smaller interval to repeat the process.

An overview of system flow given in figure 4.2. This method produces an interesting multilayered visual plot with a zooming effect toward the lowest energy conformer. An example of how this might look for a two-dihedral molecule is given in figure 4.3.

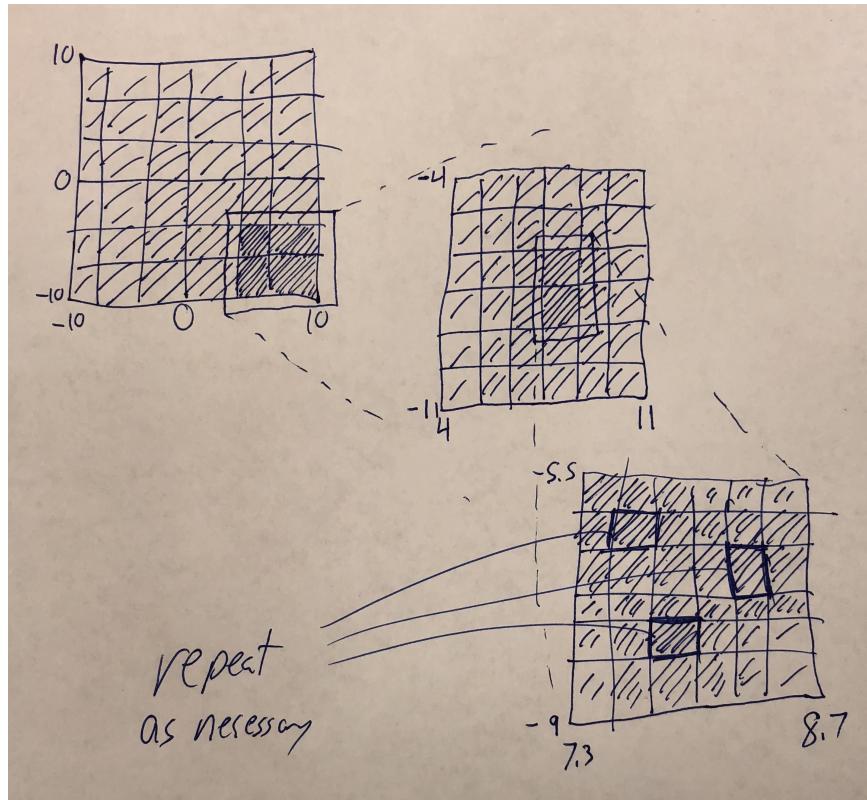


Figure 4.3: Example variable resolution chart (hand-drafted, CGI pending)

4.3 Design of System

This system designed in Python for ease of development and compiled via Cython for computational efficiency. While it currently utilizes Gaussian 09 for energy minimization and UCSF Chimera for conformer generation, it can be redesigned for any computational programs that accomplish the desired tasks.

4.3.1 Variation of Theory and Basis Set Usage by System Size and largest atom type

Given that computational requirements increase with the number of atoms in a molecule and both the accuracy of the theory and basis set used, an initial focus on a manageable amount of conformers with a sufficiently simple theory and basis set is essential to success. The system should estimate quantity and cost of calculations based on physical computational constraints for various theory-basis set pairings. Sys-

tem optimizes calculation types for the scope of the landscape. Effectively, it balances between running the first broad-scope search at relatively low accuracy and a final near-final conformation space with relatively high accuracy methods.

4.3.2 Computational Optimization by Varying Resolution

A common problem in all works on this topic is that the scale of truly searching the conformation landscape is expansive in even the most restrictive designs. The manual efforts in the design of this tool are to build checkers for impossible conformations, including overlapping atom spaces. Additional considerations are that only the most bare, three conformations per rotatable bond angle be considered initially. After the first round of calculations, the scope of candidates should be reduced by several orders of magnitude by refining the search about lower energy regions in the landscape.

4.3.3 Inherent Complications

The single greatest complication of this and any energy landscape tool is the number of rotatable bonds in the target molecule and, to a lesser extent, the elements contained. Consider the hexagermane molecule of interest in chapter 3 and the general focus of this work. One can focus on the number of torsions available to be adjusted in the energy landscape, as shown in figure 4.4. Even with the minimal three rotations per bond, these $19 = 1,162,261,467$ conformers, which is realistically impossible to explore even with a computational method requiring five seconds to compute as 184 years of computation time would be required. This is where the balance between recognizing impossible conformations comes in. Especially with bulky molecules like this hexagermane, many conformations could be eliminated by way of checking for overlapping atoms.

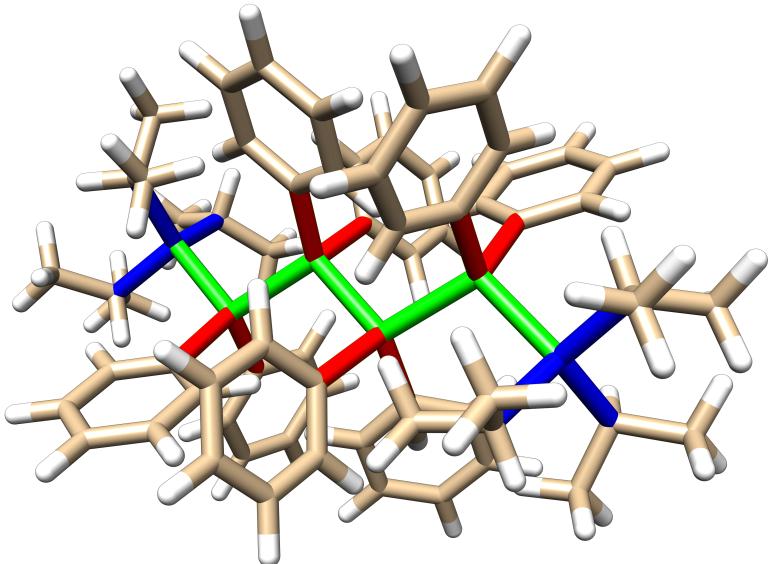


Figure 4.4: Highlighted torsions of the hexagermane molecule by type of bond, where green, red, and blue represent Ge-Ge, Ge-phenyl, and Ge-isopropyl torsion centers, respectively.

4.4 Results

Due to the scale of the hexagermane molecule, a clear answer has not yet been discovered. However, a much more simple run with o-nitrophenol, with only two rotatable bonds, was successful in finding the known lowest energy conformer shown in figures 4.5 and 4.6.

While these would have ideally been produced through a self-perpetuating system at increasing precisions and computation accuracy, the automated tool remains to be realized.

4.4.1 Difficulties and Anticipated Future Approaches

A key difficulty in automation of this tool is defining an abstract computation level based on arbitrary hardware limitations. While currently limited to the Cowboy cluster at Oklahoma State University, the goal is that this tool be made available for chemists everywhere one day. A potential solution for this abstract definition would be a small series of test runs to determine computational cost and general resource

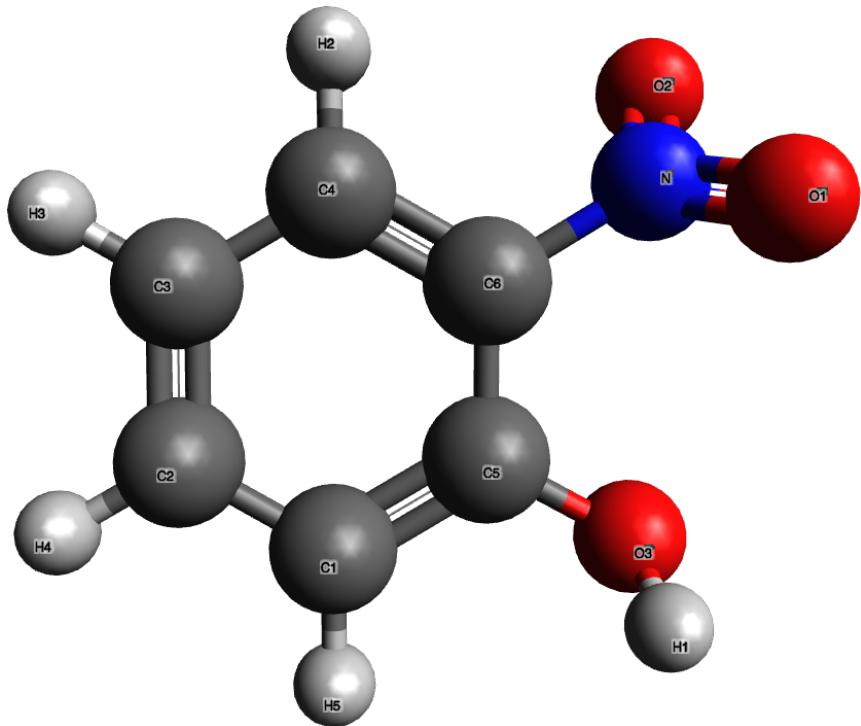


Figure 4.5: Highest energy conformer of o-nitrophenol, ignoring any ring strain conformations. Noticably was unable to rotate and form the hydrogen bond between ortho nitrogen and hydrogen.

availability.

Additionally, the number of rotatable bonds yields the single largest barrier to searching the full conformation space. With continued investigation and the inclusiveness with other works, it seems feasible that the insurmountable barrier to entry may yet be simplified in an objective way that does not perturb the system from finding the lowest energy conformer in any reasonably small molecule.

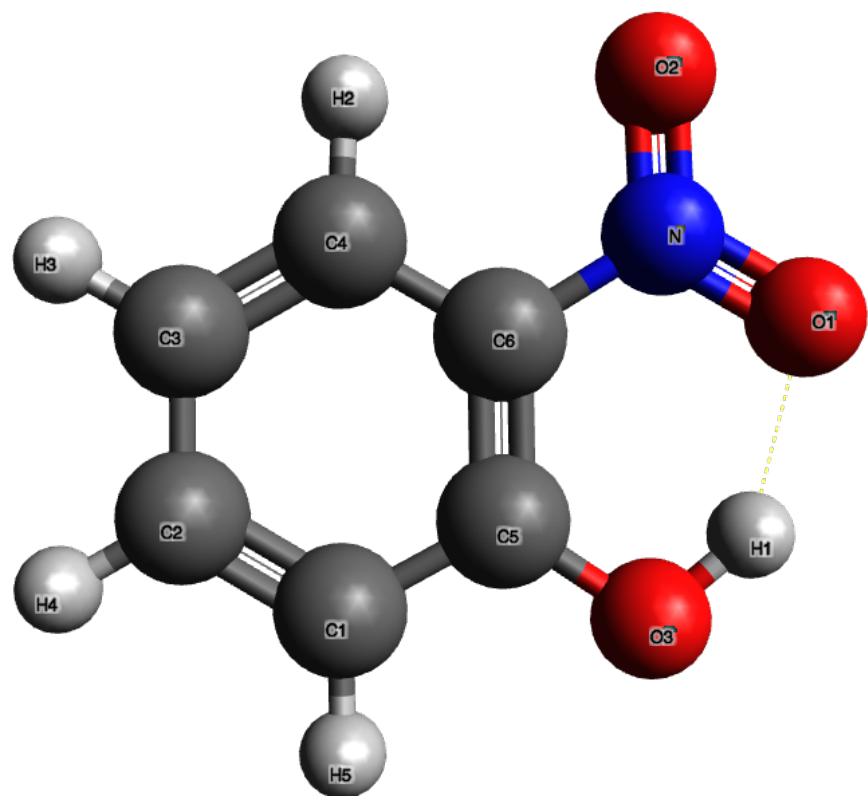


Figure 4.6: Lowest energy conformer of o-nitrophenol. Formed the expected hydrogen bond between the ortho nitrogen and hydrogen.

APPENDIX A

Ice Ih to Ice XI Conversion

Listed below is the source code utilized in the conversion of a PDB Ice Ih structure into an Ice XI structure. This code is functional in a Python 2.7 environment with NumPy and SciPy packages included.

A.1 Code: Crystal Disorganizer Tool

```
1  #!/usr/bin/python
2
3
4 # Author = Gentry Smith
5 # Copyright 2016, all rights reserved
6
7 # this reads in a .PDB file , takes an argument for deformities per
8 # molecules , and randomly organizes the crystal
9 # structure into a disordered proton formation
10
11 # import sample: python PDDBDisorganize.py arg1 arg2 arg3
12 # where:
13 # arg1 = source pdb file to be read (ex: acetone.pdb or acetone)
14 # arg2 = number of defects per molecule (in H2O, num of non-hydrogen-
15 # bonds. from 0 to 4)
16 # arg3 = desired output pdb file name
17
18 import sys
19 print sys.path
20 import string
21 import numpy as np
22 import math
23 import random
24
25
26 sys.setrecursionlimit(10000000) # maximum recursive depth. Set to
27 # (10,000,000) as under maximum
28
29
30 pdbIN = file(sys.argv[1])      # source PDB file
31 maxErr = int(sys.argv[2])       # max errors allowed
32 pdbOUT = str(sys.argv[3])       # output file name
33 finalData = [ [ [ 0 for i in range(3) ] for j in range(3) ] for k in
34 range(300) ]
35
36 # looks at args validity
37 def checkArgs(arg1, arg2, arg3):
38     returnBool = False
```

```

34     if type(arg1) != file: # check arg1
35         print"Bad arg", arg1, " must be a file "
36         returnBool = True
37     if type(arg3) != str: # check arg3
38         print"Bad arg", arg3, ", must be a file name"
39         checkPDBSuffix(arg3)
40         print arg3
41         returnBool = True
42     if type(arg2) != int: # check arg2 type
43         print "Bad arg2: ", arg2, " is not an int."
44         returnBool = True
45     elif type(arg2) == int:
46         if arg2 < 0 or arg2 > 4: # check arg2 range
47             print "arg2 is not in a valid range 0 <= arg2 <= 4"
48             returnBool = True
49     return returnBool
50
51 def checkPDBSuffix(pdbFile):
52     if string.find(pdbFile, '.pdb', 0, len(pdbFile)) == -1:
53         print("did not find 'pdb' in ", pdbFile, ". Appending...")
54         pdbFile += '.pdb'
55
56
57
58 # reads in file ,
59 def readFile(fileName):
60     print "Reading file..."
61     # gets number of atoms
62     atoms = 0
63     for line in fileName:
64         data = line.split()
65         if len(data) > 0:
66             if data[0] != "CONECT" and data[0] != "END":
67                 atoms += 1
68     # print "atoms: ", atoms
69     numMol = atoms / 3 # assumes 3-atom water molecule
70     dataTable = [ [ [ 0 for i in range(3) ] for j in range(3) ] for k in
71     range(numMol) ]
72     fileName.seek(0)
73     iter0 = 0
74     iter1 = 0
75     pdbType = -1
76     for line in fileName:
77         data = line.split()
78         if pdbType == -1:
79             if data[0] == "ATOM":
80                 pdbType = 0
81             elif data[0] == "HETATM":
82                 pdbType = 1
83             # print "LineTuple= ", data
84             if len(data) > 1 and ( data[0] == "ATOM" or data[0] == "HETATM"
85             ):
86                 if data[0] == "ATOM":
87                     newData = getDataATOM( data )

```

```

86         for i in range(3):
87             #data[molecule][atom][X/Y/Z]
88             dataTable[iter0][iter1 % 3][i] = newData[i]
89     elif data[0] == "HETATM":
90         dataTable[iter0][iter1 % 3] = getDataHETATM(data)
91     if iter1 == 2:
92         iter0 += 1
93         iter1 = 0
94     elif iter1 != 2:
95         iter1 += 1
96 # print "DataTable: ", dataTable
97 print "File read"
98 return dataTable, pdbType
99
100
101 # Split by index
102 # if having a problem with reading data, check .pdb to see if data
103 # has a space between each value
104
105 # reads XYZ coordinate data from ATOM-type pdb
106 def getDataATOM(strLine):
107     # print "Getting ATOM Data..."
108     dataLine = strLine[5:8]
109     # print "dataline: ", dataLine
110     i = 0
111     while i < 3:
112         # print "dataline[", i, "]: ", dataLine[i]
113         dataLine[i] = float(dataLine[i])
114         # print "dataline[", i, "] type: ", type(dataLine[i])
115         i += 1
116     return dataLine
117
118 # reads XYZ coordinate data from HETATM-type pdb
119 def getDataHETATM(strLine):
120     # print "Getting HETATM Data..."
121     dataLine = strLine[5:8]
122     # print "dataline: ", dataLine
123     i = 0
124     while i < 3:
125         # print "dataline[", i, "]: ", dataLine[i]
126         dataLine[i] = float(dataLine[i])
127         # print "dataline[", i, "] type: ", type(dataLine[i])
128         i += 1
129     return dataLine
130
131
132 # gets all four position vectors of hydrogen/lone pair as offset of
133 # oxygen molecule
134 def getOrientations( molecule ):
135     # 120 degrees = ( 2 * pi ) / 3 radians
136     theta = ( ( 2 * math.pi ) / 3 )
137     newMol = zeroOrientation(molecule)
138     returnInt1 = rotateMolecule(newMol[1], newMol[2], theta)

```

```

138     returnInt2 = rotateMolecule(newMol[1], newMol[2], (-1 * theta) )
139     return [returnInt1, returnInt2]
140
141
142 # randomly selects new orientation, returns two unique ints, from 0 to 3
143 # inclusively
144 def newRandOrientation( positions ):
145     # print "Changing orientation"
146     randVal1 = random.randint(0,3)
147     randVal2 = random.randint(0,3)
148     while randVal1 == randVal2:
149         randVal2 = random.randint(0,3)
150     newMol = [ [ 0, 0, 0 ],
151                positions[ randVal1 ] ,
152                positions[ randVal2 ] ]
153     return newMol
154
155 # selects new orientation from list. Reduces computational overhead in
156 # re-orientation option traversal
157 def newSetOrientation( positions, pos1, pos2 ):
158     newMol = [ [ 0, 0, 0 ],
159                positions[ pos1 ],
160                positions[ pos2 ] ]
161     return newMol
162
163 # sets molecule coordinates so that oxygen is the origin
164 def zeroOrientation(source):
165     # print "Zeroing Molecule..."
166
167     oxy = source[0]
168     hyd1 = source[1]
169     hyd2 = source[2]
170
171     # print "Oxygen pos: ", oxy
172     # print "Hydrogen 1: ", hyd1
173     # print "Hydrogen 2: ", hyd2
174
175     zeroedOrigin = [0, 0, 0]
176     zeroedHyd1 = [0, 0, 0]
177     zeroedHyd2 = [0, 0, 0]
178     for i in range(3):
179         zeroedHyd1[i] = hyd1[i] - oxy[i]
180         zeroedHyd2[i] = hyd2[i] - oxy[i]
181
182     # print "Zeroed Hydrogen 1: ", zeroedHyd1
183     # print "Zeroed Hydrogen 2: ", zeroedHyd2
184
185     # return new molecule position
186     newMol = [zeroedOrigin, zeroedHyd1, zeroedHyd2]
187     return newMol
188
189 # resets the zeroed molecule to the original oxygen position
190 def resetOrientation(oxygenPos, molecule):

```

```

190 # print "Resetting molecule..."
191 rO = oxygenPos
192 rH1 =[0 ,0 ,0]
193 rH2 =[0 ,0 ,0]
194 newMol = []
195 for i in range(3):
196     rH1[i] = molecule[1][i] + rO[i]
197     rH2[i] = molecule[2][i] + rO[i]
198     newMol = [rO, rH1, rH2]
199 # print "Rebuilt Molecule: ", newMol
200 return newMol
201
202 # rotates vector about axis for theta degrees
203 # Handler for rotationMatrix function below
204 def rotateMolecule(vector, axis, theta):
205     rotMatx = rotationMatrix(axis, theta)
206     return np.dot(rotMatx, vector)
207
208
209 # Creates Rotation matrix for a given axis and theta
210 # from stackoverflow user unutbu
211 # page: http://stackoverflow.com/questions/6802577/python-rotation-of-3d
212 -vector
212 def rotationMatrix(axis, theta):
213     """
214
215     :type axis: list
216     :type theta: union
217     """
218     axis = np.asarray(axis)
219     theta = np.asarray(theta)
220     axis /= math.sqrt(np.dot(axis, axis))
221     a = math.cos(theta/2.0)
222     b, c, d = -axis*math.sin(theta/2.0)
223     aa, bb, cc, dd = (a * a), (b * b), (c * c), (d * d)
224     bc, ad, ac, ab, bd, cd = (b * c), (a * d), (a * c), (a * b), (b * d),
225     (c * d)
226     return np.array([
227         [(aa + bb - cc - dd), (2 * (bc + ad)), (2 *
228         (bd - ac))],
229         [(2 * (bc - ad)), (aa + cc - bb - dd), (2 *
230         (cd + ab))],
231         [(2 * (bd + ac)), (2 * (cd - ab)), (aa +
232         dd - bb - cc)]])
233
234
235 # gets results from rotateAboutAxis plus two Hydrogens to get the
236 # tetrahedron positions
237 def getTetrahedronPositions(molecule):
238     positions = [[0 for i in range(3)] for j in range(4)]
239     newMol = zeroOrientation(molecule) # zero molecule
240     positions[0] = newMol[1]
241     positions[1] = newMol[2]
242     newPos = getOrientations(molecule) # get final two positions
243     positions[2] = list(newPos[0])

```

```

238     positions[3] = list(newPos[1])
239     return positions # return all four positions
240
241
242 # checks distance of new positions from zero
243 def checkDist(posArray):
244     distance = [0 for i in range(len(posArray))]
245     for i in range(len(posArray)):
246         distance[i] = ( (posArray[i][0] * posArray[i][0]) +
247                         (posArray[i][1] * posArray[i][1]) +
248                         (posArray[i][2] * posArray[i][2]) )
249         # print "Distance", i, ":", distance[i]
250     avg = 0
251     for i in range(len(posArray)):
252         avg += distance[i]
253     averageDistance = ( avg / len(posArray) )
254     # print "Average Distance: ", averageDistance
255     return averageDistance
256
257
258 # prints data given a 3D table of water molecules
259 def printData(data):
260     print "Data: "
261     strData = [ "O" , "H1" , "H2" ]
262     dimData = [ "X" , "Y" , "Z" ]
263     bigAvg = 0
264     numAtoms = 0
265     for mol in range(len(data)):
266         for atom in range(len(data[mol])):
267             printStr = str(mol) + ":" + strData[atom] + ":" +
268             for dimension in range(3):
269                 printStr += dimData[atom] + ":" + "{:7.3f}".format(data[
270 mol][atom][dimension]) + "\t"
271             print printStr
272             bigAvg += checkDist(zeroOrientation(data[mol])[1:])
273             numAtoms += 1
274             print ""
275     print "total average distance: ", bigAvg / numAtoms
276
277
278 # checks validity of molecule
279 def isDefectiveCheck(err, neighborData, posData, index):
280     # find nearby molecules (avg oxygen distance??)
281     print "checking for defects at index", index, "... "
282     print "neighbor indices: ", neighborData[index]
283     returnBool = False
284     neighbors = 4
285     for i in range(4): # count real neighbors
286         if neighborData[index][1][i] == -1:
287             neighbors -= 1
288         if neighbors <= err: # de facto good if num(neighbors) <
maxErrAllowed
289             # print "Fewer neighbors than allowed errors. de facto Good
Orientation"

```



```

339     # re-rotate molecule through all positions (iterated through all
340     # orientations)
341     positions = getTetrahedronPositions(posData[index])
342     zeroedMol = newRandOrientation(positions)
343     # print "isGood CHECK", isGood
344     isGood, posData = iterThroughRotations(err, neighborData,
345     posData, index)
346     posData[index] = resetOrientation(posData[index][0], zeroedMol)
347     if timeCount >= 13: # { (1 - 1/6)^n < 0.05 } says n = 17
348         # BROKEN - need to rebuild
349         # 0. evaluated molecule has too many defects
350         # 1. reorient molecule statistically probable amount of
351         # times to cover all orientations
352         # 2. Repeat 1. with neighbor 1
353         # 2a repeat 1. with original molecule
354         # 3. Repeat 2. with neighbor 2, 3, 4, as/if necessary
355         for neighborIndex in range(4):
356             if neighborData[index][1][neighborIndex] != -1:
357                 positions = getTetrahedronPositions(posData[
358                 neighborIndex])
359                 zeroedMol = newRandOrientation(positions)
360                 posData[neighborIndex] = resetOrientation(posData[
361                 neighborIndex][0], zeroedMol)
362                 # isGood = isDefectiveCheck(err, neighborData,
363                 posData, neighborIndex)
364                 isGood = isDefectiveCheck(err, neighborData, posData, index)
365                 if not isGood:
366                     isGood, posData = rerunMolAndNeighbors(err, neighborData,
367                     posData, neighborData[index][1][neighborIndex])
368                     finalData = posData
369             return True, finalData
370
371 # iterates molecule through all possible rotations
372 def iterThroughRotations(err, neighborData, posData, index):
373     isGood = False
374     pos1 = 0 # tetrahedral position for H1
375     pos2 = 0 # tetrahedral position for H2
376     while not isGood or (pos1 != 3 and pos2 != 3): # iterates through
377         # all orientations, stops if good orientation
378         if pos1 != pos2:
379             posData[index] = newSetOrientation(posData[index][0], pos1,
380             pos2)
381             isGood = isDefectiveCheck(err, neighborData, posData, index)
382             if pos2 < 3:
383                 pos2 += 1
384             elif pos2 == 3:
385                 if pos1 < 3:
386                     pos1 += 1
387                     pos2 = 0
388     return isGood, posData
389 # determines minimum hydrogen distance between two atoms
390 def minHydrogenDistance(mol1, mol2):
391     minDist = 100
392     for first in range(2):

```

```

384     for second in range(2):
385         newDist = getDistBetweenAtoms(mol1[first+1], mol2[second+1])
386         if newDist < minDist:
387             minDist = newDist
388     return minDist
389
390
391
392
393
394 # finds neighboring molecules of each molecule
395 def getNeighbors(data):
396     returnData = [ [ [ 0 for i in range(4) ] for j in range(2) ] for k
397     in range(len(data)) ] # data[molecule][distance ,index ][ four values ]
398     for mol1 in range(len(data)):
399         minDist = [100, 100, 100, 100]
400         minIndex = [0, 0, 0, 0]
401         for mol2 in range(len(data)):
402             if mol1 != mol2:
403                 newMin = getDistBetweenAtoms(data[mol1][0], data[mol2]
404                 )[0])
405
406                 bigIndex = indexOfBiggest(minDist)
407                 if newMin < minDist[bigIndex]:
408                     minDist[bigIndex] = newMin
409                     minIndex[bigIndex] = mol2
410
411                 for i in range(4):
412                     if minDist[i] >= 9:
413                         minDist[i] = -1
414                         minIndex[i] = -1
415
416                 # print "Four smallest Distances of", mol1, ":", minDist
417                 # print "Four smallest Indices of", mol1, ":", minIndex
418                 returnData[mol1] = [minDist, minIndex]
419
420     return returnData
421
422
423
424
425 # finds distance between oxygen atoms
426 def getDistBetweenAtoms( mol1 , mol2 ):
427     distance = ( ( ( mol1[0] - mol2[0] ) * ( mol1[0] - mol2[0] ) ) +
428                 ( ( mol1[1] - mol2[1] ) * ( mol1[1] - mol2[1] ) ) +
429                 ( ( mol1[2] - mol2[2] ) * ( mol1[2] - mol2[2] ) ) )
430
431     return distance
432
433
434
435 # gets index of largest item from a list
436 def indexOfBiggest(check):
437     bigIndex = 0
438     for i in range(len(check)):
439         if check[i] > check[bigIndex]:
440             bigIndex = i
441
442     return bigIndex
443
444
445 # writes data to PDB file
446 def writeDataPDB(data , pdbType):

```

```

436 print "Writing Data to", str(pdbOUT)
437 fileName = str(pdbOUT)
438 output = open(fileName, 'w')
439 if pdbType == 0:
440     writeDataPDBATOM(data, output)
441 elif pdbType == 1:
442     writeDataPDBHETATM(data, output)
443 output.close()
444
445
446 # Writes data to PDB file style = ATOM
447 def writeDataPDBATOM(data, inFile):
448     iterator = 0
449     for molecule in range(len(data)):
450         for atom in range(3):
451             iterator += 1
452             outStr = "ATOM "
453             outStr += str(iterator)
454             while len(outStr) < 11:
455                 outStr = outStr[:6] + " " + outStr[6:]
456             outStr += " "
457             if atom == 0:
458                 outStr += " O " + " WAT"
459             elif atom == 1:
460                 outStr += " H1 " + " WAT"
461             elif atom == 2:
462                 outStr += " H2 " + " WAT"
463             outStr += str(molecule)
464             while len(outStr) < 26:
465                 outStr = outStr[:20] + " " + outStr[20:]
466             outStr += " "
467             outStr += "{:8.3f}".format(data[molecule][atom][0])
468             outStr += "{:8.3f}".format(data[molecule][atom][1])
469             outStr += "{:8.3f}".format(data[molecule][atom][2])
470             outStr += " 1.00 " + " 0.00"
471             outStr += " "
472             if atom == 0:
473                 outStr += " O "
474             elif atom == 1:
475                 outStr += " H "
476             elif atom == 2:
477                 outStr += " H "
478             outStr += "\n"
479             inFile.write(outStr)
480
481
482 # Writes data to PDB file style = HETATOM
483 def writeDataPDBHETATM(data, inFile):
484     iterator = 0
485     for molecule in range(len(data)):
486         for atom in range(3):
487             iterator += 1
488             outStr = "HETATOM"
489             outStr += str(iterator)

```

```

490     while len(outStr) < 11:
491         outStr = outStr[:6] + " " + outStr[6:]
492     outStr += " "
493     if atom == 0:
494         outStr += " O " + " WAT"
495     elif atom == 1:
496         outStr += " H1 " + " WAT"
497     elif atom == 2:
498         outStr += " H2 " + " WAT"
499     outStr += str(molecule)
500     while len(outStr) < 26:
501         outStr = outStr[:20] + " " + outStr[20:]
502     outStr += " "
503     outStr += "{:8.3f}".format(data[molecule][atom][0])
504     outStr += "{:8.3f}".format(data[molecule][atom][1])
505     outStr += "{:8.3f}".format(data[molecule][atom][2])
506     outStr += " 1.00" + " 0.00"
507     outStr += " "
508     if atom == 0:
509         outStr += " O "
510     elif atom == 1:
511         outStr += " H "
512     elif atom == 2:
513         outStr += " H "
514     outStr += "\n"
515     inFile.write(outStr)
516
517
518 # runs program
519 def testRun(inFile, err, outFile):
520     print "Running Test Version of Program..."
521
522
523 # this is the parent runner for the program
524 def runPgm(inFile, err):
525     print "Running Program..."
526     data, pdbType = readFile(inFile)
527     newData = [ [ [ 0 for i in range(3) ] for j in range(3) ] for k in
528     range(len(data)) ]
529     print "Reordering Molecules..."
530     for i in range(len(data)):
531         positions = getTetrahedronPositions(data[i])
532         zeroedMol = newRandOrientation(positions)
533         newMol = resetOrientation( data[i][0], zeroedMol )
534         newData[i] = newMol
535     print "Molecules Reordered"
536     connectedMolecules = getNeighbors(newData) # -1 index = not
537     neighboring
538     finalData = newData
539     for i in range(len(connectedMolecules)):
540         # print "check defects"
541         isFine = isDefectiveCheck(err, connectedMolecules, finalData, i)
542         # print "isFINE CHECK", isFine
543         if not isFine:

```

```

542     # print "fixing defects"
543     while not isFine:
544         # print "RerunMol"
545         isFine , finalData = rerunMolAndNeighbors( err ,
546         connectedMolecules , finalData , i )
547         # print "rerunDone"
548         writeDataPDB( finalData , pdbType )
549         # printData( newData )
550
551 badArgs = checkArgs( pdbIN , maxErr , pdbOUT ) # stop in case of bad
      argument
552
553 # check input args
554 if not badArgs: # stop in case of bad argument
      print "Good Arguments , Initializing Reorientation with" , maxErr , "
      maximum defects"
555     # testRun( pdbIN , maxErr , pdbOUT )
556     runPgm( pdbIN , maxErr )
557 elif badArgs:
558     print "Bad Arguments , Quitting . . . "

```

APPENDIX B

Germanium Landscape

B.1 Sample Gaussian 09 Germanium File

Command files like the one below were built using Dr. Fennell's Gaussian 09 run builder script and proved very effective in producing command files.

```

1#!/bin/bash
2g09 <<EOF > B3LYP_STO-3G_1_hexagermane_transall_first_reorder.out
3%Chk=B3LYP_STO-3G_1_hexagermane_transall_first_reorder
4%NProcShared=12
5#B3LYP/STO-3G OPT

6
7 Title: hexagermane_transall_first_reorder system
8
9 0 1
10 Ge -4.5430000000000 -0.076000000000 0.598000000000
11 Ge -2.121000000000 0.068000000000 0.086000000000
12 C -4.774000000000 -1.010000000000 2.336000000000
13 C -5.549000000000 -1.127000000000 -0.760000000000
14 C -5.371000000000 1.719000000000 0.829000000000
15 C -1.311000000000 1.357000000000 1.303000000000
16 Ge -0.754000000000 -1.978000000000 0.323000000000
17 C -1.340000000000 -2.994000000000 1.865000000000
18 Ge 1.675000000000 -1.503000000000 0.567000000000
19 C -1.186000000000 -3.039000000000 -1.235000000000
20 C 1.925000000000 -0.300000000000 2.094000000000
21 C 2.315000000000 -0.535000000000 -0.983000000000
22 Ge 3.126000000000 -3.539000000000 0.606000000000
23 C 2.522000000000 -4.845000000000 1.926000000000
24 Ge 5.558000000000 -3.333000000000 1.109000000000
25 C 2.938000000000 -4.327000000000 -1.151000000000
26 C 6.648000000000 -2.029000000000 0.052000000000
27 C 6.362000000000 -5.157000000000 0.995000000000
28 C 5.463000000000 -2.780000000000 3.007000000000
29 C -4.808000000000 -2.522000000000 2.129000000000
30 C -6.030000000000 -0.572000000000 3.098000000000
31 H -3.909000000000 -0.757000000000 2.974000000000
32 H -6.954000000000 -0.777000000000 2.525000000000
33 H -5.990000000000 0.508000000000 3.335000000000
34 H -6.094000000000 -1.114000000000 4.066000000000
35 H -5.710000000000 -2.829000000000 1.564000000000
36 H -4.817000000000 -3.048000000000 3.107000000000
37 H -3.927000000000 -2.841000000000 1.555000000000
38 H -5.250000000000 -2.187000000000 -0.668000000000
39 C -5.199000000000 -0.705000000000 -2.186000000000
40 C -7.061000000000 -1.057000000000 -0.533000000000

```

41	H	-4.1260000000000	-0.8790000000000	-2.3610000000000
42	H	-5.4330000000000	0.3560000000000	-2.3850000000000
43	H	-5.7540000000000	-1.3310000000000	-2.9170000000000
44	H	-7.3270000000000	-1.4620000000000	0.4590000000000
45	H	-7.5880000000000	-1.6770000000000	-1.2900000000000
46	H	-7.4380000000000	-0.0190000000000	-0.6080000000000
47	C	5.1710000000000	-1.2860000000000	3.1080000000000
48	C	6.7030000000000	-3.1420000000000	3.8230000000000
49	H	4.6120000000000	-3.3360000000000	3.4560000000000
50	H	7.6170000000000	-2.6600000000000	3.4250000000000
51	H	6.8340000000000	-4.2420000000000	3.8350000000000
52	H	6.5690000000000	-2.8200000000000	4.8770000000000
53	H	6.0030000000000	-0.6730000000000	2.7150000000000
54	H	4.9880000000000	-0.9950000000000	4.1640000000000
55	H	4.2810000000000	-1.0550000000000	2.5110000000000
56	C	6.8810000000000	-2.4910000000000	-1.3860000000000
57	C	8.0000000000000	-1.6920000000000	0.6950000000000
58	H	6.0930000000000	-1.0840000000000	0.0270000000000
59	H	8.6630000000000	-2.5690000000000	0.7760000000000
60	H	7.8710000000000	-1.2570000000000	1.7000000000000
61	H	8.5270000000000	-0.9270000000000	0.0850000000000
62	H	7.5310000000000	-3.3840000000000	-1.4160000000000
63	H	7.3870000000000	-1.6900000000000	-1.9660000000000
64	H	5.9290000000000	-2.7200000000000	-1.8880000000000
65	C	6.0420000000000	-5.8440000000000	-0.3400000000000
66	H	5.9120000000000	-5.7470000000000	1.8170000000000
67	C	7.8820000000000	-5.1770000000000	1.1930000000000
68	H	8.3970000000000	-4.6880000000000	0.3460000000000
69	H	8.2400000000000	-6.2280000000000	1.2290000000000
70	H	8.1890000000000	-4.6900000000000	2.1340000000000
71	H	4.9590000000000	-5.9050000000000	-0.5140000000000
72	H	6.4360000000000	-6.8830000000000	-0.3370000000000
73	H	6.4870000000000	-5.3110000000000	-1.1990000000000
74	H	-6.3620000000000	1.5630000000000	1.3030000000000
75	C	-5.6460000000000	2.4560000000000	-0.4830000000000
76	C	-4.5230000000000	2.5900000000000	1.7560000000000
77	H	-4.3490000000000	2.0800000000000	2.7250000000000
78	H	-5.0420000000000	3.5500000000000	1.9600000000000
79	H	-3.5480000000000	2.8210000000000	1.2850000000000
80	H	-6.3580000000000	1.8940000000000	-1.1100000000000
81	H	-4.7250000000000	2.6290000000000	-1.0570000000000
82	H	-6.1170000000000	3.4400000000000	-0.2730000000000
83	C	-0.5320000000000	2.4210000000000	0.8170000000000
84	C	-1.5290000000000	1.2580000000000	2.6840000000000
85	H	-2.1290000000000	0.4690000000000	3.0880000000000
86	C	-0.9960000000000	2.2060000000000	3.5610000000000
87	C	-0.0010000000000	3.3710000000000	1.6940000000000
88	C	-0.2370000000000	3.2670000000000	3.0660000000000
89	H	0.5960000000000	4.1880000000000	1.3100000000000
90	H	-1.1800000000000	2.1220000000000	4.6240000000000
91	H	0.1740000000000	4.0020000000000	3.7450000000000
92	C	-1.7770000000000	-4.3220000000000	1.7250000000000
93	C	-0.2170000000000	-3.3920000000000	-2.1750000000000
94	C	-2.2320000000000	-5.0370000000000	2.8380000000000

95	H	-1.775000000000	-4.812000000000	0.763000000000
96	C	-1.348000000000	-2.404000000000	3.134000000000
97	C	-0.568000000000	-4.133000000000	-3.309000000000
98	H	0.799000000000	-3.079000000000	-2.038000000000
99	C	-2.513000000000	-3.435000000000	-1.440000000000
100	C	-2.250000000000	-4.433000000000	4.097000000000
101	H	-2.571000000000	-6.058000000000	2.723000000000
102	C	-1.802000000000	-3.118000000000	4.246000000000
103	H	-1.007000000000	-1.394000000000	3.262000000000
104	C	-2.868000000000	-4.180000000000	-2.567000000000
105	H	-3.268000000000	-3.168000000000	-0.721000000000
106	C	-1.893000000000	-4.529000000000	-3.504000000000
107	H	0.183000000000	-4.395000000000	-4.040000000000
108	H	-3.896000000000	-4.482000000000	-2.715000000000
109	H	-2.164000000000	-5.101000000000	-4.381000000000
110	H	-2.602000000000	-4.985000000000	4.958000000000
111	H	-1.809000000000	-2.651000000000	5.222000000000
112	C	3.101000000000	-6.123000000000	1.997000000000
113	C	2.378000000000	-5.604000000000	-1.315000000000
114	C	3.370000000000	-3.619000000000	-2.281000000000
115	H	2.025000000000	-6.170000000000	-0.467000000000
116	C	2.272000000000	-6.169000000000	-2.590000000000
117	C	1.513000000000	-4.525000000000	2.832000000000
118	C	2.686000000000	-7.047000000000	2.960000000000
119	H	3.865000000000	-6.421000000000	1.310000000000
120	C	1.687000000000	-6.704000000000	3.869000000000
121	H	3.142000000000	-8.028000000000	3.002000000000
122	C	1.100000000000	-5.441000000000	3.804000000000
123	H	1.054000000000	-3.568000000000	2.784000000000
124	C	2.720000000000	-5.462000000000	-3.708000000000
125	H	1.844000000000	-7.156000000000	-2.710000000000
126	C	3.263000000000	-4.184000000000	-3.554000000000
127	H	3.780000000000	-2.628000000000	-2.178000000000
128	H	3.599000000000	-3.631000000000	-4.421000000000
129	H	2.636000000000	-5.900000000000	-4.694000000000
130	H	1.366000000000	-7.414000000000	4.620000000000
131	H	0.327000000000	-5.175000000000	4.510000000000
132	C	1.504000000000	-0.326000000000	-2.095000000000
133	C	1.670000000000	-0.714000000000	3.412000000000
134	C	3.620000000000	-0.040000000000	-0.992000000000
135	C	1.987000000000	0.362000000000	-3.212000000000
136	H	0.510000000000	-0.709000000000	-2.093000000000
137	C	2.402000000000	1.008000000000	1.889000000000
138	C	1.890000000000	0.147000000000	4.490000000000
139	H	1.314000000000	-1.696000000000	3.632000000000
140	C	4.114000000000	0.652000000000	-2.102000000000
141	H	4.233000000000	-0.179000000000	-0.124000000000
142	C	3.296000000000	0.851000000000	-3.216000000000
143	H	1.348000000000	0.514000000000	-4.073000000000
144	H	5.127000000000	1.034000000000	-2.095000000000
145	H	3.673000000000	1.385000000000	-4.079000000000
146	C	2.374000000000	1.435000000000	4.270000000000
147	H	1.691000000000	-0.189000000000	5.500000000000
148	C	2.630000000000	1.865000000000	2.969000000000

```

149 H 2.603000000000 1.384000000000 0.900000000000
150 H 3.001000000000 2.867000000000 2.795000000000
151 H 2.548000000000 2.101000000000 5.105000000000
152 C -2.041000000000 0.841000000000 -1.709000000000
153 C -1.767000000000 0.059000000000 -2.841000000000
154 C -2.300000000000 2.209000000000 -1.888000000000
155 C -1.732000000000 0.632000000000 -4.115000000000
156 H -1.595000000000 -0.996000000000 -2.753000000000
157 C -2.263000000000 2.785000000000 -3.160000000000
158 H -2.521000000000 2.839000000000 -1.039000000000
159 C -1.977000000000 1.997000000000 -4.275000000000
160 H -1.519000000000 0.016000000000 -4.979000000000
161 H -2.458000000000 3.843000000000 -3.281000000000
162 H -1.950000000000 2.441000000000 -5.262000000000
163 H -0.322000000000 2.526000000000 -0.236000000000
164
165 EOF
166 formchk B3LYP_STO-3G_1_hexagermane_transall_first_reorder.chk
167 newzmat -ichk -opdb -step 999 B3LYP_STO-3
    G_1_hexagermane_transall_first_reorder.chk final_B3LYP_STO-3
    G_1_hexagermane_transall_first_reorder.pdb
168 echo
169 echo "Job done"

```

B.2 Building Group 4 Chains

While briefly mentioned and the subject of research for some time, the butyl-IV chain builder is detailed below. Ultimately unsuccessful in the initial trials, these scripts may serve a purpose in further work.

This first script builds a parent set of all possible C, Si, and Ge butylalkyl chains.

```

1 #!/usr/bin/python
2
3 import sys
4 import subprocess
5
6 # argument: sys.argv[1]
7 # Replacement: sed -i -e 's/IN/OUT/g' FILE > NEWFILE
8
9 inFile = file(sys.argv[1])
10
11 def DoIT():
12     for first in { 'C', 'Si', 'Ge' }:
13         name1 = "%s" % (first.lstrip(' '))
14         out1 = open(name1, "w")
15         cmdStr = "sed -e 's/1 GE/1 %s/g' ./%s >> ./%s.pdb" % (first,
16         inFile, name1)
17         # subprocess.call(cmdStr, shell=True, stdout=out1)
18         subprocess.Popen(cmdStr, shell=True, executable='/bin/bash')
19         out1.close()
20         for second in { 'C', 'Si', 'Ge' }:
21             name2 = name1 + "%s" % (second.lstrip(' '))
22             out2 = open(name2, "w")

```

```

22     cmdStr = "sed -e 's/2 GE/2 %s/g' ./%s.pdb >> ./%s.pdb" % (
23         second, name1, name2)
24         # subprocess.call(cmdStr, shell=True, stdout=out2)
25         subprocess.Popen(cmdStr, shell=True, executable='/bin/bash')
26         out2.close()
27         for third in {'C', 'Si', 'Ge'}:
28             name3 = name2 + "%s" % (third.lstrip(' '))
29             out3 = open(name3, "w")
30             cmdStr = "sed -e 's/3 GE/3 %s/g' ./%s.pdb >> ./%s.pdb" %
31             (third, name2, name3)
32             # subprocess.call(cmdStr, shell=True, stdout=out3)
33             subprocess.Popen(cmdStr, shell=True, executable='/bin/
34             bash')
35             out3.close()
36             for fourth in {'C', 'Si', 'Ge'}:
37                 name4 = name3 + "%s" % (fourth.lstrip(' '))
38                 out4 = open(name4, "w")
39                 cmdStr = "sed -e 's/4 GE/4 %s/g' ./%s.pdb >> ./%s.
40                 pdb" % (fourth, name3, name4)
41                 # subprocess.call(cmdStr, shell=True, stdout=out4)
42                 subprocess.Popen(cmdStr, shell=True, executable='/
43                 bin/bash')
44                 out4.close()
45
46 DoIT()

```

This second script takes the original trans-all butyl chain and enumerates 72 torsional rotations into a folder.

```

1 from chimera import runCommand as rc
2 from chimera import replyobj
3 import sys
4 import os
5
6 #standard sys.argv[] for script args?
7 # sys.argv[0] = directory
8 os.chdir(sys.argv[0])
9
10 file_names = [fn for fn in os.listdir(".") if fn.endswith(".pdb")]
11 fn = file_names[0]
12 # inPDB = chimera.openModels.open('/Users/gentry/Desktop/test/testmol.
13     pdb', type="PDB")
14
15 rc("open " + fn)
16 rc("rotation 1 reverse #0:1.HET@/serialNumber=2 #0:1.HET@/serialNumber=3
17 ")
18 for i in range(72):
19     #replyobj.status("Processing " + fn)
20     #rc("open " + fn)
21     #rc("rotation 1 reverse #0:1.HET@/serialNumber=2 #0:1.HET@/
22     serialNumber=3")
23     rc("rotation 1 5")
24     newName = (fn[:-3] + str((i*5)) + ".pdb")

```

```

24 rc("write format pdb 0 " + newName)
25 #rc("close ")
26
27
28 # chimera.runCommand("rotation 2 3 5")
29 # newName = ( inPDB[:-3] + i*5 + ".pdb" )
30 # chimera.runCommand("write format pdb " + newName)

```

B.3 Collecting and Comparing Torsional Data

These two scripts were utilized to reduce the output data into an energy value with normalized intensity from 0 to 1. The third script compares two of these files and looks for any additive or multiplicative trend.

This first file reads energy data and creates a list of absolute energy values per torsion degree.

```

1#!/usr/bin/python
2
3### Author: Gentry Smith, Oklahoma State University
4### Created: August 7, 2017, 3PM
5### Last Edited: August 7, 2017
6
7### Takes a stationary_points.txt file and will copy .pdb files of the
8### same name from a split_conformers.pdb/ folder
9### into a new folder "stationary_conformers"
10# This does not use any args and instead relies on the stationary points
11# file being "stationary_points.txt" and the
12# conformers residing in a "split_conformers.pdb/" directory on the same
13# level. It will create the new folder "stationary_conformers"
14
15import os
16
17def IOValidator():
18    returnBool = [False, False]
19    try:
20        file1 = open('stationary_points.txt', 'r')
21        file1.close()
22        returnBool[0] = True
23    except IOError:
24        print("Did not find 'stationary_points.txt' file. Quitting...")
25        quit()
26    try:
27        wkdir = os.getcwd()
28        file2 = os.chdir('split_conformers.pdb')
29        os.chdir(wkdir)
30        returnBool[1] = True
31    except OSError:
32        print("Did not find 'split_conformers.pdb' folder. Quitting...")
33        quit()
34    if returnBool[0] & returnBool[1]:
35        return True
36    else:

```

```

35         return False
36
37
38 def GetPDBs():
39     pdbNames = []
40     inFile = open('stationary_points.txt', 'r')
41     for line in inFile:
42         pdbNames.append(line.split()[1])
43     return pdbNames
44
45
46 def CopyPDBs(pdbList):
47     wkdir = os.getcwd()
48     for i in range(len(pdbList)):
49         pstring = ('cp ' + 'split-conformers.pdb/' + str(pdbList[i]) +
50         ' stationary-conformers/')
51         os.popen(pstring)
52
53 def Runner():
54     if IOValidator():
55         print('Valid Args. Running... ')
56         pdbList = GetPDBs()
57         try:
58             os.mkdir('stationary-conformers')
59             CopyPDBs(pdbList)
60         except OSError:
61             print("stationary-conformers' directory already exists.
62 Erase directory and run again. Quitting...")
63             quit()
64
65 Runner()

```

This second file converts the first file into a relative scale from 0 to 1.

```

1#!/usr/bin/python
2
3### Author: Gentry Smith, Oklahoma State University
4### Created: July 31, 2017, 12PM
5### Last Edited: July 31, 2017
6
7### takes file arg with format [ [energy] [pdb_name] ], alters to [ [
8    energy] [torsion] ], and creates copy with
9    [ [relative energy] [torsion] ].
10
11
12
13 def IOValidator():
14     isValid = False
15     try:
16         inFile = sys.argv[1]
17         isValid = True
18     except IOError:

```

```

19     print("Input arg is not a file.\nQuitting...")
20     exit()
21 return isValid
22
23
24 def GetFileData():
25     inData = []
26     inFile = open(sys.argv[1], 'r')
27     iter = 0
28     for line in inFile:
29         inLine = line.split()
30         inData.append(float(inLine[0]))
31         iter = iter + 1
32     inFile.close()
33 return inData
34
35
36 def Relativize(energies):
37     minimum = min(energies)
38     # print("Relativize: minimum=" + str(minimum))
39     newEnergies = []
40     for i in range(len(energies)):
41         # print("Relativize: index=" + str(i))
42         # print("Relativize: energy=" + str(energies[i]))
43         newMin = (float(energies[i]) - float(minimum))
44         # print("Relativize: newMin=" + str(newMin))
45         newEnergies.append((newMin))
46         # print("Relativize: newEnergies=" + str(newEnergies))
47     return newEnergies
48
49
50 def UnifiedScale(energies):
51     # print("unifying scale...")
52     maxi = max(energies)
53     # print("Unify: max=" + str(maxi))
54     newEnergies = []
55     for i in range(len(energies)):
56         # print("Unify: energy=" + str(energies[i]))
57         newEner = (float(energies[i]) / maxi)
58         # print("Unify: scaled energy=" + str(newEner))
59         newEnergies.append(newEner)
60     return newEnergies
61
62
63 def CriticalHit(energies, torsions):
64     isIncreasing = True
65     crits = []
66     tors = []
67     prev = 0
68     for i in range(len(energies)):
69         if (energies[i] == 0):
70             crits.append(energies[i])
71             tors.append(torsions[i])
72         if ((isIncreasing) & (energies[i] < prev)) or ((not

```

```

    isIncreasing) & (energies[i] > prev) ):
73        crits.append(energies[i-1])
74        tors.append(torsions[i-1])
75        isIncreasing = not isIncreasing
76        prev = float(energies[i])
77    returnThing = [crits, tors]
78    return returnThing
79
80
81 def MakeFile(energies, torsions, fileName):
82     outFile = open(fileName, 'w')
83     for i in range(len(energies)):
84         strOut = ('{:.11e}'.format(energies[i]) + " " + str(torsions[i])
85         + "\n")
86         outFile.write(strOut)
87     outFile.close()
88
89 def Runner():
90     if IOValidator():
91         energies = GetFileData()
92         torsions = [180]
93         i = 185
94         while i != 180:
95             if i == 360:
96                 i = 0
97             torsions.append(i)
98             i = i + 5
99         MakeFile(energies, torsions, 'abs_energ.txt')
100        relativeEnergies = Relativize(energies)
101        MakeFile(relativeEnergies, torsions, 'rel_energ.txt')
102        MakeFile(UnifiedScale(relativeEnergies), torsions, 'uni_energ.
103        txt')
104        crits = CriticalHit(relativeEnergies, torsions)
105        MakeFile(crits[0], crits[1], 'crit_pts.txt')
106
107 Runner()

```

This third script compares two generated files using the prior scripts. It can compare the generated absolute energy with the relative energy files. It was often run as a loop through every permutation of the group 4 builder.

```

1 #!/usr/bin/python
2
3 #### Author: Gentry Smith, Oklahoma State University
4 #### Created: July 31, 2017, 3PM
5 #### Last Edited: August 1, 2017
6
7 #### Takes data created by teatAbsEnergies and compares values via
8 #### additive and multiplicative comparison
9 ####      with abs or rel data. Math in terms of File 2 sub/div File 1.
10
11 # sys.argv[1] = file 1, working directory here.
12 # sys.argv[2] = file 2, compared with file 1.

```

```

12
13
14 import sys
15 import numpy
16 import math
17
18 def IOValidator():
19     isValid1 = False
20     isValid2 = False
21     try:
22         inFile1 = open(sys.argv[1])
23         isValid1 = True
24     except IOError:
25         print("Arg File 1 is invalid.")
26         isValid1 = False
27     try:
28         inFile1 = open(sys.argv[2])
29         isValid2 = True
30     except IOError:
31         print("Arg File 2 is invalid.")
32         isValid2 = False
33     if (isValid1 & isValid2 & (sys.argv[1] != sys.argv[2])):
34         print('Valid Args. Running... ')
35         return True
36     else:
37         if (sys.argv[1] == sys.argv[2]):
38             print ('args are indentical. Skipping... ')
39         else:
40             print("Invalid args. Quitting... ")
41         exit()
42
43
44 def ExtractData(data):
45     inFile = open(data, 'r')
46     inData = []
47     inTorsions = []
48     # print('Extracting Data... ')
49     for line in inFile:
50         # print('line=' + str(line))
51         # print('line.split()=' + str(line.split()))
52         # print('line.split()[1]=' + str(line.split()[1]))
53         inData.append(float(line.split()[0]))
54         inTorsions.append(int(line.split()[1]))
55     # print(str(inTorsions))
56     # print('Done. ')
57     return [inData, inTorsions]
58
59
60 def Comparator(data1, data2, func):
61     # func: 0=add, 1=mult
62     newData = []
63     if func == 0:
64         for i in range(len(data2)):
65             newData.append(float(data2[i] - data1[i]))

```

```

66     elif func == 1:
67         for i in range(len(data2)):
68             try:
69                 newData.append(float(data2[i] / data1[i]))
70             except ZeroDivisionError:
71                 newData.append(0.0)
72     return newData
73
74
75 def WriteFile(data1, data2, tors, compData, comp, sigs):
76     # writes data of comparison. Format:
77     #   File1 = {file1}
78     #   File2 = {file2}
79     #   Source: {absolute, relative}
80     #   Comparison: {additive, multiplicative}
81     #   comp: {min/max/avg/stdev of all comp values}
82     #   Raw Data: {includes header of File1, File2, Torsions, Comp
83     # defining each column}
84     # print("Writing file...")
85     # print('File2=' + str((sys.argv[2]).split("/")))
86     source = ""
87     if str(sys.argv[1])[:3] == "abs":
88         source = "absolute"
89     elif str(sys.argv[1])[:3] == "rel":
90         source = "relative"
91     elif str(sys.argv[1])[:3] == "uni":
92         source = "unified relative scale"
93     else:
94         print(str(sys.argv[1])[:2])
95     comparison = ""
96     if comp == 0:
97         comparison = "additive"
98     elif comp == 1:
99         comparison = "multiplicative"
100    headerLines = [0]*10
101    headerLines[0] = ('File1 = ' + sys.argv[1] + '\n')
102    headerLines[1] = ('File2 = ' + sys.argv[2] + '\n')
103    headerLines[2] = ('Source: ' + source + '\n')
104    headerLines[3] = ('Comparison: ' + comparison + '\n')
105    headerLines[4] = ('Comparison min: ' + str(sigs[0]) + '\n')
106    headerLines[5] = ('Comparison max: ' + str(sigs[1]) + '\n')
107    headerLines[6] = ('Comparison avg: ' + str(sigs[2]) + '\n')
108    headerLines[7] = ('Comparison stdev: ' + str(sigs[3]) + '\n')
109    headerLines[8] = ('Raw Data: ' + '\n')
110    f1ColSize = len(str(data1[0]))
111    f2ColSize = len(str(data2[0]))
112    headerLines[9] = ('File1'.ljust(18) + 'File2'.ljust(18) + 'Tors'.
113    ljust(5) + 'Comp'.ljust(18) + '\n')
114    fileName = (str((sys.argv[2]).split("/"))[-2]) + "_" + str(sys.argv
115    [1])[:3] + "_" + comparison + ".txt")
116    outFile = open(fileName, 'w')
117    for i in range(len(headerLines)):
118        outFile.write(str(headerLines[i]))
119    for i in range(len(data1)):

```

```
117     # print('str(tors[i]).ljust(5)=' + str(tors[i]).ljust(5))
118     string = (str(data1[i])[:17].ljust(18) + ' ' + str(data2[i])
119     [:17].ljust(18) + str(tors[i]).ljust(5) + str(compData[i])[:17].
120     ljust(18) + '\n')
121     outFile.write(string)
122
123 def GetCompSigs(data):
124     sigs = []
125     sigs.append(min(data))
126     sigs.append(max(data))
127     sigs.append((float(sum(data))/float(len(data))))
128     sigs.append(numpy.std(data, axis=0))
129
130
131 def Runner():
132     if IOValidator():
133         [data1, torsions1] = ExtractData(sys.argv[1])
134         [data2, torsions2] = ExtractData(sys.argv[2])
135         if (len(data1) == len(data2)) & (len(torsions1) == len(torsions2)):
136            aData = Comparator(data1, data2, 0)
137             aSigs = GetCompSigs(aData)
138             WriteFile(data1, data2, torsions1, aData, 0, aSigs)
139             mData = Comparator(data1, data2, 1)
140             mSigs = GetCompSigs(mData)
141             WriteFile(data1, data2, torsions1, mData, 1, mSigs)
142             print('Complete.')
143 Runner()
```

APPENDIX C

Conformation Landscapes

Listed below are two example Germanium PDB files. The first is for the end-goal hexagermane in the trans-trans-trans conformation with isopropyl groups on the terminal Ge atoms. The second is for the simplified butagermane with fully protonated Germanium atoms.

C.1 Code: hexagermane-transall.pdb

```
1 HEADER
2 REMARK Title: hexagermane_transall system
3 HETATM    1  Ge          1      -4.399   0.008   0.355   0.00   0.00
4           Ge
5 HETATM    2  Ge          1      -1.965   0.138   -0.022   0.00   0.00
6           Ge
7 HETATM    3  C          1      -4.822   1.886   0.961   0.00   0.00
8           C
9 HETATM    4  C          1      -5.008   -1.297   1.715   0.00   0.00
10          C
11 HETATM   5  C          1      -5.256   -0.261   -1.445   0.00   0.00
12          C
13 HETATM   6  C          1      -1.213   1.435   1.157   0.00   0.00
14          C
15 HETATM   7  Ge         1      -0.756   -1.988   0.223   0.00   0.00
16           Ge
17 HETATM   8  C          1      -1.297   -2.917   1.805   0.00   0.00
18           C
19 HETATM   9  Ge         1      1.647   -1.496   0.371   0.00   0.00
20           Ge
21 HETATM  10  C          1      -1.182   -3.010   -1.339   0.00   0.00
22           C
23 HETATM  11  C          1      2.131   -0.425   1.877   0.00   0.00
24           C
25 HETATM  12  C          1      2.111   -0.634   -1.269   0.00   0.00
26           C
27 HETATM  13  Ge         1      2.889   -3.585   0.738   0.00   0.00
28           Ge
29 HETATM  14  C          1      2.287   -4.358   2.378   0.00   0.00
30           C
31 HETATM  15  Ge         1      5.327   -3.386   1.080   0.00   0.00
32           Ge
33 HETATM  16  C          1      2.766   -4.685   -0.813   0.00   0.00
34           C
35 HETATM  17  C          1      5.688   -2.615   2.887   0.00   0.00
36           C
```

20	HETATM	18	C		1	6.239	-2.415	-0.417	0.00	0.00
		C								
21	HETATM	19	C		1	5.893	-5.324	0.888	0.00	0.00
		C								
22	HETATM	20	C		1	-3.527	2.543	1.328	0.00	0.00
		C								
23	HETATM	21	C		1	-5.754	1.844	2.133	0.00	0.00
		C								
24	HETATM	22	H		1	-5.303	2.355	0.072	0.00	0.00
		H								
25	HETATM	23	H		1	-5.269	1.358	2.999	0.00	0.00
		H								
26	HETATM	24	H		1	-6.679	1.287	1.913	0.00	0.00
		H								
27	HETATM	25	H		1	-6.047	2.856	2.449	0.00	0.00
		H								
28	HETATM	26	H		1	-3.043	2.019	2.171	0.00	0.00
		H								
29	HETATM	27	H		1	-3.683	3.585	1.642	0.00	0.00
		H								
30	HETATM	28	H		1	-2.818	2.559	0.490	0.00	0.00
		H								
31	HETATM	29	H		1	-4.336	-1.167	2.589	0.00	0.00
		H								
32	HETATM	30	C		1	-4.907	-2.680	1.154	0.00	0.00
		C								
33	HETATM	31	C		1	-6.417	-0.909	2.051	0.00	0.00
		C								
34	HETATM	32	H		1	-3.858	-2.985	0.982	0.00	0.00
		H								
35	HETATM	33	H		1	-5.433	-2.773	0.187	0.00	0.00
		H								
36	HETATM	34	H		1	-5.349	-3.420	1.836	0.00	0.00
		H								
37	HETATM	35	H		1	-6.488	0.167	2.302	0.00	0.00
		H								
38	HETATM	36	H		1	-6.802	-1.477	2.909	0.00	0.00
		H								
39	HETATM	37	H		1	-7.103	-1.094	1.205	0.00	0.00
		H								
40	HETATM	38	C		1	5.200	-6.127	1.944	0.00	0.00
		C								
41	HETATM	39	C		1	7.384	-5.453	0.966	0.00	0.00
		C								
42	HETATM	40	H		1	5.523	-5.590	-0.126	0.00	0.00
		H								
43	HETATM	41	H		1	7.790	-4.974	1.874	0.00	0.00
		H								
44	HETATM	42	H		1	7.885	-4.994	0.099	0.00	0.00
		H								
45	HETATM	43	H		1	7.691	-6.509	0.992	0.00	0.00
		H								
46	HETATM	44	H		1	5.502	-5.821	2.960	0.00	0.00
		H								

47	HETATM	45	H		1	5.436	-7.197	1.849	0.00	0.00
			H							
48	HETATM	46	H		1	4.106	-6.027	1.879	0.00	0.00
			H							
49	HETATM	47	C		1	6.243	-1.232	2.746	0.00	0.00
			C							
50	HETATM	48	C		1	6.612	-3.524	3.636	0.00	0.00
			C							
51	HETATM	49	H		1	4.684	-2.582	3.376	0.00	0.00
			H							
52	HETATM	50	H		1	7.535	-3.731	3.068	0.00	0.00
			H							
53	HETATM	51	H		1	6.139	-4.497	3.853	0.00	0.00
			H							
54	HETATM	52	H		1	6.913	-3.088	4.599	0.00	0.00
			H							
55	HETATM	53	H		1	7.243	-1.234	2.279	0.00	0.00
			H							
56	HETATM	54	H		1	6.347	-0.742	3.725	0.00	0.00
			H							
57	HETATM	55	H		1	5.589	-0.589	2.128	0.00	0.00
			H							
58	HETATM	56	C		1	5.630	-1.055	-0.555	0.00	0.00
			C							
59	HETATM	57	H		1	6.024	-3.039	-1.315	0.00	0.00
			H							
60	HETATM	58	C		1	7.712	-2.342	-0.145	0.00	0.00
			C							
61	HETATM	59	H		1	7.923	-1.890	0.839	0.00	0.00
			H							
62	HETATM	60	H		1	8.227	-1.728	-0.898	0.00	0.00
			H							
63	HETATM	61	H		1	8.188	-3.335	-0.163	0.00	0.00
			H							
64	HETATM	62	H		1	4.573	-1.106	-0.861	0.00	0.00
			H							
65	HETATM	63	H		1	6.155	-0.455	-1.314	0.00	0.00
			H							
66	HETATM	64	H		1	5.675	-0.486	0.391	0.00	0.00
			H							
67	HETATM	65	H		1	-5.890	-1.163	-1.302	0.00	0.00
			H							
68	HETATM	66	C		1	-4.220	-0.487	-2.505	0.00	0.00
			C							
69	HETATM	67	C		1	-6.093	0.945	-1.729	0.00	0.00
			C							
70	HETATM	68	H		1	-6.841	1.122	-0.939	0.00	0.00
			H							
71	HETATM	69	H		1	-6.644	0.838	-2.676	0.00	0.00
			H							
72	HETATM	70	H		1	-5.478	1.858	-1.818	0.00	0.00
			H							
73	HETATM	71	H		1	-3.754	-1.481	-2.414	0.00	0.00
			H							

74	HETATM	72	H		1	-3.411	0.262	-2.459	0.00	0.00
			H							
75	HETATM	73	H		1	-4.659	-0.429	-3.512	0.00	0.00
			H							
76	HETATM	74	C		1	-1.706	1.681	2.429	0.00	0.00
			C							
77	HETATM	75	C		1	-0.128	2.155	0.679	0.00	0.00
			C							
78	HETATM	76	H		1	0.268	1.941	-0.323	0.00	0.00
			H							
79	HETATM	77	C		1	0.451	3.147	1.465	0.00	0.00
			C							
80	HETATM	78	C		1	-1.134	2.678	3.216	0.00	0.00
			C							
81	HETATM	79	C		1	-0.058	3.415	2.731	0.00	0.00
			C							
82	HETATM	80	H		1	-1.525	2.873	4.219	0.00	0.00
			H							
83	HETATM	81	H		1	1.306	3.716	1.086	0.00	0.00
			H							
84	HETATM	82	H		1	0.391	4.199	3.349	0.00	0.00
			H							
85	HETATM	83	C		1	-1.557	-4.274	1.694	0.00	0.00
			C							
86	HETATM	84	C		1	-0.365	-3.088	-2.455	0.00	0.00
			C							
87	HETATM	85	C		1	-2.027	-4.985	2.796	0.00	0.00
			C							
88	HETATM	86	H		1	-1.378	-4.792	0.742	0.00	0.00
			H							
89	HETATM	87	C		1	-1.446	-2.277	3.025	0.00	0.00
			C							
90	HETATM	88	C		1	-0.752	-3.866	-3.544	0.00	0.00
			C							
91	HETATM	89	H		1	0.592	-2.548	-2.482	0.00	0.00
			H							
92	HETATM	90	C		1	-2.386	-3.699	-1.304	0.00	0.00
			C							
93	HETATM	91	C		1	-2.219	-4.336	4.011	0.00	0.00
			C							
94	HETATM	92	H		1	-2.237	-6.056	2.707	0.00	0.00
			H							
95	HETATM	93	C		1	-1.915	-2.983	4.130	0.00	0.00
			C							
96	HETATM	94	H		1	-1.159	-1.217	3.132	0.00	0.00
			H							
97	HETATM	95	C		1	-2.771	-4.484	-2.388	0.00	0.00
			C							
98	HETATM	96	H		1	-3.043	-3.610	-0.422	0.00	0.00
			H							
99	HETATM	97	C		1	-1.952	-4.568	-3.509	0.00	0.00
			C							
100	HETATM	98	H		1	-0.105	-3.928	-4.425	0.00	0.00
			H							

101	HETATM	99	H		1	-3.721	-5.027	-2.358	0.00	0.00
			H							
102	HETATM	100	H		1	-2.253	-5.182	-4.364	0.00	0.00
			H							
103	HETATM	101	H		1	-2.596	-4.891	4.876	0.00	0.00
			H							
104	HETATM	102	H		1	-2.041	-2.474	5.091	0.00	0.00
			H							
105	HETATM	103	C		1	2.487	-3.679	3.571	0.00	0.00
			C							
106	HETATM	104	C		1	1.701	-5.563	-0.935	0.00	0.00
			C							
107	HETATM	105	C		1	3.733	-4.618	-1.807	0.00	0.00
			C							
108	HETATM	106	H		1	0.940	-5.615	-0.140	0.00	0.00
			H							
109	HETATM	107	C		1	1.598	-6.382	-2.057	0.00	0.00
			C							
110	HETATM	108	C		1	1.690	-5.609	2.382	0.00	0.00
			C							
111	HETATM	109	C		1	2.102	-4.259	4.776	0.00	0.00
			C							
112	HETATM	110	H		1	2.956	-2.680	3.567	0.00	0.00
			H							
113	HETATM	111	C		1	1.520	-5.523	4.784	0.00	0.00
			C							
114	HETATM	112	H		1	2.260	-3.721	5.716	0.00	0.00
			H							
115	HETATM	113	C		1	1.311	-6.197	3.585	0.00	0.00
			C							
116	HETATM	114	H		1	1.504	-6.131	1.431	0.00	0.00
			H							
117	HETATM	115	C		1	2.562	-6.313	-3.057	0.00	0.00
			C							
118	HETATM	116	H		1	0.754	-7.074	-2.153	0.00	0.00
			H							
119	HETATM	117	C		1	3.630	-5.430	-2.933	0.00	0.00
			C							
120	HETATM	118	H		1	4.590	-3.931	-1.700	0.00	0.00
			H							
121	HETATM	119	H		1	4.391	-5.376	-3.718	0.00	0.00
			H							
122	HETATM	120	H		1	2.481	-6.954	-3.941	0.00	0.00
			H							
123	HETATM	121	H		1	1.223	-5.984	5.731	0.00	0.00
			H							
124	HETATM	122	H		1	0.844	-7.187	3.587	0.00	0.00
			H							
125	HETATM	123	C		1	1.878	0.732	-1.306	0.00	0.00
			C							
126	HETATM	124	C		1	1.530	-0.534	3.120	0.00	0.00
			C							
127	HETATM	125	C		1	2.642	-1.289	-2.370	0.00	0.00
			C							

128	HETATM	126	C		1	2.179	1.455	-2.458	0.00	0.00
		C								
129	HETATM	127	H		1	1.444	1.239	-0.432	0.00	0.00
		H								
130	HETATM	128	C		1	3.179	0.461	1.679	0.00	0.00
		C								
131	HETATM	129	C		1	2.005	0.227	4.186	0.00	0.00
		C								
132	HETATM	130	H		1	0.661	-1.197	3.265	0.00	0.00
		H								
133	HETATM	131	C		1	2.940	-0.568	-3.524	0.00	0.00
		C								
134	HETATM	132	H		1	2.840	-2.370	-2.334	0.00	0.00
		H								
135	HETATM	133	C		1	2.710	0.804	-3.567	0.00	0.00
		C								
136	HETATM	134	H		1	1.989	2.533	-2.491	0.00	0.00
		H								
137	HETATM	135	H		1	3.358	-1.081	-4.396	0.00	0.00
		H								
138	HETATM	136	H		1	2.944	1.370	-4.475	0.00	0.00
		H								
139	HETATM	137	C		1	3.067	1.105	3.998	0.00	0.00
		C								
140	HETATM	138	H		1	1.534	0.140	5.170	0.00	0.00
		H								
141	HETATM	139	C		1	3.650	1.229	2.740	0.00	0.00
		C								
142	HETATM	140	H		1	3.633	0.553	0.682	0.00	0.00
		H								
143	HETATM	141	H		1	4.480	1.926	2.585	0.00	0.00
		H								
144	HETATM	142	H		1	3.439	1.703	4.836	0.00	0.00
		H								
145	HETATM	143	C		1	-2.039	0.838	-1.804	0.00	0.00
		C								
146	HETATM	144	C		1	-1.525	0.195	-2.916	0.00	0.00
		C								
147	HETATM	145	C		1	-2.655	2.077	-1.927	0.00	0.00
		C								
148	HETATM	146	C		1	-1.618	0.802	-4.168	0.00	0.00
		C								
149	HETATM	147	H		1	-1.048	-0.789	-2.818	0.00	0.00
		H								
150	HETATM	148	C		1	-2.746	2.686	-3.175	0.00	0.00
		C								
151	HETATM	149	H		1	-3.084	2.566	-1.036	0.00	0.00
		H								
152	HETATM	150	C		1	-2.223	2.047	-4.296	0.00	0.00
		C								
153	HETATM	151	H		1	-1.210	0.296	-5.049	0.00	0.00
		H								
154	HETATM	152	H		1	-3.229	3.663	-3.275	0.00	0.00
		H								

155	HETATM	153	H		1	-2.292	2.524	-5.279	0.00	0.00
		H								
156	HETATM	154	H		1	-2.539	1.081	2.827	0.00	0.00
		H								
157	CONECT	3	1	20	21	22				
158	CONECT	4	1	29	30	31				
159	CONECT	5	1	65	66	67				
160	CONECT	6	74	75	2					
161	CONECT	8	83	87	7					
162	CONECT	10	84	90	7					
163	CONECT	11	124	128	9					
164	CONECT	12	123	125	9					
165	CONECT	14	103	108	13					
166	CONECT	16	104	105	13					
167	CONECT	17	15	47	48	49				
168	CONECT	18	56	57	58	15				
169	CONECT	19	15	38	39	40				
170	CONECT	20	3	26	27	28				
171	CONECT	21	3	23	24	25				
172	CONECT	30	4	34	32	33				
173	CONECT	31	4	35	36	37				
174	CONECT	38	19	44	45	46				
175	CONECT	39	19	41	42	43				
176	CONECT	47	54	55	17	53				
177	CONECT	48	17	50	51	52				
178	CONECT	56	62	63	64	18				
179	CONECT	58	59	60	61	18				
180	CONECT	66	71	72	73	5				
181	CONECT	67	68	69	70	5				
182	CONECT	74	78	154	6					
183	CONECT	75	76	77	6					
184	CONECT	77	75	79	81					
185	CONECT	78	74	79	80					
186	CONECT	79	77	78	82					
187	CONECT	83	85	86	8					
188	CONECT	84	88	89	10					
189	CONECT	85	83	91	92					
190	CONECT	87	93	94	8					
191	CONECT	88	84	97	98					
192	CONECT	90	95	96	10					
193	CONECT	91	85	93	101					
194	CONECT	93	87	91	102					
195	CONECT	95	90	97	99					
196	CONECT	97	88	95	100					
197	CONECT	103	109	110	14					
198	CONECT	104	106	107	16					
199	CONECT	105	117	118	16					
200	CONECT	107	104	115	116					
201	CONECT	108	113	114	14					
202	CONECT	109	103	111	112					
203	CONECT	111	109	113	121					
204	CONECT	113	108	111	122					
205	CONECT	115	107	117	120					
206	CONECT	117	105	115	119					

207	CONECT	123	126	127	12
208	CONECT	124	129	130	11
209	CONECT	125	131	132	12
210	CONECT	126	123	133	134
211	CONECT	128	139	140	11
212	CONECT	129	124	137	138
213	CONECT	131	125	133	135
214	CONECT	133	126	131	136
215	CONECT	137	129	139	142
216	CONECT	139	128	137	141
217	CONECT	143	144	145	2
218	CONECT	144	143	146	147
219	CONECT	145	143	148	149
220	CONECT	146	144	150	151
221	CONECT	148	145	150	152
222	CONECT	150	146	148	153
223	CONECT	1	2	3	4
224	CONECT	2	1	143	6
225	CONECT	7	2	8	9
226	CONECT	9	7	11	12
227	CONECT	13	9	14	15
228	CONECT	15	13	17	18
229	CONECT	22	3		
230	CONECT	23	21		
231	CONECT	24	21		
232	CONECT	25	21		
233	CONECT	26	20		
234	CONECT	27	20		
235	CONECT	28	20		
236	CONECT	29	4		
237	CONECT	32	30		
238	CONECT	33	30		
239	CONECT	34	30		
240	CONECT	35	31		
241	CONECT	36	31		
242	CONECT	37	31		
243	CONECT	40	19		
244	CONECT	41	39		
245	CONECT	42	39		
246	CONECT	43	39		
247	CONECT	44	38		
248	CONECT	45	38		
249	CONECT	46	38		
250	CONECT	49	17		
251	CONECT	50	48		
252	CONECT	51	48		
253	CONECT	52	48		
254	CONECT	53	47		
255	CONECT	54	47		
256	CONECT	55	47		
257	CONECT	57	18		
258	CONECT	59	58		
259	CONECT	60	58		
260	CONECT	61	58		

```
261 CONECT  62   56
262 CONECT  63   56
263 CONECT  64   56
264 CONECT  65    5
265 CONECT  68   67
266 CONECT  69   67
267 CONECT  70   67
268 CONECT  71   66
269 CONECT  72   66
270 CONECT  73   66
271 CONECT  76   75
272 CONECT  80   78
273 CONECT  81   77
274 CONECT  82   79
275 CONECT  86   83
276 CONECT  89   84
277 CONECT  92   85
278 CONECT  94   87
279 CONECT  96   90
280 CONECT  98   88
281 CONECT  99   95
282 CONECT 100   97
283 CONECT 101   91
284 CONECT 102   93
285 CONECT 106  104
286 CONECT 110  103
287 CONECT 112  109
288 CONECT 114  108
289 CONECT 116  107
290 CONECT 118  105
291 CONECT 119  117
292 CONECT 120  115
293 CONECT 121  111
294 CONECT 122  113
295 CONECT 127  123
296 CONECT 130  124
297 CONECT 132  125
298 CONECT 134  126
299 CONECT 135  131
300 CONECT 136  133
301 CONECT 138  129
302 CONECT 140  128
303 CONECT 141  139
304 CONECT 142  137
305 CONECT 147  144
306 CONECT 149  145
307 CONECT 151  146
308 CONECT 152  148
309 CONECT 153  150
310 CONECT 154   74
311 END
```

The above molecule contains 154 atoms and 153 bonds, making it extremely computationally expensive for regular QM calculations. This made utilizing the large

molecule as a trial system unreasonable due to the prohibitively long computation time for each conformation, assuming the conformation calculation would complete at all.

The below PDB file is the simplified butagermane with fully protonated Germanium atoms. As a significantly smaller system with only 14 atoms and 13 bonds, the relatively short computation time allowed the trial system to move with relative ease.

C.2 Code: ge4h.pdb

```

1 COMPND      UNNAMED
2 AUTHOR      GENERATED BY OPEN BABEL 2.3.90
3 HETATM      1  GE   UNL    1     -3.520   1.842  -0.078   1.00   0.00
4           Ge3-
5 HETATM      2  GE   UNL    1     -1.368   2.888  -0.034   1.00   0.00
6           Ge2-
7 HETATM      3  GE   UNL    1      0.324   1.200   0.059   1.00   0.00
8           Ge3-
9 HETATM      4  GE   UNL    1      2.475   2.248   0.099   1.00   0.00
10          Ge
11 HETATM     5   H   UNL    1     -4.622   2.930  -0.135   1.00   0.00
12          H
13 HETATM     6   H   UNL    1     -3.699   0.985   1.202   1.00   0.00
14          H
15 HETATM     7   H   UNL    1     -3.621   0.932  -1.328   1.00   0.00
16          H
17 HETATM     8   H   UNL    1     -1.258   3.797   1.217   1.00   0.00
18          H
19 HETATM     9   H   UNL    1     -1.178   3.740  -1.314   1.00   0.00
20          H
21 HETATM    10   H   UNL    1      0.213   0.288  -1.189   1.00   0.00
22          H
23 HETATM    11   H   UNL    1      0.135   0.352   1.342   1.00   0.00
24          H
25 HETATM    12   H   UNL    1      2.655   3.095  -1.186   1.00   0.00
26          H
27 HETATM    13   H   UNL    1      3.578   1.161   0.165   1.00   0.00
28          H
29 HETATM    14   H   UNL    1      2.574   3.167   1.343   1.00   0.00
30          H

17 CONECT     1     2     5     6     7
18 CONECT     2     1     3     8     9
19 CONECT     3     2     4    10    11
20 CONECT     4     3    12    13    14
21 CONECT     5     1
22 CONECT     6     1
23 CONECT     7     1
24 CONECT     8     2
25 CONECT     9     2
26 CONECT    10     3
27 CONECT    11     3
28 CONECT    12     4
29 CONECT    13     4
30 CONECT    14     4

```

31	MASTER	0	0	0	0	0	0	0	14	0	14	0
32	END											

C.3 Progress on Torsion Minimizer System

While incomplete and largely nonfunctioning, this code is the current progress toward the implementation of the torsion minimizer system as outlined in 4.2.

```

1  ### Author: Gentry Smith
2  ### Date: April 22, 2017
3  ### Description: This is the runner file that is the primary executable
4  ###      for the torsion minimizer. Currently is the
5  ###      only file utilized.
6
7  # Inputs:
8  # Arg1: the molecule file to be minimized (currently only accepts a pdb
9  #       file)
10
11 import sys
12 import subprocess
13 import math
14
15 # IO Validator: validates user-submitted molecule.
16 def IOValidator():
17     isValid = False
18     # Check for valid length of args (2)
19     if len(sys.argv) == 2:
20         # Check arg to make sure it's a file.
21         argFile = sys.argv[1]
22         try:
23             inputFile = open(argFile)
24             # Finally, make sure the file is a .pdb
25             if inputFile[-4:] == ".pdb":
26                 isValid = True
27             else:
28                 print("This is not a .pdb file. Please try again with a
29 .pdb file.\n")
30                 inputFile.close()
31             except IOError:
32                 print("System was not able to open '", str(argFile), "'.")
33             # too long
34             elif len(sys.argv) > 2:
35                 print("You have too many arguments. Call the file as 'Runner.py
36 [molecule file]' and try again.\n")
37             # too short
38             else:
39                 print("You do not have enough arguments. Start the program as '
40 Runner.py [molecule file]' and try again.\n")
41             # return validity boolean
42             return isValid
43
44 # Get Torsions: initiates function to get user-specified torsion bonds.
45 # Returns bonds as int[[a,b],[a,b]] list
46 def getTorsions():
47     torsions = [[0, 0]]

```

```

42     newTorsion = "first"
43     firstTime = True
44     doneCheck = ""
45     badIn = False
46
47     # loop for all torsions until user types "done"
48     while newTorsion != "":
49         if firstTime:
50             print("It's time to define the torsions of the molecule and")
51             print("declare which bonds you would like to rotate.\n")
52             print("Before going any further, it's important to note at")
53             print("this time that version 0.2 (current) will assume the torsions you")
54             print("enter are completely correct. You'll see a bunch of error messages")
55             print("soon if it isn't correct.\n")
56             print("Open the .pdb file and identify the numbers of the")
57             print("atoms on the .pdb that will make the bond (the first number on the")
58             print("line of each atom)\n\n")
59             print("Now it's time to enter in the numbers of the two")
60             print("atoms. We'll do it one at a time.")
61
62             firstTor = raw_input("Type in the number of the first atom")
63             in the bond and hit enter. \nEx: type 3 and then hit enter.\n")
64
65             try:
66                 confFirstTor = int(firstTor)
67             except ValueError:
68                 print("You typed in '", firstTor, "'", which is not a")
69                 print("number. Let's start again.")
70                 badIn = True
71
72             secondTor = raw_input("Type in the number of the second atom")
73             in the bond and hit enter. \nEx: type 3 and then hit enter.\n")
74
75             try:
76                 confSecondTor = int(secondTor)
77             except ValueError:
78                 print("You typed in '", secondTor, "'", which is not a")
79                 print("number. Let's start again.")
80                 badIn = True
81                 firstTime = False
82
83             else:
84                 print("Open the .pdb file and identify the numbers of the")
85                 print("atoms on the .pdb that will make the bond (the first number on the")
86                 print("line of each atom)\n\n")
87
88             firstTor = raw_input("Type in the number of the first atom")
89             in the bond and hit enter. \nEx: type 3 and then hit enter.\n")
90
91             try:
92                 confFirstTor = int(firstTor)
93             except ValueError:
94                 print("You typed in '", firstTor, "'", which is not a")
95                 print("number. Let's start again.")

```

```

81         badIn = True
82
83     secondTor = raw_input("Type in the number of the second atom
84     in the bond and hit enter. \nEx: type 3 and then hit enter.\n")
85
86     try:
87         confSecondTor = int(secondTor)
88     except ValueError:
89         print("You typed in '", secondTor, "' , which is not a
90     number. Let's start again.")
91         badIn = True
92         firstTime = False
93
94     if badIn == False:
95         newTorsion = [confFirstTor, confSecondTor]
96         if torsions == [[0, 0]]:
97             print("You added a new torsion: ", newTorsion, "\n")
98             torsions = newTorsion
99         else:
100             torsions.append(newTorsion)
101             print("The current torsions you have created are:\n")
102             for each in torsions:
103                 print(each, "\n")
104             doneCheck = raw_input("If you would like to add another
105             torsion, press enter. If you are finished adding torsions, type
106             'done' and press enter\n")
107
108             if str(doneCheck) == "done":
109                 print("Finished entering torsions. Beginning the work.\n")
110             else:
111                 newTorsion = "first"
112
113
114     return torsions
115
116 # Get Conformation Count: determines conformations needed. Returns list
117 # in form: [#conf, rotDeg, rotRng]
118 def getConformationInfo(depth, torsions):
119     # rotates 60 degrees on the first search, then logarithmic decrease
120     # from 10 for each subsequent search.
121     rotDeg = [60, 10]
122     # full torsion range for first search, logarithmic decrease from 50
123     # for each subsequent search
124     rotRng = [360, 50]
125     # number of conformations needed
126     numConf = 0
127     # degrees per rotation
128     deg = 0
129     # rotation range

```

```

127 rng = 0
128 # number of rotations per torsion
129 rotTick = 0
130
131 # determine counts from depth
132 if depth >= 2:
133     deg = math.pow(10, (2-depth))
134     rng = deg*5
135 elif depth <2:
136     deg = rotDeg[depth]
137     rng = rotRng[depth]
138 if depth == 1:
139     rotTick = 6
140 elif depth >= 1:
141     rotTick = 11
142
143 numConf = math.pow(torsions, rotTick)
144
145 return [numConf, deg, rng]
146
147
148
149
150
151
152
153
154 def Launcher():
155     valid = IOValidator()
156     if valid:
157         # do everything
158         depth = 0
159
160         InitWD()
161
162     else:
163         print("There was a problem while reading in the molecule file.
164             Please try again.\n")
165         exit()
166
167
168 # Initiates proper working directory.
169 def InitWD():
170
171
172
173
174 # Recursive search through molecule torsions
175 def RecursiveSearch(depth):
176
177     torsions = getTorsions()
178
179

```

180
181 Launcher()

VITA

Gentry H. Smith

Candidate for the Degree of

Master of Science

Thesis: EXPLORING CRITICAL CONFORMATIONS

Major Field: Chemistry

Biographical:

Personal Information: Born in Olathe, KS in November 1993.

Education:

Received a Bachelors of Science in Chemistry at Southern Nazarene University in May 2016.

Completed the requirements for the degree of Master of Science with a major in Chemistry at Oklahoma State University in December 2018.

Experience:

Teaching Assistant, various undergraduate chemistry courses, Southern Nazarene University, Aug. 2014 - May 2016

Graduate Teaching Assistant, CHEM 1314, 1414 at Oklahoma State University, Aug. 2016 - Dec. 2018

Professional Affiliations:

American Chemical Society

Awards

Colonel Andre Whitely Scholarship in Chemistry