EXPLORING CRITICAL CONFORMATIONS: STATE SEARCHING AND SAMPLING IN BOTH GERMANIUM CHAINS AND ICE

By

GENTRY H. SMITH

Bachelor of Science in Chemistry

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

Thesis Approved:

Dr. Christopher J. Fennell

Thesis Advisor

Dr. Jindal K. Shah

Dr. Jimmie D. Weaver

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: CHEMISTRY

Abstract: Molecular conformation plays a critical role in the properties of systems in both the condensed or vapor states. The ensemble of conformations dictates structural properties, average energy, 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

LIST OF TABLES

Table Page

LIST OF FIGURES

Figure Page

CHAPTER I

Introduction

I.1 Computational Chemistry: Chemistry on the Computer

For nearly a century, computational methods have greatly assisted chemists in their efforts of research and discovery. Five computational chemists have been awarded the Nobel Prize in Chemistry. Laureates include Walter Kohn and John Pople in 1998 and Martin Karplus, Michael Levitt, and Arieh Warshel in 2013. Since the early 1960s, chemists have specialized in using computer systems to solve chemical problems.

Computational chemistry is now recognized as its own field rather than a subspecialty within physical chemistry as computational chemists continue to develop efficient methods to calculate large and complex simulations. 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.[3,4,5] This introduction serves to introduce necessary background information generally relevant to the methods developed and utilized in the following chapters.

I.2 Relevant Computational Methods

Analytical descriptions of molecular systems are ideal simulation goals as they provide a complete description of a process. However, it is often impossible to provide analytic solutions for complex systems. This complexity usually drives numer-

ical approaches to instead approximate chemical systems of interest. While not exact, these numerical approximations can produce values consistent with experimental data. Usually limited by the size of a system, multiple numerical methods exist to analytically solve or closely approximate a system by way of solving or approximating the quantum mechanical wave function. Methods relevant to this work include *ab initio*, density functional, semi-empirical, and Monte Carlo methods. 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 cases like the dihydrogen cation, due to the electron-electron correlation term being situationally dependent.[6] Here, we will focus on systems of such complexity that numerical approaches will be of greatest interest.

I.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 exactly for many-body systems, different methods use different approximations to balance between accuracy of the approximation and efficiency of computation.

I.2.1.1 *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 in Chemistry in 1998 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.[7] 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 decreased 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.

I.2.1.2 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.[8] These simplifications are 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. DFT methods exist as pure DFT methods or as hybrid functional methods. Pure DFT methods excel in computing systems much more ef-

ficiently than with HF methods, but at the cost of accuracy. These pure functionals do not rely as much on HF terms and instead use a more general expression. Hybrid functional methods act as DFT methods but with the inclusion of HF terms that require additional computation. Both DFT and hybrid functional methods use an exchange and correlation part.[8] The exchange part attempts to fix density problems from DFT approximations while the correlation parts fixes electron correlation problems including two electrons of identical spin occupying the same position.

One pure DFT method used in this work is BLYP, which utilizes the Becke exchange with the Lee-Yang-Parr correlation part.[9] 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.[10] The PBE method, developed by Perdew, Burke, and Ernzerhof, is another method with similar levels of accuracy to B3LYP that attempts to increase the number of HF-exchanged functionals.[11]

I.2.1.3 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 systems with hundreds of atoms.[12] Additionally, various semi-empirical methods have been designed to produce results with close accuracies to specific sets of experimental data. Two methods used in this work, AM1[13] and

PM3,[14] reproduce well heats of formation, dipole moments, ionization potentials, and structural geometries. Unlike the other methods described so far, basis sets are not used at all in the calculation of energies and properties.

I.2.1.4 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 with atomic orbitals or plane waves, this work focus primarily on basis sets designed with 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,[15] function as a linear combination of atomic orbitals (LCAO) adopted as a molecular orbital. STOs notably exhibit similar features as Schrödinger-based orbitals, excepting that STOs have no radial nodes.

Gaussian-type orbitals (GTOs), introduced by S. Francis Boys in 1950,[4] also function as orbitals in the LCAO method. GTOs are similar to STOs in premise, but have further reduced realism when compared to Schrödinger-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. 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.[3] 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 approximation 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 was introduced by John Pople as the X-YZg form and are commonly referred to as Pople basis sets.[16] These follow the form that each orbital basis function is comprised of X Gaussians. The Y and Z represent an additional linear combination of Gaussian functions made of Y and Z Gaussians 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 expand 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.

I.2.2 Monte Carlo Molecular Modeling

Another method of simulating chemical systems is known as Monte Carlo methods, or MC. While not named until the 1950s, MC methods were first seen in the 18^{th} century thought experiment Buffon's needle.[17] 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 the of the lines, h, by the n needles as $P = \frac{h}{n}$, the approximation can be expressed as $\pi = \frac{2l*n}{th}$.

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 Electronic Numerical Integrator and Computer (ENIAC) 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.[18] Later dubbed Markov Chain Monte Carlo (MCMC) sampling, this allowed for random sampling to instead become a virtual statistically-appropriate sampling method. At the most common level, MC methods apply probabilistic forces to a random interaction to generate a numeric approximation. Eventually published in 1949 by Metropolis and Ulam, this introduced MC methods to chemical simulation packages.[19]

I.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 its successor Electronic Discrete Variable Automatic Computer (EDVAC) 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 2012, this cluster collectively offers the computing power of 3048 cores and 8576 GB of RAM, totaling 48.8 trillion FLoating point Operations Per Second (Tera FLOPS or TFLOPS).

I.4 Software

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

I.4.1 Programs

While chemical 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, was the first popular *ab initio* computation program. Released as Gaussian 70 in 1970, it has received regular updates and capability expansions, 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 influence in the computational community for being one of the oldest packages around.

In addition to Gaussian, many other chemical computational packages exist. Two additional packages used in this work are GAMESS,[5] a package also in active development since the 1970s led by Mark Gordon, and NWChem,[20] a popular open source package developed by Pacific Northwest National Laboratory since the late 2000s.

Once a set of calculations 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.[21] UCSF Chimera, developed by the Resource for Biocomputing, Visualization, and Informatics (RBVI) at the University of California, San Francisco, focuses on more advanced represen-

tations 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.[22]

I.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 allow 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 that are compiled into machine code. Because these tools directly interact with hardware to complete an immense number of calculations, efficiency is key.

One language almost exclusively used in this work is Python.[23] 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 language, Python is not compiled for specific hardware like code written in 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[24] and SciPy[25] 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[26] 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 an automated function that can operate as a tool within a tool. One aspect of this is abstracting out methods and basis sets to that of a computational requirement and level of accuracy, which will be discussed in chapter IV.

CHAPTER II

On Algorithms for Building and Sampling Disordered Crystal States

II.1 States and Properties of Ice

II.1.1 Bernal-Fowler Ice Rules

First described in John Desmond Bernal and Ralph H. Fowler's 1933 paper, the Bernal-Fowler Ice Rules are the foundational observations of how water molecules interact in an ice structure.[27] Although a bent, divalent molecule, water possesses an electronic tetrahedral structure that allows for four interactions on each molecule. The two protons each allow for a hydrogen bond with a lone pair from a neighboring oxygen atom. Similarly, the oxygen atom's two lone pairs each allow for a hydrogen bond with a neighboring proton. While a hydrogen bond is typically defined as an attractive interaction between a proton and one lone pair of electrons on Nitrogen, Oxygen, or Fluorine, this work restricts the definition to a computational implication. Here, a hydrogen bond refers to the space between two oxygen atoms in a crystal where exactly one proton and lone pair are directed toward one another according to Bernal-Fowler ice rules. 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. 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, once hydrogen bonded, each of the four attached water molecules can occupy one of three rotational positions. Including the 6 orientations of the central water, 486

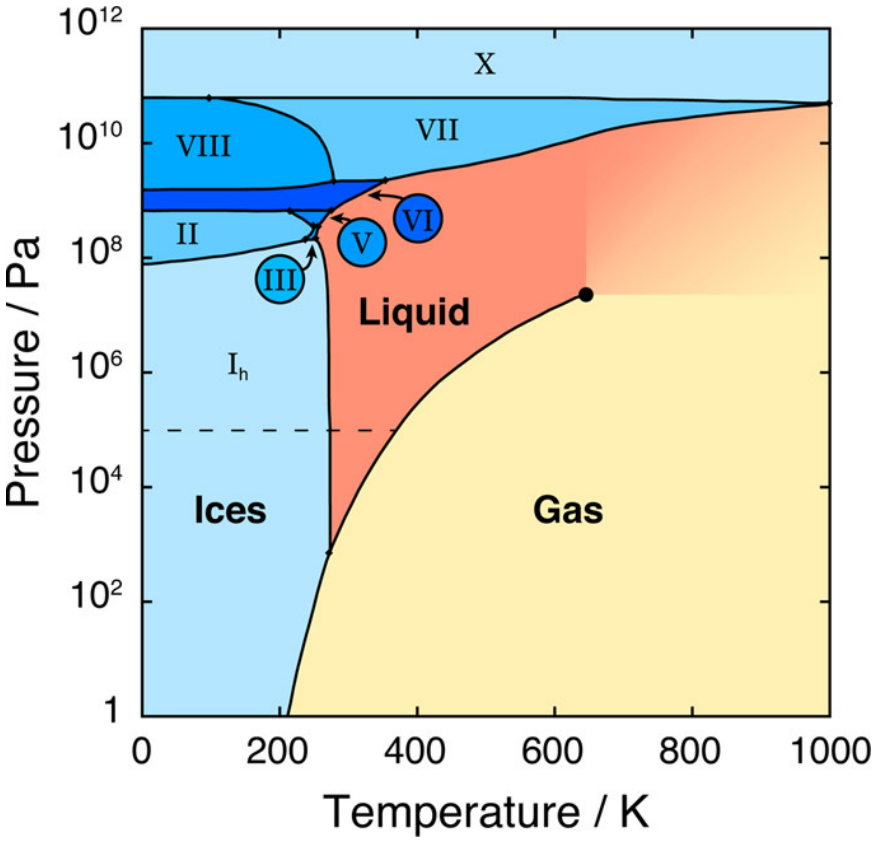

Figure II.1: Water phase diagram. Taken from Brini et al.[1]

microstates exist from these five waters.

II.1.2 Forms of Ice

While ubiquitous in the 'I_h' form, ice water has many phases. As of the writing of this work, there are 18 experimentally established forms of ice. These forms usually occur in cubic, hexagonal, and orthorhombic crystal structures. As can be seen in figure II.1, the system pressure and temperature are primary characteristics of which phase will form. The subject of this work will be on the proton-ordered orthorhombic ice XI and its proton-disordered isomer, ice I_h.

II.1.3 Ice I_h

Ice I_h naturally forms at temperatures below 273.15 K at pressures in the 1 Pa to 100 MPa range,[28] with some temperature curving off into the vapour and liquid phases

at very high and very low pressures as seen in figure II.1. As the most commonly found form on earth, ice I_h is the most relevant form for computational studies involving ice systems.

As famously discussed by Linus Pauling, hexagonal ice water contains a residual entropy at very low temperatures.[29] This residual entropy in ice goes according to Boltzmann's entropy equation $S = K_B LnW$ where $W = (\frac{3}{2})^N$ for N molecules in the crystal. At near absolute zero temperatures, the residual entropy will not reach zero as the disordered water could settle into one of many microstates that fit the "disordered" description. Pauling additionally predicted that an ice structure with perfectly ordered protons may exist at sufficiently low temperatures with zero residual entropy.

II.1.4 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 of water molecules. 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. 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. The interaction of proton with proton or lone pair with lone pair are not hydrogen bonds and are considered defects in the lattice. These are known as Bjerrum defects and referred as D with two protons or L with

two lone pairs interacting.[30] Conversely, hydrogen bonding does not occur if Bjerrum L or D defects occur between the oxygens. 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 crystal, leading to stability problems while running simulations. In generating the crystal, the cause of these defects must be considered and countered effectively. While other stable hydrogen bonding structures may exist, they would either break the Bernal-Fowler ice rules or alter the structure away from the specified form.

II.2 Method Design

II.2.1 Method Tools and Information Management

The primary objective 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.

Python was chosen as the programming language of the tool due to its versatility and the ease of development due to the "pseudocode" written style and the availability of scientific packages including SciPy and NumPy. Python version 2.7 was specifically utilized. Crystal files where 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. An example of this is provided in Appendix A.

To create an ice XI .pdb file, an ice XI cell of eight water molecules can be tiled to create a sufficiently large crystal. The primarily used crystal consists of a 3 x 3 x 6 cell repetition totaling 432 water molecules.

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. This is functionally identitical to the .pdb format data, but compresses the data across multidimensional arrays for iterative use.

Identifying the neighboring molecules proved computationally difficult. The most effective method is to find the closest four molecules by computing a distance between every two oxygen atoms. This ensures every molecule is considered, but also presents significant hurdles. First, a distance calculation utilizes a computationally-inefficient square root calculation. The inefficiency lies in the binary-based command for calculating a square root that often utilizes either a logarithmic solution or a Newtonian approximation that typically requires 16-64 processor cycles. This square root computation can be entirely bypassed by instead comparing the squared-distance between molecules and finding the lowest values. These squared-distances scale identically to the square root value for all distances greater than one, which is true for the ice XI structures sampled in this work.

Second, molecules positioned along the sides will not have four neighbors in a 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.

Once these four neighboring oxygen atoms have been discovered for each water, the four hydrogen-bond interactions according for Bernal-Fowler ice rules with the neighbors describe an orientation defined by the location of each water's protons and lone pairs located at coordinates called tetrahedral positions.

An important aspect of pseudorandom selection is the existence of a bank of op-

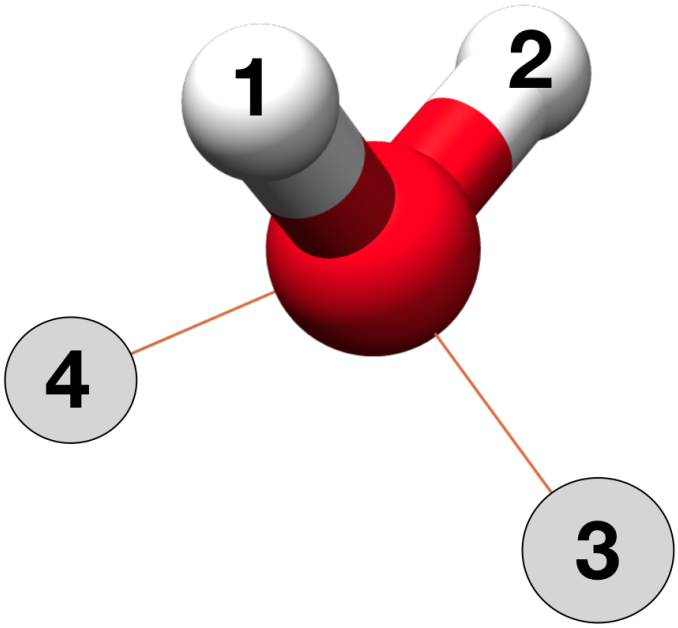

Figure II.2: Example Tetrahedral positions of a water molecule. The two spheres represent potential proton positions roughly occupied by lone pairs.

tions. Utilizing the ingestion portion of the tool to calculate and store all orientational possibilities proves effective for tracking position options. In this work, tetrahedral positions are defined as the four positions that a proton may occupy about a water molecule as the four electron groups extend from the oxygen. For each water molecule, the first two tetrahedral positions are defined 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. Prior to rotation, the third and fourth positions are occupied by lone pairs. According to the .pdb file style, though, lone pairs are implied from the atom data and are not explicitly stated in the file data. This allows for passive relocation of the lone pairs by redefining the proton positions about the water. A visualization of these four tetrahedral positions, two read and two generated, are shown and labeled in figure II.2.

In a equally-repulsed tetrahedral molecule, electron group angles are 109.5°. This method does not produce an exactly correct tetrahedral position of potential hy-

drogen atoms due to the slightly acute104.5° 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.

II.2.2 Pseudorandom Rearrangement of Water Molecules and Generation of Bjerrum Defects

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. The act of rotating each proton about the corresponding oxygen atom in a crystal is as simple as iterating through and pseudorandomly selecting two tetrahedral positions from each water for protons to occupy. The new position data is saved to a new crystal array file similar to the parent generated during the initial file read. These new positions are determined sequentially and "instantaneously" in the time-independent manipulation of the crystal. An important note is that this rearrangement does not consider the orientations of neighboring molecules and likely introduces Bjerrum defects. The likelihood of a defect-free interaction lattice forming is nearly zero and is presumed to have a large number of defects within the lattice. For example, the first molecule reoriented will have a $\frac{5}{6}$ chance of containing a defect.

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 interacting tetrahedral positions between two neighboring waters and confirming that they do not both contain or lack a hydrogen atom. Each water maintains a count of how many defects are present among the four positions, which can be collectively averaged for a per-molecule defect average. Likewise, these defects can be summed and halved to produce a total number of defects in the crystal. Each molecule holding its own defect count allows for contextual changes during the correction step.

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 because of the highest-density entropy introduced into the system. These 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

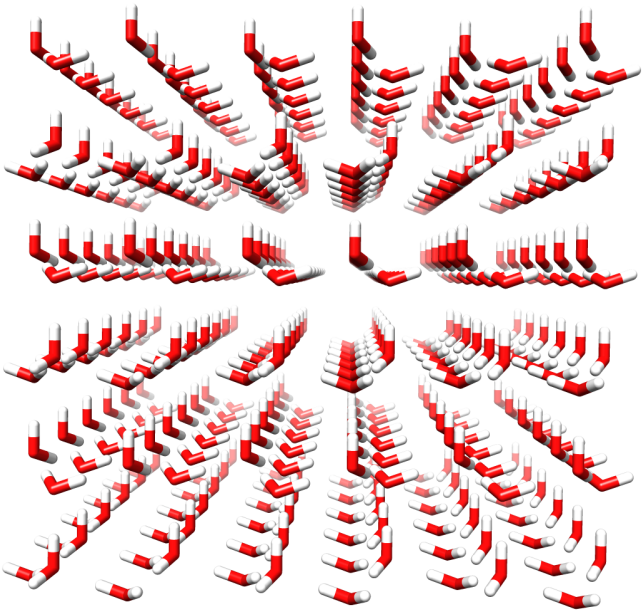

Figure II.3: "Before" image of Ice XI

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. The memory overflow is due to the infinite recursion instead of repeatedly storing new crystal data.

II.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 II.3 and II.4. As can be seen, the "after" image has experienced rotation and can no longer be classified as ice XI. 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

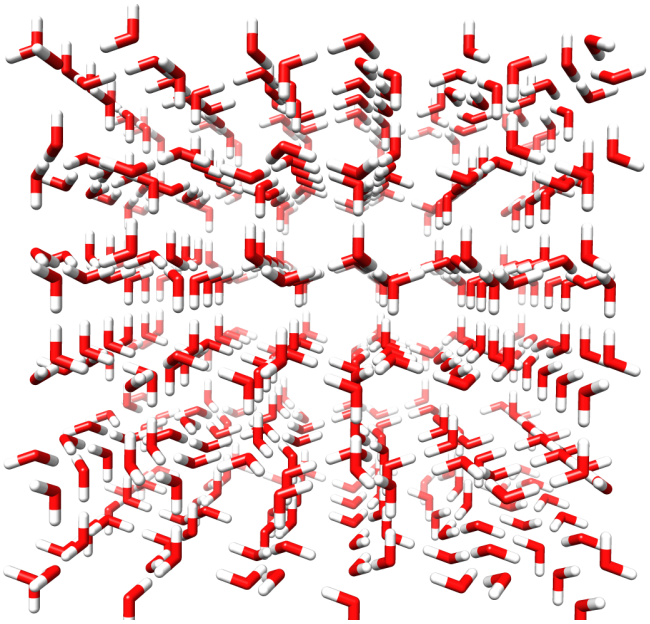

Figure II.4: "After" image of generated ice I_h

consistent. Some of these are a uniform rotation of both hydrogen atoms consistent across rows. These consistent rows can be observed in figure II.4 toward the center-left and center-right along the into-the-page axis. 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. A scoring function to analyze the "randomness" of the crystal would confirm whether this pattern is imagined or real.

II.4 Comparison to Buch's Method

In her 1998 paper, Victoria Buch proposed a MC-based system for converting ice XI to ice I_h.[31] In that method, an ice XI crystal would have all protons dissociated from oxygens by moving them to halfway between corresponding oxygens. By placing protons in the middle of two oxygens, this allowed MC methods to pseudorandomly move the protons toward one or another oxygen. Once moved, the Bernal-Fowler rules are applied to increase the chance of a proton association switch being accepted for invalid waters.

As a comparison to this work, Buch's method is more likely to successfully produce a defect-free ice I_h crystal. In its current state, this work's method is not as efficient nor as effective as Buch's method. As a potential for future development, this method allows for defects to exist as a state value which could be used for annealing studies.

II.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 treated uniquely. 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 specific water 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. No special considerations for waters with fewer than four neighbors are necessary as periodic generation would allow "edge" waters to interact with the periodic continuation waters. 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 II.1. If effective, a scoring function like below would build a better queue for the defect correction step in an MC fashion as it works toward identifying

and reducing the defect density.

$$Value = \sum_{l=1}^{N_{levels}} \left[\frac{1}{l} * \frac{1}{N_{molecules}} * \sum_{m=1}^{N_{molecules}} [N_{defects,m}] \right] \tag{II.1}$$

CHAPTER III

Germanium Compounds and QM Concerns

III.1 The Initial Problem: Germanium Study

During Fall 2016, 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 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.

III.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.[32] 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.[33,34]

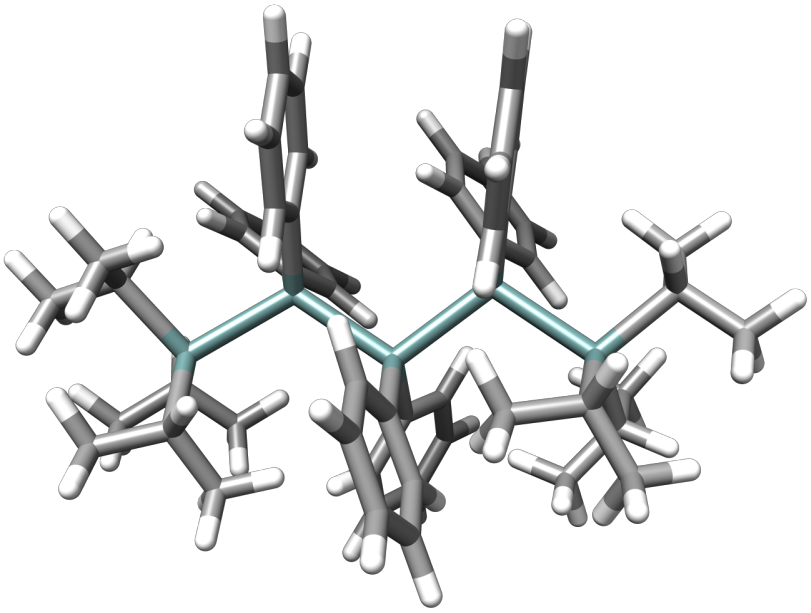

Figure III.1: Fully *trans* configuration of pentagermanium-based compound.

Conformation	Energy (E_h)	Δ Energy (E_h)	Δ Energy ($\frac{\text{kJ}}{\text{mol}}$)
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 III.1: Collaborator's Hexagermanium Energies by Conformation (unspecified DFT, 6-31G(d) basis set, energy in Hartrees and kJ/mol)

III.1.2 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 $Pr^i_3Ge(GePh_2)_nGePr^i_3$ 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 III.1 and III.2.

Dr. Weinert had worked previously with a collaborator who provided conformation data supplied in table III.1. An unspecified DFT method with the 6-31G(d) basis set was used. Additionally, the *cis* and *trans* terms were not explicitly defined.

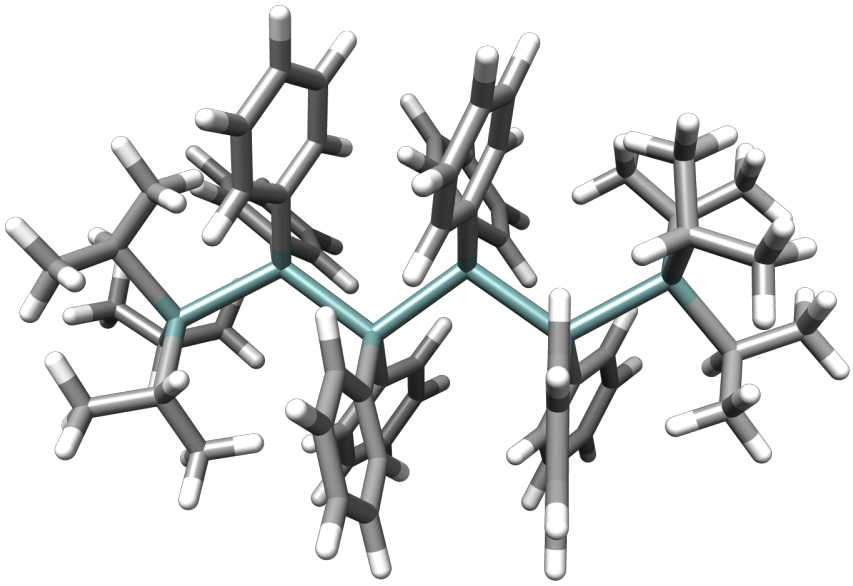

Figure III.2: Fully *trans* configuration of hexagermanium-based compound.

Unfortunately, the collaborator is no longer active in research and was inaccessible for clarification.

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 III.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-periplanar in organic structure nomenclature These *cis* and *trans* terms are preferred over gauche and anti as the dihedral angles are not necessarily restricted to eclipsed or staggered angles. Terminal germanium atoms are not considered as a part of the conformation nomenclature. 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

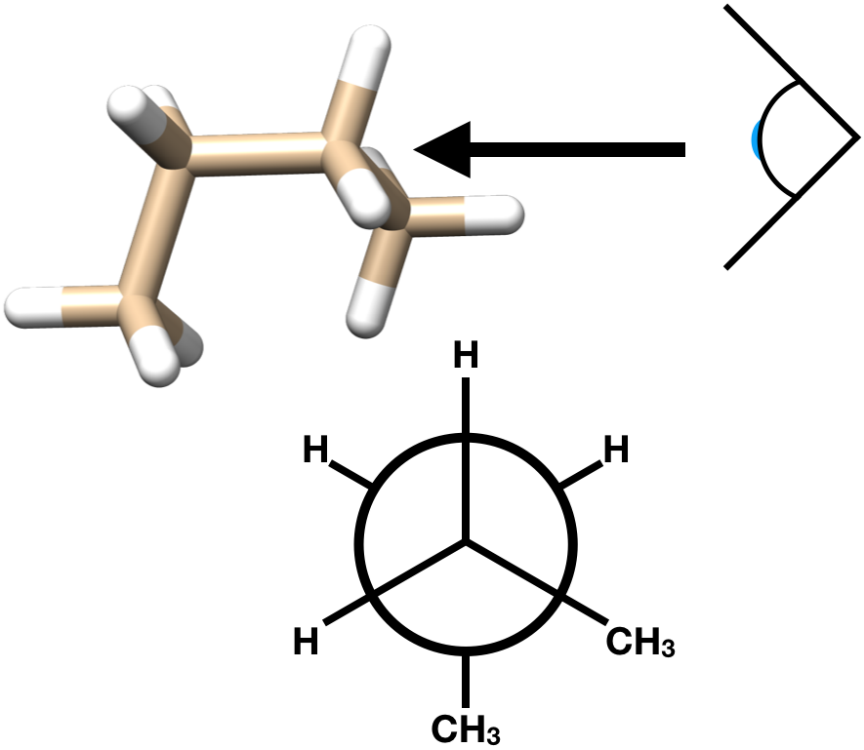

Figure III.3: Sample Newman projection of *cis*-butane.

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.

III.1.3 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.

III.1.3.1 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 with-

out consideration for the phenyl rings populating the non-terminal Ge atoms. Each molecule 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.

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.

III.1.3.2 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 initialized in the fully *trans* conformer.

To prevent potentially strong interactions between adjacent phenyl rings, an additional steepest-descent minimization from Avogadro was computed 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 π 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 III.2 and III.3.

Immediately obvious in the table are the considerable number of nonconverged results. A 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. 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 III.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.

Internal Species	Terminal Species	Conformer	Final Energy (Hartrees)	Δ Energy (Hartrees)	Δ 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 III.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)	Δ Energy (Hartrees)	Δ 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 III.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.

III.1.3.3 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 a 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. 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. A visual of these bonds are given in figure III.4. 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

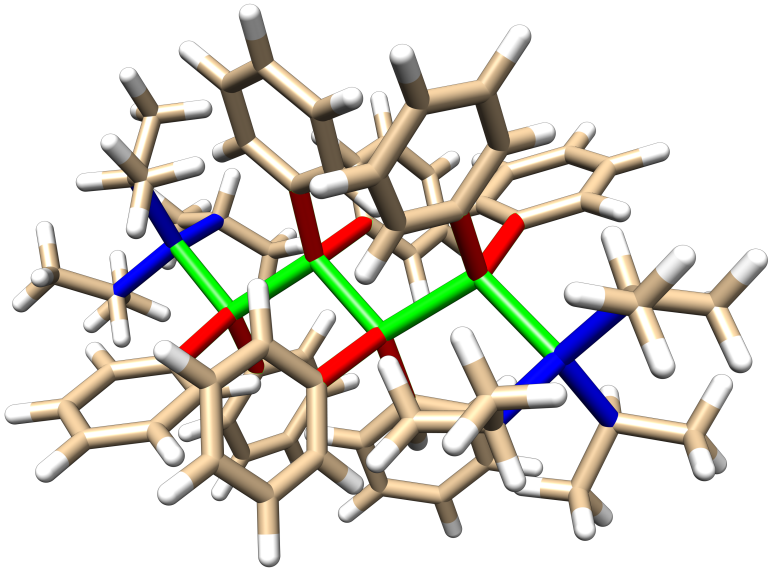

Figure III.4: Visualization of rotatable bonds in hexagermane molecule colored by bonded atoms. Green: Ge-Ge, red: Ge-phenyl C, blue: Ge-isopropyl C.

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 at least two orders of magnitude.

III.1.4 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 III.5 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 π 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 IV, while the remainder of this

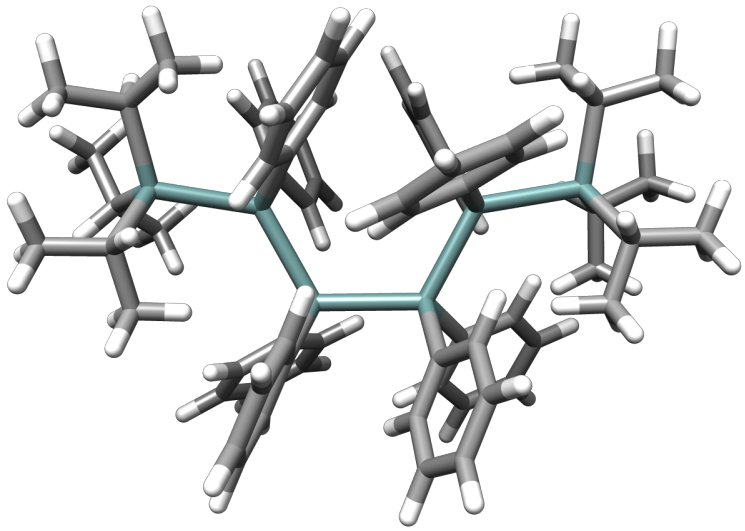

Figure III.5: Visualization of a *trans-cis-trans* hexagermane structure.

chapter details additional efforts of calculating these germanium compounds.

III.1.5 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% increase 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.

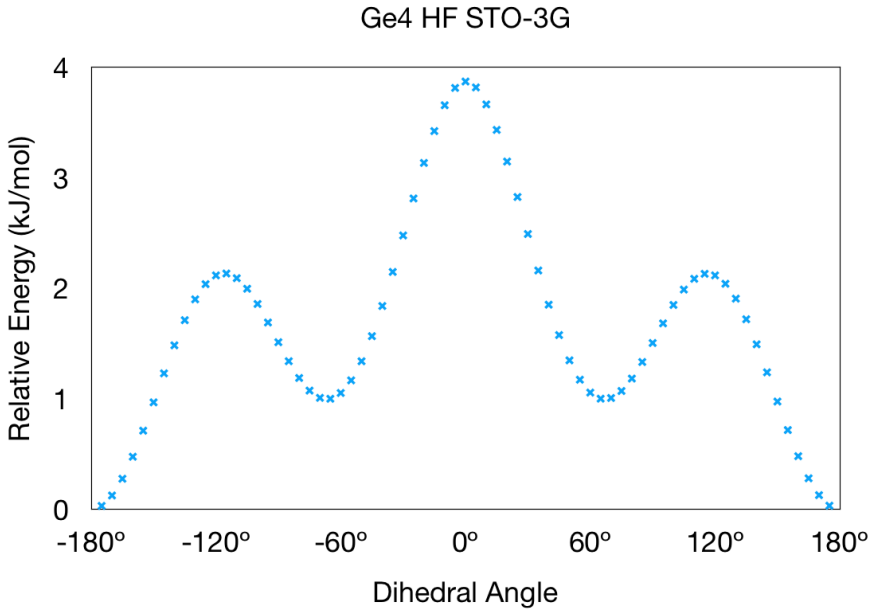

Figure III.6: Sample torsion plot at reduced energy scale.

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.

An example plot of this torsion drive is shown in figure III.6 Once multiple torsion drives had completed in multiple group four elements (butane, butasilane, and butagermane were all built and tested), the energies could be compared and analyzed for any relative or absolute scaling at the additive or multiplicative reference. Relative scaling involved two approaches. The first relative scaling approach involved subtracting each data point by the minimum energy. The second approach involved reducing the first approach to a scale from 0 to 1. This allowed the data points to be considered as percentage energies for additive scaling. The script to collect and scale data points is detailed in Appendix B.

For a full comparative set, 3456 points of analyzed data were generated for each reference molecule's potential in comparison with the others. The script to accomplish each molcule-centric analysis is detailed in Appendix B. No simple correction factor

arose by method of a simple additive or multiplicative term applied toward all torsion points with either absolute or relative energy values. 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 of 5832 conformers. These were then subject to the same data comparison method as before, 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 III.7. 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 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-

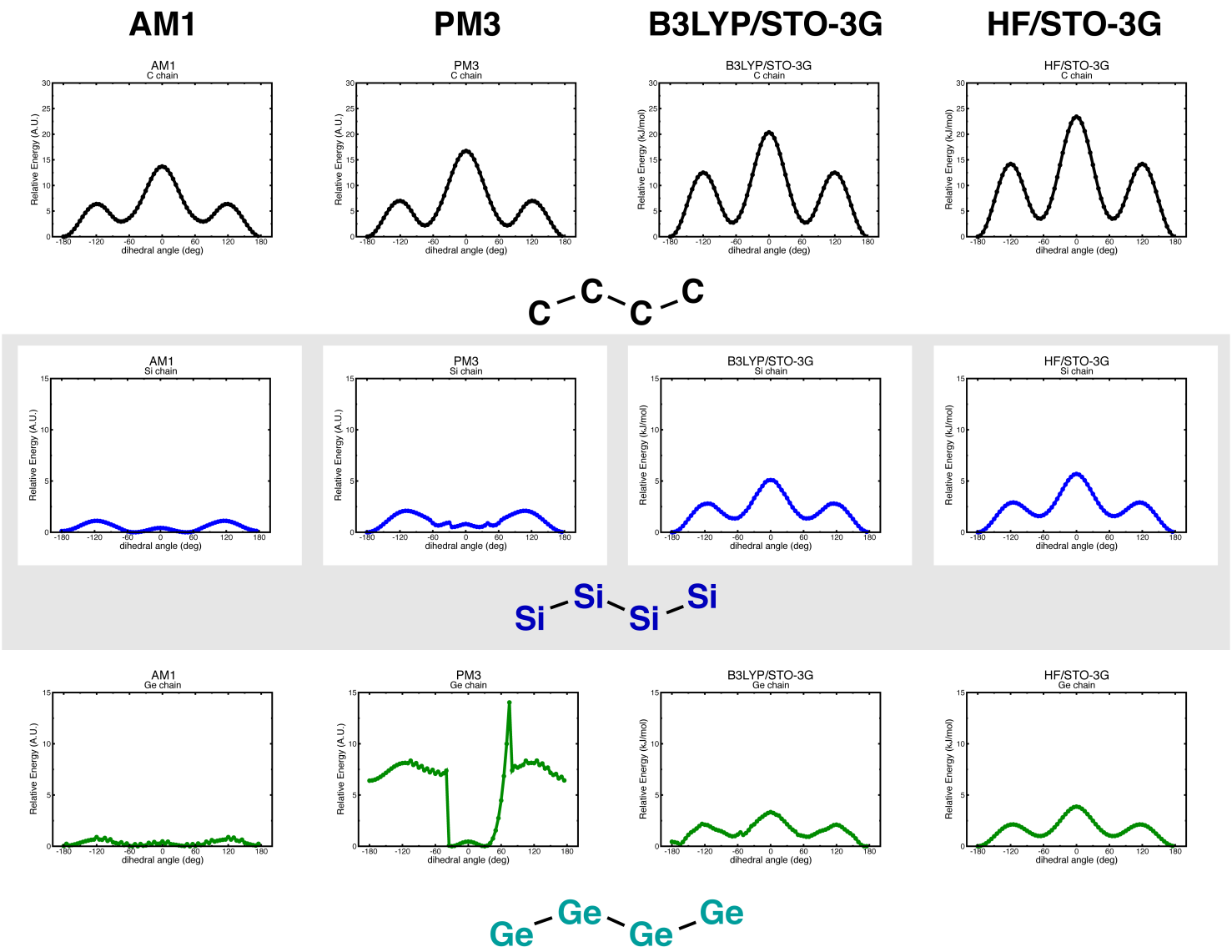

Figure III.7: Visualization of a multiple pure group IV torsions at various theories and basis sets

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.

III.2 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 III.8, 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 appears to be inconsistent with basis set trends.

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) basis set with the B3LYP theory yielded the same inverted 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 figure III.9.

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

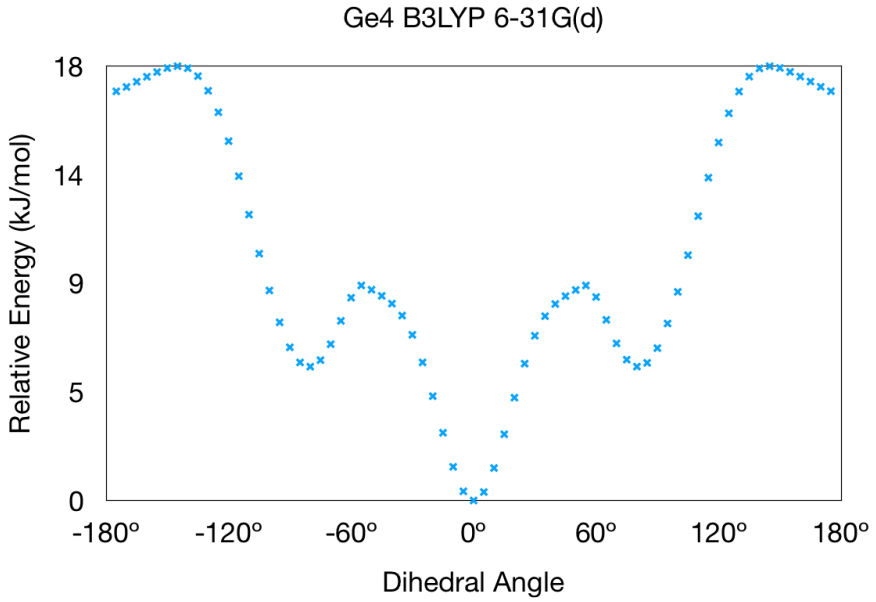

Figure III.8: A curious seemingly-inverted torsion plot of butagermane.

Program	*Trans* Energy (Hartree)	*Cis* Energy (Hartree)	Δ Energy *trans - cis* (Hartree)	Δ Energy *trans - cis* (kJ / mol)
Gaussian	-8298.8259	-8298.8268	-0.0009	-2.4163
GAMESS	-8306.1290	-8306.1250	0.0040	10.4495
NWChem	-8306.1290	-8306.1250	0.0040	10.4495

Table III.4: Energy comparison of HF theory with 6-31G(d) basis set across multiple computational programs. The expected ΔE should be positive.

III.10. 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 III.11, 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 III.4. As can be seen, critical energetic difference was negative for Gaussian 09 and positive for both GAMESS[5] and NWChem.[20] Since the expected conformations should yield a positive difference, it was concluded that both Gaussian 03 and 09 contain bad 6-31G(d) basis set data for germanium.

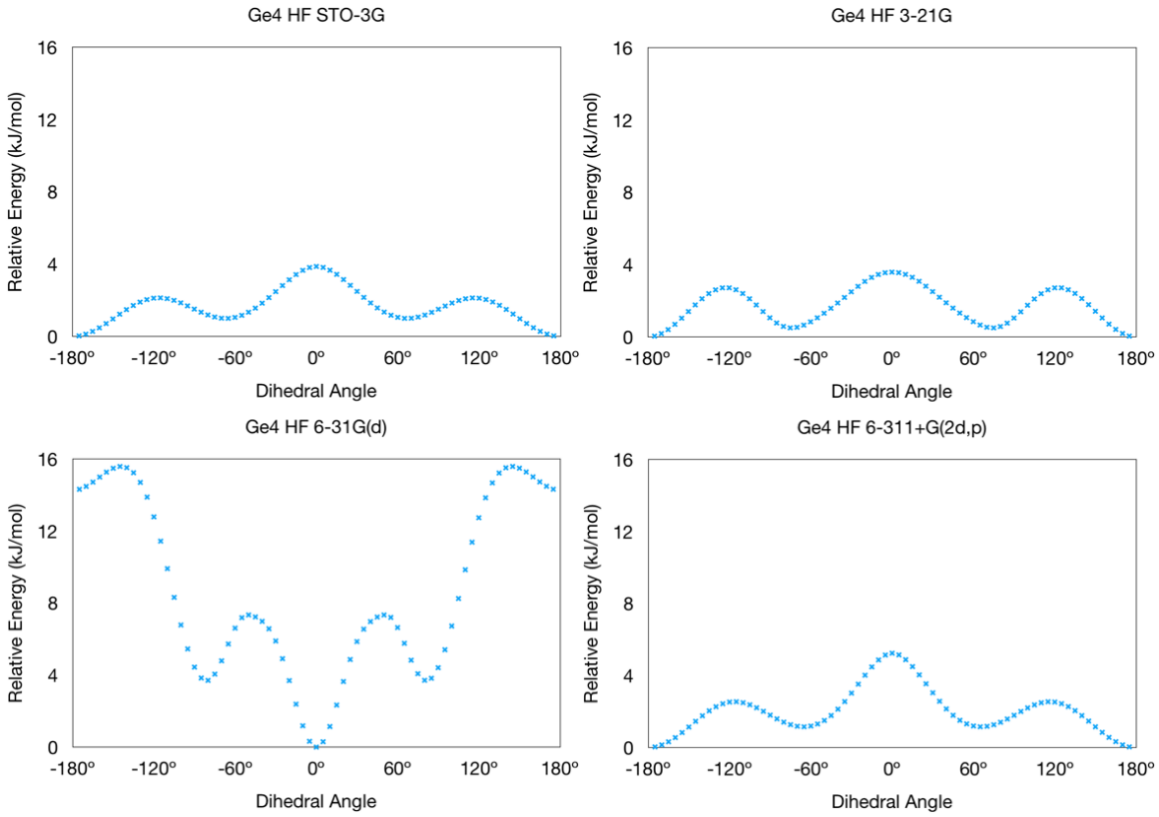

Figure III.9: Hartree Fock energy minimization of butagermane torsion run at varying basis sets.

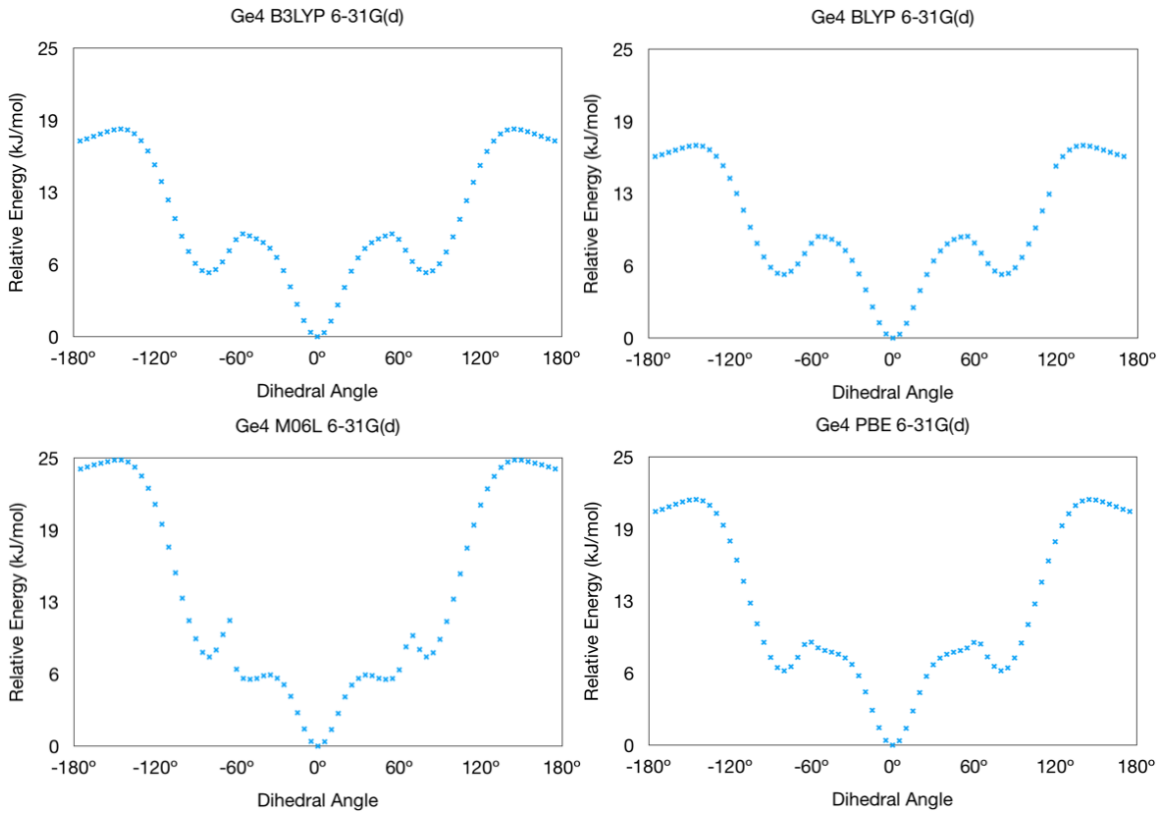

Figure III.10: Minimization of butagermane torsion run at varying theories and the 6-31G(d) basis set.

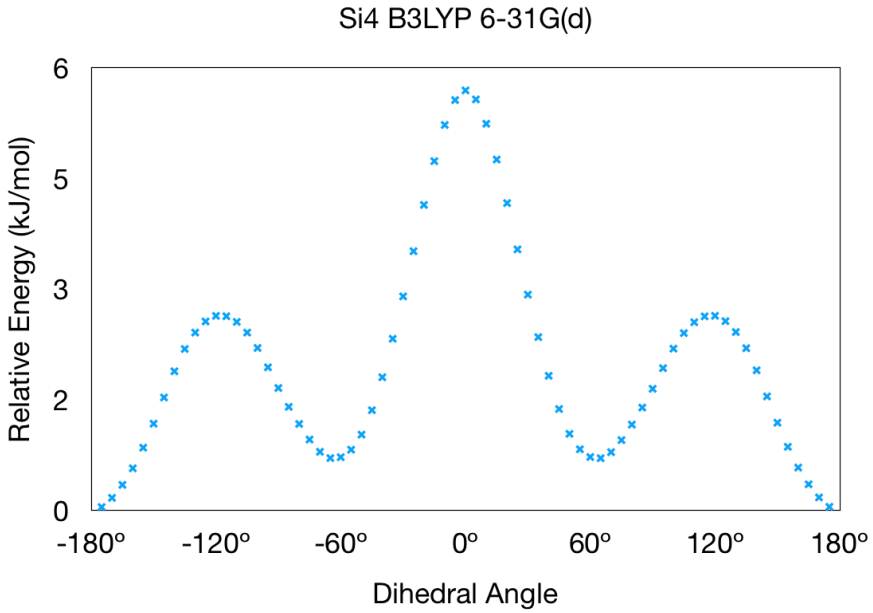

Figure III.11: B3LYP energy minimization of butasilane torsion run at 6-31G(d) basis set.

III.3 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 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 conducting computational studies of germanium either replace the 6-31G(d) basis set data, use another basis set, or instead use a program like GAMESS or NWChem for that final computation.

CHAPTER IV

Sampling Conformation Landscapes by Rotatable Bond Degrees of Freedom

IV.1 A Brief History on Conformation Landscapes

IV.1.1 Levinthal's Paradox

In 1969, a molecular biologist by the name of Cyrus Levinthal proposed a thought experiment regarding protein formation[35]:

Consider a relatively small 150-residue peptide chain completely unfolded. This protein will have 149 peptide bonds and therefore 149 phi angles and 149 psi angles. Assuming three possible angle positions each, the number of possible folds of this protein follows as 3^{298}. 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.[36] How, then, do proteins fold so quickly and efficiently? The answer lies in energy cascades through a visualization tool called a golf course.

IV.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 may not be a clean pathway of energetic difference for the ball to roll toward

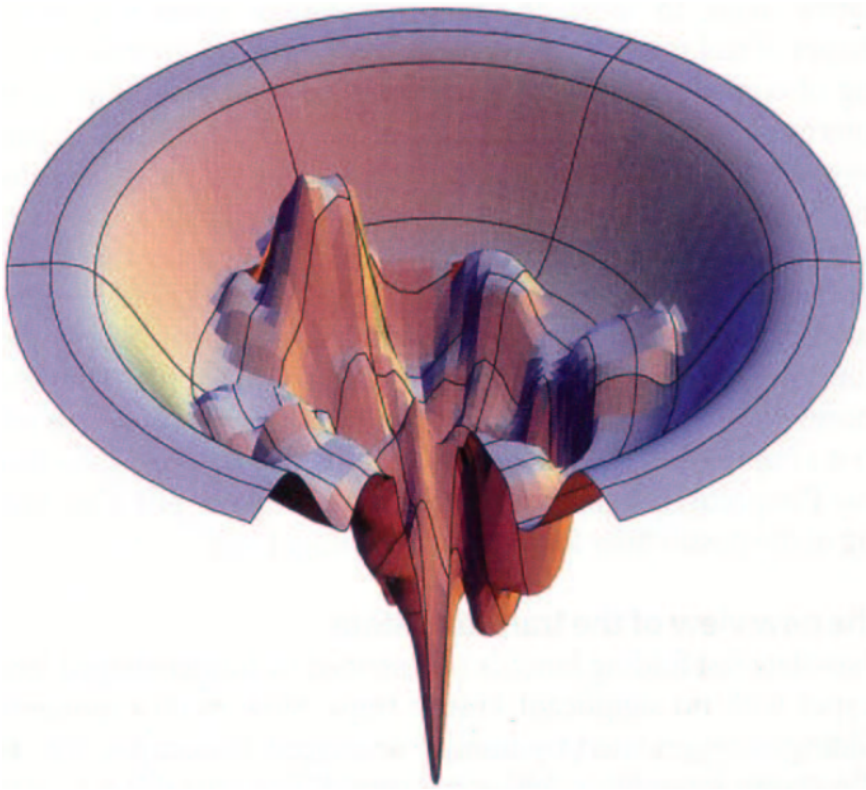

Figure IV.1: Example Levinthal Golf Course taken from Dill et al.[2].

the global minima. 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 IV.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 minimum that is hopefully the true global minimum.

IV.2 Purpose of Project

As introduced in chapter III, there may be a generic solution toward determining the 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 impossible or duplicate conformers. The tool would roughly take shape though a design flow

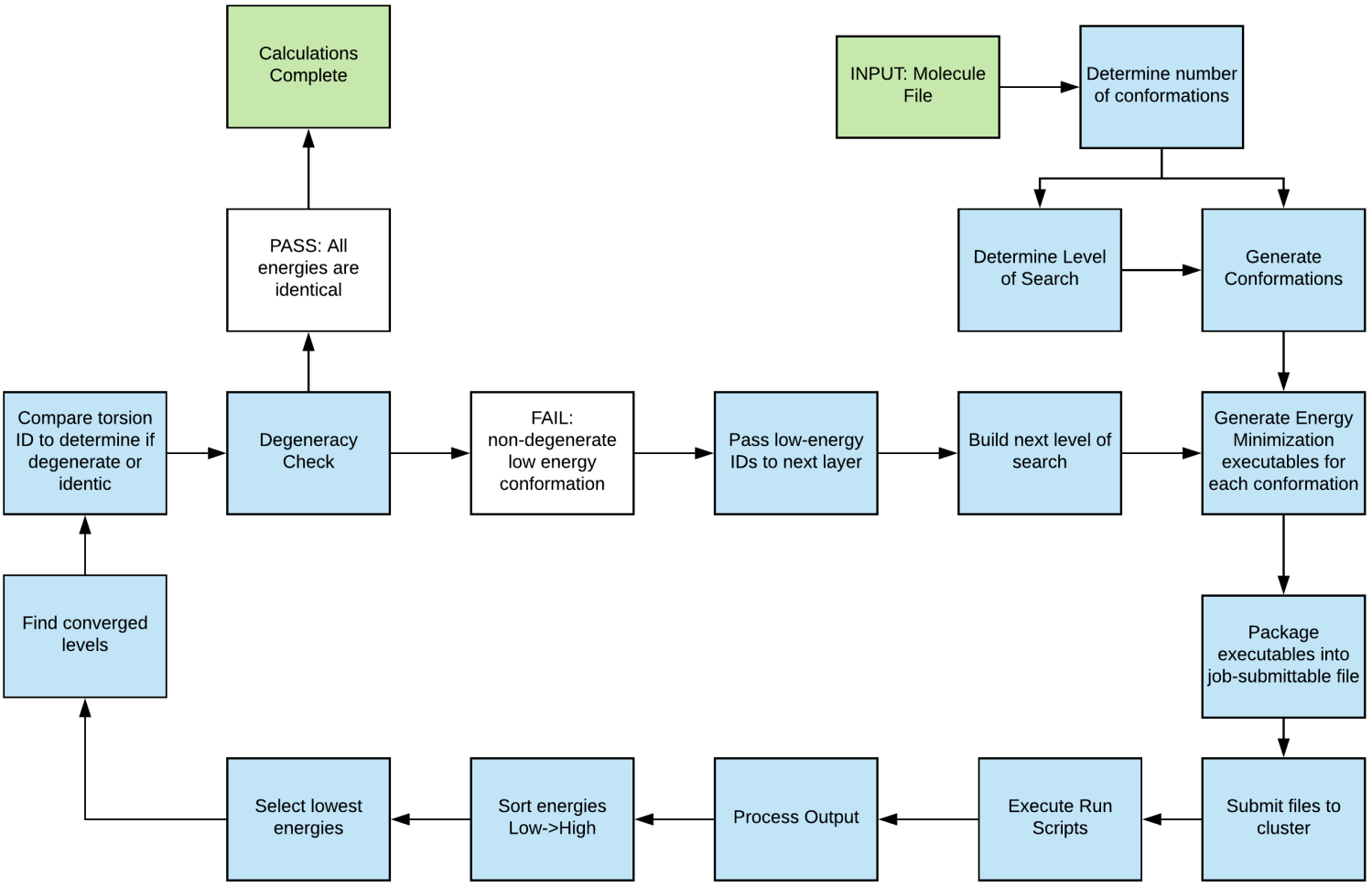

Figure IV.2: Flow of method design for variable resolution conformation landscape search.

detailed below.

First, the system takes an input molecule and generates a number of conformers based on rotatable dihedrals. Second, a time-effective geometry optimization theory and basis set if necessary is selected and run files are generated and submitted to a cluster to compute. Third, the results are collected and analyzed; low-energy dihedral values are passed back through the system while high-energy dihedrals are logged and discarded. This restricts the conformation space to reduce the overall number of conformers generated and allows for more accurate and computationally-expensive theories and methods to calculate more reliable energies.

An overview of system flow given in figure IV.2. This method produces an in-

teresting 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 IV.3. The outlined black boxes represent found regions of interest for future iterations of the method. This would repeat as necessary until regions converge to one energy.

IV.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.

IV.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 physcal computational constraints for various theory-basis set pairings. The system 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.

IV.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,

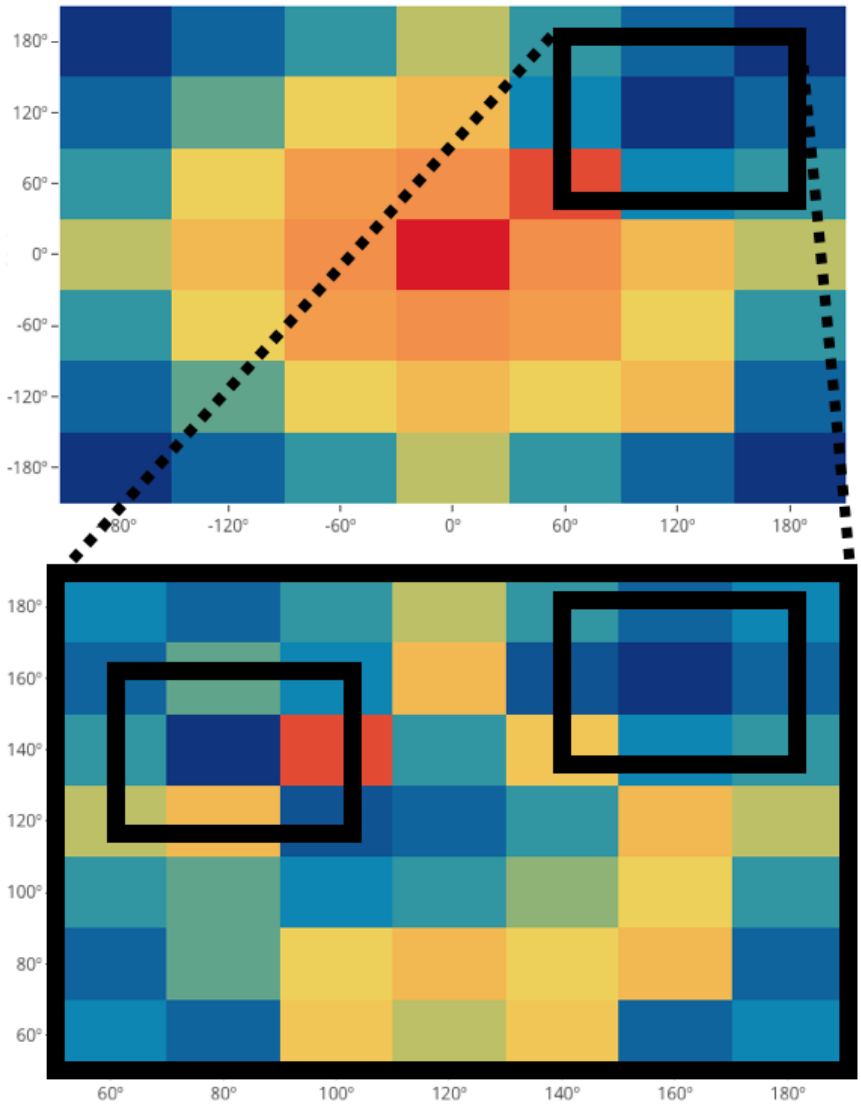

Figure IV.3: Example variable resolution search chart of two dihedrals with low-energy blue to high-energy red.

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.

IV.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 III 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 IV.4. Even with the minimal three rotations per bond, these 19 rotatable bonds produce $3^{19} = 1,162,261,467$ conformers, which is realistically impossible to explore even with a computational method requiring five seconds to compute. 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.

IV.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 highest and lowest energy conformer shown in figures IV.5 and IV.6, respectively.

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.

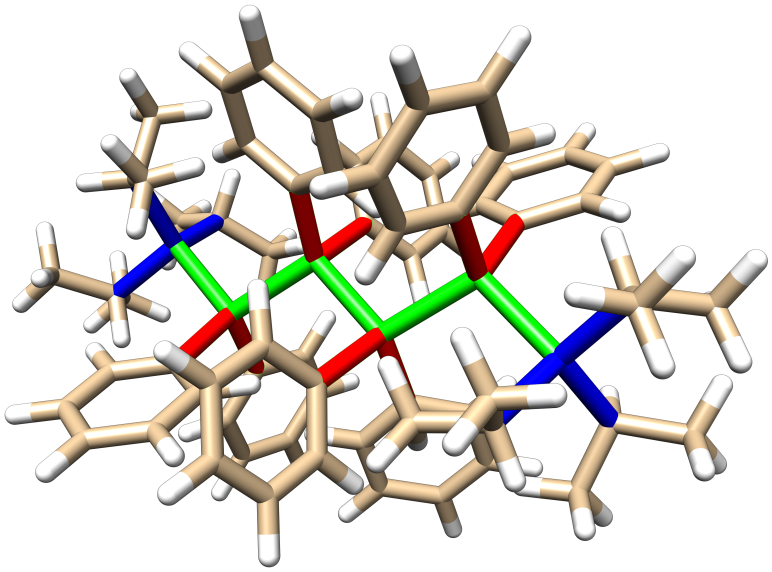

Figure IV.4: Highlighted torsions of the hexagermane molecule by type of bond, where green, red, and blue represent Ge-Ge, Ge-phenyl, and Ge-isopropyl torsion centers, respectively.

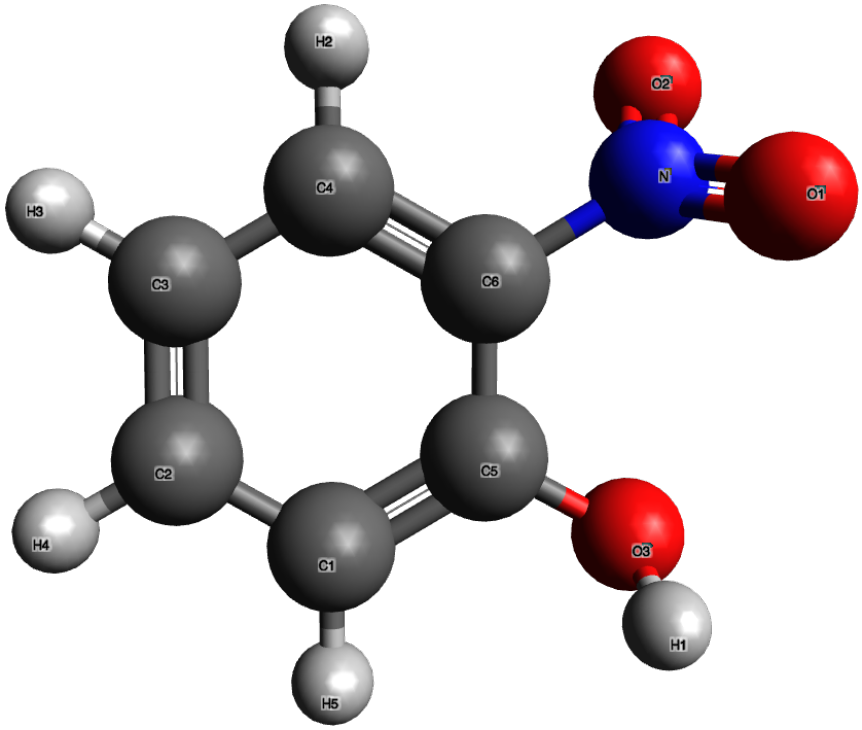

Figure IV.5: Highest energy conformer of o-nitrophenol, ignoring any ring strain conformations. This structure was notably unable to rotate and form the expected hydrogen bond between the ortho nitro and hydroxyl.

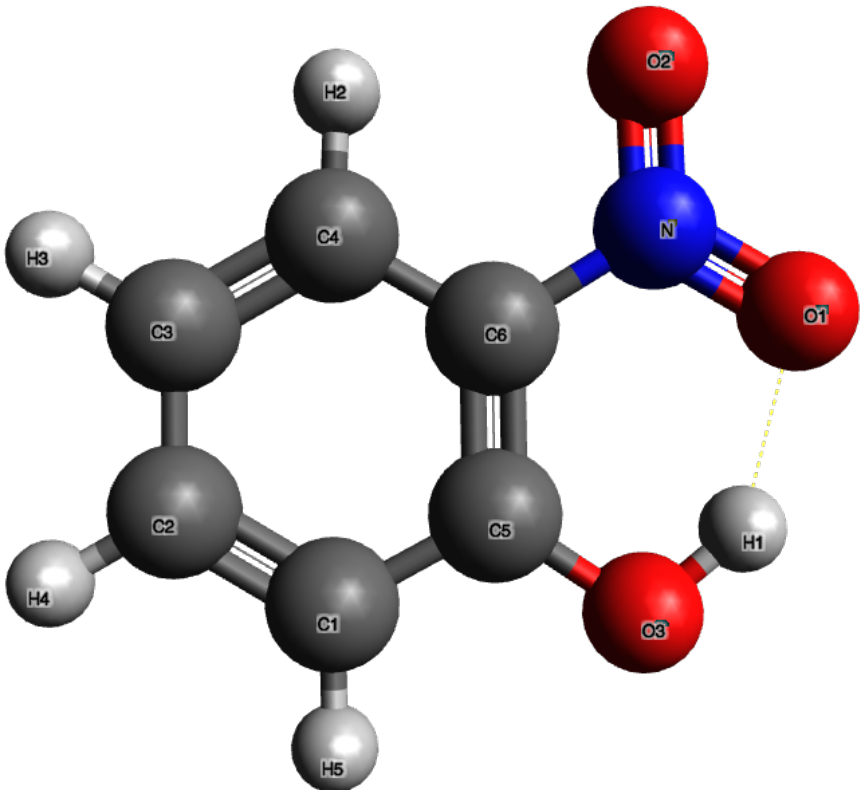

Figure IV.6: Lowest energy conformer of o-nitrophenol. Formed the expected hydrogen bond between the ortho nitro and hydroxyl.

IV.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 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 prevent the system from finding the lowest energy conformer in any reasonably small molecule.

References

[1] Brini, E.; Fennell, C. J.; Fernandez-Serra, M.; Hribar-Lee, B.; Lukšič, M.; Dill, K. A. *Chemical Reviews* **2017**, *117*, 12385–12414.

[2] Dill, K. A.; Chan, H. S. *Nature Structural Biology* **1997**, *4*, 10–19.

[3] Hehre, W. J.; Stewart, R. F.; Pople, J. A. *Journal of Computational Chemistry* **1969**, *51*, 2657–2664.

[4] Boys, S. F. *Proceedings of the Royal Society of London Series A* **1950**, *200*, 542–554.

[5] Schmidt, M.; Baldridge, K.; Boatz, J.; Elbert, S.; Gordon, M.; Jensen, J.; Koseki, S.; Matsunaga, N.; Nguyen, K.; Su, S.; Windus, T.; Dupuis, M.; Montgomery, J. *Journal of Computational Chemistry* **1993**, *14*, 1347–1363.

[6] Thouless, D. *The quantum mechanics of many-body systems*; Pure and applied physics; Academic Press, 1972.

[7] Hartree, D. R. *Mathematical Proceedings of the Cambridge Philosophical Society* **1928**, *24*, 111–132.

[8] Kohn, W.; Sham, L. J. *Physical Review* **1965**, *140*, A1133–A1138.

[9] Lee, C.; Yang, W.; Parr, R. G. *Physical Review B* **1988**, *37*, 785–789.

[10] Zhao, Y.; Truhlar, D. G. *Journal of Computational Chemistry* **2006**, *125*, 194101.

[11] Perdew, J. P.; Burke, K.; Ernzerhof, M. *Physical Review Letters* **1996**, *77*, 3865–3868.

[12] Hückel, E. *Zeitschrift für Physik* **1933**, *83*, 632–668.

[13] Dewar, M. J. S.; Zoebisch, E. G.; Healy, E. F.; Stewart, J. J. P. *Journal of the American Chemical Society* **1985**, *107*, 3902–3909.

[14] Stewart, J. J. P. *Journal of Computational Chemistry* **1989**, *10*, 209–220.

[15] Slater, J. C. *Physical Review* **1930**, *36*, 57–64.

[16] Ditchfield, R.; Hehre, W. J.; Pople, J. A. *Journal of Computational Chemistry* **1971**, *54*, 724–728.

[17] Buffon, G. *Histoire naturelle, générale et particulière, servant de suite à la Théorie de la Terre et d'introduction à l'Histoire des Minéraux...Supplément Tome premier [septième]*; Histoire naturelle, générale et particulière, servant de suite à la Théorie de la Terre et d'introduction à l'Histoire des Minéraux...Supplément Tome premier [septième]; de l'imprimerie royale, 1777.

[18] Metropolis, N. *Los Alamos Science* **1987**, *15*, 125–130.

[19] Metropolis, N.; Ulam, S. *Journal of the American Statistical Association* **1949**, *44*, 335–341.

[20] Valiev, M.; Bylaska, E.; Govind, N.; Kowalski, K.; Straatsma, T.; Dam, H. V.; Wang, D.; Nieplocha, J.; Apra, E.; Windus, T.; de Jong, W. *Computer Physics Communications* **2010**, *181*, 1477–1489.

[21] Hanwell, M. D.; Curtis, D. E.; Lonie, D. C.; Vandermeersch, T.; Zurek, E.; Hutchison, G. R. *Journal of Cheminformatics* **2012**, *4*, 17.

[22] Pettersen, E.; Goddard, T.; Huang, C.; Couch, G.; Greenblatt, D.; Meng, E. *Journal of Computational Chemistry* **2004**, *25*, 1605–1612.

[23] van Rossum, G. *Python tutorial, Technical Report CS-R9526*; 1995.

[24] Oliphant, T. *Guide to NumPy*; Trelgol Publishing, 2006.

[25] Jones, E.; Oliphant, T.; Peterson, P. SciPy: Open source scientific tools for Python. 2001; `http://www.scipy.org/`.

[26] Behnel, S.; Bradshaw, R.; Citro, C.; Dalcin, L.; Seljebotn, D.; Smith, K. *Computing in Science Engineering* **2011**, *13*, 31–39.

[27] Bernal, J. D.; Fowler, R. H. *The Journal of Chemical Physics* **1933**, *1*, 515–548.

[28] Yen, F.; Chi, Z. *Physical Chemistry Chemical Physics* **2015**, *17*, 12458–12461.

[29] Pauling, L. *Journal of the American Chemical Society* **1935**, *57*, 2680–2684.

[30] Bjerrum, N. *Science* **1952**, *115*, 385–390.

[31] Buch, V.; Sandler, P.; Sadlej, J. *The Journal of Physical Chemistry B* **1998**, *102*, 8641–8653.

[32] Brown, Z. D.; Guo, J.-D.; Nagase, S.; Power, P. P. *Organometallics* **2012**, *31*, 3768–3772.

[33] Roewe, K. D.; Rheingold, A. L.; Weinert, C. S. *Chemical Communications* **2013**, *49*, 8380–8382.

[34] Komanduri, S. P.; Shumaker, F. A.; Roewe, K. D.; Wolf, M.; Uhlig, F.; Moore, C. E.; Rheingold, A. L.; Weinert, C. S. *Organometallics* **2016**, *35*, 3240–3247.

[35] Levinthal, C. How to Fold Graciously. Mossbauer Spectroscopy in Biological Systems: Proceedings of a meeting held at Allerton House. 1969; pp 22–24.

[36] Zwanzig, R.; Szabo, A.; Bagchi, B. *Proceedings of the National Academy of Sciences* **1992**, *89*, 20–22.

APPENDIX A

Ice Ih to Ice XI Conversion

Listed below is the source code utilized in the conversion of a .pdb Ice Ice I_h structure into an Ice XI structure. This code is functional in a Python 2.7 environment with the included packages: NumPy version 1.14.3 and SciPy version 1.1.0.

A.1 Brief Sample of Ice XI .PDB File

```
HETATM    1    O    O    1     -10.483   -5.440   10.189
HETATM    2  1H1    H    1     -10.473   -4.440   10.185
HETATM    3  2H1    H    1     -10.015   -5.781    9.374
HETATM    4    O    O    2      -9.186   -6.385    7.933
HETATM    5  1H2    H    2      -9.655   -6.049    7.115
HETATM    6  2H2    H    2      -8.241   -6.059    7.931
HETATM    7    O    O    3      -6.569   -5.486    7.929
HETATM    8  1H3    H    3      -6.559   -4.486    7.925
HETATM    9  2H3    H    3      -6.101   -5.827    7.114
HETATM   10    O    O    4      -5.274   -6.412   10.193
HETATM   11  1H4    H    4      -5.741   -6.077    9.375
HETATM   12  2H4    H    4      -4.327   -6.087   10.191
HETATM   13    O    O    5      -6.569   -5.468   12.449
HETATM   14  1H5    H    5      -6.559   -4.468   12.445
HETATM   15  2H5    H    5      -6.101   -5.809   11.633
HETATM   16    O    O    6      -9.186   -6.366   12.453
HETATM   17  1H6    H    6      -9.655   -6.031   11.634
HETATM   18  2H6    H    6      -8.241   -6.041   12.451
HETATM   19    O    O    7     -10.526  -10.053   10.207
HETATM   20  1H1    H    7     -11.466   -9.710   10.206
HETATM   21  2H1    H    7     -10.052   -9.720   11.022
HETATM   22    O    O    8      -9.212   -9.151    7.944
HETATM   23  1H2    H    8      -9.203   -8.151    7.940
HETATM   24  2H2    H    8      -9.688   -9.477    8.762
HETATM   25    O    O    9      -6.612  -10.099    7.947
HETATM   26  1H3    H    9      -7.552   -9.756    7.946
HETATM   27  2H3    H    9      -6.138   -9.766    8.763
HETATM   28    O    O   10      -5.300   -9.179   10.204
HETATM   29  1H4    H   10      -5.290   -8.179   10.200
HETATM   30  2H4    H   10      -5.774   -9.504   11.021
```

A.2 Code: Crystal Disorganizer Tool

```

#!/usr/bin/python

# Author = Gentry Smith
```

```
# Copyright 2016, all rights reserved

# this reads in a .PDB file, takes an argument for deformities per molecules, and randomly organizes the crystal
# structure into a disordered proton formation

# import sample: python PDBDisorganize.py arg1 arg2 arg3
# where:
# arg1 = source pdb file to be read (ex: acetone.pdb or acetone)
# arg2 = number of defects per molecule (in H20, num of non-hydrogen-bonds. from 0 to 4)
# arg3 = desired output pdb file name

import sys
print sys.path
import string
import numpy as np
import math
import random

sys.setrecursionlimit(10000000) # maximum recursive depth. Set to (10,000,000) as under maximum


pdbIN = file(sys.argv[1])     # source PDB file
maxErr = int(sys.argv[2])    # max errors allowed
pdbOUT = str(sys.argv[3])    # output file name
finalData = [ [ [ 0 for i in range(3) ] for j in range(3) ] for k in range(300) ]

# looks at args validity
def checkArgs(arg1, arg2, arg3):
    returnBool = False
    if type(arg1) != file: # check arg1
        print"Bad arg", arg1, " must be a file "
        returnBool = True
    if type(arg3) != str: # check arg3
        print"Bad arg", arg3, ", must be a file name"
        checkPDBSuffix(arg3)
        print arg3
        returnBool = True
    if type(arg2) != int:  # check arg2 type
        print "Bad arg2: ", arg2, " is not an int."
        returnBool = True
    elif type(arg2) == int:
        if arg2 < 0 or arg2 > 4:  # check arg2 range
            print "arg2 is not in a valid range 0 <= arg2 <= 4"
            returnBool = True
    return returnBool

def checkPDBSuffix(pdbFile):
    if string.find(pdbFile, '.pdb', 0, len(pdbFile)) == -1:
        print("did not find 'pdb' in ", pdbFile, ". Appending...")
        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
       range(numMol) ]
71     fileName.seek(0)
72     iter0 = 0
73     iter1 = 0
74     pdbType = -1
75     for line in fileName:
76         data = line.split()
77         if pdbType == -1:
78             if data[0] == "ATOM":
79                 pdbType = 0
80             elif data[0] == "HETATM":
81                 pdbType = 1
82         # print "LineTuple= ", data
83         if len(data) > 1 and ( data[0] == "ATOM" or data[0] == "HETATM"
       ):
84             if data[0] == "ATOM":
85                 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
       has a space between each value
103
104 # reads XYZ coordinate data from ATOM-type pdb
105 def getDataATOM(strLine):
```

```
106     # print "Getting ATOM Data..."
107     dataLine = strLine[5:8]
108     # print "dataline: ", dataLine
109     i = 0
110     while i < 3:
111         # print "dataline[", i, "]: ", dataLine[i]
112         dataLine[i] = float(dataLine[i])
113         # print "dataline[", i, "] type: ", type(dataLine[i])
114         i += 1
115     return dataLine
116 
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
    oxygen molecule
133 def getOrientations( molecule ):
134     # 120 degrees = ( 2 * pi ) / 3 radians
135     theta = ( ( 2 * math.pi ) / 3 )
136     newMol = zeroOrientation(molecule)
137     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
     inclusively
143 def newRandOrientation( positions ):
144     # print "Changing orientation"
145     randVal1 = random.randint(0,3)
146     randVal2 = random.randint(0,3)
147     while randVal1 == randVal2:
148         randVal2 = random.randint(0,3)
149     newMol = [ [ 0, 0, 0 ],
150                  positions[ randVal1 ] ,
151                  positions[ randVal2 ]  ]
152     return newMol
153 
154 # selects new orientation from list. Reduces computational overhead in
    re-orientation option traversal
155 def newSetOrientation( positions, pos1, pos2 ):
156     newMol = [ [ 0, 0, 0 ],
```

```
157                 positions[ pos1 ],
158                 positions[ pos2 ] ]
159     return newMol
160
161
162 # sets molecule coordinates so that oxygen is the origin
163 def zeroOrientation(source):
164     # print "Zeroing Molecule..."
165
166     oxy = source[0]
167     hyd1 = source[1]
168     hyd2 = source[2]
169
170     # print "Oxygen pos: ", oxy
171     # print "Hydrogen 1: ", hyd1
172     # print "Hydrogen 2: ", hyd2
173
174     zeroedOrigin = [0, 0, 0]
175     zeroedHyd1 = [0, 0, 0]
176     zeroedHyd2 = [0, 0, 0]
177     for i in range(3):
178         zeroedHyd1[i] = hyd1[i] - oxy[i]
179         zeroedHyd2[i] = hyd2[i] - oxy[i]
180
181     # print "Zeroed Hydrogen 1: ", zeroedHyd1
182     # print "Zeroed Hydrogen 2: ", zeroedHyd2
183
184     # return new molecule position
185     newMol = [zeroedOrigin, zeroedHyd1, zeroedHyd2]
186     return newMol
187
188 # resets the zeroed molecule to the original oxygen position
189 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
```

```
# page: http://stackoverflow.com/questions/6802577/python-rotation-of-3d-vector
def rotationMatrix(axis, theta):
    """

    :type axis: list
    :type theta: union
    """
    axis = np.asarray(axis)
    theta = np.asarray(theta)
    axis /= math.sqrt(np.dot(axis, axis))
    a = math.cos( theta/2.0 )
    b, c, d = -axis*math.sin(theta/2.0)
    aa, bb, cc, dd = (a * a), (b * b), (c * c), (d * d)
    bc, ad, ac, ab, bd, cd = (b * c), (a * d), (a * c), (a * b), (b * d), (c * d)
    return np.array( [ [ (aa + bb - cc - dd), ( 2 * ( bc + ad ) ), ( 2 * ( bd - ac ) ) ],
                       [ ( 2 * ( bc - ad ) ), (aa + cc - bb - dd), ( 2 * ( cd + ab ) ) ],
                       [ ( 2 * ( bd + ac ) ), ( 2 * ( cd - ab ) ), (aa + dd - bb - cc) ] ] )


# gets results from rotateAboutAxis plus two Hydrogens to get the tetrahedron positions
def getTetrahedronPositions(molecule):
    positions = [ [ 0 for i in range(3) ] for j in range(4) ]
    newMol = zeroOrientation(molecule)  # zero molecule
    positions[0] = newMol[1]
    positions[1] = newMol[2]
    newPos = getOrientations(molecule)  # get final two positions
    positions[2] = list(newPos[0])
    positions[3] = list(newPos[1])
    return positions                    # return all four positions


# checks distance of new positions from zero
def checkDist(posArray):
    distance = [0 for i in range(len(posArray))]
    for i in range(len(posArray)):
        distance[i] =  ( (posArray[i][0] * posArray[i][0]) +
                         (posArray[i][1] * posArray[i][1]) +
                         (posArray[i][2] * posArray[i][2]) )
        # print "Distance", i, ": ", distance[i]
    avg = 0
    for i in range(len(posArray)):
        avg += distance[i]
    averageDistance = ( avg / len(posArray) )
    # print "Average Distance: ", averageDistance
    return averageDistance


# 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[
    mol][atom][dimension]) + "\t"
270             print printStr
271         bigAvg += checkDist(zeroOrientation(data[mol])[1:])
272         numAtoms += 1
273         print ""
274     print "total average distance: ", bigAvg / numAtoms
275 
276 
277 # checks validity of molecule
278 def isDefectiveCheck(err, neighborData, posData, index):
279     # find nearby molecules (avg oxygen distance???)
280     print "checking for defects at index", index, "..."
281     print "neighbor indices: ", neighborData[index]
282     returnBool = False
283     neighbors = 4
284     for i in range(4):  # count real neighbors
285         if neighborData[index][1][i] == -1:
286             neighbors -= 1
287     if neighbors <= err:    # de facto good if num(neighbors) <
    maxErrAllowed
288         # print "Fewer neighbors than allowed errors. de facto Good
    Orientation"
289         returnBool = True
290     elif neighbors > err:   # enough neighbors to require check
291         # print "More neighbors than error threshold"
292         defectCount = 0
293         for neighbor in range(4):   # check each neighbor
294             if neighborData[index][1][neighbor] != -1:  # skip over non-
    existant neighbors
295                 molA = posData[index]
296                 molB = posData[ neighborData[index][1][neighbor] ]
297                 oxyDist = getDistBetweenAtoms(molA[0], molB[0])
298 
299                 if minHydrogenDistance(molA, molB) > oxyDist: # check
    for facing lone pairs
300                     print "Double Lone Pair defect"
301                     defectCount += 1
302                     break
303                 else:   # check for facing protons
304                     smallerHydrogenDistanceCount = 0
305                     isDefective = False
306                     for first in range(2):
307                         if not isDefective:
```

```
                            for second in range(2):
                                newDist = getDistBetweenAtoms(molA[first + 1], molB[second + 1])
                                if newDist < oxyDist:
                                    smallerHydrogenDistanceCount += 1
                            if smallerHydrogenDistanceCount > 1:
                                print "Double Hydrogen defect"
                                defectCount += 1
                                isDefective = True
        # print "Defects found:", defectCount
        if defectCount > 4:
            print "IMPOSSIBLE AMOUNT OF DEFECTS DETECTED!!!! AHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHH"
        if defectCount > err:
            # print "Found a bad molecule!"
            returnBool = False
        else:
            # print "Molecule is within parameters."
            returnBool = True

    return returnBool


# randomly re-reorients molecule and neighbors, rechecks all
def rerunMolAndNeighbors(err, neighborData, posData, index):
    # print "Re-reordering molecule at", index
    # err - max errors allowed
    # neighborData - int[4] of neighbor indices
    # posData - array of all molecule position vectors
    # index - location of focus molecule in posData
    isGood = False
    timeCount = 0
    while not isGood:
        # re-rotate molecule through all positions (iterated through all orientations)
        positions = getTetrahedronPositions(posData[index])
        zeroedMol = newRandOrientation(positions)
        # print "isGood CHECK", isGood
        isGood, posData = iterThroughRotations(err, neighborData, posData, index)
        posData[index] = resetOrientation(posData[index][0], zeroedMol)
        if timeCount >= 13: # { (1 - 1/6)^n < 0.05 } says n = 17
            # BROKEN - need to rebuild
            # 0. evaluated molecule has too many defects
            # 1. reorient molecule statistically probable amount of times to cover all orientations
            # 2. Repeat 1. with neighbor 1
            # 2a repeat 1. with original molecule
            # 3. Repeat 2. with neighbor 2, 3, 4, as/if necessary
            for neighborIndex in range(4):
                if neighborData[index][1][neighborIndex] != -1:
                    positions = getTetrahedronPositions(posData[neighborIndex])
                    zeroedMol = newRandOrientation(positions)
```

```
                    posData[neighborIndex] = resetOrientation(posData[
    neighborIndex][0], zeroedMol)
                    # isGood = isDefectiveCheck(err, neighborData,
    posData, neighborIndex)
            isGood = isDefectiveCheck(err, neighborData, posData, index)
            if not isGood:
                isGood, posData = rerunMolAndNeighbors(err, neighborData
    , posData, neighborData[index][1][neighborIndex])
    finalData = posData
    return True, finalData

# iterates molecule through all possible rotations
def iterThroughRotations(err, neighborData, posData, index):
    isGood = False
    pos1 = 0  # tetrahedral position for H1
    pos2 = 0  # tetrahedral position for H2
    while not isGood or (pos1 != 3 and pos2 != 3):  # iterates through
    all orientations, stops if good orientation
        if pos1 != pos2:
            posData[index] = newSetOrientation(posData[index][0], pos1,
    pos2)
            isGood = isDefectiveCheck(err, neighborData, posData, index)
        if pos2 < 3:
            pos2 += 1
        elif pos2 == 3:
            if pos1 < 3:
                pos1 += 1
                pos2 = 0
    return isGood, posData
# determines minimum hydrogen distance between two atoms
def minHydrogenDistance(mol1, mol2):
    minDist = 100
    for first in range(2):
        for second in range(2):
            newDist = getDistBetweenAtoms(mol1[first+1], mol2[second+1])
            if newDist < minDist:
                minDist = newDist
    return minDist




# finds neighboring molecules of each molecule
def getNeighbors(data):
    returnData = [ [ [ 0 for i in range(4) ] for j in range(2) ] for k
    in range(len(data)) ] # data[molecule][distance,index][four values]
    for mol1 in range(len(data)):
        minDist = [100, 100, 100, 100]
        minIndex = [0, 0, 0, 0]
        for mol2 in range(len(data)):
            if mol1 != mol2:
                newMin = getDistBetweenAtoms(data[mol1][0], data[mol2
    ][0])
```

```
403
404                 bigIndex = indexOfBiggest(minDist)
405                 if newMin < minDist[bigIndex]:
406                     minDist[bigIndex] = newMin
407                     minIndex[bigIndex] = mol2
408         for i in range(4):
409             if minDist[i] >= 9:
410                 minDist[i] = -1
411                 minIndex[i] = -1
412         # print "Four smallest Distances of", mol1, ": ", minDist
413         # print "Four smallest Indices of", mol1, ": ", minIndex
414         returnData[mol1] = [minDist, minIndex]
415     return returnData
416
417
418 # finds distance between oxygen atoms
419 def getDistBetweenAtoms( mol1, mol2 ):
420     distance = ( ( ( mol1[0] - mol2[0] ) * ( mol1[0] - mol2[0] ) ) +
421                  ( ( mol1[1] - mol2[1] ) * ( mol1[1] - mol2[1] ) ) +
422                  ( ( mol1[2] - mol2[2] ) * ( mol1[2] - mol2[2] ) ) )
423     return distance
424
425 # gets index of largest item from a list
426 def indexOfBiggest(check):
427     bigIndex = 0
428     for i in range(len(check)):
429         if check[i] > check[bigIndex]:
430             bigIndex = i
431     return bigIndex
432
433
434 # writes data to PDB file
435 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 += " "
```

```
            if atom == 0:
                outStr += " O   " + " WAT"
            elif atom == 1:
                outStr += " H1  " + " WAT"
            elif atom == 2:
                outStr += " H2  " + " WAT"
            outStr += str(molecule)
            while len(outStr) < 26:
                outStr = outStr[:20] + " " + outStr[20:]
            outStr += "     "
            outStr += "{:8.3f}".format(data[molecule][atom][0])
            outStr += "{:8.3f}".format(data[molecule][atom][1])
            outStr += "{:8.3f}".format(data[molecule][atom][2])
            outStr += "  1.00" + "  0.00"
            outStr += "           "
            if atom == 0:
                outStr += " O  "
            elif atom == 1:
                outStr += " H  "
            elif atom == 2:
                outStr += " H  "
            outStr += "\n"
            inFile.write(outStr)


# Writes data to PDB file style = HETATOM
def writeDataPDBHETATM(data, inFile):
    iterator = 0
    for molecule in range(len(data)):
        for atom in range(3):
            iterator += 1
            outStr = "HETATM"
            outStr += str(iterator)
            while len(outStr) < 11:
                outStr = outStr[:6] + " " + outStr[6:]
            outStr += " "
            if atom == 0:
                outStr += " O   " + " WAT"
            elif atom == 1:
                outStr += " H1  " + " WAT"
            elif atom == 2:
                outStr += " H2  " + " WAT"
            outStr += str(molecule)
            while len(outStr) < 26:
                outStr = outStr[:20] + " " + outStr[20:]
            outStr += "     "
            outStr += "{:8.3f}".format(data[molecule][atom][0])
            outStr += "{:8.3f}".format(data[molecule][atom][1])
            outStr += "{:8.3f}".format(data[molecule][atom][2])
            outStr += "  1.00" + "  0.00"
            outStr += "           "
            if atom == 0:
                outStr += " O  "
            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
    range(len(data)) ]
528     print "Reordering Molecules..."
529     for i in range(len(data)):
530         positions = getTetrahedronPositions(data[i])
531         zeroedMol = newRandOrientation(positions)
532         newMol = resetOrientation( data[i][0], zeroedMol )
533         newData[i] = newMol
534     print "Molecules Reordered"
535     connectedMolecules = getNeighbors(newData)  # -1 index = not
    neighboring
536     finalData = newData
537     for i in range(len(connectedMolecules)):
538         # print "check defects"
539         isFine = isDefectiveCheck(err, connectedMolecules, finalData, i)
540         # print "isFINE CHECK", isFine
541         if not isFine:
542             # print "fixing defects"
543             while not isFine:
544                 # print "RerunMol"
545                 isFine, finalData = rerunMolAndNeighbors(err,
    connectedMolecules, finalData, i)
546                 # print "rerunDone"
547     writeDataPDB(finalData, pdbType)
548     # printData(newData)
549 
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
555     print "Good Arguments, Initializing Reorientiation with", maxErr, "
    maximum defects"
556     # testRun(pdbIN, maxErr, pdbOUT)
557     runPgm(pdbIN, maxErr)
558 elif badArgs:
559     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.

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

Title: hexagermane_transall_first_reorder system

0  1
Ge        -4.543000000000      -0.076000000000       0.598000000000
Ge        -2.121000000000       0.068000000000       0.086000000000
 C        -4.774000000000      -1.010000000000       2.336000000000
 C        -5.549000000000      -1.127000000000      -0.760000000000
 C        -5.371000000000       1.719000000000       0.829000000000
 C        -1.311000000000       1.357000000000       1.303000000000
Ge        -0.754000000000      -1.978000000000       0.323000000000
 C        -1.340000000000      -2.994000000000       1.865000000000
Ge         1.675000000000      -1.503000000000       0.567000000000
 C        -1.186000000000      -3.039000000000      -1.235000000000
 C         1.925000000000      -0.300000000000       2.094000000000
 C         2.315000000000      -0.535000000000      -0.983000000000
Ge         3.126000000000      -3.539000000000       0.606000000000
 C         2.522000000000      -4.845000000000       1.926000000000
Ge         5.558000000000      -3.333000000000       1.109000000000
 C         2.938000000000      -4.327000000000      -1.151000000000
 C         6.648000000000      -2.029000000000       0.052000000000
 C         6.362000000000      -5.157000000000       0.995000000000
 C         5.463000000000      -2.780000000000       3.007000000000
 C        -4.808000000000      -2.522000000000       2.129000000000
 C        -6.030000000000      -0.572000000000       3.098000000000
 H        -3.909000000000      -0.757000000000       2.974000000000
 H        -6.954000000000      -0.777000000000       2.525000000000
 H        -5.990000000000       0.508000000000       3.335000000000
 H        -6.094000000000      -1.114000000000       4.066000000000
 H        -5.710000000000      -2.829000000000       1.564000000000
 H        -4.817000000000      -3.048000000000       3.107000000000
 H        -3.927000000000      -2.841000000000       1.555000000000
 H        -5.250000000000      -2.187000000000      -0.668000000000
 C        -5.199000000000      -0.705000000000      -2.186000000000
 C        -7.061000000000      -1.057000000000      -0.533000000000
```

H	−4.126000000000	−0.879000000000	−2.361000000000
H	−5.433000000000	0.356000000000	−2.385000000000
H	−5.754000000000	−1.331000000000	−2.917000000000
H	−7.327000000000	−1.462000000000	0.459000000000
H	−7.588000000000	−1.677000000000	−1.290000000000
H	−7.438000000000	−0.019000000000	−0.608000000000
C	5.171000000000	−1.286000000000	3.108000000000
C	6.703000000000	−3.142000000000	3.823000000000
H	4.612000000000	−3.336000000000	3.456000000000
H	7.617000000000	−2.660000000000	3.425000000000
H	6.834000000000	−4.242000000000	3.835000000000
H	6.569000000000	−2.820000000000	4.877000000000
H	6.003000000000	−0.673000000000	2.715000000000
H	4.988000000000	−0.995000000000	4.164000000000
H	4.281000000000	−1.055000000000	2.511000000000
C	6.881000000000	−2.491000000000	−1.386000000000
C	8.000000000000	−1.692000000000	0.695000000000
H	6.093000000000	−1.084000000000	0.027000000000
H	8.663000000000	−2.569000000000	0.776000000000
H	7.871000000000	−1.257000000000	1.700000000000
H	8.527000000000	−0.927000000000	0.085000000000
H	7.531000000000	−3.384000000000	−1.416000000000
H	7.387000000000	−1.690000000000	−1.966000000000
H	5.929000000000	−2.720000000000	−1.888000000000
C	6.042000000000	−5.844000000000	−0.340000000000
H	5.912000000000	−5.747000000000	1.817000000000
C	7.882000000000	−5.177000000000	1.193000000000
H	8.397000000000	−4.688000000000	0.346000000000
H	8.240000000000	−6.228000000000	1.229000000000
H	8.189000000000	−4.690000000000	2.134000000000
H	4.959000000000	−5.905000000000	−0.514000000000
H	6.436000000000	−6.883000000000	−0.337000000000
H	6.487000000000	−5.311000000000	−1.199000000000
H	−6.362000000000	1.563000000000	1.303000000000
C	−5.646000000000	2.456000000000	−0.483000000000
C	−4.523000000000	2.590000000000	1.756000000000
H	−4.349000000000	2.080000000000	2.725000000000
H	−5.042000000000	3.550000000000	1.960000000000
H	−3.548000000000	2.821000000000	1.285000000000
H	−6.358000000000	1.894000000000	−1.110000000000
H	−4.725000000000	2.629000000000	−1.057000000000
H	−6.117000000000	3.440000000000	−0.273000000000
C	−0.532000000000	2.421000000000	0.817000000000
C	−1.529000000000	1.258000000000	2.684000000000
H	−2.129000000000	0.469000000000	3.088000000000
C	−0.996000000000	2.206000000000	3.561000000000
C	−0.001000000000	3.371000000000	1.694000000000
C	−0.237000000000	3.267000000000	3.066000000000
H	0.596000000000	4.188000000000	1.310000000000
H	−1.180000000000	2.122000000000	4.624000000000
H	0.174000000000	4.002000000000	3.745000000000
C	−1.777000000000	−4.322000000000	1.725000000000
C	−0.217000000000	−3.392000000000	−2.175000000000
C	−2.232000000000	−5.037000000000	2.838000000000

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

```
H          2.603000000000       1.384000000000       0.900000000000
H          3.001000000000       2.867000000000       2.795000000000
H          2.548000000000       2.101000000000       5.105000000000
C         -2.041000000000       0.841000000000      -1.709000000000
C         -1.767000000000       0.059000000000      -2.841000000000
C         -2.300000000000       2.209000000000      -1.888000000000
C         -1.732000000000       0.632000000000      -4.115000000000
H         -1.595000000000      -0.996000000000      -2.753000000000
C         -2.263000000000       2.785000000000      -3.160000000000
H         -2.521000000000       2.839000000000      -1.039000000000
C         -1.977000000000       1.997000000000      -4.275000000000
H         -1.519000000000       0.016000000000      -4.979000000000
H         -2.458000000000       3.843000000000      -3.281000000000
H         -1.950000000000       2.441000000000      -5.262000000000
H         -0.322000000000       2.526000000000      -0.236000000000

EOF
formchk B3LYP_STO-3G_1_hexagermane_transall_first_reorder.chk
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
echo
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.

```
#!/usr/bin/python

import sys
import subprocess

# argument: sys.argv[num]
# Replacement: sed -i -e 's/IN/OUT/g' FILE > NEWFILE

inFile = file(sys.argv[1])

def DoIT():
    for first in {' C', 'Si', 'Ge'}:
        name1 = "%s" % (first.lstrip(' '))
        out1 = open(name1, "w")
        cmdStr = "sed -e 's/1 GE/1 %s/g' ./%s >> ./%s.pdb" % (first,
    inFile, name1)
        # subprocess.call(cmdStr, shell=True, stdout=out1)
        subprocess.Popen(cmdStr, shell=True, executable='/bin/bash')
        out1.close()
        for second in {' C', 'Si', 'Ge'}:
            name2 = name1 + "_%s" % (second.lstrip(' '))
            out2 = open(name2, "w")
```

```
            cmdStr = "sed -e 's/2 GE/2 %s/g' ./%s.pdb >> ./%s.pdb" % (second, name1, name2)
            # subprocess.call(cmdStr, shell=True, stdout=out2)
            subprocess.Popen(cmdStr, shell=True, executable='/bin/bash')
            out2.close()
            for third in {' C', 'Si', 'Ge'}:
                name3 = name2 + "_%s" % (third.lstrip(' '))
                out3 = open(name3, "w")
                cmdStr = "sed -e 's/3 GE/3 %s/g' ./%s.pdb >> ./%s.pdb" % (third, name2, name3)
                # subprocess.call(cmdStr, shell=True, stdout=out3)
                subprocess.Popen(cmdStr, shell=True, executable='/bin/bash')
                out3.close()
                for fourth in {' C', 'Si', 'Ge'}:
                    name4 = name3 + "_%s" % (fourth.lstrip(' '))
                    out4 = open(name4, "w")
                    cmdStr = "sed -e 's/4 GE/4 %s/g' ./%s.pdb >> ./%s.pdb" % (fourth, name3, name4)
                    # subprocess.call(cmdStr, shell=True, stdout=out4)
                    subprocess.Popen(cmdStr, shell=True, executable='/bin/bash')
                    out4.close()

DoIT()
```

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

```
from chimera import runCommand as rc
from chimera import replyobj
import sys
import os

#standard sys.argv[] for script args?
# sys.argv[0] = directory
os.chdir(sys.argv[0])

file_names = [fn for fn in os.listdir(".") if fn.endswith(".pdb")]
fn = file_names[0]
# inPDB = chimera.openModels.open('/Users/gentry/Desktop/test/testmol.pdb', type="PDB")

rc("open " + fn)

rc("rotation 1 reverse #0:1.HET@/serialNumber=2 #0:1.HET@/serialNumber=3")

for i in range(72):
    #replyobj.status("Processing " + fn)
    #rc("open " + fn)
    #rc("rotation 1 reverse #0:1.HET@/serialNumber=2 #0:1.HET@/serialNumber=3")
    rc("rotation 1 5")
    newName = (fn[:-3] + str((i*5)) + ".pdb")
```

```
    rc("write format pdb 0 " + newName)
    #rc("close ")


    # chimera.runCommand("rotation 2 3 5")
    # newName = ( inPDB[:-3] + i*5 + ".pdb" )
    # 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.

```
#!/usr/bin/python

### Author: Gentry Smith, Oklahoma State University
### Created: August 7, 2017, 3PM
### Last Edited: August 7, 2017

### Takes a stationary_points.txt file and will copy .pdb files of the same name from a split_conformers.pdb/ folder
### into a new folder "stationary_conformers"

# This does not use any args and instead relies on the stationary points file being "stationary_points.txt" and the
# conformrs residing in a "split_conformers.pdb/" directory on the same level. It will create the new folder "stationary_conformers"

import os

def IOValidator():
    returnBool = [False,False]
    try:
        file1 = open('stationary_points.txt', 'r')
        file1.close()
        returnBool[0] = True
    except IOError:
        print("Did not find 'stationary_points.txt' file. Quitting...")
        quit()
    try:
        wkdir = os.getcwd()
        file2 = os.chdir('split_conformers.pdb')
        os.chdir(wkdir)
        returnBool[1] = True
    except OSError:
        print("Did not find 'split_conformers.pdb' folder. Quitting...")
        quit()
    if returnBool[0] & returnBool[1]:
        return True
    else:
```

```
        return False


def GetPDBs():
    pdbNames = []
    inFile = open('stationary_points.txt', 'r')
    for line in inFile:
        pdbNames.append(line.split()[1])
    return pdbNames


def CopyPDBs(pdbList):
    wkdir = os.getcwd()
    for i in range(len(pdbList)):
        pstring = ( 'cp ' + 'split_conformers.pdb/' + str(pdbList[i]) +
    ' stationary_conformers/' )
        os.popen(pstring)


def Runner():
    if IOValidator():
        print('Valid Args. Running...')
        pdbList = GetPDBs()
        try:
            os.mkdir('stationary_conformers')
            CopyPDBs(pdbList)
        except OSError:
            print("'stationary_conformers' directory already exists.
    Erase directory and run again. Quitting...")
            quit()


Runner()
```

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

```
#!/usr/bin/python

### Author: Gentry Smith, Oklahoma State University
### Created: July 31, 2017, 12PM
### Last Edited: July 31, 2017

### takes file arg with format [ [energy] [pdb_name] ], alters to [ [
    energy]  [torsion] ], and creates copy with
###     [ [relative energy]  [torsion] ].

import sys


def IOValidator():
    isValid = False
    try:
        inFile = sys.argv[1]
        isValid = True
    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) ):
            crits.append(energies[i-1])
            tors.append(torsions[i-1])
            isIncreasing = not isIncreasing
        prev = float(energies[i])
    returnThing = [crits, tors]
    return returnThing


def MakeFile(energies, torsions, fileName):
    outFile = open(fileName, 'w')
    for i in range(len(energies)):
        strOut = ('{:.11e}'.format(energies[i]) + " " + str(torsions[i])
    + "\n")
        outFile.write(strOut)
    outFile.close()


def Runner():
    if IOValidator():
        energies = GetFileData()
        torsions = [180]
        i = 185
        while i != 180:
            if i == 360:
                i = 0
            torsions.append(i)
            i = i + 5
        MakeFile(energies, torsions, 'abs_energ.txt')
        relativeEnergies = Relativize(energies)
        MakeFile(relativeEnergies, torsions, 'rel_energ.txt')
        MakeFile(UnifiedScale(relativeEnergies), torsions, 'uni_energ.
    txt')
        crits = CriticalHit(relativeEnergies, torsions)
        MakeFile(crits[0], crits[1], 'crit_pts.txt')


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.

```
#!/usr/bin/python

### Author: Gentry Smith, Oklahoma State University
### Created: July 31, 2017, 3PM
### Last Edited: August 1, 2017

### Takes data created by teatAbsEnergies and compares values via
    additive and multiplicative comparison
###     with abs or rel data. Math in terms of File 2 sub/div File 1.

# sys.argv[1] = file 1, working directory here.
# 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]))
```

```
    elif func == 1:
        for i in range(len(data2)):
            try:
                newData.append(float(data2[i] / data1[i]))
            except ZeroDivisionError:
                newData.append(0.0)
    return newData


def WriteFile(data1, data2, tors, compData, comp, sigs):
    # writes data of comparison. Format:
    #   File1 = {file1}
    #   File2 = {file2}
    #   Source: {absolute, relative}
    #   Comparison: {additive, multiplicative}
    #   comp: {min/max/avg/stdev of all comp values}
    #   Raw Data: {includes header of File1, File2, Torsions, Comp defining each column}
    # print("Writing file...")
    # print('File2=' + str((sys.argv[2]).split("/")))
    source = ""
    if str(sys.argv[1])[:3] == "abs":
        source = "absolute"
    elif str(sys.argv[1])[:3] == "rel":
        source = "relative"
    elif str(sys.argv[1])[:3] == "uni":
        source = "unified relative scale"
    else:
        print(str(sys.argv[1])[:2])
    comparison = ""
    if comp == 0:
        comparison = "additive"
    elif comp == 1:
        comparison = "multiplicative"
    headerLines = [0]*10
    headerLines[0] = ('File1 = ' + sys.argv[1] + '\n')
    headerLines[1] = ('File2 = ' + sys.argv[2] + '\n')
    headerLines[2] = ('Source: ' + source + '\n')
    headerLines[3] = ('Comparison: ' + comparison + '\n')
    headerLines[4] = ('Comparison min: ' + str(sigs[0]) + '\n')
    headerLines[5] = ('Comparison max: ' + str(sigs[1]) + '\n')
    headerLines[6] = ('Comparison avg: ' + str(sigs[2]) + '\n')
    headerLines[7] = ('Comparison stdev: ' + str(sigs[3]) + '\n')
    headerLines[8] = ('Raw Data:' + '\n')
    f1ColSize = len(str(data1[0]))
    f2ColSize = len(str(data2[0]))
    headerLines[9] = ('File1'.ljust(18) + 'File2'.ljust(18) + 'Tors'.ljust(5) + 'Comp'.ljust(18) + '\n')
    fileName = (str((sys.argv[2]).split("/")[-2]) + "_" + str(sys.argv[1])[:3] + "_" + comparison + '.txt')
    outFile = open(fileName, 'w')
    for i in range(len(headerLines)):
        outFile.write(str(headerLines[i]))
    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])
    [:17].ljust(18) + str(tors[i]).ljust(5) + str(compData[i])[:17].
    ljust(18) + '\n')
119         outFile.write(string)
120
121
122 def GetCompSigs(data):
123     sigs = []
124     sigs.append(min(data))
125     sigs.append(max(data))
126     sigs.append((float(sum(data))/float(len(data))))
127     sigs.append(numpy.std(data, axis=0))
128     return sigs
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

```
HEADER
REMARK Title: hexagermane_transall system
HETATM    1 Ge             1      -4.399    0.008    0.355  0.00  0.00          Ge
HETATM    2 Ge             1      -1.965    0.138   -0.022  0.00  0.00          Ge
HETATM    3  C             1      -4.822    1.886    0.961  0.00  0.00           C
HETATM    4  C             1      -5.008   -1.297    1.715  0.00  0.00           C
HETATM    5  C             1      -5.256   -0.261   -1.445  0.00  0.00           C
HETATM    6  C             1      -1.213    1.435    1.157  0.00  0.00           C
HETATM    7 Ge             1      -0.756   -1.988    0.223  0.00  0.00          Ge
HETATM    8  C             1      -1.297   -2.917    1.805  0.00  0.00           C
HETATM    9 Ge             1       1.647   -1.496    0.371  0.00  0.00          Ge
HETATM   10  C             1      -1.182   -3.010   -1.339  0.00  0.00           C
HETATM   11  C             1       2.131   -0.425    1.877  0.00  0.00           C
HETATM   12  C             1       2.111   -0.634   -1.269  0.00  0.00           C
HETATM   13 Ge             1       2.889   -3.585    0.738  0.00  0.00          Ge
HETATM   14  C             1       2.287   -4.358    2.378  0.00  0.00           C
HETATM   15 Ge             1       5.327   -3.386    1.080  0.00  0.00          Ge
HETATM   16  C             1       2.766   -4.685   -0.813  0.00  0.00           C
HETATM   17  C             1       5.688   -2.615    2.887  0.00  0.00           C
HETATM   18  C             1       6.239   -2.415   -0.417  0.00  0.00           C
HETATM   19  C             1       5.893   -5.324    0.888  0.00  0.00           C
HETATM   20  C             1      -3.527    2.543    1.328  0.00  0.00           C
HETATM   21  C             1      -5.754    1.844    2.133  0.00  0.00           C
HETATM   22  H             1      -5.303    2.355    0.072  0.00  0.00           H
HETATM   23  H             1      -5.269    1.358    2.999  0.00  0.00           H
HETATM   24  H             1      -6.679    1.287    1.913  0.00  0.00           H
HETATM   25  H             1      -6.047    2.856    2.449  0.00  0.00           H
HETATM   26  H             1      -3.043    2.019    2.171  0.00  0.00           H
HETATM   27  H             1      -3.683    3.585    1.642  0.00  0.00           H
HETATM   28  H             1      -2.818    2.559    0.490  0.00  0.00           H
HETATM   29  H             1      -4.336   -1.167    2.589  0.00  0.00           H
HETATM   30  C             1      -4.907   -2.680    1.154  0.00  0.00           C
HETATM   31  C             1      -6.417   -0.909    2.051  0.00  0.00           C
HETATM   32  H             1      -3.858   -2.985    0.982  0.00  0.00           H
HETATM   33  H             1      -5.433   -2.773    0.187  0.00  0.00           H
HETATM   34  H             1      -5.349   -3.420    1.836  0.00  0.00           H
HETATM   35  H             1      -6.488    0.167    2.302  0.00  0.00           H
HETATM   36  H             1      -6.802   -1.477    2.909  0.00  0.00           H
HETATM   37  H             1      -7.103   -1.094    1.205  0.00  0.00           H
HETATM   38  C             1       5.200   -6.127    1.944  0.00  0.00           C
HETATM   39  C             1       7.384   -5.453    0.966  0.00  0.00           C
HETATM   40  H             1       5.523   -5.590   -0.126  0.00  0.00           H
HETATM   41  H             1       7.790   -4.974    1.874  0.00  0.00           H
HETATM   42  H             1       7.885   -4.994    0.099  0.00  0.00           H
HETATM   43  H             1       7.691   -6.509    0.992  0.00  0.00           H
```

```
HETATM   44  H   1      5.502   −5.821   2.960  0.00  0.00           H
HETATM   45  H   1      5.436   −7.197   1.849  0.00  0.00           H
HETATM   46  H   1      4.106   −6.027   1.879  0.00  0.00           H
HETATM   47  C   1      6.243   −1.232   2.746  0.00  0.00           C
HETATM   48  C   1      6.612   −3.524   3.636  0.00  0.00           C
HETATM   49  H   1      4.684   −2.582   3.376  0.00  0.00           H
HETATM   50  H   1      7.535   −3.731   3.068  0.00  0.00           H
HETATM   51  H   1      6.139   −4.497   3.853  0.00  0.00           H
HETATM   52  H   1      6.913   −3.088   4.599  0.00  0.00           H
HETATM   53  H   1      7.243   −1.234   2.279  0.00  0.00           H
HETATM   54  H   1      6.347   −0.742   3.725  0.00  0.00           H
HETATM   55  H   1      5.589   −0.589   2.128  0.00  0.00           H
HETATM   56  C   1      5.630   −1.055   −0.555 0.00  0.00           C
HETATM   57  H   1      6.024   −3.039   −1.315 0.00  0.00           H
HETATM   58  C   1      7.712   −2.342   −0.145 0.00  0.00           C
HETATM   59  H   1      7.923   −1.890   0.839  0.00  0.00           H
HETATM   60  H   1      8.227   −1.728   −0.898 0.00  0.00           H
HETATM   61  H   1      8.188   −3.335   −0.163 0.00  0.00           H
HETATM   62  H   1      4.573   −1.106   −0.861 0.00  0.00           H
HETATM   63  H   1      6.155   −0.455   −1.314 0.00  0.00           H
HETATM   64  H   1      5.675   −0.486   0.391  0.00  0.00           H
HETATM   65  H   1      −5.890  −1.163   −1.302 0.00  0.00           H
HETATM   66  C   1      −4.220  −0.487   −2.505 0.00  0.00           C
HETATM   67  C   1      −6.093  0.945    −1.729 0.00  0.00           C
HETATM   68  H   1      −6.841  1.122    −0.939 0.00  0.00           H
HETATM   69  H   1      −6.644  0.838    −2.676 0.00  0.00           H
HETATM   70  H   1      −5.478  1.858    −1.818 0.00  0.00           H
HETATM   71  H   1      −3.754  −1.481   −2.414 0.00  0.00           H
HETATM   72  H   1      −3.411  0.262    −2.459 0.00  0.00           H
HETATM   73  H   1      −4.659  −0.429   −3.512 0.00  0.00           H
HETATM   74  C   1      −1.706  1.681    2.429  0.00  0.00           C
HETATM   75  C   1      −0.128  2.155    0.679  0.00  0.00           C
HETATM   76  H   1      0.268   1.941    −0.323 0.00  0.00           H
HETATM   77  C   1      0.451   3.147    1.465  0.00  0.00           C
HETATM   78  C   1      −1.134  2.678    3.216  0.00  0.00           C
HETATM   79  C   1      −0.058  3.415    2.731  0.00  0.00           C
HETATM   80  H   1      −1.525  2.873    4.219  0.00  0.00           H
HETATM   81  H   1      1.306   3.716    1.086  0.00  0.00           H
HETATM   82  H   1      0.391   4.199    3.349  0.00  0.00           H
HETATM   83  C   1      −1.557  −4.274   1.694  0.00  0.00           C
HETATM   84  C   1      −0.365  −3.088   −2.455 0.00  0.00           C
HETATM   85  C   1      −2.027  −4.985   2.796  0.00  0.00           C
HETATM   86  H   1      −1.378  −4.792   0.742  0.00  0.00           H
HETATM   87  C   1      −1.446  −2.277   3.025  0.00  0.00           C
HETATM   88  C   1      −0.752  −3.866   −3.544 0.00  0.00           C
HETATM   89  H   1      0.592   −2.548   −2.482 0.00  0.00           H
HETATM   90  C   1      −2.386  −3.699   −1.304 0.00  0.00           C
HETATM   91  C   1      −2.219  −4.336   4.011  0.00  0.00           C
HETATM   92  H   1      −2.237  −6.056   2.707  0.00  0.00           H
HETATM   93  C   1      −1.915  −2.983   4.130  0.00  0.00           C
HETATM   94  H   1      −1.159  −1.217   3.132  0.00  0.00           H
HETATM   95  C   1      −2.771  −4.484   −2.388 0.00  0.00           C
HETATM   96  H   1      −3.043  −3.610   −0.422 0.00  0.00           H
HETATM   97  C   1      −1.952  −4.568   −3.509 0.00  0.00           C
HETATM   98  H   1      −0.105  −3.928   −4.425 0.00  0.00           H
HETATM   99  H   1      −3.721  −5.027   −2.358 0.00  0.00           H
HETATM  100  H   1      −2.253  −5.182   −4.364 0.00  0.00           H
HETATM  101  H   1      −2.596  −4.891   4.876  0.00  0.00           H
HETATM  102  H   1      −2.041  −2.474   5.091  0.00  0.00           H
HETATM  103  C   1      2.487   −3.679   3.571  0.00  0.00           C
HETATM  104  C   1      1.701   −5.563   −0.935 0.00  0.00           C
HETATM  105  C   1      3.733   −4.618   −1.807 0.00  0.00           C
HETATM  106  H   1      0.940   −5.615   −0.140 0.00  0.00           H
HETATM  107  C   1      1.598   −6.382   −2.057 0.00  0.00           C
HETATM  108  C   1      1.690   −5.609   2.382  0.00  0.00           C
HETATM  109  C   1      2.102   −4.259   4.776  0.00  0.00           C
HETATM  110  H   1      2.956   −2.680   3.567  0.00  0.00           H
HETATM  111  C   1      1.520   −5.523   4.784  0.00  0.00           C
```

```
HETATM  112  H          1       2.260  -3.721   5.716  0.00  0.00           H
HETATM  113  C          1       1.311  -6.197   3.585  0.00  0.00           C
HETATM  114  H          1       1.504  -6.131   1.431  0.00  0.00           H
HETATM  115  C          1       2.562  -6.313  -3.057  0.00  0.00           C
HETATM  116  H          1       0.754  -7.074  -2.153  0.00  0.00           H
HETATM  117  C          1       3.630  -5.430  -2.933  0.00  0.00           C
HETATM  118  H          1       4.590  -3.931  -1.700  0.00  0.00           H
HETATM  119  H          1       4.391  -5.376  -3.718  0.00  0.00           H
HETATM  120  H          1       2.481  -6.954  -3.941  0.00  0.00           H
HETATM  121  H          1       1.223  -5.984   5.731  0.00  0.00           H
HETATM  122  H          1       0.844  -7.187   3.587  0.00  0.00           H
HETATM  123  C          1       1.878   0.732  -1.306  0.00  0.00           C
HETATM  124  C          1       1.530  -0.534   3.120  0.00  0.00           C
HETATM  125  C          1       2.642  -1.289  -2.370  0.00  0.00           C
HETATM  126  C          1       2.179   1.455  -2.458  0.00  0.00           C
HETATM  127  H          1       1.444   1.239  -0.432  0.00  0.00           H
HETATM  128  C          1       3.179   0.461   1.679  0.00  0.00           C
HETATM  129  C          1       2.005   0.227   4.186  0.00  0.00           C
HETATM  130  H          1       0.661  -1.197   3.265  0.00  0.00           H
HETATM  131  C          1       2.940  -0.568  -3.524  0.00  0.00           C
HETATM  132  H          1       2.840  -2.370  -2.334  0.00  0.00           H
HETATM  133  C          1       2.710   0.804  -3.567  0.00  0.00           C
HETATM  134  H          1       1.989   2.533  -2.491  0.00  0.00           H
HETATM  135  H          1       3.358  -1.081  -4.396  0.00  0.00           H
HETATM  136  H          1       2.944   1.370  -4.475  0.00  0.00           H
HETATM  137  C          1       3.067   1.105   3.998  0.00  0.00           C
HETATM  138  H          1       1.534   0.140   5.170  0.00  0.00           H
HETATM  139  C          1       3.650   1.229   2.740  0.00  0.00           C
HETATM  140  H          1       3.633   0.553   0.682  0.00  0.00           H
HETATM  141  H          1       4.480   1.926   2.585  0.00  0.00           H
HETATM  142  H          1       3.439   1.703   4.836  0.00  0.00           H
HETATM  143  C          1      -2.039   0.838  -1.804  0.00  0.00           C
HETATM  144  C          1      -1.525   0.195  -2.916  0.00  0.00           C
HETATM  145  C          1      -2.655   2.077  -1.927  0.00  0.00           C
HETATM  146  C          1      -1.618   0.802  -4.168  0.00  0.00           C
HETATM  147  H          1      -1.048  -0.789  -2.818  0.00  0.00           H
HETATM  148  C          1      -2.746   2.686  -3.175  0.00  0.00           C
HETATM  149  H          1      -3.084   2.566  -1.036  0.00  0.00           H
HETATM  150  C          1      -2.223   2.047  -4.296  0.00  0.00           C
HETATM  151  H          1      -1.210   0.296  -5.049  0.00  0.00           H
HETATM  152  H          1      -3.229   3.663  -3.275  0.00  0.00           H
HETATM  153  H          1      -2.292   2.524  -5.279  0.00  0.00           H
HETATM  154  H          1      -2.539   1.081   2.827  0.00  0.00           H
CONECT    3    1   20   21   22
CONECT    4    1   29   30   31
CONECT    5    1   65   66   67
CONECT    6   74   75    2
CONECT    8   83   87    7
CONECT   10   84   90    7
CONECT   11  124  128    9
CONECT   12  123  125    9
CONECT   14  103  108   13
CONECT   16  104  105   13
CONECT   17   15   47   48   49
CONECT   18   56   57   58   15
CONECT   19   15   38   39   40
CONECT   20    3   26   27   28
CONECT   21    3   23   24   25
CONECT   30    4   34   32   33
CONECT   31    4   35   36   37
CONECT   38   19   44   45   46
CONECT   39   19   41   42   43
CONECT   47   54   55   17   53
CONECT   48   17   50   51   52
CONECT   56   62   63   64   18
CONECT   58   59   60   61   18
CONECT   66   71   72   73    5
CONECT   67   68   69   70    5
```

```
CONECT   74   78  154    6
CONECT   75   76   77    6
CONECT   77   75   79   81
CONECT   78   74   79   80
CONECT   79   77   78   82
CONECT   83   85   86    8
CONECT   84   88   89   10
CONECT   85   83   91   92
CONECT   87   93   94    8
CONECT   88   84   97   98
CONECT   90   95   96   10
CONECT   91   85   93  101
CONECT   93   87   91  102
CONECT   95   90   97   99
CONECT   97   88   95  100
CONECT  103  109  110   14
CONECT  104  106  107   16
CONECT  105  117  118   16
CONECT  107  104  115  116
CONECT  108  113  114   14
CONECT  109  103  111  112
CONECT  111  109  113  121
CONECT  113  108  111  122
CONECT  115  107  117  120
CONECT  117  105  115  119
CONECT  123  126  127   12
CONECT  124  129  130   11
CONECT  125  131  132   12
CONECT  126  123  133  134
CONECT  128  139  140   11
CONECT  129  124  137  138
CONECT  131  125  133  135
CONECT  133  126  131  136
CONECT  137  129  139  142
CONECT  139  128  137  141
CONECT  143  144  145    2
CONECT  144  143  146  147
CONECT  145  143  148  149
CONECT  146  144  150  151
CONECT  148  145  150  152
CONECT  150  146  148  153
CONECT    1    2    3    4    5
CONECT    2    1  143    6    7
CONECT    7    2    8    9   10
CONECT    9    7   11   12   13
CONECT   13    9   14   15   16
CONECT   15   13   17   18   19
CONECT   22    3
CONECT   23   21
CONECT   24   21
CONECT   25   21
CONECT   26   20
CONECT   27   20
CONECT   28   20
CONECT   29    4
CONECT   32   30
CONECT   33   30
CONECT   34   30
CONECT   35   31
CONECT   36   31
CONECT   37   31
CONECT   40   19
CONECT   41   39
CONECT   42   39
CONECT   43   39
CONECT   44   38
CONECT   45   38
CONECT   46   38
```

```
CONECT   49   17
CONECT   50   48
CONECT   51   48
CONECT   52   48
CONECT   53   47
CONECT   54   47
CONECT   55   47
CONECT   57   18
CONECT   59   58
CONECT   60   58
CONECT   61   58
CONECT   62   56
CONECT   63   56
CONECT   64   56
CONECT   65    5
CONECT   68   67
CONECT   69   67
CONECT   70   67
CONECT   71   66
CONECT   72   66
CONECT   73   66
CONECT   76   75
CONECT   80   78
CONECT   81   77
CONECT   82   79
CONECT   86   83
CONECT   89   84
CONECT   92   85
CONECT   94   87
CONECT   96   90
CONECT   98   88
CONECT   99   95
CONECT  100   97
CONECT  101   91
CONECT  102   93
CONECT  106  104
CONECT  110  103
CONECT  112  109
CONECT  114  108
CONECT  116  107
CONECT  118  105
CONECT  119  117
CONECT  120  115
CONECT  121  111
CONECT  122  113
CONECT  127  123
CONECT  130  124
CONECT  132  125
CONECT  134  126
CONECT  135  131
CONECT  136  133
CONECT  138  129
CONECT  140  128
CONECT  141  139
CONECT  142  137
CONECT  147  144
CONECT  149  145
CONECT  151  146
CONECT  152  148
CONECT  153  150
CONECT  154   74
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

```
COMPND    UNNAMED
AUTHOR    GENERATED BY OPEN BABEL 2.3.90
HETATM    1 GE   UNL     1      -3.520   1.842  -0.078  1.00  0.00          Ge3-
HETATM    2 GE   UNL     1      -1.368   2.888  -0.034  1.00  0.00          Ge2-
HETATM    3 GE   UNL     1       0.324   1.200   0.059  1.00  0.00          Ge3-
HETATM    4 GE   UNL     1       2.475   2.248   0.099  1.00  0.00          Ge
HETATM    5  H   UNL     1      -4.622   2.930  -0.135  1.00  0.00           H
HETATM    6  H   UNL     1      -3.699   0.985   1.202  1.00  0.00           H
HETATM    7  H   UNL     1      -3.621   0.932  -1.328  1.00  0.00           H
HETATM    8  H   UNL     1      -1.258   3.797   1.217  1.00  0.00           H
HETATM    9  H   UNL     1      -1.178   3.740  -1.314  1.00  0.00           H
HETATM   10  H   UNL     1       0.213   0.288  -1.189  1.00  0.00           H
HETATM   11  H   UNL     1       0.135   0.352   1.342  1.00  0.00           H
HETATM   12  H   UNL     1       2.655   3.095  -1.186  1.00  0.00           H
HETATM   13  H   UNL     1       3.578   1.161   0.165  1.00  0.00           H
HETATM   14  H   UNL     1       2.574   3.167   1.343  1.00  0.00           H
CONECT    1    2    5    6    7
CONECT    2    1    3    8    9
CONECT    3    2    4   10   11
CONECT    4    3   12   13   14
CONECT    5    1
CONECT    6    1
CONECT    7    1
CONECT    8    2
CONECT    9    2
CONECT   10    3
CONECT   11    3
CONECT   12    4
CONECT   13    4
CONECT   14    4
MASTER        0    0    0    0    0    0    0    0   14    0   14    0
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 IV.2.

```
### Author: Gentry Smith
### Date: April 22, 2017
### Description: This is the runner file that is the primary executable
    for the torsion minimizer. Currently is the
###     only file utilized.

# Inputs:
# Arg1: the molecule file to be minimized (currently only accepts a pdb
    file)

import sys
import subprocess
import math

# IO Validator: validates user-submitted molecule.
def IOValidator():
```

```
15      isValid = False
16      # Check for valid length of args (2)
17      if len(sys.argv) == 2:
18          # Check arg to make sure it's a file.
19          argFile = sys.argv[1]
20          try:
21              inputFile = open(argFile)
22              # Finally, make sure the file is a .pdb
23              if inputFile[-4:] == ".pdb":
24                  isValid = True
25              else:
26                  print("This is not a .pdb file. Please try again with a
    .pdb file.\n")
27              inputFile.close()
28          except IOError:
29              print("System was not able to open '", str(argFile), "'.")
30      # too long
31      elif len(sys.argv) > 2:
32          print("You have too many arguments. Call the file as 'Runner.py
    [molecule file]' and try again.\n")
33      # too short
34      else:
35          print("You do not have enough arguments. Start the program as '
    Runner.py [molecule file]' and try again.\n")
36      # return validity boolean
37      return isValid
38
39  # Get Torsions: initiates function to get user-specified torsion bonds.
    Returns bonds as int[[a,b],[a,b]] list
40  def getTorsions():
41      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
    declare which bonds you would like to rotate.\n")
51              print("Before going any further, it's important to note at
    this time that version 0.2 (current) will assume the torsions you
    enter are completely correct. You'll see a bunch of error messages
    soon if it isn't correct.\n")
52              print("Open the .pdb file and identify the numbers of the
    atoms on the .pdb that will make the bond (the first number on the
    line of each atom)\n\n")
53              print("Now it's time to enter in the numbers of the two
    atoms. We'll do it one at a time.")
54
55              firstTor = raw_input("Type in the number of the first atom
    in the bond and hit enter. \nEx: type 3 and then hit enter.\n")
56
```

```
57              try:
58                  confFirstTor = int(firstTor)
59              except ValueError:
60                  print("You typed in '", firstTor, "', which is not a
    number. Let's start again.")
61                  badIn = True
62
63              secondTor = raw_input("Type in the number of the second atom
     in the bond and hit enter. \nEx: type 3 and then hit enter.\n")
64
65              try:
66                  confSecondTor = int(secondTor)
67              except ValueError:
68                  print("You typed in '", secondTor, "', which is not a
    number. Let's start again.")
69                  badIn = True
70              firstTime = False
71
72          else:
73              print("Open the .pdb file and identify the numbers of the
    atoms on the .pdb that will make the bond (the first number on the
    line of each atom)\n\n")
74
75              firstTor = raw_input("Type in the number of the first atom
    in the bond and hit enter. \nEx: type 3 and then hit enter.\n")
76
77              try:
78                  confFirstTor = int(firstTor)
79              except ValueError:
80                  print("You typed in '", firstTor, "', which is not a
    number. Let's start again.")
81                  badIn = True
82
83              secondTor = raw_input("Type in the number of the second atom
     in the bond and hit enter. \nEx: type 3 and then hit enter.\n")
84
85              try:
86                  confSecondTor = int(secondTor)
87              except ValueError:
88                  print("You typed in '", secondTor, "', which is not a
    number. Let's start again.")
89                  badIn = True
90              firstTime = False
91
92          if badIn == False:
93              newTorsion  = [confFirstTor, confSecondTor]
94              if torsions == [[0, 0]]:
95                  print("You added a new torsion: ", newTorsion, "\n")
96                  torsions = newTorsion
97              else:
98                  torsions.append(newTorsion)
99                  print("The current torsions you have created are:\n")
100                 for each in torsions:
101                     print(each, "\n")
```

```
102             doneCheck = raw_input("If you would like to add another
    torsion, press enter. If you are finished adding torsions, type '
    done' and press enter\n")
103 
104             if str(doneCheck) == "done":
105                 print("Finished entering torsions. Begining the work.\n"
    )
106             else:
107                 newTorsion = "first"
108 
109         if badIn == True:
110             firstTime = True
111             badIn = False
112             newTorsion = "first"
113 
114     return torsions
115 
116 # Get Conformation Count: determines conformations needed. Returns list
    in form: [#conf, rotDeg, rotRng]
117 def getConformationInfo(depth, torsions):
118     # rotates 60 degrees on the first search, then logarithmic decrease
    from 10 for each subsequent search.
119     rotDeg = [60, 10]
120     # full torsion range for first search, logarithmic decrease from 50
    for each subsequent search
121     rotRng = [360, 50]
122     # number of conformations needed
123     numConf = 0
124     # degrees per rotation
125     deg = 0
126     # 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 def Launcher():
```

```
150     valid = IOValidator()
151     if valid:
152         # do everything
153         depth = 0
154 
155         InitWD()
156 
157 
158     else:
159         print("There was a problem while reading in the molecule file.
    Please try again.\n")
160         exit()
161 
162 
163 # Initiates proper working directory.
164 def InitWD():
165 
166 
167 # Recursive search through molecule torsions
168 def RecursiveSearch(depth):
169 
170     torsions = getTorsions()
171 
172 
173 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:

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

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

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