Automation of Quantum Chemistry Workflows

by

Victoria M. Ingman

(Under the Direction of Steven E. Wheeler)

Abstract

Quantum chemistry has made tremendous strides in recent years, and modern methods now enable accurate predictions of the properties of a wide range of chemical systems. However, what now limits the use of quantum chemistry in many cutting-edge chemical applications is the sheer number of computations required to provide predictions ahead of experiment. One potential solution to this growing problem is the automation of routine tasks or even entire quantum chemistry workflows. Toward this end, we describe the development of robust command line and graphical tools to manipulate molecular structures, submit and monitor jobs, and automate quantum chemistry workflows. First, we describe QChASM, our suite of free, open-source tools for quantum chemistry automation and structure manipulation. Central to QChASM is AaronTools, a Python package for building and manipulating complex molecular structures and performing routine computational tasks. Next, we describe AaronJr, which is a new workflow manager for QChASM that provides a simple yet flexible command line interface to build and execute quantum chemistry workflows across many popular quantum chemistry packages. Finally, we show example applications that can be automated using AaronJr in order to showcase both the ease with which new workflows can be described and the power of AaronJr to automate a wide range of quantum chemistry applications. The ultimate hope is that by automating the mundane tasks that currently dominate day-to-day life of the computational chemist (building molecular structures, creating input files, submitting and monitoring jobs, checking and parsing output files, and analyzing results), QChASM will allow computational chemists to instead focus on the chemistry, not the computations.

Index words: quantum chemistry, density functional theory, automation tools

Automation of Quantum Chemistry Workflows

by

Victoria M. Ingman

B.S., University of Tennessee, 2016

A Dissertation Submitted to the Graduate Faculty of the
University of Georgia in Partial Fulfillment of the Requirements for the Degree

Doctor of Philosophy

Athens, Georgia

2021

Automation of Quantum Chemistry Workflows

by

Victoria M. Ingman

Major Professor:	Steven E. Wheeler
Committee:	Henry F. Schaefer III Kyle Johnsen

Dedication

To Charlie, for always believing in me. And to my family, for ensuring I had the opportunity to reach my full potential.

Contents

List of Figures

Chapter 1

Introduction

1.1 Virtual Screening of Chemical Systems

While advancements in computing are continuously making computational chemistry a cheaper and quicker way to investigate chemical systems, the manual application of these methods for many projects can be a daunting task. For example, generating the input files for the hundreds of transition state (TS) structures that are often necessary to estimate experimentally measurable properties, such as enantiomeric excess (*% ee*), is almost impossible without well-honed skills at the command line. Additionally, modifying each of these structures in a graphical molecule builder, such as GaussView[1] or Avogadro,[2] to test the effects of functional group modifications, alternative ligands, or reaction conditions, is an incredibly time consuming and error prone process. It follows that some amount of automation is necessary to efficiently screen chemical systems computationally.

Furthermore, many experimental chemists who may not consider themselves "computer people" can miss out on insights they might have gleaned from a computational study thanks to the high barrier of entry to interact with computational software. A small (but potentially disastrous) typographical error in an input file may go unnoticed even by the experienced computational chemist, and it is not always easy to discern where in the output file to look for information on what caused the error. Something as routine as collecting the energies from several output files and organizing them into a spreadsheet can become a time consuming and error-prone task for someone with little scripting experience.

The QChASM[3] suite of tools was designed to help experienced computational chemists more efficiently conduct their research, while also reducing the barrier to entry for inexperienced users and thereby catalyzing the adoption of computational tools amongst a broader audience. With more data generated, there is increased potential for insights via cheminformatics studies and for explaining or expanding chemical intuition. Additionally, computational screening can help identify promising leads to pursue experimentally, thus saving time, money, and resources. The field of chemistry as a whole can benefit from increased computational activity and easy to use tools for generating and analyzing this data.

1.2 Perl-based AARON and AaronTools

The Perl version of AaronTools was developed by Guan, Ingman, Rooks, and Wheeler starting around 2014.[4–7] It provides object-oriented functionality capable of measuring and manipulating molecular structures, as well as tools for data analysis and job submission. Many of the functionalities of AaronTools are now also available as command line scripts. These tools allow for the automation of many common tasks in computational chemistry. For example, AaronTools makes it trivial to perform a dihedral scan around a bond and write these new geometries to files for later computations. AaronTools can scan through a set of output files and compile data of interest, such as bond lengths, the RMSD compared to a reference structure, or thermochemical data. It is also possible to make functional group substitutions and map new ligands or organocatalyst onto a previously built structure. These tools simplify making adjustments to work already done without having to start over from square one; the user can build it once and let AaronTools do the rest.

AARON[7] (An Automated Reaction Optimizer for New catalysts) is a toolkit built on the Perl version of AaronTools for the specific purpose of predicting enatio- and regioselectivity of catalytic reactions. While originally designed for a specific class of bipyridine N,N′-dioxide catalyzed asymmetric alkylation reactions,[4,5] AARON can now handle both transition metal centered catalysts as well as fully organic catalysts. AARON interfaces with the computational package Gaussian 09[8] to automate optimizations for structures generated from template structures. AARON will submit jobs on the user's behalf and monitor their progress, making adjustments to molecular structures or computational parameters to fix errors detected along the way. After the initial set up, AARON can be run in the background, gathering data for screening ligands or exploring substrate scope for catalytic systems. In this way, the user is able to focus on chemical exploration and discovery rather than the monotonous process of submitting hundreds of jobs by hand.

1.3 AARON Applications

AARON has been applied to a number of catalytic reactions.[4–6,9] By providing automated predictions of catalytic activity and stereoselectivity, AARON opens the door for the computational screening of virtual libraries of catalysts for targeted reactions.

One of the primary uses of AARON is to screen potential ligands or organocatalysts for asymmetric reactions, with the goal of identifying potential new catalysts with superior selectivity and/or activity. For example, Doney et al. used an early version of AARON to make predictions for a library of 59 bipyridine-N,N'-dioxide derived catalysts for the asymmetric propargylation of benzaldehyde (see Figures 1.1A and B).[5] These 59 catalysts were built by appending one of ten functional groups on to six parent scaffolds (one catalyst was too sterically crowded to be viable), a task greatly facilitated using the mapping and substitution functionalities provided by AaronTools. For each of the C_2 symmetric catalysts, five different geometric arrangements of the ligands are possible (see Figure 1.1C), and the alkyl nucleophile can add to either face of the benzaldehyde — doubling the number of TS structures that must be considered for

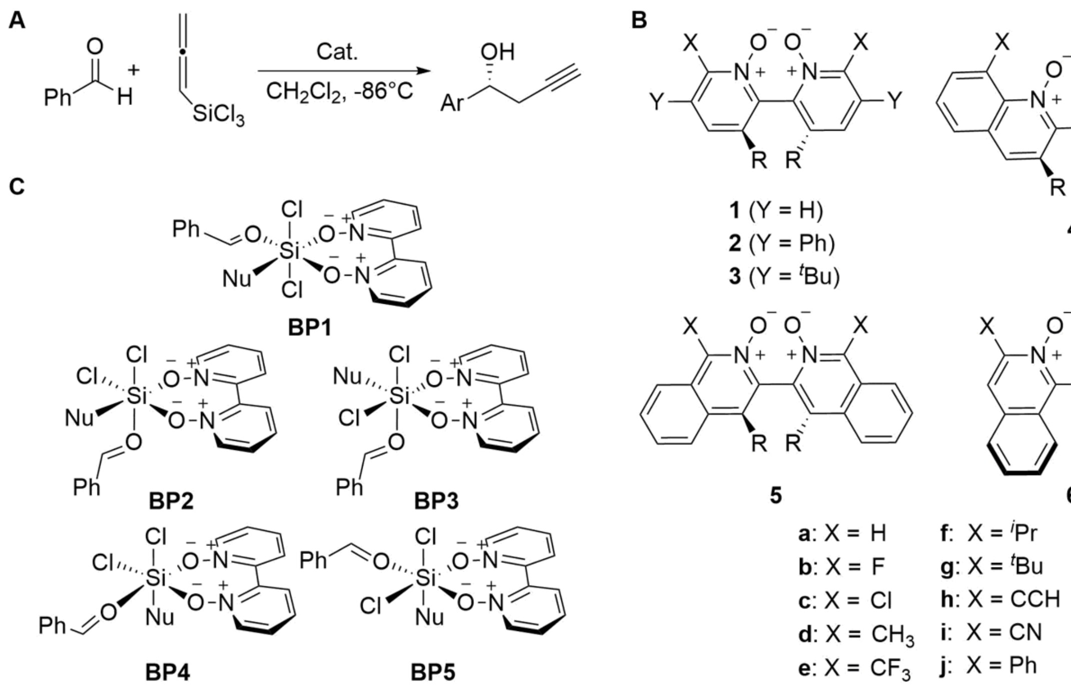


Figure 1.1: A) Asymmetric propargylation of benzaldehyde studied by Doney et al.[5] B) 60 potential catalysts built by considering ten substituents appended to six scaffolds. C) Five unique configurations of the C_2-symmetric bidentate catalyst, two chlorines, alkyl nucleophile, and aromatic aldehyde compatible with this reaction.

each catalyst in the library. AARON was used to automatically optimize these TS structures for each of the 59 catalysts.

The predicted enantioselectivities ranged from 45% *ee* (*S*) to 99% *ee* (*R*), with 12 catalysts predicted to perform in excess of 95% *ee* (*R*). When comparing these catalysts, there is quite a bit of variation in the energetic distribution of TS structures, illustrating the importance of considering all accessible TS structures, rather than just the lowest-lying *R*- and *S*-structures. For example, modest improvements in selectivity across the substituted varieties based on scaffolds 1 and 5 are generally due to changes in relative energies seen for the higher energy TS structures, while the energy gap between the two lowest-lying TS structures is essentially the same. Since AARON automatically calculates the Boltzmann weighted selectivities and presents a summary table with relative energies, enthalpies, and free energies (using both RRHO and quasi-RRHO schemes),[10] these subtle effects on the performance of catalysts can be predicted and analyzed easily.

As alluded to previously, the ease of gathering data for all 59 catalysts allowed more time for investigating what changes to the catalyst have the most impact on the reaction and why this may be the case. Although the effect the substituents had on selectivity for each catalyst scaffold was not always straightforward, it was noted that BP2(R) (i.e. the TS leading to the (R)-product built from configuration BP2) could be preferentially stabilized by tuning the electrostatic environment of the formyl C$-$H on the benzaldehyde. In light of this insight, a new catalyst scaffold was designed with two cyclohexane rings fused to scaffold 1. With more conformational flexibility due to the cyclohexane rings, which in some cases broke the C_2 symmetry (since each ring could adopt a different conformation than the other) and doubled the number of possible ligand arrangements, the number of TS structures needed to predict performance was drastically increased. Starting from 60 structures from a simplified version of the final catalyst, AARON was used to determine the most important TS structures for full analysis. This new catalyst was predicted to give greater than 99% *ee* and to outperform other scaffolds analyzed with similar substituents. This success illustrates how valuable it is to be able to generate computational data for large numbers of similar catalysts. Because a legion of effects can impact the performance (both positively and negatively) of any given catalyst, computational screening can inform new developments much more easily when enough data is present to see larger trends.

Subsequently, Malkov et al.[11] synthesized and tested close analogs of some of the catalysts screened by Doney et al.[5] While not all of these catalysts performed exactly as predicted, Malkov et al. were able to identify a catalyst exhibiting far greater activity and selectivity across a broad range of substrates for this reaction than previously available. Subsequent computational analyses of this new catalyst revealed that the origin of selectivity matched that of the original computationally designed version, representing a triumph for both AARON and computational catalyst design in general.

AARON has also been used by Guan and Wheeler[6] to screen chiral ligands for transition metal catalyzed reactions, including for the asymmetric hydrogenation of (E)-β-aryl-N-acetyl enamides (see Figure 1.2).[12] Without automation, computational investigation of this reaction for even just one ligand would have been tedious at best. Two hydride transfers from the metal to the substrate are necessary, at the α- and β-position, and the order in which these transfers occur is dependent on the nature of both the catalyst and the substrate. Because the ligand may influence the order of these hydride transfers, it is necessary to model both steps when investigating the performance of a new ligand. Additionally, the arrangement of the ligands around the metal center yields two possible transition state structures for either α- or β-hydride transfer. Finally, the rearrangement of the hydride-complex leads to two possible TS structures as well, although either arrangement for the initial hydride transfer will lead to the same two hydride-complex rearrangements. Thus, there are eight TS structures, four from the first step and four from the second step. However, these TS structures only take into account one enantiomeric product, increasing the final number of TS structures that must be considered to 16. Furthermore, each of the TS structures can spawn many conformations due to rotatable groups. Using AARON, 32 distinct pathways were found for L1, comprising 42 TS structures within 5 kcal/mol of the lowest TS structure. Because many pathways leading to the major and minor products are so close in energy, a Boltzmann weighting of these accessible pathways was necessary to predict the selectivity of L1. Similarly, for L2-L5, hundreds of TS structures were identified for each ligand, and the Boltzmann weighted selectivity predictions were in good agreement with experiment.

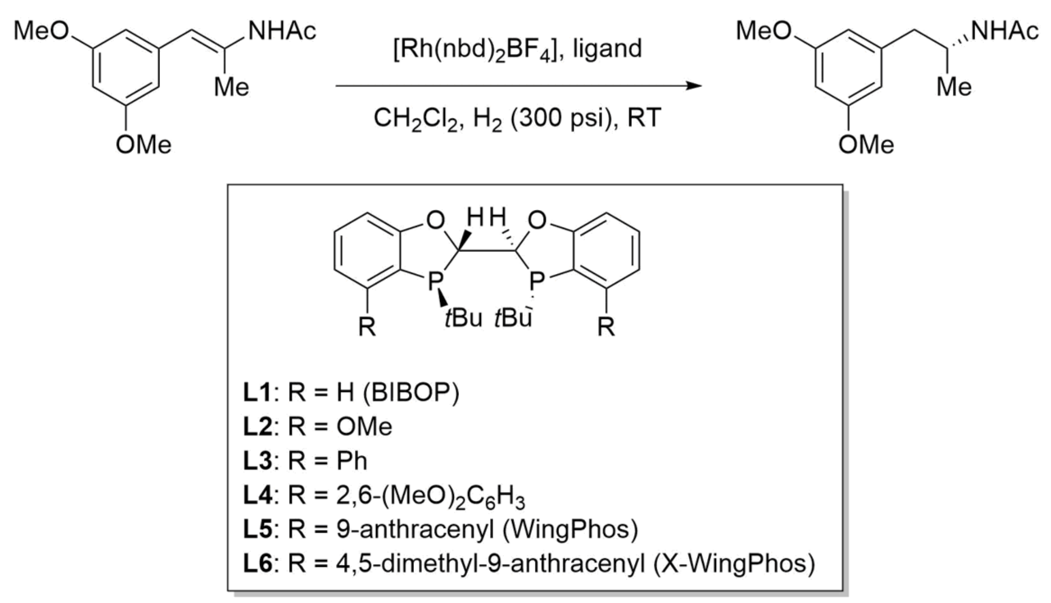


Figure 1.2: Asymmetric hydrogenation studied by Guan et al.[6]

Once again, AARON alleviated the tedium of finding the plethora of distinct TS structures needed to predict the selectivity of these six reactions. Because AARON can not only make substitutions to functional groups, but also map new ligands to the catalyst (in this case, using the phosphorus atoms as anchors to map a new ligand), it is trivial to screen a catalog of ligands using the templates already developed. This means that the extra time and effort to screen multiple ligands, compared to a single ligand, is essentially all computation time. Due to the efficiency of utilizing the AARON workflow, the authors could put their efforts towards discovery. By noting the unfavorable steric interactions between the aryl groups of the catalyst and substrate for the TS structures leading to the (*S*)-product for L5, it was postulated that extending this ligand could further exaggerate the energetic favorability of the (*R*)-pathway over the (*S*). Two methyl groups were added in the 4 and 5 positions on the 9-anthracenyl group of WingPhos (L5) to give X-WingPhos (L6). Theoretical results gathered by AARON indicate an increase in both activity and selectivity for this new ligand, compared to L5, with a Boltzmann weighted free-energy difference between the (*S*) and (*R*) pathways of 9.5 kcal/mol (compared to 6.2 kcal/mol for L5). Thus, the AARON workflow can be invaluable by automating the optimization of many catalysts and bringing us closer to true computationally-driven rational catalyst design.

Finally, AARON was recently used by Schaefer et al.[13] to benchmark popular DFT methods for an Ir-catalyzed CH-activation reaction (see Figure 1.3). Identifying the best DFT level of theory for a given problem generally depends on benchmarks of the performance of different DFT functionals, basis sets, and effective core potentials (ECPs) for similar reactions. While there have been countless benchmarks of DFT methods for organic systems, there are relatively few benchmarks of properties of transition

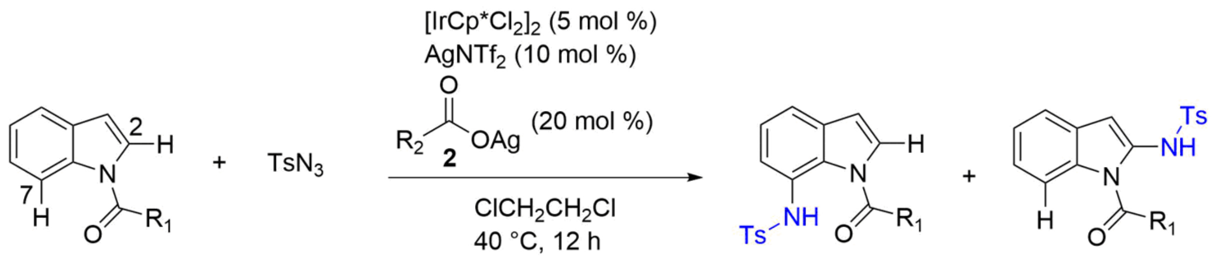


Figure 1.3: Regioselective Ir-catalyzed CH activation studied by Schaefer et al.[13]

metal containing compounds. There are even fewer, if any, benchmarks of the performance of DFT methods for problems in selective transition metal catalyzed reactions. This is, presumably, in large part due to the tedium associated with optimizing the hundreds or thousands of TS structures needed to assess predictions from DFT methods across a statistically meaningful set of reactions. Leveraging the power of AARON, such a project becomes feasible. By altering the computational parameters specified in a copy of the AARON input file, the structures can be computed at the new level of theory without any further user intervention, and results parsing methods greatly simplify the data collection process.

1.4 Overcoming the Shortcomings of AARON and AaronTools

Despite the successful application of AARON to many of important catalytic systems, a number of shortcomings of AARON have become clear. Many of these stem from AaronTools itself: it is neither user-friendly nor easily extensible.

To help address the first issue, command line scripts were made for many AaronTools methods, allowing users to take advantage of our tools without needing to learn the Perl programming language. However, attempting to expand the capabilities of our tools exposed deeper flaws in the data structures used to represent molecular systems and in the implementation of the methods. This fragility of the underlying code structure meant developing new tools was an incredibly bug-prone process and often required the developer to build new tools in a less than optimal way in order to integrate with the rest of the code base. This, in turn, made future developments even more difficult to accomplish.

As a result of these shortcomings, the entire AaronTools code base was re-written in Python, with a guiding principle of facilitating future extensions and applications. Using the Python language instead of Perl automatically increased the number of users able to take advantage of our tools in their own research, since Python is a much more popular language among scientists (particularly those not stuck in the past). We have also made the toolkit available on the Python Package Index for installation via `pip`, making it much easier for new users to get started. Additionally, a comprehensive testing suite was developed to help ensure future changes to the code base do not break existing implementations of the tools already developed. New test cases are easily added when unexpected behavior is discovered and addressed.

However, the rewrite was not a simple translation from Perl to Python. Ensuring the ability to integrate our tools with other libraries and software, as well as providing a good user experience, has been achieved by staying mindful and forward-thinking with regards to data structure design, thorough documentation, and elegant error handling.

First, the data structure for molecules was redesigned (Figure 1.4). In the Perl implementation, information about the atoms in the molecule — such as the atoms' coordinates, their elements, and their connections to other atoms — were stored in parallel arrays with each index in the array associated with one atom. This was inefficient and made it much too easy for a developer to introduce bugs. For example, using the Perl AaronTools the arrays for each attribute must all be changed when adding or removing atoms or molecular fragments (e.g. when adding substituents to molecular structures). Furthermore, the index associated with a particular atom must remain conserved across all arrays, meaning any change in the order of items in any of the arrays must be reflected in all the other attribute arrays. One can see how this inherently fragile data structure did little to facilitate the development of new tools and methods.

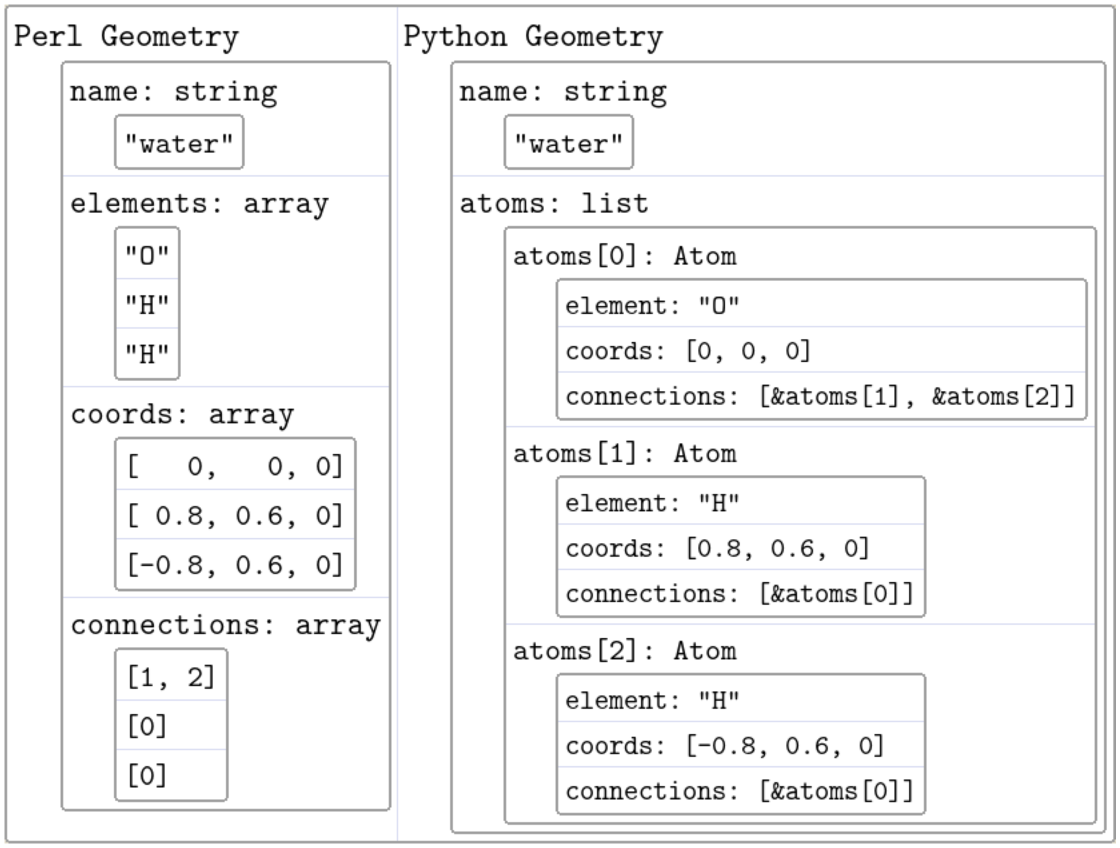


Figure 1.4: Data structure comparison of select attributes for a water molecule. The "&" symbol is used to denote an object reference.

Instead, a new `Atom` object was created to store all the necessary attributes together, with a single `Atom` array as the basis of the `Geometry` object that contains all the information that was once spread over several attributes. This change to the overall data structure used for representing molecules within AaronTools made modifications to the molecular structure much easier to code and eliminated many avenues for mistakes to occur. Additionally, the atom's index in the `Atom` array was no longer the only way to find a single `Atom`. Pointers to individual `Atom` objects could be stored in the connectivity attribute, thus eliminating the reliance on consistent atom ordering within the `Geometry` object's `Atom` array. This also makes partitioning of the molecule into sub-structural components (such as ligands or functional groups) faster and cleaner as graph-based algorithms (such as breadth-first search) can now be easily applied to the inter-linked `Atom` objects.

To ease modification or measurement, a `tag` and `name` attribute is also given to each `Atom` allowing one to search for particular atoms based on its membership to sub-structural components as well as its index in the original input molecule, respectively. Membership to a ligand, functional group, etc. is determined upon initialization of the Geometry object and used to tag the atoms. The atoms are assigned a `name` attribute based on the atom indices of the original structure, and the atoms each maintain their `name` regardless of any other changes made to the molecule. It is also now much easier for a programmer to interact with AaronTools, as one can easily search for particular atoms, such as finding all oxygen atoms that are part of an alcohol group or finding the atoms in a ligand that are coordinated with the metal center of a catalytic system. Furthermore, one can open the original structure in a graphical molecular viewer and know that the indices displayed there correspond to the `name` given to each atom in the structure, thus making it easy to modify different aspects of the structure without worrying about modifications disrupting the atom index used when calling various AaronTools methods.

Similar to the Perl version of AaronTools, the Perl implementation of AARON[7] was difficult to extend and debug. The implementation was designed with asymmetric catalysis in mind, and it was necessary to "trick" AARON to handle other types of systems. This, by definition, required the user to have detailed knowledge of the inner-workings of the program, which was made even more hopeless due to sparse documentation and a lack of proper error handling techniques. To make matters worse, logging and tracking procedures left much to be desired; an untold number of hours were wasted attempting to simply find or reproduce bugs, let alone actually fixing the issue.

Additionally, while AARON can produce conformers stemming from functional group rotations, there are other conformational changes, such as ring flips, that remain outside of its purview. Grimme, et al. have developed a conformer/rotamer generation program, Crest,[14] which uses a combination of vibrational mode following, molecular dynamics simulations, and "genetic" structure crossing to build an ensemble of structures. Vibrational mode following computes the harmonic vibrational frequencies of an optimized structure. Then, displacements along low frequency normal modes are used to find new minima along this now one-dimensional potential energy surface. This is done in a stepwise fashion, where partial geometry optimizations are performed as steps are taken along this displacement vector. By relaxing everything other than the displacement of interest, the procedure effectively includes non-linear effects. Next, these generated structures are subjected to molecular dynamics runs at three different temperatures; snapshots are taken and added to the ensemble. If at any point a lower lying conformer is found, the harmonic frequencies of this new structure are computed and the procedure is restarted. Finally, the "genetic crossing" portion of the procedure takes the lowest lying structure found during the mode following and molecular dynamics steps and modifies it by replacing portions of the structure's Z-matrix with that of other structures. Due to the design of AARON, seamlessly interfacing Crest with AARON proved problematic.

A new, Python-based tool (AaronJr: Automates All Reactions and Optimizations, Normally Just Right) was developed as a replacement for AARON that addressed these and other shortcomings, while also providing a much more general means of defining and executing quantum chemistry workflows. Thanks to the greater extensibility of the AaronJr code base, interfacing with Crest is now possible, and the user can take advantage of this robust conformer space sampling procedure.

The switch to Python itself was beneficial to the project. Python has built in exception handling and its object-based approach makes it easy for developers to design specific exceptions for the objects they implement. Additionally, these exceptions are part of a hierarchy, allowing one to handle particular types of errors while allowing other unexpected errors to cause the program to exit and print a traceback of where the error occurred. As an example, if one is expecting a numerical argument to a function, but a string is passed, one can catch a `TypeError` and attempt to convert the string to a numerical type. If this conversion is not possible, a `ValueError` is instead raised. One could also catch any error produced (the `Exception` error is the most general exception and will catch any error) and modify the error message to include additional information needed for debugging or to help the user identify any errors in their input files or user-made scripts.

In contrast, while Perl does have exception handling modules available through CPAN, they are not part of the standard library. These modules are essentially emulating a feature that is built into almost every other programming language. Being forced to use non-standard modules inherently makes installation more difficult for new users, but not using them makes for graceless code (as a best case) and could introduce security bugs (as a worst case). Indeed, to emulate error handling blocks in Perl requires testing `eval` blocks for failures, meaning it is possible to execute dangerous code if one is either malicious or simply not careful.

The net result of the Python re-write of AaronTools and the development of AaronJr is a much more robust suite of tools for structure manipulation and quantum chemistry automation.

1.5 Aim of This Dissertation

The aim of this dissertation is to describe the development of new Python tools that overcome the shortcomings of the Perl-based AARON and AaronTools, thereby providing powerful new tools for automated quantum chemistry workflows. I first introduce QChASM (Chapter 2), which is a new collection of free, open-source tools for quantum chemistry automation and structure manipulation, with a focus on the new Python implementation of AaronTools. I then describe AaronJr in Chapter 3, which is a complete rewrite of AARON to enable the automation of any quantum chemistry workflow. AaronJr utilizes the Python-based AaronTools, combined with the FireWorks workflow software,[15] to enable much more flexible automation of modern quantum chemistry chemical applications. I then describe example applications of AaronJr in Chapter 4, followed by concluding remarks in Chapter 5.

Chapter 2

QChASM: Quantum Chemistry Automation and Structure Manipulation[1]

[1]Ingman, V. M.; Schaefer, T. J.; Andreola, L. R.; Wheeler, S. E. *WIREs Comp. Mol. Sci.* 11, e1510 (2021). Copyright 2020 John Wiley & Sons. Reprinted here with permission of the publisher.

2.1 Abstract

As the tools of computational quantum chemistry have continued to mature, larger and more complex molecular systems have become amenable to computational study. However, studies of these complex systems often require the execution of enormous numbers of computations, which can be a tedious and error-prone process if done manually. We have developed a suite of free, open-source tools to facilitate the automation of quantum chemistry workflows. These tools are collected under the organization QChASM (Quantum Chemistry Automation and Structure Manipulation) and include functionality for building and manipulating complex molecular structures and performing routine tasks (AaronTools), a toolkit for automating TS optimizations and predictions of the outcomes of selective homogeneous catalytic reactions, and a plug-in for UCSF ChimeraX that provides a graphical interface for building complex molecular structures and representing output from quantum chemistry computations. These tools are described below, with a focus on the recent Python implementation of AaronTools.

2.2 Introduction

A growing challenge in modern applications of computational quantum chemistry is managing the sheer number of computations that must be performed when tackling complex molecular systems. This problem is particularly severe in the realm of homogeneous catalysis, in which the reliable prediction of catalyst activity and selectivity often requires the optimization of hundreds of transition state (TS) structures.[16–18] Studies of potential new catalysts are often limited to only a few examples of a given reaction because finding so many TS structures for just one system requires a lot of time, and expanding the search to more variations of a given reaction is not always feasible within time constraints.

To address this and other challenges, we are developing a set of free, open source tools for structure manipulation and automation, collected under the organization QChASM (Quantum Chemistry Automation and Structure Manipulation, see Figure 2.1). QChASM currently comprises three main packages: AaronTools, AARON, and SEQCROW. AaronTools is a collection of tools (available as Perl modules or as a Python package) for building, measuring, manipulating, and comparing molecular structures; constructing input and parsing output files; submitting and monitoring jobs in high-performance computing environments; and analyzing data. AARON is a computational toolkit, written using AaronTools, to automate the geometry optimization of the many TS structures and energy minima required to predict the activity and selectivity of homogeneous catalytic reactions. Finally, SEQCROW is a plug-in for UCSF ChimeraX[19] that adds tools to build and modify complex molecular structures, map new catalysts and ligands onto previously-computed structures, manage AaronTools libraries, construct input files for quantum chemistry packages, and run and manage jobs.

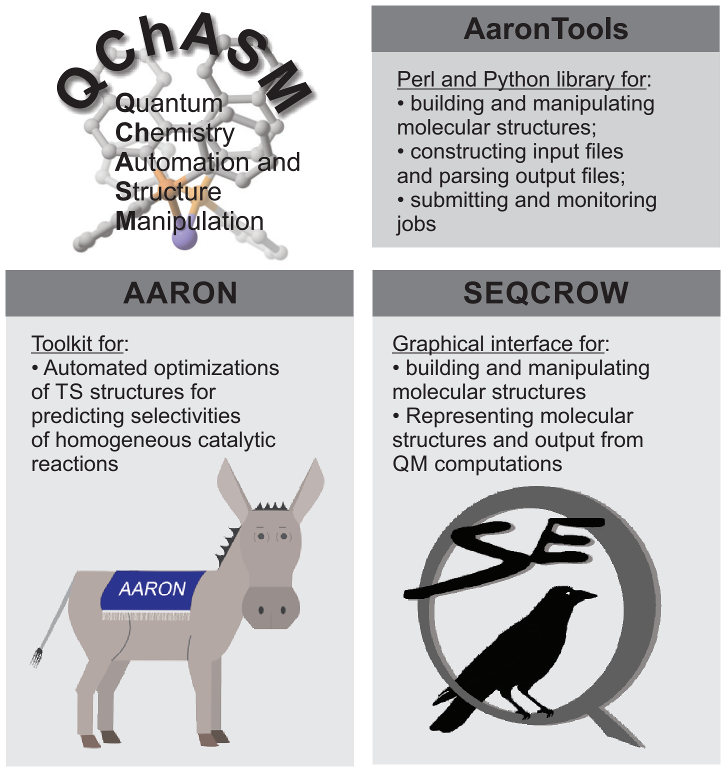

QChASM
Quantum
Chemistry
Automation and
Structure
Manipulation
AaronTools
Perl and Python library for:
• building and manipulating molecular structures;
• constructing input files and parsing output files;
• submitting and monitoring jobs
AARON
Toolkit for:
• Automated optimizations of TS structures for predicting selectivities of homogeneous catalytic reactions
AARON
SEQCROW
Graphical interface for:
• building and manipulating molecular structures
• Representing molecular structures and output from QM computations
SE

Figure 2.1: Main components of QChASM

These tools, described below, together facilitate the automation of quantum chemistry applications to complex molecular systems. Because AARON has been described recently[7] and a separate publication on SEQCROW is in preparation, we focus primarily on AaronTools, particularly the recent Python implementation. Our goal is to show that with these tools, users will be able to tackle more challenging problems by spending their time thinking about molecules instead of frittering away the hours building molecular structures, generating input files, submitting jobs, and parsing output files.

2.3 AaronTools

In many quantum chemistry applications, a model system is used as a common framework to construct coordinates of many new molecules. For instance, you might consider a series of substituted analogs of a given molecule or need to optimize structures for different substrates or catalysts along the same reaction pathway. Typically, coordinates for such structures are generated using any of a number of available graphical molecular builders; however, this requires the user to make changes file by file, which is time consuming and can lead to mistakes, inconsistencies in atom numbering, etc. To avoid these mistakes, we have made central to AaronTools utilities to streamline the generation of complex molecular structures from either a simple script or the command line. These tools enable the generation of structures quickly and methodically, and are concise and flexible enough to be incorporated in workflows in many types of quantum chemistry applications, not limited to catalysis. There are two primary ways to use AaronTools, through either a series of stand-alone command line scripts or the underlying Perl or Python objects.

2.3.1 Command Line Scripts

The AaronTools command line scripts provide basic functionality for structure measurement, manipulation, and comparison, as well as tools for output parsing and analysis. Most of these scripts provide a single specific function, such as modifying a distance, angle, or dihedral angle, adding or changing a substituent, or extracting thermochemical information from an output file. Internally, all structure manipulations are performed in Cartesian coordinates. Each command provides basic usage documentation and command line conveniences (e.g. file name globbing, pipes, and output redirection) are supported. In many cases, a simple shell script can be used to string together individual command line scripts to build up an automated workflow. A few examples are provided here (additional examples and usage help can be found on the AaronTools GitHub Wiki; see Additional Reading, below). While the Perl and Python AaronTools provide command line scripts with similar functionality, below we demonstrate the Python-based scripts.

A simple example using an AaronTools command line script is measuring a particular dihedral angle from a set of structures using the `dihedral` command with the `--measure` argument. For instance, the following will determine the dihedral angle defined by atoms 1, 2, 3, and 4 for each XYZ file in the current directory and print the value of the corresponding angle:

```
dihedral.py --measure 1 2 3 4 *.xyz
```

This and other command line scripts can also take output files from Gaussian,[20] Psi4,[21] or ORCA,[22] from which they will automatically extract the last set of molecular coordinates (e.g. from a geometry optimization).

A slightly more complex example uses `dihedral` to generate structures for a torsional scan. The `--set` argument accepts the four atoms defining the dihedral angle followed by the desired angle. Alternatively, the `--change` argument can be used to alter the selected torsional angle by the specified amount. We note that currently, `dihedral` can not change a dihedral angle involving four atoms within a ring and does not automatically resolve steric clashes or close contants that occur as the result of a change in dihedral angle. Combined with a simple `for` loop, the structures for each point along a torsional scan can be created and saved using the `--output` argument. For example, the following bash loop uses the coordinates from `original.xyz` to generate 35 new structures (each saved as a new XYZ file) by rotating the dihedral angle defined by atoms 1, 2, 3 and 4, in 10° increments:

```
for d in $(seq 0 10 350); do
  dihedral.py original.xyz --set 1 2 3 4 "$d" --output "rotated_$d.xyz"
done
```

Structures or components of structures can be translated using the `translate` command or rotated around any specified bond or axis using the `rotate` command. This includes simple rotations around the Cartesian axes as well as axes defined between centroids of groups of atoms or perpendicular to sets of atoms. For instance, Figure 2.2a shows the rotation of one component (atoms 20-34) of the stacked dimer in `dimer.xyz` by 60° around the axis perpendicular to the plane defined by atoms 20-30:

```
  rotate.py dimer.xyz --targets 20-34 --perpendicular 20-30 --angle 60
```

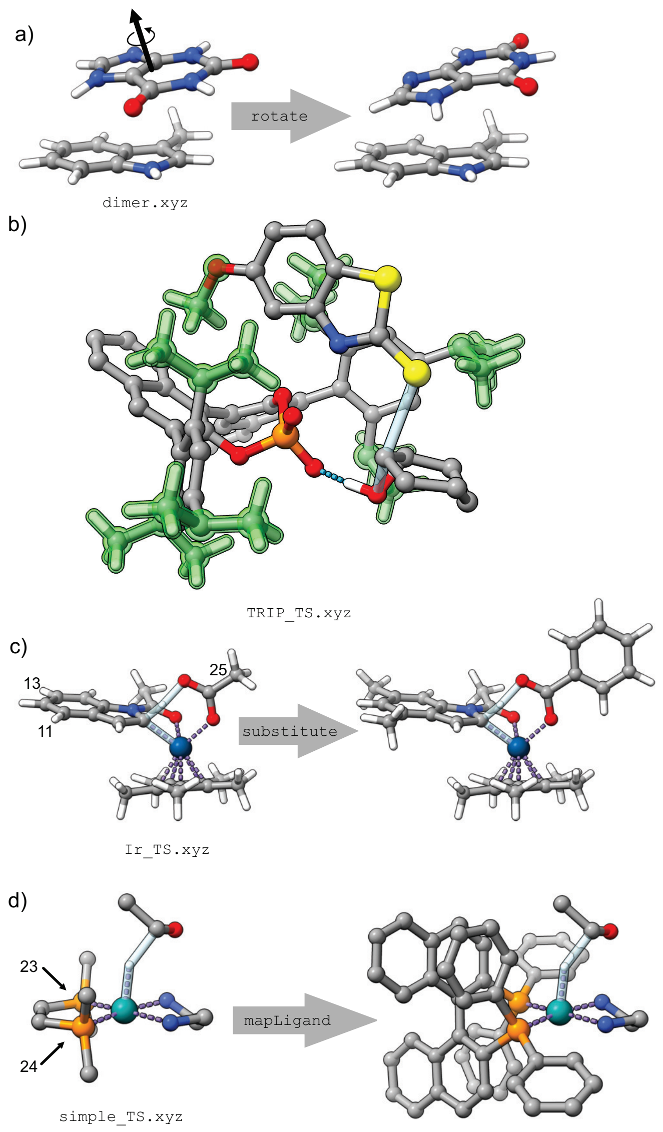


Figure 2.2: a) Dimer of 3-methylindole and 3,9-dihydro-purine-2,6-dione from Ref. [23] before and after rotation of 3,9-dihydro-purine-2,6-dione by 60° around the vector normal to the ring plane; b) TS for a stereoselective epoxide ring opening catalyzed by the chiral phosphoric acid TRIP, from Ref. [24], highlighting the six iPr and one OMe that will be rotated by `makeConf` (selected hydrogen atoms removed for clarity); c) TS for an Ir-catalyzed CH activation reaction before and after substitution of atoms 11 and 13 with Me groups and replacement of the Me group at atom 25 with a Ph ring using `substitute`; d) Use of `mapLigand` to replace a model ZDMP ligand with (S)-BINAP in a TS structure for a Noyori asymmetric hydrogenation of acetaldehyde.

A common task encountered in quantum chemistry applications is the generation of rotamers of substituents (e.g. iPr, OMe, etc). In systems with multiple rotatable groups, the number of potential conformations can be astronomical. The `makeConf` command automatically generates new conformers of a given structure by rotating all or selected substituents. AaronTools contains a library of common substituents along with information about the number of unique rotamers and the angles separating these rotamers. For example, in the AaronTools library a Ph substituent can exist as two rotamers, separated by 90°. Users can easily augment this library with their own, custom library, or override the number of rotamers considered for built-in substituents. `makeConf` uses this information to build rotameric structures. For instance, provided the TS structure shown in Figure 2.2b, `makeConf` will, by default, generate 2916 rotamers by considering three rotamers each of the six iPr groups and two rotamers of the OMe and OH groups. In this case, we would not need to consider rotations of the OH group, because we are confident that the OH—O hydrogen bond will be maintained in all low-lying TS structures. As such, we could use `makeConf` to automatically generate the 1458 rotamers by considering rotations of only the iPr and OMe groups, requesting that the generated conformers be written as separate XYZ files in a directory called `Conformers`:

```
makeConf.py TRIP_TS.xyz --substituent OMe iPr --output Conformers
```

Alternatively, specific iPr substituents could be rotated as identified by atom numbers. The structures generated by `makeConf.py` could then be optimized at a chosen level of theory to identify accessible TS structures, for example. Often, conformers generated by `makeConf` will exhibit steric clashes. Options allow for severe steric clashes to be automatically resolved or for such structures to simply not be printed.

The `substitute` command provides a flexible means of replacing any monovalent atom or substituent with a substituent from the built-in or user-defined libraries. To demonstrate this, Figure 2.2c shows the replacement of hydrogens 11 and 13 with methyl groups and the methyl group at atom 25 with a Ph ring in a TS structure for an Ir-catalyzed CH activation reaction via:

```
substitute.py Ir_TS.xyz -s 11,13=Me 25=Ph --minimize
```

With the `--minimize` argument, `substitute` will rotate the added substituents to minimize the Lennard-Jones energy, ensuring that new substituents are added in relatively low-energy orientations.

Additionally, `rmsdAlign` can be used to align two structures, minimizing the root mean squared deviation (RMSD) between the atom positions. This includes an efficient canonicalization scheme (triggered by the option `--sort`) that will automatically sort atoms within each structure to account for changes in numbering or degenerate rotations of substituents. For example, the following will calculate the RMSD between all XYZ files in a directory and a reference structure (`ref.xyz`), printing the results as comma-separated values:

```
rmsdAlign.py align/*.xyz --reference ref.xyz --sort -csv --output results.csv
```

AaronTools contains a library of nearly 100 achiral and chiral ligands as well as common organocatalysts. As with the substituent library, users can easily add their own custom ligand library. The `mapLigand` command provides an efficient means of replacing specified ligand(s) or catalyst(s) with other ones from either the built-in or individual user library. One could use `mapLigand` to take previously computed structures along a reaction pathway and replace the catalyst with another across each structure. The resulting geometries can then be optimized at the chosen level of theory to quickly explore reaction mechanisms for variations of a given reaction. As an example, Figure 2.2d shows the use of `mapLigand` to replace a simple bidentate ligand (ZDMP, bound to the metal through atoms 23 and 24) with the chiral ligand (S)-BINAP in a TS structure for a Noyori asymmetric hydrogenation of acetaldehyde:[25]

```
mapLigand.py simple_TS.xyz --ligand 23,24=S-BINAP
```

AaronTools can also extract thermochemical information from Gaussian (G09 or G16), ORCA, or Psi4 output files while also computing Grimme's quasi-RRHO[10] free energies. Options enable the user to automatically combine higher-level single point energies with thermochemical corrections computed at lower levels of theory, recalculate thermochemical corrections at a different temperature, and recursively search directories for output files. For example, the following extracts the energy from a set of single point jobs and combines these with enthalpy, RRHO free energy, and quasi-RRHO free energy corrections based on data extracted from the corresponding frequency jobs. The resulting data is printed to a `csv` file, along with warnings for any cases in which the single point and frequency jobs appear to use different geometries.

```
grabThermo.py --recursive *freq.log -sp *sp.log -csv --output thermo.csv
```

Finally, the commands `makeInput.py` and `submitJob.py` allow for the creation of Gaussian, Psi4, and Orca input files from the command line and submission to common queueing systems, respectively. For the creation of input files, level of theory and options are specified on the command line. Command line options also allow for the specification of bond constraints, DFT integration grids, etc., allowing the user to build sophisticated input files for these three computational chemistry packages all from the command line. Another command, `fetchMolecule.py` can generate initial molecular coordinates from SMILES or IUPAC names. As with all AaronTools command line scripts, output from one script can be read as STDIN for another, which makes it possible to fetch coordinates and place these directly into an input file. For instance, the following will build an Orca input file for the geometry optimization the structure of 1,3,5-trinitrotoluene at the B3LYP/def2TZVP level of theory:

```
fetchMolecule.py --iupac "1,3,5-trinitrotoluene" |
makeInput.py --method b3lyp --basis def2tzvp --optimize --output-format orca
```

The resulting input file could subsequently be submitted using `submitJob.py`.

Command line scripts are also available to measure or change inter-atomic distances and bond angles, change the element of a given atom (e.g. convert a C to an N, including automatic adjustment of hydrogen atoms to satisfy valency), change the chirality of a chiral center, calculate Sterimol parameters,[26] and follow imaginary vibrational modes, among others. These scripts provide command line access to common tasks encountered in quantum chemistry applications, allowing users to streamline and automate large portions of their workflows.

2.3.2 Scripting with Perl and Python

While the combination of AaronTools command line scripts enable the construction of automated workflows, more complex tools can be developed by directly using the objects implemented in the AaronTools modules (e.g. AARON and SEQCROW; see below). Some features of the Perl-based AaronTools have been described previously;[7] here, we provide a complementary description of the new Python AaronTools package.

The `Atom` class

Unlike the Perl version of AaronTools,[7] which uses indexed arrays within a `Geometry` object to keep track of atom information, the Python package introduces a new `Atom` class. Each `Atom` object contains all information associated with an individual atom, including the element and its coordinates as well as connections to other atoms, associated Lennard-Jones parameters, and tags useful for filtering and sorting. This data structure allows for more graceful and powerful interactions with molecular structures. For example, a molecular graph can be easily obtained by following the `Atom.connections` attributes throughout the structure. Additionally, the `Atom.tags` attribute allows AaronTools processes, as well as the programmer, to quickly identify structural components. Finally, extensibility is facilitated since sub-classes can be built on top of the `Atom` class without needing to redefine large swaths of code to use them. For instance, the `Atom` class was recently extended for use in parsing output files and writing input files for ONIOM computations,[27] which requires additional information to properly delineate regions of molecules that will be treated at different levels of theory.

The `Geometry` class

The most general way of interacting with a molecular structure is by using a `Geometry` object. Atoms and their coordinates are read from a number of common file formats, including XYZ files, Gaussian input and output files, and output files from Psi4[21] and ORCA[22] or from the AaronTools libraries. Atom connectivity is automatically determined. Utilities are available to transform or analyze the molecular structure or to prepare it for a workflow. For instance, one can perform substitutions, locate substituents by either name (e.g. Me, Ph, and iPr) or the atoms to which they are connected, determine the shortest path between specified atoms, identify molecular fragments, etc. After the desired changes are made, the `Geometry` object can be written to either an XYZ file or a Gaussian, ORCA, or Psi4 input file (see Managing Jobs, below).

The following illustrates the use of a `Geometry` object to add substituents by reading the coordinates of benzene and replacing atom 7 with a methyl group and atoms 8, 9, and 12 with NO_2 groups to generate TNT:

```
geom = Geometry("benzene.xyz")
geom.substitute("Me", "7")
o_and_p_positions = geom.find("8,9,12")
for position in o_and_p_positions:
  geom.substitute("NO2", position)
geom.write(outfile="tnt.xyz")
```

The above example used `find` to locate atoms by atom number, which obviously requires the programmer to know the atom numbering in the file `benzene.xyz`. However, `find` offers far more flexibility, allowing the programmer to locate atoms based on combinations of attributes including the element, the number and identity of bonded neighbors, and the distance from a given atom either spatially or connectively. As an example, the following will perform the same transformation of benzene to TNT, but without any prior knowledge of the atom ordering by first finding any hydrogen atom (h1), then locating the hydrogens that are three and five bonds away (the *o*- and *p*-hydrogens). The former is then substituted with Me and the latter with nitro groups.

```
geom = Geometry("benzene.xyz")
h1 = geom.find("H")[0]
o_and_p_positions = geom.find([BondsFrom(h1, 3), BondsFrom(h1, 5)], "H")
geom.substitute("Me", h1)
for position in o_and_p_positions:
  geom.substitute("NO2", position)
geom.write(outfile="tnt.xyz")
```

Additionally, substituents can be detected and identified by name. For example, the following reads the geometry of TNT (`tnt.xyz`), detects all attached substituents, and then removes the methyl group using `remove_fragment` to leave 1,3,5-trinitrobenzene:

```
geom = Geometry("tnt.xyz")
geom.detect_substituents()
for sub in geom.substituents:
  if sub.name == "Me":
    methyl_carbon = geom.find("Me", "C")
    geom.remove_fragment(methyl_carbon, sub.end)
geom.write("trinitrobenzene.xyz")
```

In this example, `remove_fragment` requires the first and last atom of a fragment, so we use `sub.end` to request the last atom within the substituent.

Another feature of `Geometry` is the ability to add fused rings to structures. For example, one can convert benzene to naphthalene by first locating any hydrogen, then finding a second hydrogen three bonds away, and finally using `ring_substitute` to replace these two hydrogens with a new benzene ring:

```
geom = Geometry("benzene.xyz")
h1 = geom.find("H")[0]
h2 = geom.find(BondsFrom(h1, 3), "H")[0]
geom.ring_substitute([h1, h2], "benzene")
geom.write(outfile="naphthalene.xyz")
```

More complicated transformations can be achieved using combinations of `find`, `substitute`, and `ring_substitute`, all without knowledge of atom ordering.

Finally, the Geometry class provides the function map_ligand that allows the programmer to replace any specified ligand(s) with another ligand with the same total denticity. For instance, one can replace a bidentate ligand with another bidentate ligand, or a combination of a bidentate and monodentate ligand with a tridentate ligand. As an example, the following reads the coordinates of a TS structure for the enantioselective Markovnikov hydroboration of an alkene from TS.xyz,[28] locates the two phosphorus atoms of the original ligand using find, and then replaces this ligand with (S,S)-Me-DuPhos using map_ligand (see Figure 2.3):

```
oldcat = Geometry("TS.xyz")
Ps = oldcat.find("P")
oldcat.map_ligand("SS-Me-DuPhos", Ps)
```

In more complex cases (e.g. systems with additional P atoms or multiple ligands), one could locate the correct ligand atoms through the use of more refined attributes, as discussed above.

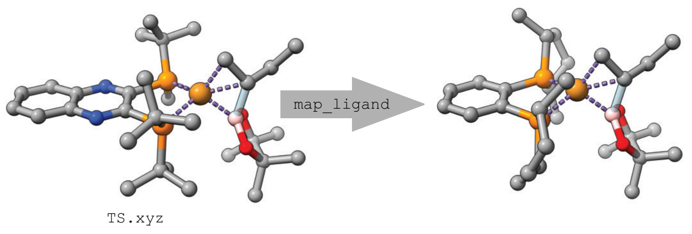


Figure 2.3: Use of the function map_ligand to replace the ligand in a TS structure for an enantioselective Markovnikov hydroboration (from Ref. [29]) of a terminal alkene with (S,S)-Me-DuPhos

The Geometry class provides many other functions, including computing Lennard-Jones energies, aligning structures for evaluating RMSDs, measuring and changing bond distances, angles, and dihedral angles, comparing connectivities between structures, canonicalizing atom ordering, identifying the shortest path between atoms, and performing substitutions, rotations, translations, etc. Information on these is readily available on the command line through pydoc.

Managing Jobs

Finally, AaronTools provides functions to create input files for Gaussian, ORCA, and Psi4 and to submit and monitor jobs using popular queuing software (Slurm, Torque/MOAB, LSF, and SGE). Combined with the structure manipulation capabilities described above, these functions enable the development of complex toolkits and automated workflows. For this purpose, a Theory class provides an intuitive means of setting the level of theory as well as job type (optimization, vibrational frequencies, etc), charge and multiplicity, and other job-specific options (number of processors, integration grids for DFT computations, etc). A Theory object can contain a BasisSet object to facilitate specification of more complex basis sets (e.g. ECPs, auxiliary basis sets, etc.). Once constructed, a Theory object can be passed to a Geometry object in order to automatically construct the corresponding input file for the chosen software package. The resulting input file can then be submitted using a SubmitProcess object, with an option to await all running jobs in the current directory to finish before proceeding.

As an example, the following will read the geometry from dimer.xyz (e.g. Figure 2.2a) and optimize this structure using Gaussian at the ωB97X-D/def2-TZVP level of theory. Once this job is complete, the script reads the optimized geometry and then uses ORCA to run a DLPNO-CCSD(T)/cc-pVQZ single point.

```
#read initial coordinates
geom = Geometry("dimer.xyz")

#method for optimization
dft_theory = Theory(method="wB97X-D", basis="def2-TZVP", processors=14,
                    memory=24, job_type=OptimizationJob())

#write optimization input file and submit, waiting for job to finish
geom.write(outfile="opt.com", theory=dft_theory)
opt_job = SubmitProcess(fname="opt.com", walltime=24, processors=14, memory=28)
opt_job.submit(wait=True)

#read the optimized structure
opt_geom = Geometry("opt.log")
#basis set and method for DLPNO-CCSD(T) single point
dlpno_basis = BasisSet([Basis("cc-pVQZ"), Basis("cc-pVQZ", aux_type="C")])
dlpno_theory = Theory(method="DLPNO-CCSD(T)", basis=dlpno_basis, processors=4,
                      memory=28, job_type=SinglePointJob())

#Write DLPNO-CCSD(T) input file and submit
opt_geom.write(outfile="dlpno.inp", theory=dlpno_theory, simple=["TightSCF"])
dlpno_job = SubmitProcess(fname="dlpno.inp", walltime=24, processors=4, memory=32)
dlpno_job.submit(wait=True)
```

2.3.3 Remote Job Submission with FireWorks

We are currently extending AaronTools to work with the FireWorks workflow software,[15] which will allow more rapid deployment of AaronTools-based tools across different HPC environments. FireWorks is a free, open-source toolkit for managing HPC workflows over arbitrary computing resources, including those that have queueing systems. As such, the integration of AaronTools and AARON with FireWorks will also enable remote job submission across multiple computer clusters and job monitoring and management via a web interface. This added functionality will be part of the new Python-based AARON (AARON2), which is currently in development.

2.4 AARON

One example of a complex tool developed using the Perl-based AaronTools is AARON, which has been described in detail before.[7] Briefly, AARON interfaces with Gaussian09 or Gaussian16 to automate the optimization of TS structures derived from templates through substitutions or changes of ligand/organocatalyst. First, the user provides the TS configurations to be considered in the form of a TS library (i.e. template structures leading to the *R*- or *S*-product of an enantioselective reaction), or they are requested from the built-in template library. Next, any requested changes to the template are handled, such as mapping a new ligand or making substitutions. Additionally, conformers of rotatable groups are sampled automatically. After structure generation, an initial refinement is performed, freezing the unchanged parts of the structure and relaxing the new additions, generally at an inexpensive level of theory (e.g. semi-empirical). Next, geometry optimizations at the density functional theory (DFT) level are done, first with any forming or breaking bonds frozen, and then with no constraints. After a full geometry optimization, harmonic frequencies and thermochemistry can be computed. If structures exhibit the wrong number of imaginary vibrational modes, AARON will attempt to automatically resolve the problem through additional constrained and unconstrained optimizations. If desired, a final single-point computation can be done at a higher level of theory, which is then combined with the thermochemical corrections derived from the original level of theory. While all these computations are running, AARON is continuously monitoring their status — handling common errors, resubmitting failed jobs, and resolving unexpected structural changes (for example, by constraining a problematic bond) — all without user intervention. Finally, after all computations are finished, AARON parses the output files, analyzes the thermochemistry to generate a summary of the results and predict product ratios based on a Boltzmann weighting of accessible pathways, and generates the associated Supporting Information.

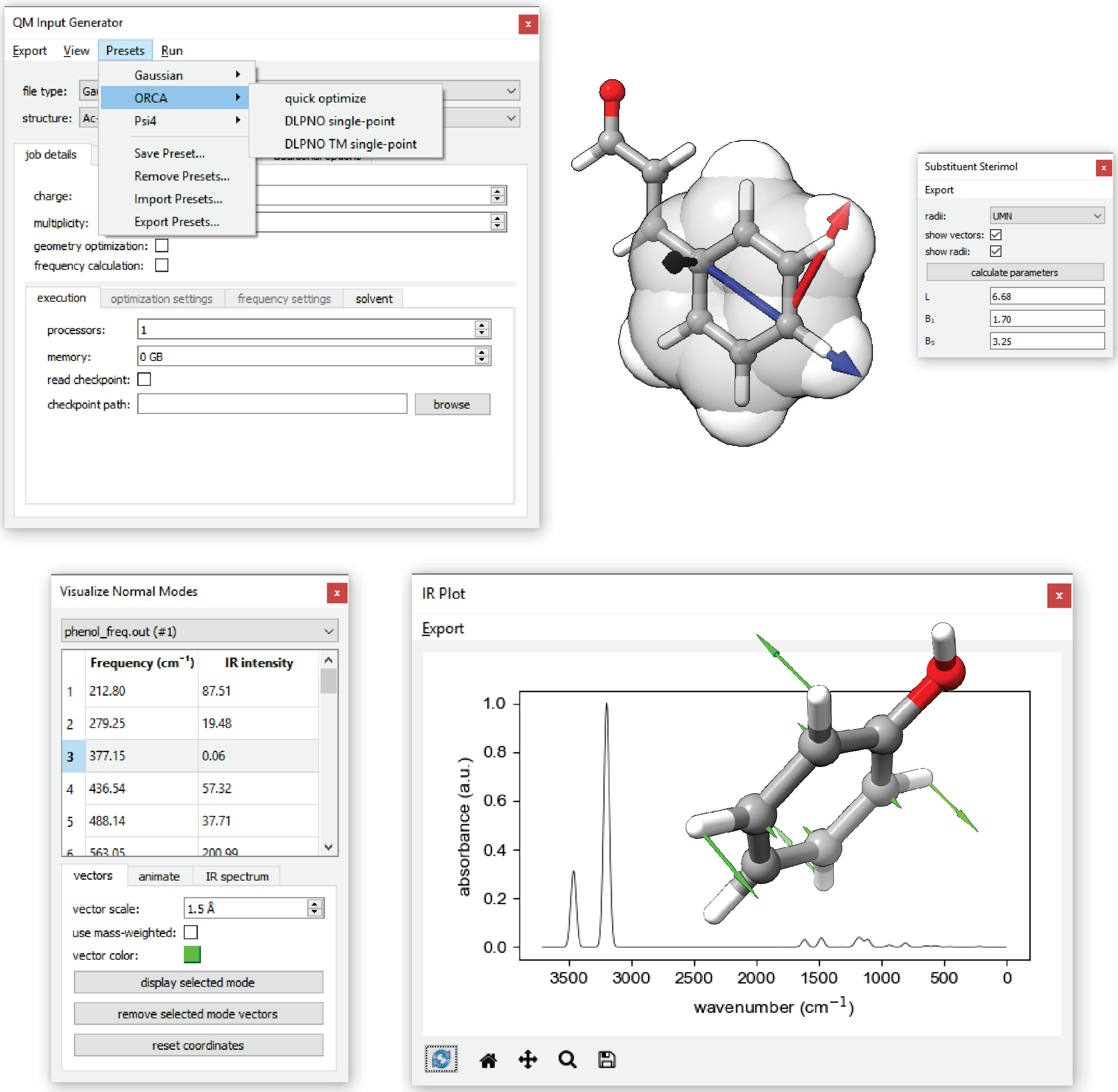
The primary use of AARON is to accelerate predictions of the activity and selectivity of homogeneous catalysts by automating optimization of TS structures. This automation allows computational predictions of selectivities for more examples of a given reaction than could reasonably be done without it. This opens the door for a number of exciting possibilities, including virtually screening catalysts with DFT methods and benchmarking DFT methods on statistically significant numbers of catalysts. AARON has been applied to a number of catalytic reactions,[5,6,9] several of which were recently summarized[7] and is currently being used to benchmark DFT methods for an Ir-catalyzed CH activation reaction (See Figure 2.2c). Overall, the automation provided by AARON brings us closer to true computationally-driven rational catalyst design. The current version of AARON[7] is built using the Perl-based AaronTools. A more powerful Python version of AARON (AARON2) is currently in development that will exploit the added flexibility of the Python implementation of AaronTools.

2.4.1 Automated Conformer Searches using XTB/Crest

Vital to successful applications of AARON is the generation of all reasonable configurations and conformers in the construction of a TS template library. We have had recent success doing this by combining Grimme's XTB/Crest automated conformational search[14] with AARON. For instance, starting from one conformation of each configuration, Crest can be used to generate a collection of conformations for an initial TS template library (constraining any forming or breaking bonds), which can then be further refined using AARON. This avoids the often cumbersome task of enumerating conformations when constructing a TS template library, particularly in cases of ring-flips and other more complex conformational changes that are not handled automatically by AARON.

2.5 SEQCROW

While the main utility of AaronTools is the ability to build and manipulate complex molecular structures from the command line or from within a Perl or Python script, use of a graphical interface is often advantageous. While graphical molecular builders abound, few are well suited for the rapid construction of the structures encountered in studies of homogeneous catalysis. We are developing a plug-in for UCSF ChimeraX[19] called SEQCROW that provides the power of AaronTools within a graphical user interface. For example, SEQCROW can be used to generate and explore potential TS structures for transition metal catalyzed reactions using different chiral ligands or perform substitutions and add fused rings to structures. SEQCROW also contains graphics presets for the generation of publication-quality images of small to medium sized molecular structures and extends the capabilities of ChimeraX to be able to generate input files for popular quantum chemistry packages and run these jobs, plot simulated spectra, calculate thermal energy corrections, etc. (see Figure 2.4). SEQCROW is still in development, but is available through the ChimeraX 'Toolshed.' It will be described more fully in a forthcoming publication.



file type: Ga
structure: Ac
charge:
multiplicity:
geometry optimization: ☐
frequency calculation: ☐
processors: 1
memory: 0 GB
read checkpoint: ☐
checkpoint path:

radii: UMN
show vectors: ☑
show radii: ☑
L 6.68
B_1 1.70
B_5 3.25

phenol_freq.out (#1)

	Frequency (cm^{-1})	IR intensity
1	212.80	87.51
2	279.25	19.48
3	377.15	0.06
4	436.54	57.32
5	488.14	37.71
6	563.05	200.99

vector scale: 1.5 Å
use mass-weighted: ☐
vector color: 🟩

Figure 2.4: Screenshots of SEQCROW showing the QM Input Generator allowing users to save 'Presets' of common input file types for Gaussian, Orca, and Psi4, the calculation of Sterimol parameters, and the Normal Mode Visualization and IR Spectra Plotting

2.6 Conclusions

As quantum chemists have gravitated to larger and more complex molecular systems, new challenges have emerged regarding the execution of large numbers of computations, parsing the corresponding output files, and verifying and managing the resulting data. In the context of catalysis, reliable predictions of catalyst performance often require consideration of hundreds of potential TS structures. The time required to generate input files, run the required jobs, and check the resulting output files often prevents the assessment of large numbers of potential catalysts, precluding effective computational catalyst design. We have developed a suite of open-source tools (QChASM) to meet some of these challenges. The workhorse of QChASM is AaronTools, which consists of Perl and Python tools for building, measuring, and manipulating molecular structures. Using AaronTools, one can quickly and intuitively perform substitutions or exchange chiral and achiral ligands, submit and monitor jobs, and construct input and parse output files from popular electronic structure packages. AARON is a more specialized toolkit for automated predictions of selectivities for catalytic reactions. Finally, SEQCROW provides a graphical interface to exploit the power of AaronTools, build input files for QM computations, and generate molecular images.

While these tools were initially developed for applications in computational catalysis, they should be of general use by the quantum chemistry community, AaronTools being particularly broad in its applicability. For example, we have relied extensively on AaronTools in our studies of stacking interactions and supramolecular complexes.[30] Furthermore, as free, open-source tools, contributions from others are welcomed in order to build even more powerful and useful tools for the broader computational chemistry community.

2.7 Acknowledgements

We thank B. J. Rooks and Y. Guan for their efforts in the development of AARON and the Perl-based AaronTools as well as M. R. Haas, A. C. Doney, and A. N. Bootsma for feedback and testing. Molecular graphics were generated in UCSF ChimeraX, developed by the Resource for Biocomputing, Visualization, and Informatics at the University of California, San Francisco. We have no conflicts of interest to declare.

2.8 Funding Information

This work was funded by National Science Foundation Grants CHE-1266022 and CHE-1665407.

Chapter 3

AaronJr: A Python Toolkit for Automating Quantum Chemistry Workflows[1]

[1]Ingman, V. M.; Wheeler, S. E. To be submitted to the Journal of Chemical Theory and Computation.

3.1 Abstract

Modern quantum chemistry applications often require the execution of hundreds or even thousands of individual computations, sometimes using multiple quantum chemistry packages. Manually creating the corresponding input files, checking the resulting output files, and managing the generated data can quickly become cumbersome. To combat this growing problem, we have developed AaronJr (Automates Any Reaction or Optimization, Normally Just Right), which provides a simple yet powerful command line interface to define and execute automated quantum chemistry workflows which use the packages Gaussian, ORCA, Psi4, Crest, and/or xtb. Sensible defaults for common tasks allow for the definition and deployment of workflows from within a simple, user-readable INI formatted input file, while the flexibility of the input and configuration files facilitates the definition of complex new workflows or modification of existing workflows. Key features include the automated resolution of errors, including structural validation which checks for unexpected changes in atom connectivity during (seemingly successful) geometry optimizations, the storage and display of key results, and the generation of resource usage summaries.

3.2 Introduction

With continued hardware and algorithmic advances, many quantum chemists have turned their attention to increasingly complex chemical applications.[31–33] A side-effect of this natural evolution of the field is the rapidly growing number of computations that must be performed for a given project. For example, mechanistic investigations of a single transition metal catalyzed reaction can require the geometry optimization of dozens of transition state (TS) structures and intermediates.[34,35] For each of these stationary points, one often needs to consider many potential conformations or configurations of the substrate(s) and ligand(s) around the metal, resulting in the need to carry out hundreds of optimizations of TS structures and local minima.[16,17] Furthermore, many applications also involve single point computations at higher levels of theory, the evaluation of solvent effects, bonding analyses, etc. The net result is that looking at a single reaction mechanism can require hundreds or even thousands of individual computations, each generating data in the form of molecular structures, energies, and other values that must be parsed from text files with content formatting varying among different software packages. Each of these computations can result in errors, requiring user intervention to first identify and then resolve errors through either modification of the input file (e.g. adding new keywords) or modifying the initial molecular structure (e.g. adjusting bond lengths to ensure a saddle point optimization corresponds to the correct TS structure). Studying multiple examples of a given reaction, which is required for any iterative computational design process,[9,18,36] further multiplies the number of computations that must be performed. Manually building initial molecular structures, creating the corresponding input files, parsing the resulting output files, resolving errors, and managing the data generated by such expansive projects quickly becomes cumbersome. An unfortunate consequence is that many potentially impactful applications of quantum chemistry to complex chemical systems are simply avoided because they would be too onerous to complete.

A number of tools have been developed to try to combat this growing problem by automating some or all aspects of quantum chemistry computations. These are part of a growing ecosystem of tools that do not themselves perform quantum chemistry computations, but instead facilitate the efficient use of existing quantum chemistry packages or analyses of results. Some are targeted to specialized tasks or particular types of quantum chemical applications. For instance, we previously developed AARON (An Automated Reaction Optimizer for New Catalysts),[7] which is a toolkit for automating the TS optimizations required for predicting the outcome of stereoselective catalytic reactions.[5,6,9] Tools have been developed for building molecular and supramolecular structures,[37–40] automatically computing reaction energy profiles for organic and organometallic reactions,[41] and even generating supporting information,[42] among other tasks.

Other tools allow for the development of more general automated workflows. QMFlows[43] allows users to build a workflow, including input file generation, intelligent parallel job execution, and output file post-processing, via a concise but flexible Python API. Several computational chemistry software packages are supported; the use of generalized keywords for defining computational tasks (e.g. geometry optimizations, vibrational frequency computations) as well as for computational parameters (e.g. DFT functional, basis set) allows users to switch between software packages easily, while software-specific keywords are available for advanced use cases. Similarly, the open-source quantum chemistry package Psi4[44] allows for the definition of workflows within the input file or within a Python script. However, while powerful, both QMFlows and Psi4 have some limitations in terms of workflow development and deployment. For instance, they require users to know Python in order to develop even simple workflows, limiting their use. Moreover, even though failed jobs can be identified and restarted, and job dependencies defined, neither Psi4 nor QMFlows provides automated resolution of failed computations. That is, the user must explicitly code the appropriate response if something as simple as an SCF convergence failure occurs during a given step in a workflow. Additionally, more subtle errors such as unexpected changes in atom connectivity during geometry optimizations or the optimization to an undesired TS structure go undetected. This can result in the automated execution of computations that will ultimately be discarded, representing an unfortunate waste of computational resources.

We have developed a suite of tools for quantum chemistry automation and structure manipulation (QChASM)[3] designed for both experienced computational chemists and newcomers to the field. Below, we describe a new QChASM tool called AaronJr (Automates Any Reaction or Optimization, Normally Just Right). AaronJr provides a general, easy to use command line interface for creating and executing quantum chemistry workflows using any combination of the quantum chemistry packages Gaussian,[20] ORCA,[22,45] Psi4,[21] Crest,[14] and xtb.[46] By automating job generation and submission, as well as parsing and storing results and resolving errors, AaronJr allows computational chemists to focus on chemistry instead of managing the vast number of computations that must be run for modern quantum chemistry applications.

3.3 Overview of AaronJr

AaronJr is written using the Python package AaronTools[3] for structure manipulation, input file creation, and output parsing. Simple configuration files allow the user to easily define workflows (default workflows are available for use or extension), set computational parameters, and connect to high-performance computing systems. Basic workflow management, queue submission, resource tracking, and computational output storage is done using the FireWorks Python library,[15] which is a free, open-source toolkit for managing workflows over arbitrary computing resources, including those with queueing systems. This library provides tools to submit jobs to high performance computing (HPC) systems, define dynamic workflows, and record the results of computations in a central database. Using these tools, the dynamic workflow implementation in AaronJr tracks job progress, records any errors detected in the computational output, and automatically resolves common errors encountered during quantum chemistry computations. The overall workflow of AaronJr is depicted in Figure 3.1. More details on the implementation are provided below (See AaronJr Implementation).

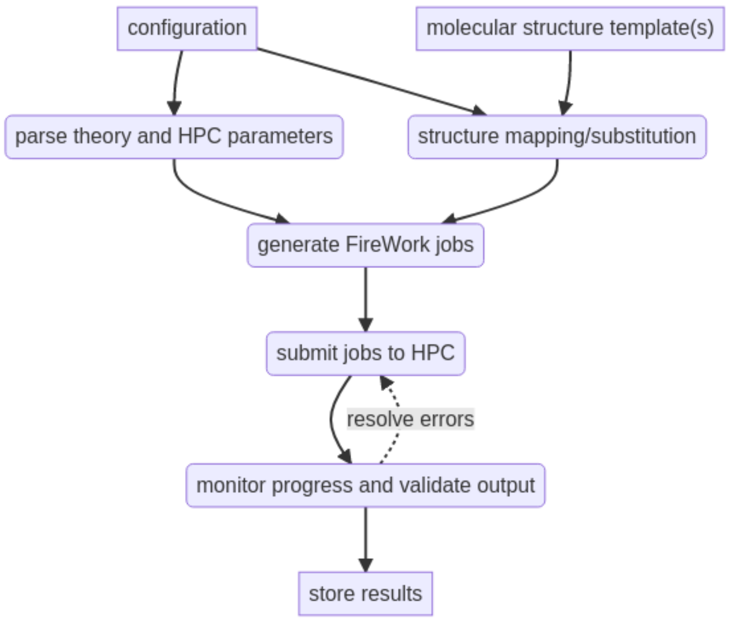

configuration
molecular structure template(s)
parse theory and HPC parameters
structure mapping/substitution
generate FireWork jobs
submit jobs to HPC
resolve errors
monitor progress and validate output
store results

Figure 3.1: AaronJr workflow overview

Most generally, a quantum chemistry workflow requires the specification and/or construction of a set of initial molecular structures followed by a series of computations at potentially different levels of theory using one or more quantum chemistry packages. Input and configuration files for AaronJr are INI-format text files (see examples below), allowing even inexperienced users to develop robust workflows without any programming experience. Comments can provide further notes and clarity within these files. The AaronJr input file allows the user to define a workflow from scratch or by using/modifying one of the provided default workflows and to supply one or more molecular structure template files. These template structures can be used as-is or can be modified using the AaronTools package.[3] As such, users can exploit the plethora of ligands and functional groups from the AaronTools libraries for quick and easy structural generation/modification; tools are also available to facilitate adding ligands and functional groups to a personal library. Many file formats for these template structures are supported, including XYZ and SDF files, as well as input or output files from Gaussian,[20] ORCA,[22,45] and Psi4.[21] Molecular structures can also be specified using SMILES. These template structures will be either used as-is or modified to build the desired structures.

While complete workflows for a particular series of computations can be defined within an AaronJr input file, AaronJr allows the creation of system-wide (`AaronTools/config.ini`) or user-defined (`$AARONLIB/config.ini`) configuration files that define default workflows or other parameters used to generate jobs. These default workflows can be imported and modified within an AaronJr input INI file, allowing for very compact and simple AaronJr input files for routine applications and the quick definition of new, modified workflows built on existing defaults.

Once a workflow has been defined, AaronJr will execute all required computations as separate jobs on the designated local or remote HPC resource, while also respecting job dependencies thanks to the dynamic workflow implementation. Adapters are included for popular queueing systems and can be modified to work with particular implementations. By running jobs separately when possible, AaronJr provides efficient coarse-grained parallelization to maximally exploit modern HPC resources. The user can set a default executable (e.g. Gaussian, ORCA, etc.) and other job parameters (number of nodes, cores, memory, walltime, etc.) for all steps or for individual steps within the workflow.

As is vital for any automation tool, AaronJr continually monitors all running jobs to identify and resolve errors. Any errant job is killed, fixed, and resubmitted, ensuring that wasted CPU cycles are minimized. This includes errors arising from unexpected changes in atom connectivity during geometry optimizations, as well as more mundane errors such as SCF convergence errors or simply running out of requested walltime.

3.3.1 AaronJr Input File

The AaronJr input file has three required sections, `[Job]`, `[Theory]`, and `[Geometry]`, which specify the steps in the workflow, the levels of theory for these steps, and the structural template files, respectively. For example, an AaronJr input file requesting B3LYP/6-31G(d) geometry optimizations[47] and vibrational frequency computations for all structures in the directory `Example1` (and recursively in any subdirectories) is shown in Figure 3.2 (left).

The `[Job]` section is used to define a workflow as a series of job types each preceded by a step number (currently supported job types are depicted in Figure 3.3). These step numbers not only designate the order that the steps in the workflow will be executed but also serve as reference labels for step-specific parameters in this or other sections of the input file.

Alternatively, the `[Job]` section can import default workflows and then use these as-is or make modifications. For example, Figure 3.2 (right) shows essentially the same input file but now imports the built-in workflow 'Minimum.' This default workflow, taken from the `config.ini` installed with AaronJr, is

```
[Job]
1 type = optimize
2 type = frequencies

[Theory]
method = b3lyp
basis = 6-31G(d)

[Geometry]
structure = Example1
```

```
[Job]
Include = Minimum

[Theory]
method = b3lyp
basis = 6-31G(d)

[Geometry]
structure = Example1
```

Figure 3.2: Equivalent AaronJr input files requesting the geometry optimization and vibrational frequencies at the B3LYP/6-31G(d) level of theory for all geometries in the directory `Example1`

```
single-point
gradient
optimize
optimize.changed
optimize.constrained
optimize.ts
frequencies
conformers
conformers.constrained
```

Figure 3.3: Currently available job types in AaronJr

shown in Figure 3.4. In step 1, any new atoms or groups added to the template structure(s) (e.g. a new sub-structural substitution or mapping) are relaxed with the conserved portion of the geometry frozen; if no changes were made to the template(s), this step is automatically skipped. This is followed by a full geometry optimization to an energy minimum and vibrational frequencies in steps 2 and 3, respectively.

```
[Job.Minimum]
1 type=optimize.changes
2 type=optimize
3 type=frequencies

[Job.TS]
1 type=optimize.changes
2 type=optimize.constrained
3 type=optimize.ts
4 type=frequencies

[Job.CrestMinimum]
1 type=optimize.changes
1 exec_type = xtb
2 type=conformers
2 exec_type=crest
3 type=optimize
4 type=frequencies

[Job.CrestTS]
1 type=optimize.changes
1 exec_type = xtb
2 type=conformers.constrained
2 exec_type=crest
3 type=optimize.constrained
4 type=optimize.ts
5 type=frequencies
```

Figure 3.4: Default workflows included with AaronJr. Users can define new workflows either from scratch or by importing and modifying these default workflows.

AaronJr will repeatedly attempt to resolve any errors encountered during each step by modifying parameters present in the input file or molecular structure, or even returning to previous steps in the workflow if necessary. For instance, if there is an SCF convergence failure, AaronJr will add the appropriate keywords for the corresponding quantum chemistry package to successfully converge the SCF iterations. Moreover, if a change in the atom connectivity is detected during any geometry optimization, AaronJr will attempt to fix this through small structural modifications (i.e. by shortening or lengthening problematic bonds). This avoids the unnecessary continuation of doomed geometry optimizations. This is particularly important in the case of TS optimizations, for which one often needs to try multiple initial geometries before successfully locating the correct first order saddle point. By default, each subsequent step in a given workflow requires the error-free completion of the previous step. Alternatively, the user can request that two or more steps be run concurrently given successful completion of a previous step. For instance, one could run an NBO analysis,[48] solvent single point energy, and higher-order single point energy all concurrently once the geometry optimization step is completed.

The `[Theory]` section allows the user to specify levels of theory for all or selected steps in the workflow defined in the `[Job]` section. In Figure 3.2, the B3LYP/6-31G(d) level of theory is used for all steps. The level of theory and other parameters (e.g. solvent, temperature, etc.) can be set for individual steps by preceding keywords with the corresponding step number from the `[Job]` section (see examples below). Additional job types, beyond those listed in Figure 3.3, can be designated by specifying program-specific input parameters within the `[Theory]` section. For example, setting "route = pop NBO" in the `[Theory]` section of the AaronJr input file will include "pop=NBO" in the corresponding Gaussian input files, triggering an NBO analysis[48] even though this is not one of the built-in job types.

The user can also modify the default workflows from within the AaronJr input file. For example, one could augment the default Minimum workflow by adding a single point energy at a different level of theory (see Figure 3.5). In Figure 3.5, a DLPNO-CCSD(T)/cc-pVQZ single point energy[49–53] is added to the imported workflow as a fourth step. Because the method and basis set keywords are preceded by '4', these modifications to the level of theory will only be applied to step 4 within the workflow; all other steps will be done with B3LYP/6-31G(d). Furthermore, this example requests that ORCA be used for step 4 (via "exec-type = orca" in the `[Job]` section), while the default executable defined in the user's default configuration is used for all other steps. The user can also insert intermediate steps before or into a default workflow by specifying appropriately numbered steps (e.g. adding step 2.5 will result in a new step being inserted between steps 2 and 3).

```
[Job]
Include = Minimum
4 type = single-point
4 exec-type=orca

[Theory]
method = b3lyp
basis = 6-31G(d)
4 method = dlpno-ccsd(t)
4 basis = cc-pVQZ
4         aux basis cc-pVQZ/C

[Geometry]
structure = Example1
```

Figure 3.5: Extending a default workflow, with alternate theory parameters for the added step.

Mixed basis sets or basis set/ECP combinations are simply specified within the [Theory] section. For instance, the user can request the 6-31G(d) basis set be used in all non-transition metal ('!tm') atoms and LANL2DZ basis and ECP[54] be used on any transition metal ('tm') atoms via

```
basis = !tm 6-31G(d)
        tm LANL2DZ
ecp   = tm LANL2DZ
```

within the [Theory] section.

Finally, the [Geometry] section identifies one or more structure templates for which the defined workflow will be executed. This could be the name of a directory, as in the above examples, in which case AaronJr executes the workflow for all geometry-containing files in that directory (and subdirectories therein). Alternatively, 'structure' could be set to one or more SMILES strings or names of files containing molecular coordinates.

Additionally, the [Geometry] section allows for far more general and much finer control over the specification of template structures. Use of the &call keyword allows the user to take advantage of any of the AaronTools methods defined for Geometry objects.[3] Figure 3.6 shows a simple example in which the carbon atoms of ethane (defined using the SMILES string 'CC') are changed to silicon, and the Si−Si bond length is adjusted accordingly (setting dist=None in the call to change_distance() will default to the distance defined by the corresponding atomic radii).

```
[Geometry]
structure = CC
&call:
  structure.change_element("1", "Si")
  structure.change_element("2", "Si")
  structure.change_distance("1", "2", dist=None)
```

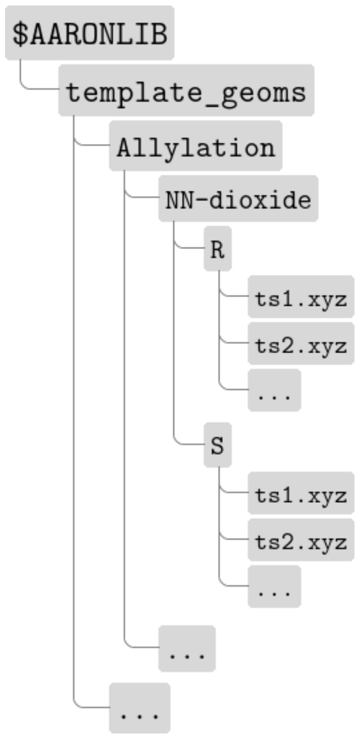
Figure 3.6: Example [Geometry] section converting the structure of ethane to disilane

As a more complex example, Figure 3.7 shows the [Geometry] specification for generating the structures corresponding to a dihedral scan of ethane. This snippet showcases the use of the &for keyword, iterating over angles from 0–360° (inclusive) at 10° increments (specified using the Python built-in range() method) and using the change_dihedral() method implemented in AaronTools to set the H-C-C-H dihedral appropriately. AaronJr would then run the desired workflow on each of these structures.

```
[Geometry]
structure.0 = CC
&for i in range(0, 370, 10):
  structure.i = structure.0.copy()
  structure.i.change_dihedral("3", "1", "2", "6", i)
```

Figure 3.7: Dihedral scan of ethane

The successful use of templates for initial structures for complex molecular systems requires templates that are as close as possible to the targeted geometries. In the case of organic and organometallic reactions, this often means using slightly different templates for a given reaction depending on the modifications that will be performed. This is particularly important in cases in which intermediates or TS structures change qualitatively depending on the nature of the substrate or catalyst. AaronJr makes this simple by allowing the user to specify "reaction" and "template" keywords within a special `[Reaction]` section. While initially provided for backwards-compatibility for AARON,[7] using this template structure specification style provides two convenient features. First, structure files with names starting with "int" or "INT" will automatically be assigned the `Job.Minimum` workflow and those starting with "ts" or "TS" will be assigned the `Job.TS` workflow (unless the "include" option is used to override this behavior in the AaronJr input file). Second, selectivity can be automatically detected based on subdirectory names (*cis*, *trans*, *R*, *S*, *E*, or *Z*) and used to provide selectivity summaries in the results display. An example directory structure, with the accompanying AaronJr input file, is given in Figure 3.8. This example will automatically optimize all structures in the `$AARONLIB/template_geoms/Allylation/NN-dioxide` directory at the B3LYP/6-31G(d) level of theory to TS structures. Note that this input file could be further simplified by setting default method and basis set keywords in the user's `config.ini`.



```
[Reaction]
reaction = Allylation
template = NN-dioxide

# no [Job] section necessary,
# workflow assignment detected
# automatically

[Theory]
method = B3LYP
basis = 6-31G(d)
```

Figure 3.8: Using the `[Reaction]` section for structure specification

Structure Modification

Above, structural templates were used unaltered (or with simple and minimal changes, as in the `&call` and `&for` usage examples). This is useful if the user wants to execute a given workflow using structures generated by or extracted from some external source or from previous computations. However, the AaronJr input file has a number of optional sections that can be used to modify the specified template structures, making it easy to automatically screen a variety of changes to many template files at once. For example, the user could request all combinations of a set of template structures modified by adding any number of different substituents or changing a ligand on a metal center. AaronJr can also automatically build and optimize structures for all unique coordination compounds built from adding specified ligands to a given metal center using the templates provided by Simas et al.[55] More fine-tuned changes to template structures (adjusting bond lengths, angles, dihedral angles, etc.) can be made by directly calling AaronTools functions. In this way, AaronJr facilitates the execution of a given workflow for large libraries of molecular structures. For instance, one can use AaronJr to examine the impact of multiple substituents on some molecular system or screen potential ligands for a given reaction, automatically computing all stationary points along a catalytic reaction pathway for any number of chiral or achiral ligands. Moreover, all of this can be done from a single, simple input file.

Figure 3.9 gives an example using combinations of substrate substitutions and ligand mappings applied to eight intermediate and TS template structures (located in the `chlorocarbamoylation/Pd-PAPh` sub-directory of the user's template library). One of the template structures is shown on the right, with the atoms involved in defining modifications labeled. Running AaronJr with this input will automatically optimize 448 structures to either minima or first order saddle points (using the default level of theory defined by the user in `config.ini`), resolving all errors along the way and providing a summary of results when complete.

Conformer Generation with AaronJr

AaronJr can use Crest[14] to generate conformers using the `[Job.CrestMinimum]` and `[Job.CrestTS]` default workflows for energy minima and TS structures, respectively. After an initial optimization of any changes using XTB, these workflows then perform the `conformers` (for minima) or `conformers.constrained` (for TS structures) job types using Crest (see Figure 3.4). The conformers generated by Crest are automatically loaded during job validation and used to build child workflows that will be appended to the parent workflow. Conformer screening parameters (such as the energy window or minimum RMSD difference between conformers) can be set for the Crest run or as job validation settings using the AaronJr configuration file. For example, the user may wish to have Crest, which is quite fast, generate structures for conformers within a 10 kcal/mol window. However, performing DFT optimizations of all the generated structures can be prohibitively resource intensive. Options in the AaronJr configuration file can be used to request DFT optimization for only the structures within, say, a 5 kcal/mol window and widen this request later if needed. AaronJr will automatically screen out unwanted or duplicated conformer structures from the child workflows or pick up additional conformer structures during

```
[Geometry]
reaction = chlorocarbamoylation
template = Pd-PAPh
# 8 template structures

[Substitution]
&combinations:
  32,42,52 = F,Cl,H, Me,Et,iPr,tBu
  103      = F,Cl,Br,I
# 28 different substitutions

[Mapping]
Pd-PPh3: 62=PPh3
# 2 ligands (original and PPh3)

# 448 structures generated!
```

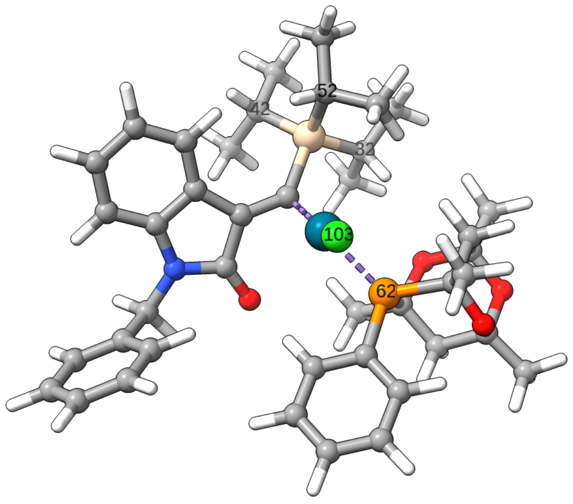


Figure 3.9: Simple AaronJr input file that would result in the geometry optimization and frequency computations of 448 unique intermediates and TS structures generated by combinations of substitutions and ligand mappings of eight previously computed template structures (one of which is shown, along with key atom numbers)

a later run if these screening parameters change. In the future, machine learning methods will be used to make AaronJr more intelligent when choosing which conformers to submit for more refined optimizations, allowing the more efficient sampling of even larger regions of conformer space without hopelessly increasing the computational workload.

The Use of Multiple Input Files

More complex applications sometimes require the use of multiple INI files that are referenced within the `[Config]` section of a single project input file (see Figure 3.10). This can be especially helpful when `[Substitution]` or `[Mapping]` sections need to be defined differently for certain structures. For example, ligand mapping on the catalyst would not be done for the dissociated reactant/product structures, while substitutions might be defined differently for reactants versus products or TS structures, such as in the case of bimolecular bond forming reactions, in order to account for changes in atom numbering.

project.ini

```
[Config]
rxn = rxn.ini
reactant = reactants.ini
product = products.ini

[Theory]
method = B3LYP
basis = !tm 6-31G(d)
        tm LANL2DZ
ecp = tm LANL2DZ
```

rxn.ini

```
[Reaction]
reaction = hydrogenation
template = S-BINAP

[Mapping]
R-BINAP: 20,21=R-BINAP
```

reactants.ini

```
[Geometry]
structure = reactants_dir

[Job]
include = Minimum
```

products.ini

```
[Geometry]
structure = products_dir

[Job]
include = Minimum
```

Figure 3.10: Using a project configuration file (project.ini) to tie together multiple configuration files (rxn.ini, reactants.ini, and products.ini)

Running AaronJr

Armed with an input file, AaronJr has four main operation modes: run, results, resources, and plot. Each of these commands have specific usage help available using `AaronJr <command> --help` at the command line.

The `run` command

The `run` command is used to run the workflow defined in the AaronJr input file:

```
  AaronJr run my_project.ini
```

If the jobs associated with the defined workflow are not present in the FireWorks database (e.g. when running a workflow for the first time), new jobs are created and assigned a FireWork ID and Workflow ID. Alternatively, if the AaronJr run has been exited and restarted, this command will resume the run, after ensuring any updates are reflected in the database; this includes downloading computational output files from the HPC resource, validating the current output files and resolution of errors, and starting new workflows from child structures (such as conformers) or newly discovered templates (e.g. if a directory was provided to the 'structure' keyword, additional templates added to that directory will be automatically picked up when AaronJr is restarted). The `update` command is also available to simply ensure the status of all jobs is up to date without continuing with the run. After the jobs are collected, a monitoring routine is called for each job, which allows for real-time tracking of the job's progress. This routine can print statuses and look for errors both during and after execution of the jobs. A `--sleep` parameter can be supplied to define how long to wait between each call to the monitoring routine (default is five minutes).

The `results` command

The `results` command is used to print thermochemistry or other pertinent values parsed from the computational output files or to save these values in a comma-separated value file for further analysis in external programs. Using command line flags, the user may specify which thermochemical values to display, whether these should be displayed in absolute or relative terms, which units to use, which structures will be included in the display, how thermal corrections will be applied, and more. When performing modifications (ligand mapping or substitutions), a separate table for each change is printed, and the lowest energy structure within that group is used to determine the relative thermochemistry (unless explicitly defined, see below). For applicable systems, detected stereo- or regio-selectivity will be used to calculate predicted selectivities, and Boltzmann-weighting can be applied to relative thermochemical values and displayed; once again, the results are grouped by modification prior to performing these calculations. Users can even hook in AaronTools command line scripts to print a variety of measurements about the structures using the `--script` flag. For example,

```
AaronJr results my_project.ini --script 'bond.py -m 1 2'
```

will print the bond length between atoms 1 and 2 for each optimized structure.

Since a workflow will generally consist of multiple steps, the `results` routine takes this into account. By default, results for the latest (i.e. with the highest step-number) completed step is loaded and additional results from lower steps are pulled in as needed (this behavior can be changed using options in the `[Results]` section, see below). For example, a higher-level single point computation, run as the last step in a workflow, will provide an electronic energy but not the vibrational frequencies needed for calculating thermal corrections; in this case, vibrational frequencies are automatically pulled in from a previous step. This allows the application of thermal corrections to higher-level single point energies. Additionally, the command line interface allows thermal corrections to be re-calculated at different temperatures than what was specified during the computational run without running any additional jobs, as well as to use different methods to determine these corrections (RRHO, quasi-RRHO, or quasi-harmonic free energy corrections are available, with user-specified frequency cutoffs if applicable).[10,56]

Finally, a `[Results]` section in the configuration file can be used to handle more complicated analysis needs. Finer-tune control over results loading is made available in the `[Results]` section of the input file: the 'load' keyword is used to specify the order in which steps should be loaded (e.g. 'load = 4 3'), and the 'calc' keyword can be used to specify exactly how certain quantities should be calculated using attributes and methods available for each step's `CompOutput` AaronTools object (e.g.: 'calc: free_energy = 4.energy + 3.calc_G_corr()') The 'relative' keyword allows the user to define which structure the thermochemistry output will be set relative to. The implementation is intelligent enough to group modifications together when displaying relative thermochemistry, meaning one merely specifies which template structure gives rise to the structure for which thermochemical values should be set relative to. For certain projects, it may be necessary to perform simple arithmetic on the results for certain structures to get meaningful relative energies, for example when comparing TS structure energies to those of the separated reactants or products. In such cases, the user can define simple functions of the templates using a keyword starting with an ampersand (&). As a simple example, say we wanted to look at the O-H deprotonation energy for a variety of alcohols, $ROH + B \rightleftharpoons RO^{-} + HB^{+}$. We have our templates, `MeOH.xyz`, `OH.xyz`, `MeO.xyz`, and `HOH.xyz`, to which we will apply various substitutions to generate different alcohol structures. We could define `relative = MeOH + OH` and `&products = MeO + HOH` in our `[Results]` section. Now, an additional row in the display tables (one for each substitution), labeled 'products', will be appended and it's value will correspond to the O—H deprotonation energy for the alcohol.

The `resources` command

The `resource` command allows for printing of CPU usage and related metrics or saving of these to a comma-separated value file. This allows users to better understand their resource usage and make modifications to resource requests if necessary. This can be especially important for HPC resources in high-demand, as requesting more resources than necessary can lead to jobs queued for long periods of time waiting unnecessarily for extra resources that are not truly needed. Especially for users new to using shared computational resources, it can greatly help the user get a feel for resource usage for their projects.

The `plot` command

Finally, the `plot` command can be used to print potential energy diagrams for multi-step reactions; a different plot will be produced for each modification (i.e. each substitution or ligand mapping combination). The `[Path]` section in the AaronJr input file is used to define which steps are to be displayed, which structures correspond to each step in the reaction, what labels to use, and allows for customizations such as colors used (e.g. *cis* is red, *trans* is blue), line thickness, or aspect ratio. Template names (or `&keywords` in the `[Results]` section) are used to specify how the plot will be organized, and a wildcard character ('*') can be used in place of selectivity specifications (e.g. *cis*, *trans*) and will be propagated to the labels appropriately. Figure 3.11 shows the `[Plot]` section specification used (left) to generate the diagram on the right. Each line in the 'path' option corresponds to a step in the diagram; the template specification is in curly brackets, followed by the label name in square brackets (in this case, the template structures are from other configuration files, which are pulled in using the `[Configs]` section, and these names prefix the template specification).

3.4 AaronJr Implementation

Collecting template structure files, parsing requested modifications from the configuration file and applying them to the templates, generating computational input files, and parsing computational output files

```
[Configs]
1a = reactant.ini
rxn = rxn.ini
2a = product.ini

[Plot]
color = red, blue
path:
  1a{INT}       [Reactant]
  rxn{INT_*}    [*-Int]
  rxn{TS_Isom}  [Isom TS]
  rxn{INT_*}    [*-Int]
  rxn{TS_RE_*}  [* RE TS]
  2a{INT_*}     [*-Product]
```

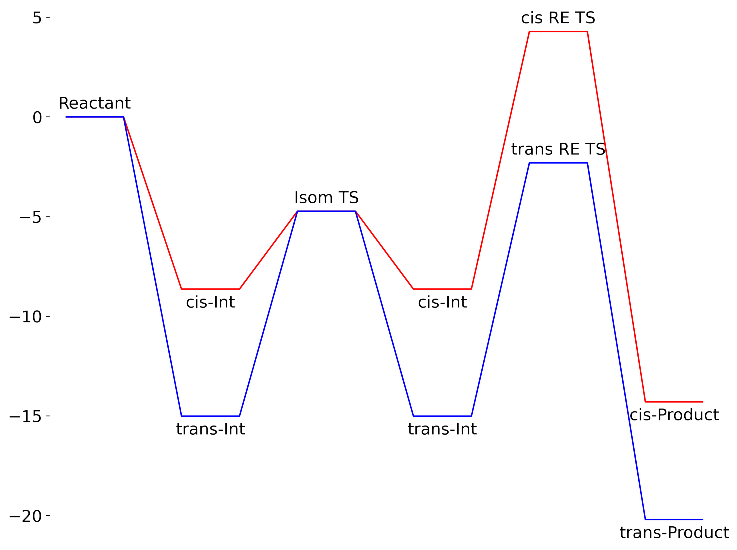


Figure 3.11: Using the `[Plot]` section for multi-step reactions

is handled by modules implemented in the AaronTools package.[3] The `AaronTools.fileIO` module currently includes support for Gaussian,[20] Orca,[22] Psi4,[21] xtb,[46] and Crest.[14] While additional work is still needed to ensure the error-resolution implementation is appropriately defined for all software packages (currently only Gaussian is fully supported for all errors detected by AaronTools) most errors can be automatically resolved and any that cannot are reported to the user in an easy to understand manner.

Building the executable scripts that are responsible for calling the different computational chemistry packages is done using the Jinja2 templating library to fill in necessary information, such as the working directory, scratch directory, and file names for the input and output files. For xtb and Crest, command line options are used extensively to specify the charge, level of theory, etc., and Jinja2 allows for flexible template design (such as if-else blocks) to handle the wide array of possibilities for appropriately generating the script. These templates are provided in the AaronJr library, but the user can alter these by placing an updated copy in their personal library, with a location defined by the `AARONLIB` environmental variable.

The Paramiko library is used for connecting to remote computing resources and handling remote file transfers, allowing users to run AaronJr on a workstation or laptop via the command line shell while submitting jobs to remote HPC resources via standard SSH and SFTP protocols using key authentication.

FireWorks[15] allows for the definition of an `FWAction` that will be performed once a job finishes running. This allows AaronJr to store computational results in the database, to update the atomic coordinates for use in the following steps, and to update the job specification parameters when resolving errors. The job progress monitoring included with FireWorks will automatically mark jobs with bad exit statuses. A job is marked as FIZZLED if problems were encountered that report a non-zero exit status to the shell. These are usually caused either by errors in the executable script templates or issues with the queue itself, so a limit of three retries is imposed. The user is provided with a description of the error (file not found, cannot connect to queue, <command> returned non-zero exit status, etc.) and instructed to re-run AaronJr after fixing the source of the error — or to contact us with a bug report if the error is on our end. However, errors found in computational output files (wrong number of negative eigenvalues, optimization step limit surpassed, syntax error in input file, etc.) are handled automatically by AaronJr. Because FireWorks only knows if a job has completed successfully, not if the content of the output file is unacceptable for moving forward in the workflow, a validation method has been written that will mark completed jobs with computational errors that need to be resolved as DEFUSED. If a job is marked as DEFUSED, an error resolution method is called that will make modifications to the job (adding keywords to the computational input file, updating the atomic coordinates, etc.) and re-run the step. In some cases, especially for TS structure optimizations, it is necessary to return to a previous step and make corrections to that job as well, and AaronJr is capable of doing so. A limit of ten retries is imposed to prevent disastrously poor template structures from being infinitely resubmitted and wasting computational resources.

The `FWAction` also allows us to specify what data or metadata should be stored in the database. Chemically pertinent information is parsed from the computational output files using AaronTools, as well as information concerning the run itself, such as the number of optimization steps, resources used, theory parameters assigned, etc. This information is converted to JSON format and stored in the FireWorks database, allowing both tracking of past runs for logging purposes as well as providing the opportunity to implement smarter error-correction schemes.

3.5 Availability and Installation

AaronJr is free and open source and available through GitHub.[57] Installation and configuration instructions are provided in our GitHub Wiki.

3.6 Conclusions

The fields of quantum chemistry and electronic structure theory have made tremendous advances over the last two decades, and we are now armed with methods capable of providing robust predictions for complex molecular systems. These advances have created a new problem arising from the need to manage the large numbers of computations that are required for modern chemical applications. Above, we described the QChASM workflow manager AaronJr, which allows for the simple automation of complex quantum chemistry workflows across the quantum chemistry packages Gaussian, Psi4, ORCA, Crest, and xtb. This includes automated error handling, data collection, and convenience commands for data analysis.

AaronJr should further open the door for widespread applications of modern quantum chemistry methods to challenging chemical problems. Most importantly, by eliminating the need for the manual creation of input files and parsing of output files, AaronJr frees the computational chemist to focus on the chemistry rather than spending their days making input files, parsing output files, and babysitting HPC job progress.

3.7 Funding Information

This work was funded by National Science Foundation Grant CHE-1665407.

Chapter 4

Example Applications of AaronJr

4.1 Introduction

AaronJr provides a flexible yet simple command line interface to automate nearly any quantum chemistry application using Gaussian, ORCA, Psi4, xtb, and Crest. To further demonstrate both the simplicity and power of AaronJr, we describe several hypothetical projects that can be carried out using AaronJr using a single input file or combination of several simple input files.

As discussed in the previous chapter, all required information can be contained in a single AaronJr input file. More practically, the user can set personal defaults that allow for more compact and concise input files for typical applications. These defaults can then be overridden for entire workflows or for individual steps within a workflow. In the following examples, to reduce the amount of clutter in the example input files, we assume the user has a personal default configuration file (stored at `$AARONLIB/config.ini`) containing the following:

```
# remote directory for saving computational input/output files
remote_dir = /home/%{$HPC:user}/chem/%{$name}

[Theory]
method = B3LYP
basis = !tm 6-31G(d)
        tm LANL2DZ
# options for relaxation of changes
1 method = PM6

[Job]
exec_type = gaussian
memory = %{$procs * 2}GB
procs = %{$ppn * $nodes}
nodes = 1
ppn = 8
wall = 12
```

```
# options for relaxation of changes
1 ppn = 2
1 wall = 2

[HPC]
user = myusername123
host = hpc.host.address
scratch_dir = /scratch/%{$user}
queue_type = SLURM
queue = batch
```

This provides defaults for all input files, with any necessary values added or overwritten in the AaronJr input file.

Note that one may use functions to define option values; these functions are enclosed in "%{..}" and variables corresponding to other option values follow the syntax "$<Option Name>" if the variable corresponds to an option in the same section or "$<Section Name>:<Option Name>" if an option from a different section is needed. For example, the number of processors requested ('procs' in the `[Job]` section) is equal to the product of the processors per node and the number of nodes, and the memory requested ('memory' in the `[Job]` section) is 2GB for each processor; now we only need to update the 'ppn' option in our project's AaronJr input file — the values for the 'procs' and 'memory' options will be resolved automatically. Similarly, the remote directory to store input and output files is built using the user name for the HPC and the name associated with the AaronJr input; the value of 'name' will default to the file name of the AaronJr input file, without the '.ini' file extension, but can be explicitly set using the 'name' keyword in the global section (the unlabeled section at the top of the file) of the input file. These functions are evaluated at runtime, allowing one to greatly simplify AaronJr input files while still ensuring flexibility. This example user configuration is also available on the AaronJr GitHub wiki page.[57]

4.2 Simple Examples

4.2.1 Generating isomers for a metal-centered coordination complex

A common task in coordination chemistry is the enumeration and optimization of all possible coordination complexes of a given set of ligands around a metal center. For multi-dentate ligands lacking symmetry, the number of such complexes can quickly reach the hundreds, and generating these manually is best avoided. The following input file instructs AaronJr to build the six unique octahedral coordination complexes of two CO ligands, two PPh_3 ligands, and two H ligands bound to Fe (see Figure 4.1). The `[Job.Minimum]` workflow is applied, which will consist of a geometry optimization and a vibrational frequencies computation. The processors per node requested is 6, which will cause the values for 'procs' and 'memory' to be updated to 6 and 12GB, respectively. Any files needed to run the jobs on the HPC will be stored under the `chem/Fe_coordination_complex` subdirectory of the user's home area on the HPC cluster.

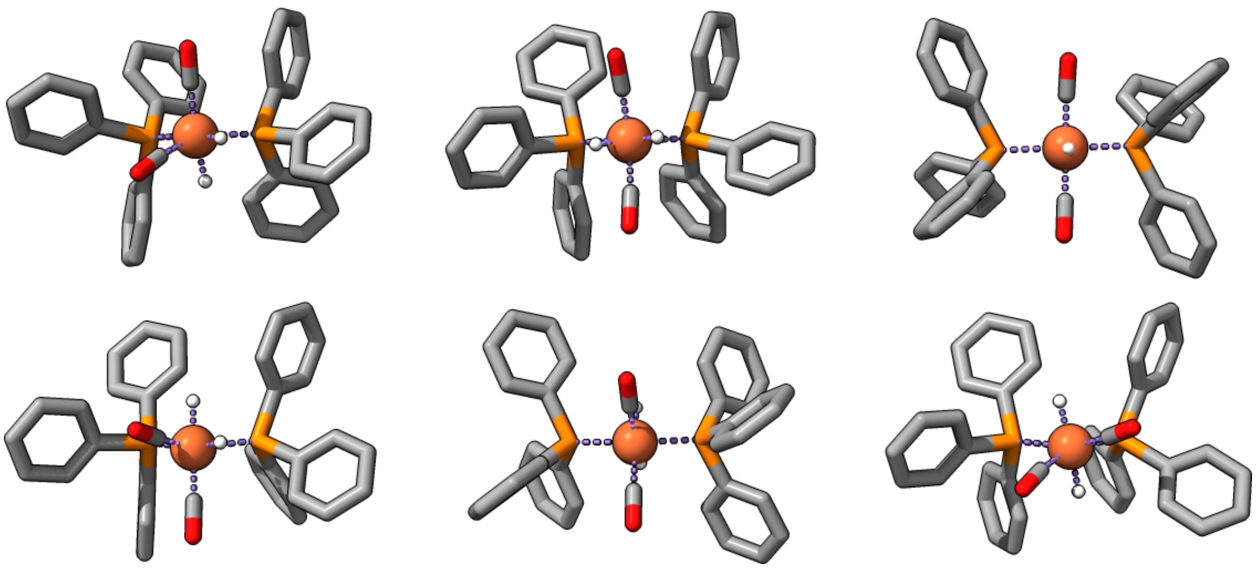


Figure 4.1: The six possible octahedral coordination complexes generated

```
name = Fe_coordination_complex

[Geometry]
structure:
  coordination_complex = octahedral
  center = Fe
  ligands = CO CO PPh3 PPh3 H H

[Job]
include = Minimum
ppn = 6
```

4.2.2 Optimizing urea and thiourea derivatives

```
name = urea_derivatives

[Geometry]
structure.O = urea.xyz
&call:
    structure.S = structure.O.copy()
    structure.S.change_element("O", "S", adjust_bonds=True)
```

```
[Substitution]
&combinations:
   6 = H, Me, Et, iPr, tBu, F, Cl, Br, I, CF3
   7 = H, Me, Et, iPr, tBu, F, Cl, Br, I, CF3

[Theory]
basis = !I 6-31G(d)
        I LANL2DZ
ecp = I LANL2DZ

[Job]
include = Minimum
ppn = 4
```

This input file will produce 200 structures total, 100 structures each from the urea and thiourea templates. In the [Geometry] section, "urea.xyz" is loaded as "structure.O", and the '&call' directive is used to call AaronTools methods to store a copy as "structure.S" and change the oxygen atom to a sulfur atom (adjusting the bond length as necessary). In the [Substitution] section, the '&combinations' directive is used to replace atoms 6 and 7 (see Figure 4.2) with all combinations of the substituents listed. Notice that although atoms 6 and 7 were originally hydrogens, the H substituent is included in the combination scheme; this is necessary to ensure that singly-substituted derivatives (atom 6 *or* atom 7 are substituted, not both) are included. We have also updated the 'basis' and 'ecp' settings, as our user-default configuration would attempt to apply the 6-31G(d) basis set to iodine (which is undefined); if this had been forgotten, AaronJr would exit the run and print an error message instructing the user to fix the basis set in their input file. The 'ppn' requested has also been reduced due to the small size of the molecules considered.

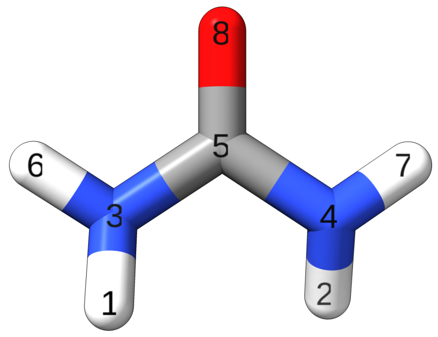


Figure 4.2: The structure saved as "urea.xyz", with atom numbering shown

4.3 Screening multiple ligands and substrates for a catalytic cycle

As a final example, we consider a more complex potential application involving the screening of multiple ligands and substrates for a Pd-catalyzed cyclization. Methylene oxindoles are found in many pharmaceutically active molecules and are useful intermediates in natural product syntheses.[58] In 2015, Schoenebeck, Lautens, et al. reported an exclusively *trans*-selective intramolecular chlorocarbamoylation of alkynes using a palladium-centered catalyst featuring a phosphoadamantane (PA-Ph) ligand to form methylene oxindoles (see Figure 4.3).[59]

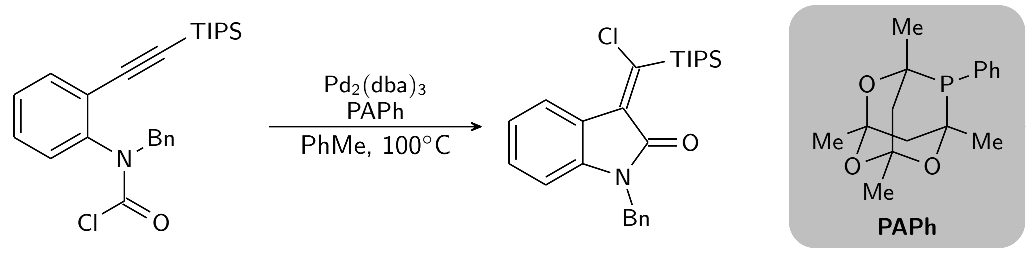


Figure 4.3: Intramolecular chlorocarbamoylation

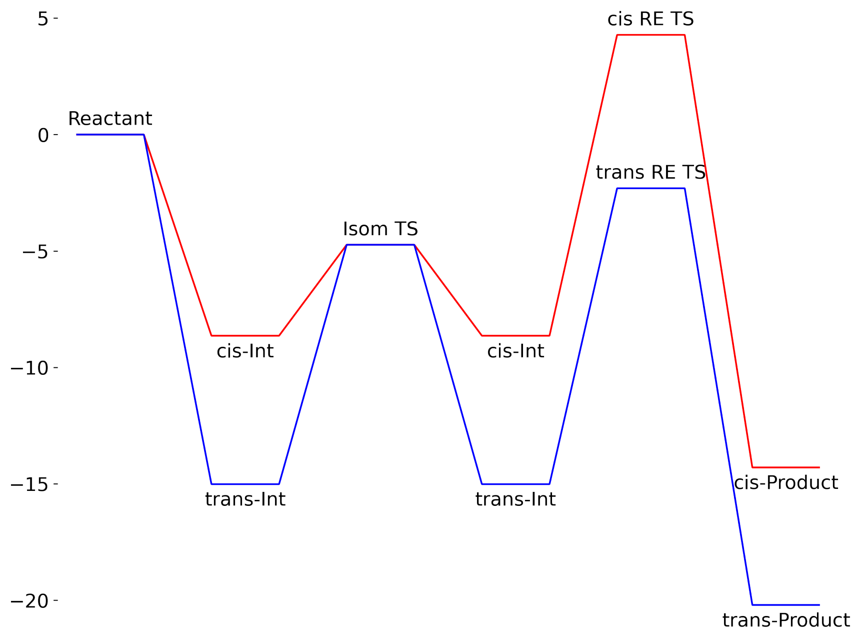
Their computational study suggested this selectivity is due to accessible *cis*/*trans* isomerization (Isom TS) of the aklyne-insertion intermediate (Int), prior to reductive elimination (RE TS) leading to the final product. Because the *trans*-intermediate and the *trans*-reductive elimination transition state are lower in energy than their corresponding *cis*-isomers, only the *trans*-product was observed experimentally.[59] However, one may wonder how changes to the alkyne substrate or PA-Ph ligand would affect the relative energies of the transition states and intermediates and how these changes would affect the final product ratios. Such a question can be easily answered by exploiting the automation provided by AaronJr. Figure 4.4 shows the reaction pathway diagrams for the original substrate and a substituted substrate created using the `AaronJr plot` command. Figure 4.5 shows the project input file, as well as one of the child configurations; the other child configurations are similar, simply with differing atom indices for the substitution requests and differing template locations.

While the results shown in Figure 4.4 show some variation in how favorable *cis*-reductive elimination is when the substrate is changed, the entire reaction path must be considered to truly determine how a given change to the reaction would affect the catalytic cycle and product ratios. In the case of changing TIPS on the reactant to a trichlorosilyl (TCS) group, the reaction intermediates corresponding to the reactant/catalyst complex are more favorable than the final products (Figure 4.4), meaning the turnover frequency of the catalyst would become detrimentally reduced. Additionally, while the *cis*-RE TS structure has a reasonable barrier, the isomerization pathway is essentially inaccessible. Thus, while selectivity was reversed, more adjustments are needed for this reaction to be promising enough to warrant experimental study.

4.4 Conclusions

As shown above, AaronJr allows for the automation of quantum chemistry applications ranging from the generation and optimization of all possible coordination compounds of a given composition, substituted analogs of some template structure, and even the screening of multiple ligands and substrates for complex

Original substrate



Modified substrate TCS: (iPr groups of TIPS substituted for Cl)

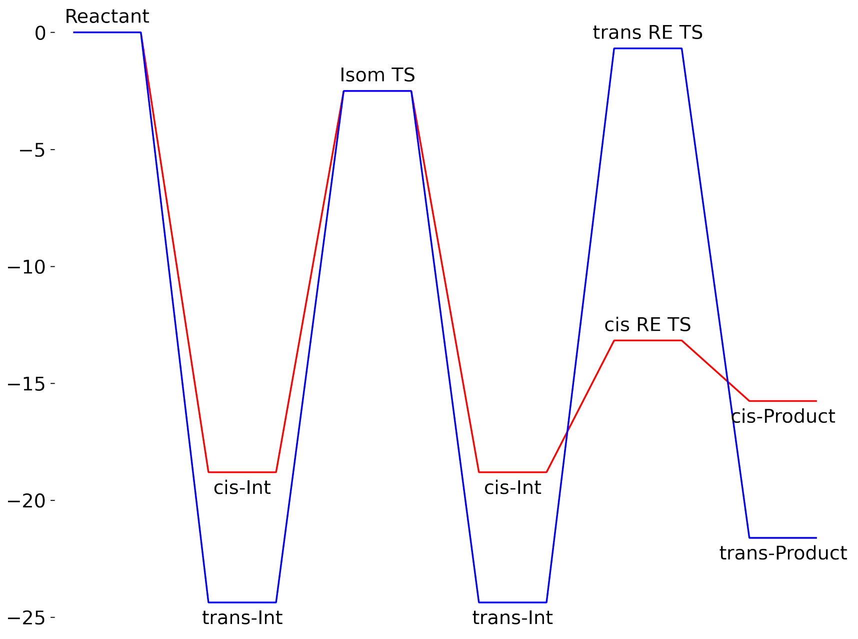


Figure 4.4: Gibbs free energy (kcal/mol) pathway for select steps in the catalytic cycle at the CPCM (toluene) M06L/def2-TZVP//B3LYP/6-31G(d) (LANL2DZ) level of theory at 373 K.

chlorocarbamoylation.ini

```
[Configs]
rxn = ./rxn.ini
1a = ./reactant.ini
2a = ./product.ini
Pd_PAPh2 = ./Pd-PAPh2.ini
PAPh = ./PAPh.ini

[Results]
&cat = Pd_PAPh2 - PAPh
&1a = 1a + cat:None
&2a = 2a + cat:None
relative = 1a{INT}
selectivity = cis, trans

[Plot]
color = red, blue
path:
  1a{INT}       [Reactant]
  rxn{INT_*}    [*-Int]
  rxn{TS_Isom}  [Isom TS]
  rxn{INT_*}    [*-Int]
  rxn{TS_RE_*}  [* RE TS]
  2a{INT_*}     [*-Product]
```

rxn.ini

```
[Reaction]
reaction = chlorocarbamoylation
template = Pd-PAPh

[Theory]
include = detect
solvent = toluene
solvent_model = cpcm
5 type = SP
5 method = M06L
5 basis = def2tzvp

[Substitution]
reopt=True
TMS: 32,42,52=Me
TES: 32,42,52=Et
TTBS: 32,42,52=tBu
TCS: 32,42,52=Cl
TFS: 32,42,52=F
TOS: 32,42,52=OH
```

Figure 4.5: Parent (left) and one of its child (right) input files for the multi-step reaction run.

organometallic reactions. Our hope is that by automating the routine tasks encountered in quantum chemistry applications we, as a field, can turn more aggressively toward even more complex and compelling molecular systems.

Chapter 5

Conclusion

The QChASM suite of tools can help make computational chemistry an even more invaluable tool for research, and these tools have already been widely adopted across academia and industry. By automating the steps taken to complete a project, AaronJr can provide computational chemists with the ability to focus on chemistry rather than on the tedious process of submitting jobs, addressing errors, and collating results. Using AaronTools — whether via AaronJr, using the SEQCROW plugin, on the command line, or in a Python script — allows one to efficiently build and modify structure files for systems of interest as well as to quickly gather the information about these systems needed for analysis.

Additionally, I believe these tools can help experimental chemists take advantage of the insights one can glean from computational studies, which will increase experimental efficiency with respect to time, money, and resources. The error correction and monitoring methods implemented in AaronJr drastically reduce the time spent determining the cause of a failed computation and adjusting the computational parameters or chemical structure to rectify the error. When automated error resolution methods fail, concise error messages make it easier to remedy the situation by hand, as even simple typos or missing files can produce cryptic error messages in the computational output. The data collection and analysis tools provided are flexible but powerful and can alleviate the need to become a command line power-user as well as limiting the possibility of data transcription mistakes that are so common when, say, copying and pasting data from output files into a spreadsheet. All this, in turn, means inexperienced and experienced users alike can turn their attention to discovery and exploration.

The flexible design of the AaronJr code base and the improved data structures in AaronTools allow for application to many varying types of systems of interest, and extension to support new computational chemistry programs or new structural modification methods or data collection methods to address specific research needs is possible. This means our tools can grow as the field of computational chemistry grows and allows us to better address the needs of our users as the adoption of our tools into research workflows increases. Overall, I believe our work greatly contributes to the growth of the field as a whole and can improve the quality and efficiency of computational investigations of chemical systems.

Appendix A

Canonical Ranking Algorithm for Atoms

One incredibly useful application of the new AaronTools data structures (specifically, the Geometry and Atom objects) is the ability to canonically order the atoms in a molecule. This allows one, for example, to perform an RMSD calculation of two similar molecules without the atoms being in the same order — a task that is necessary when comparing conformers as symmetric rotations of functional groups (Figure A.1, left). An invariant string, inspired by those described in the canonical SMILES implementation,[60] is produced for each atom which encodes the number of non-hydrogen connections, the sum of bond orders for bonds with non-hydrogen atoms, the atomic number, and the number of attached hydrogens. Using this invariant allows for chemically relevant similarities to be conserved. For example, changing the oxygen atom of an alcohol to a sulfur atom would result in the same canonical order for both the alcohol and the thiol (in most cases), allowing one to easily compare molecules with chemically similar structures despite their actual atoms being different (Figure A.1, right).

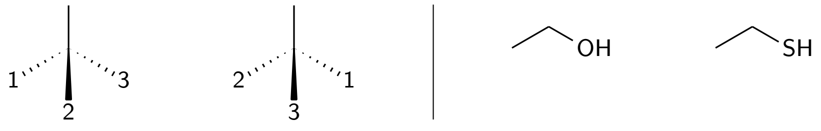


Figure A.1: RMSD of *t*Bu rotations (left) should be zero, and one should be able to align ethanol and ethanethiol (right) despite their atoms being different.

The ranking algorithm is similar to that described in work by Landrum et al.,[61] however, ties are also broken based on the relative location of atoms in Cartesian space. First, the initial invariants are used to partition the atoms into groups that share the same invariant. Next, an attempt at breaking ties in each partition containing two or more atoms is undertaken. The rank of an atom is multiplied by the ranks of each connected atom (using prime number ranks to ensure products are distinct), and then these resulting ranks are used to break ties. The atom list is then partitioned again, and partitions containing multiple atoms are iteratively improved by using the neighbors' prime-rank product until no further changes in the partitions occur. At this point, it is no longer possible to break ties based purely on atom identity and connectivity.

Tied atoms sharing a common neighbor are now broken geometrically. An axis of rotation is defined going from the center of mass of the structure to the common neighbor. A reference point is chosen to be the center of mass of the tied atoms, unless this is too close to the axis of rotation — in that case the tied-atom closest to the center of mass is used. The angle of rotation from this starting point to the tied atoms is calculated using the four-quadrant inverse tangent function; this gives radian values between $-\pi$ and π. If this does not break the tie to a reasonable level (i.e., the angles produced are equivalent to one decimal place), the angle calculation is done again using a rotational axis defined by the center of mass of atoms connected to the shared atom that are not part of the tie, instead of the center of mass of the entire molecule; this gives a more localized rotation axis and can improve the distinction between tied atoms, but it is slower to calculate and thus only used when needed. These resulting angles are used to order the tied atoms and update the partitions. Spatial tie-breaking followed by neighbors' ranks tie-breaking is repeated until no changes occur in the rankings.

This algorithm allows for chirality to be taken into account in the canonical ranking system, and maintains ties when they occur due to symmetry. The AaronTools canonical ranking code can be used in future development of symmetry detection and chirality determination methods.

Bibliography

(1) Dennington, R.; Keith, T.; Millam, J. GaussView, Version 6.1.1, 2019.

(2) Hanwell, M. D.; Curtis, D. E.; Lonie, D. C.; Vandermeersch, T.; Zurek, E.; Hutchison, G. R. *J Cheminform* **2012**, *4*, 1–17.

(3) Ingman, V. M.; Schaefer, A. J.; Andreola, A. R.; Wheeler, S. E. *WIREs Comp. Mol. Sci.* **2020**, *11*, e1510.

(4) Rooks, B. J.; Haas, M. R.; Sepúlveda, D.; Lu, T.; Wheeler, S. E. *ACS Catalysis* **2014**, *5*, 272–280.

(5) Doney, A. C.; Rooks, B. J.; Lu, T.; Wheeler, S. E. *ACS Catalysis* **2016**, *6*, 7948–7955.

(6) Guan, Y.; Wheeler, S. E. *Angew Chem Int Ed Engl* **2017**, *56*, 9101–9105.

(7) Guan, Y.; Ingman, V. M.; Rooks, B. J.; Wheeler, S. E. *J Chem Theory Comput* **2018**, *14*, 5249–5261.

(8) Frisch, M. J. et al. Gaussian 09 Revision E.01, Gaussian Inc. Wallingford CT 2009.

(9) Wheeler, S. E.; Seguin, T. J.; Guan, Y.; Doney, A. C. *Acc Chem Res* **2016**, *49*, 1061–9.

(10) Grimme, S. *Chemistry A European Journal* **2012**, *18*, 9955–64.

(11) Vaganov, V. Y.; Fukazawa, Y.; Kondratyev, N. S.; Shipilovskikh, S. A.; Wheeler, S. E.; Rubtsov, A. E.; Malkov, A. E. *Adv. Synth. Catal.* **2021**, *362*, 5467–5474.

(12) Liu, G.; Liu, X.; Cai, Z.; Jiao, G.; Xu, G.; Tang, W. *Angew Chem Int Ed Engl* **2013**, *52*, 4235–8.

(13) Schaefer, A. J.; Wheeler, S. E. **in preparation**.

(14) Pracht, P.; Bohle, F.; Grimme, S. *Phys. Chem. Chem. Phys.* **2020**, *22*, 7169–7192.

(15) Jain, A.; Ong, S. P.; Chen, W.; Medasani, B.; Qu, X.; Kocher, M.; Brafman, M.; Petretto, G.; Rignanese, G.-M.; Hautier, G.; Gunter, D.; Persson, K. A. *Concurrency and Computation: Practice and Experience* **2015**, *27*, CPE-14-0307.R2, 5037–5059.

(16) Santoro, S.; Kalek, M.; Huang, G.; Himo, F. *Acc Chem Res* **2016**, *49*, 1006–18.

(17) Vitek, A. K.; Jugovic, T. M. E.; Zimmerman, P. M. *ACS Catalysis* **2020**, *10*, 7136–7145.

(18) Foscato, M.; Jensen, V. R. *ACS Catalysis* **2020**, *10*, 2354–2377.

(19) Goddard, T. D.; Huang, C. C.; Meng, E. C.; Pettersen, E. F.; Couch, G. S.; Morris, J. H.; Ferrin, T. E. *Protein Sci* **2018**, *27*, 14–25.

(20) Frisch, M. J. et al. Gaussian 16 Revision C.01, Gaussian Inc. Wallingford CT, 2016.

(21) Turney, J. M. et al. *Wiley Interdisciplinary Reviews-Computational Molecular Science* **2012**, *2*, 556–565.

(22) Neese, F. *Wiley Interdisciplinary Reviews-Computational Molecular Science* **2018**, *8*, DOI: `132710.1002/wcms.1327`.

(23) Bootsma, A. N.; Doney, A. C.; Wheeler, S. E. *Journal of the American Chemical Society* **2019**, *141*, 11027–11035.

(24) Seguin, T. J.; Wheeler, S. E. *ACS Catalysis* **2016**, *6*, 2681–2688.

(25) Sandoval, C. A.; Ohkuma, T.; Muniz, K.; Noyori, R. *J Am Chem Soc* **2003**, *125*, 13490–503.

(26) Verloop, A.; Hoogenstraaten, W.; Tipker, J. *Drug Design* **1976**, 165–207.

(27) Vreven, T.; Byun, K. S.; Komáromi, I.; Dapprich, S.; Montgomery, J. A.; Morokuma, K.; Frisch, M. J. *Journal of Chemical Theory and Computation* **2006**, *2*, 815–826.

(28) Iwamoto, H.; Imamoto, T.; Ito, H. *Nature Communications* **2018**, *9*, 2290.

(29) Iwamoto, H.; Imamoto, T.; Ito, H. *Nat Commun* **2018**, *9*, 2290.

(30) Wheeler, S. E. *Accounts of Chemical Research* **2013**, *46*, 1029–1038.

(31) Houk, K. N.; Liu, F. *Accounts of Chemical Research* **2017**, *50*, 539–543.

(32) Lam, Y. H.; Grayson, M. N.; Holland, M. C.; Simon, A.; Houk, K. N. *Acc Chem Res* **2016**, *49*, 750–62.

(33) Peng, Q.; Duarte, F.; Paton, R. S. *Chem. Soc. Rev.* **2016**, *45*, 6093–6107.

(34) Ryu, H.; Park, J.; Kim, H. K.; Park, J. Y.; Kim, S.-T.; Baik, M.-H. *Organometallics* **2018**, *37*, 3228–3239.

(35) Ahn, S.; Hong, M.; Sundararajan, M.; Ess, D. H.; Baik, M. H. *Chem Rev* **2019**, *119*, 6509–6560.

(36) Poree, C.; Schoenebeck, F. *Acc Chem Res* **2017**, *50*, 605–608.

(37) Sobez, J. G.; Reiher, M. *J Chem Inf Model* **2020**, *60*, 3884–3900.

(38) Young, T. A.; Gheorghe, R.; Duarte, F. *J Chem Inf Model* **2020**, *60*, 3546–3557.

(39) Ioannidis, E. I.; Gani, T. Z.; Kulik, H. J. *J Comput Chem* **2016**, *37*, 2106–17.

(40) Turcani, L.; Berardo, E.; Jelfs, K. E. *J Comput Chem* **2018**, *39*, 1931–1942.

(41) Young, T. A.; Silcock, J. J.; Sterling, A. J.; Duarte, F. *Angew Chem Int Ed Engl* **2020**, DOI: `10.1002/anie.202011941`.

(42) Rodriguez-Guerra Pedregal, J.; Gomez-Orellana, P.; Marechal, J. D. *J Chem Inf Model* **2018**, *58*, 561–564.

(43) Zapata, F.; Ridder, L.; Hidding, J.; Jacob, C. R.; Infante, I.; Visscher, L. *J Chem Inf Model* **2019**, *59*, 3191–3197.

(44) Parrish, R. M. et al. *J Chem Theory Comput* **2017**, *13*, 3185–3197.

(45) Neese, F.; Wennmohs, F.; Becker, U.; Riplinger, C. *J Chem Phys* **2020**, *152*, 224108.

(46) Bannwarth, C.; Caldeweyher, E.; Ehlert, S.; Hansen, A.; Pracht, P.; Seibert, J.; Spicher, S.; Grimme, S. *WIREs Comput. Mol. Sci.* **2021**, *11*, e1493.

(47) Becke, A. D. *The Journal of Chemical Physics* **1993**, *98*, 5648–5652.

(48) Foster, J. P.; Weinhold, F. *Journal of the American Chemical Society* **1980**, *102*, 7211–7218.

(49) Pinski, P.; Riplinger, C.; Valeev, E. F.; Neese, F. *J Chem Phys* **2015**, *143*, 034108.

(50) Riplinger, C.; Sandhoefer, B.; Hansen, A.; Neese, F. *J Chem Phys* **2013**, *139*, 134101.

(51) Riplinger, C.; Neese, F. *J Chem Phys* **2013**, *138*, 034106.

(52) Neese, F.; Hansen, A.; Liakos, D. G. *J Chem Phys* **2009**, *131*, 064103.

(53) Dunning, T. H. *Journal of Chemical Physics* **1989**, *90*, 1007–1023.

(54) Hay, P. J.; Wadt, W. R. *The Journal of Chemical Physics* **1985**, *82*, 270–283.

(55) Silva, F. T.; Lins, S. L. S.; Simas, A. M. *Inorg Chem* **2018**, *57*, 10557–10567.

(56) Ribeiro, R. F.; Marenich, A. V.; Cramer, C. J.; Truhlar, D. G. *J Phys Chem B* **2011**, *115*, 14556–62.

(57) Q-ChASM: Quantum Chemistry Automation and Structure Manipulation, GitHub.

(58) Le, C. M.; Sperger, T.; Fu, R.; Hou, X.; Lim, Y. H.; Schoenebeck, F.; Lautens, M. *J. Am. Chem. Soc.* **2016**, *138*, 14441–14448.

(59) Le, C. M.; Hou, X.; Sperger, T.; Schoenebeck, F.; Lautens, M. *Angew. Chem. Int. Ed.* **2015**, *54*, 15897–15900.

(60) Weininger, D.; Weininger, A.; Weininger, J. L. *J. Chem. Inf. Model.* **1989**, *29*, 97–101.

(61) Schneider, N.; Sayle, R. A.; Landrum, G. A. *J. Chem. Inf. Model.* **2015**, *55*, 2111–2120.