CONFORMER SEARCHING

CONFORMER SEARCHING USING AN EVOLUTIONARY ALGORITHM

By JENNIFER HANNAH GARNER, B.Sc.

A Thesis Submitted to the School of Graduate Studies in Partial Fulfillment of the Requirements for the Degree Master of Science

McMaster University MASTER OF SCIENCE (2019) Hamilton, Ontario (Chemistry & Chemical Biology)

TITLE: Conformer Searching using an Evolutionary Algorithm
AUTHOR: Jennifer Hannah Garner, B.Sc. (University of Guelph)
SUPERVISOR: Professor Paul W. Ayers
NUMBER OF PAGES: xvii, 185

Lay Abstract

A conformer search affords the low-energy arrangements of atoms that can be obtained via rotation around bonds. Conformers provide insight about the chemical reactivity and physical properties of a molecule. With increasing molecule size, the number of possible conformers increases exponentially. To search the space of possible conformers, this thesis presents *Kaplan*, which is a software package that implements a novel directed, stochastic, sampling technique based on an Evolutionary Algorithm (EA). *Kaplan* uses a special type of EA that stores sets of conformers in a `ring`-based structure. Unlike other conformer-specific packages, *Kaplan* provides the means to analyse and interact with found conformers. Known conformers of amino acids are used to verify *Kaplan*. Other tools for generating conformers are discussed, including a comparison of freely available software. *Kaplan* effectively finds the conformers of small molecules, but requires additional parametrisation to find the conformers of mid-sized molecules, such as Penta-Alanine.

Abstract

Conformer searching algorithms find minima in the Potential Energy Surface (PES) of a molecule, usually by following a torsion-driven approach. The minima represent conformers, which are interchangeable via free rotation around bonds. Conformers can be used as input to computational analyses, such as drug design, that can convey molecular reactivity, structure, and function. With an increasing number of rotatable bonds, finding optima in the PES becomes more complicated, as the dimensionality explodes.

Kaplan is a new, free and open-source software package written by the author that uses a `ring`-based Evolutionary Algorithm (EA) to find conformers. The `ring`, which contains population members (or `pmems`), is designed to allow initial PES exploration, followed by exploitation of individual energy wells, such that the most energetically-favourable structures are returned.

The strengths and weaknesses of existing publicly available conformer searchers are discussed, including *Balloon*, *RDKit*, *Openbabel*, *Confab*, *Frog2*, and *Kaplan*. Since *RDKit* is usually considered to be the best free package for conformer searching, its conformers for the amino acids were optimised using the MMFF94 forcefield and compared to the conformers generated by *Kaplan*. Amino acid conformers are well characterised, and provide insight for protein substructure. Of the 20 molecules, *Kaplan* found a lower energy minima for 12 of the structures and tied for 5 of them. *Kaplan* allows the user to specify which dihedrals (by atom indices) to optimise and angles to use, a feature that is not offered by other programs. The results from *Kaplan* were compared to a known dataset of amino acid conformers. *Kaplan* identified all 57 conformers of methionine to within 1.2Å, and found identical conformers for the 5 lowest-energy structures (i.e. within 0.083Å), following forcefield optimisation.

Acknowledgements

I would like to thank my supervisor, Professor Paul Ayers, for his guidance and instruction that improved the subsequent work and for the opportunity he gave me to broaden my skill set. Thank you also to Professor Spencer Smith, whose course in software engineering allowed me to write an anti-spaghetti software package in a reasonable amount of time. Thank you to Professor David Ogborn and his group for welcoming me to the Cybernetic Orchestra and for providing a creative outlet during this degree. Thank you to Girlz of Ayers Lab, with a "z": Kumru Dikmenli and Xiaomin Huang - it was a pleasure to share the trials and tribulations of graduate school with you. Also thank you to Kumru for allowing me to work with her on the database code (Vetee), and Xiaotian (Derrick) Yang for providing much-needed software (GOpt) to complete this work. Thank you to Christina E. Gonzàlez Espinoza, who took the time to help me understand some of the basics of quantum chemistry. Thank you to my partner, Justin Schonfeld, who listened to my talks and read my drivel and gave me feedback even when I am not a happy recipient of constructive criticism. Thank you open-source contributors whose code I used. Thank you to my computers: Artificus, BABY PRINCESS, Sir Oryx Prozar, and Mooja - you guys do all the heavy lifting, and I hope you receive autonomous intelligence soon. Thank you to anyone who gave me food or made me tea. Thank you to 9GAG and those who put up with my obsession with memes. Thank you to my physiotherapist, Julia Hochstein - without you I would not be able to sit long enough to write the rest of this thesis. For support at home, thank you mumsies, Emil, Danyolla and my dad. If you read this and did not see your name, thank you too.

Contents

0.1 Typographic and Naming Conventions

Software packages are denoted using *italics*. As it relates to code written by the author (i.e. part of the *Kaplan* conformer searching package), variables and function names use `snake_case` and objects use `CamelCase`. Code from other software packages follows the same convention as taken from said software package. The use of `fixed-width font` denotes that the word or line is a:

- terminal command (example: `obenergy filename.xyz`),
- a directory or path (example: `/home/username`),
- `variable_name`,
- `function_name`,
- `method_name`,
- an instance of a `class`, or
- `Class`* name

* For the name of a class, the program and module names are implied; the `Pmem` class is from `kaplan.pmem`, and the `Ring` class is from `kaplan.ring`.

0.2 Abbreviations and Symbols

Here is a list of commonly-used abbreviations from the thesis. The complete list of programmatic inputs to *Kaplan* are not given here and are instead explained in Section 3.4. Mathematical symbols are given in Table A.1 from the Appendix.

- 2D - two-dimensional.
- 3D - three-dimensional.
- AMMOS - Automated Molecular Mechanics Optimization tool for *in silico* Screening.

- API - Application Programming Interface.
- B3LYP - Becke, 3-parameter, Lee Yang Parr (a hybrid functional used in DFT).*
- BFGS - Broyden-Fletcher-Goldfarb-Shanno, a numerical optimisation algorithm.
- EA - Evolutionary Algorithm.
- CC - Coupled Cluster.*
- CI - Configuration Interaction.*
- csv - Comma-Separated Values (a file extension).
- DFT - Density Functional Theory.*
- ETDG - Experimental-Torsion Distance Geometry. A way to search for conformers. ETKDG is a variant of ETDG that applies general knowledge "K" terms.
- GA - Genetic Algorithm, a type of Evolutionary Algorithm.
- $\hbar$ - h-bar, equivalent to $\frac{h}{2\pi}$, where h is Planck's constant ($6.626070 \times 10^{-43} J \cdot s$).
- HF - Hartree-Fock.*
- InChI - IUPAC International Chemical Identifier.
- IQR - Interquartile Range, which describes the size of the box in a boxplot.
- IUPAC - International Union of Pure and Applied Chemistry.
- LFSR - Linear Feedback Shift Register.
- `mev` - Mating Event. The iterative sequence that drives an evolutionary algorithm. In some cases, a `mev` is referred to as a generation.

- MMFF94 - Merck Molecular Force Field.
- MOGA - Multi-Objective Genetic Algorithm.
- MP - Møller-Plesset, a type of perturbation theory.*
- MP2 - second order Møller-Plesset perturbation theory.*
- NIH - National Institutes of Health.
- NMR - Nuclear Magnetic Resonance, an experimental technique for studying electronic structure.
- *NumPy* - Numerical *Python*. Also shortened to `np` in the context of written code.
- PA - Penta-Alanine peptide, a test molecule.
- PES - Potential Energy Surface.
- PDB - Protein Data Bank, an online resource for protein structures. Note that pdb is also a file extension (legacy file format for PDB).
- `pmem` - Population Member (of an Evolutionary Algorithm).
- PUG - Power User Gateway.
- `pwd` - Present Working Directory.
- rad - Radians, a unit for measuring angles.
- REST - Representational State Transfer.
- RMSD - Root-Mean-Square Deviation.
- SATP - Standard Ambient Temperature and Pressure. Defined as 298.15K (25°C), 100kPa.
- sdf - Structure-Data File (a file extension).

- SE - Schrödinger Equation.
- SMARTS - SMILES Arbitrary Target Specification.
- SMILES - Simplified Molecular-Input Line-Entry System. A string of characters that specifies the connectivity (and in some cases stereochemistry) of a molecule.
- *stdev* - standard deviation.
- STO-3G - a minimum basis set of Slater Type Orbitals, where each is estimated using 3 primitive Gaussian orbitals.
- SVD - Singular Value Decomposition.
- *VMD* - Visual Molecular Dynamics. A visualisation program used to generate molecular images.
- xyz - Cartesian xyz-coordinates file (a file extension). See Section A.2.4 for an example.

* a quantum chemistry method.

0.3 Declaration of Academic Achievement

The author has written a software package called *Kaplan*, and acknowledges the contributions of the other software packages that are integral to its function (listed in Table 2.1). Fellow graduate students Xiaotian (Derrick) Yang and Kumru Dikmenli wrote software that was used by the author for this work. A thorough analysis of amino acid conformers was performed by the author, including the *Kaplan* results in relation to known structures. Other contributors to this work include Professor Paul Ayers, who guided the student in ways to improve upon the conformer searching and perform meaningful analysis. The student was able to contribute a bug fix to *Openbabel* as a direct result of this work, and has compiled information on how to use many cheminformatics software tools in one place.

List of Figures

List of Tables

Chapter 1

Introduction

This chapter will cover the background information on all relevant topics covered in this thesis. The scope and purpose of the thesis will be outlined.

1.1 Problem Description and Motivation

This thesis discusses conformer searching. The purpose of the document is twofold. First, this document will provide the reader with the knowledge necessary to understand how conformer searching is currently conducted in freely-available software packages, such as those that might be used in academia or for personal research. Examples of how to find and evaluate conformers using the external software packages will be given. Second, the author will present a new conformer searching package - called *Kaplan*[1], including a detailed set of parameters for its use. The benefits of the new package will be discussed; it is not intended to replace the current packages, but rather to complement existing methods and fill in the gaps that the author perceived while running the other programs. The strengths and weaknesses of the current tools will be revealed, so the reader can choose a software package that best suits their needs.

[1] *Kaplan* is the Turkish word for Tiger. The name does not have any special meaning other than being the author's favourite animal.

The amino acids are used as a test suite for the running of *Kaplan*, such that the reader can be convinced of the program's viability for conformer searching. The author will also indicate how the results from a *Kaplan* conformer search might be interpreted and analysed. After reading this document, the reader should be able to conduct a conformer search on their molecules of interest (by utilising one or more of the presented tools). They should also be comfortable in assessing the quality of their conformers, and using *Kaplan* to improve upon their results. The intended audience would be a graduate student or researcher who wishes to learn more about the structure and function of small molecules (up to a couple of hundred atoms in size) via a conformational search. This document would also be a useful starting point for possible contributors to the *Kaplan* conformer searching program, for which the author has made an effort to write documentation (in the code as part of functions, methods, classes, modules, etc.). The full source code for *Kaplan* is available at: `github.com/PeaWagon/Kaplan`.

1.1.1 Definition of a Conformer

Conformational isomers (henceforth shortened to conformers) are a set of molecules, each with a different spatial arrangement, that share the same number, type, and connectivity of atoms. According to the IUPAC (International Union of Pure and Applied Chemistry) Goldbook, a conformer is a member of a set of stereoisomers that adapts to a conformation and has a distinct potential energy minimum [33]. In other words, a conformer is produced by taking a parent molecule and applying free rotation to one or more of its bonds.

The definition of free rotation, loosely, is rotation about a bond with sufficiently low energy as to be observable during an experiment [33]. For example, rotation around a double bond (see Figure 1.1) is not usually considered to be "free", since the energy barrier is high enough to prevent both such conformations existing at Standard Ambient Temperature and Pressure (SATP) conditions (298.15K and 100kPa).

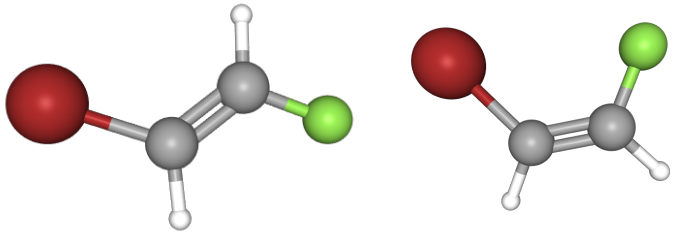


Figure 1.1: 1-bromo-2-fluoroethene has two isomers, E (left) and Z (right) that are unable to interconvert at SATP due to the presence of the double bond; therefore, these are geometric isomers rather than conformers. Figures generated using *PubChem* [91].

The prototypical example that is given for conformers is the cyclohexane molecule; this molecule has two main conformers: the boat configuration and the chair configuration. There is a barrier between these two conformers, but it is low enough such that conversion readily occurs at SATP.

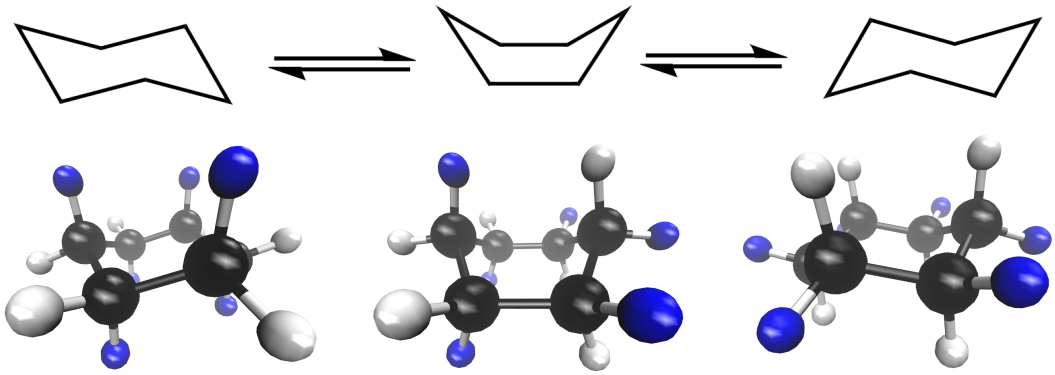


Figure 1.2: Cyclohexane is able to interconvert between the chair (left, right) and boat (middle) conformations in solution. The hydrogens in blue are in the axial position for the left chair conformation; the blue hydrogens are in the equatorial position for the right chair conformation. The top line diagram shows the carbon backbone during these interconversions.

1.1.2 The Importance of Conformers

Conformers are important in determining rates of reaction and interconversion. Conformers can be used to predict the stereochemistry of products and to elucidate reaction mechanisms. Depending on the temperature, pressure, and solvents used during an experiment, the favoured conformation may change, resulting in more or fewer side-products. In computational chemistry, an initial geometry is required to run most calculations. A conformer search can afford good starting structures from which to perform computational simulations of chemical processes.

An application of conformer searching is in the pharmaceutical industry for the purpose of identifying molecules that match a given pharmacophore [46]. A pharmacophore is a blueprint that specifies the structural and chemical features that a molecule should have to facilitate biological activity (as a drug molecule, for instance). Pharmacophore features can include hydrogen bond accepting or donating groups and hydrophobic or hydrophilic sites [73]. An example of a program that can search for 3D pharmacophores is LigandScout [49]. Important considerations for pharmacophore searching include how many conformers to produce and the energy range for conformers (usually relative to the global energy minimum). Some programs, like LigandScout, require that a training set of molecules is provided to generate a pharmacophore specification.

Enzymes (a subset of proteins) have concave binding pockets known as active sites that can catalyze chemical reactions with other molecules (specifically, substrates). Conformer searching is done to study these substrates to identify possible matches for the active site, or how often such matches occur [59][89][13][35][37][62].

1.1.3 Challenges in Conformer Searching

Since the rates of interconversion for conformers are not easy to resolve in real-time, most conformer searches are predicted using software or other computational tools. For small molecules (5-10 atoms) with few rotatable bonds, it is easy to predict the conformation that will have the lowest energy (i.e. the most stable conformation) by exhaustive search. Usually, the basic rules are to (1) space atoms out

such that larger atoms or larger groups are kept furthest apart, (2) reduce strain (such as in rings), and (3) maximise electrostatically-favourable (and minimise electrostatically-unfavourable) interactions (for example, between polar groups).

However, as the number of atoms increases and the number of rotatable bonds proliferates, the number of conformers (sampling once every 120-degree-rotation or so) increases exponentially with the number of atoms. Therefore, it becomes increasingly expensive for a human (or even a computer) to exhaustively search the possible conformations as the size of a molecule increases.

Conformer searching is multi-objective by nature; two main considerations are the energy and the relative geometries within a set of conformers, which will be reviewed in detail in the next section. For example, bioactive molecules, when binding with their target protein, may adopt a conformation that does not match the global energy minimum [45], thus necessitating a search for higher energy, yet structurally distinct, conformations.

1.2 Conformer Evaluation

The purpose of a conformer search is to provide a set of valid structures for a molecule. A valid structure is one that complies with the molecule's connectivity and spatial rules - i.e. its bond lengths and angles are within the bounds of known values for similar molecules, and there is no atom overlap. Such data can be can be attained from crystal structures [71], NMR experiments [57], and high-level computational experiments [104]. The problem then becomes to sort these valid structures into a representative sample that shows how the atoms within a molecule are most likely to arrange themselves during an experiment. To ascertain whether the "best" conformers have been found, their locations on the Potential Energy Surface (PES) can be analysed and compared, and the level of structural dissimilarity can be quantified. This section will describe methods to construct the PES and perform geometric analyses of conformers.

1.2.1 Energy Evaluation

There are two main types of energy evaluation - those that depend on classical mechanics and those that use quantum mechanics. Classical mechanics, or Newtonian mechanics, works well for massive bodies that are moving below the speed of light, and follows Newton's laws of motion.

For smaller entities, such as atoms and subatomic particles (electrons, protons, neutrons, etc.), quantum mechanics is necessary to accurately portray the motion and determine the energy of a system. In quantum mechanics, the energy and momentum of a bound system is quantized (meaning they might only have discrete values), objects exhibit particle-wave duality, and measurements are restricted in precision by the uncertainty principle. For example: the more that is known about a particle's momentum, the less that is known about the position of said particle, and vice versa.

From Section 1.1.1, a conformer was defined as representing an energy minimum, which means that there is a mapping from each given conformer geometry to an energy. To compare conformers energetically, a potential energy surface (PES) can be made. The dimension of the PES depends on the size and structure of the molecule; it is constructed by solving the Schrödinger Equation (SE) for all possible combinations of atomic positions. The most basic form of the SE is given in Equation 1.1. The SE is an eigenvalue problem affording the energy of the system, with the wavefunction representing the eigenvectors.

$$\hat{H}\Psi = E\Psi \tag{1.1}$$

In Equation 1.1, $\hat{H}$ is the Hamiltonian operator, Ψ is the wavefunction, and E is the energy. Here, an *operator* is defined (in the mathematical sense) as a transformation applied to a function that returns another function. The wavefunction by itself does not have meaning, but $|\Psi|^2$ represents a probability distribution for finding a particle at a given position. A valid wavefunction must be well-behaved; i.e. it is single-valued (returns one value per position) and $\int |\Psi|^2 d\tau > 0$, meaning that the chance of finding a particle in 3D coordinate space is finite.

As it relates to this work, the time-independent, non-relativistic, SE is used. The molecular Hamiltonian for this equation contains five main terms, which do not account for relativistic effects (such as spin-orbit coupling). The terms are: kinetic energy for nuclei (1) and electrons (2), and potential energy due to repulsion of nuclei (3), attraction of nuclei and electrons (4), and repulsions between electrons (5), as shown in Equation 1.2.

$$\hat{H} = \frac{-\hbar^2}{2}\sum_{\alpha}\frac{1}{m_\alpha}\nabla_\alpha^2 - \frac{\hbar^2}{2m_e}\sum_i \nabla_i^2 + \sum_\alpha \sum_{\beta>\alpha} \frac{Z_\alpha Z_\beta e^2}{4\pi\varepsilon_0 r_{\alpha\beta}} - \sum_\alpha \sum_i \frac{Z_\alpha e^2}{4\pi\varepsilon_0 r_{i\alpha}} + \sum_j \sum_{i>j} \frac{e^2}{4\pi\varepsilon_0 r_{ij}} \quad (1.2)$$

where $\hbar = \frac{h}{2\pi}$, m_e is the mass of an electron, m_α is the mass of a nucleus, α and β are nuclei, i and j are electrons, Z is atomic number, e is the charge of an electron, and ε_0 is the permittivity of free space. The variables $r_{\alpha\beta}$, $r_{i\alpha}$, and r_{ij} represent the distances between two nuclei, an electron and a nucleus, and two electrons, respectively. The del operator ∇ (also called nabla) is shortform for the vector of partial derivatives in all dimensions. The Laplacian operator (∇^2) represents the sum of the second partial derivatives for each of the 3D - x, y and z: $\nabla^2 = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} + \frac{\partial^2}{\partial z^2}$.

To evaluate the Hamiltonian, such that it might be useful for calculating the properties of molecules, some assumptions are made. Since nuclei have a much greater mass than electrons ($m_\alpha \gg m_e$), the approximation is made that the nuclei remain fixed while the electrons move. Therefore, terms (1) and (3) can be reasonably neglected from Equation 1.2 to afford the electronic Hamiltonian, which is given in Equation 1.3. Separating nuclear and electronic motion to simplify the SE is known as the Born-Oppenheimer approximation. Term (3) is in fact a constant (since nuclear position is now fixed), and is instead added to the electronic energy of the system.

$$\hat{H}_{el} = -\frac{\hbar^2}{2m_e}\sum_i \nabla_i^2 - \sum_\alpha \sum_i \frac{Z_\alpha e^2}{4\pi\varepsilon_0 r_{i\alpha}} + \sum_j \sum_{i>j} \frac{e^2}{4\pi\varepsilon_0 r_{ij}} \quad (1.3)$$

The electronic Hamiltonian can be used to solve the electronic SE (given in Equation 1.4 below), which contains U (the electronic energy plus nuclear repulsion

energy) and V_{NN} (term (3) from Equation 1.2).

$$(\hat{H}_{el} + V_{NN})\Psi_{el} = U\Psi_{el} \tag{1.4}$$

Therefore, by solving the electronic SE for a set of atomic positions, a PES can be constructed and conformations can be judged based on their relative potential energy. To solve the SE and get the energy of the system, two components are required: (1) a method, and (2) a basis set. Some families of quantum chemistry methods are: Hartree-Fock (HF), Coupled-Cluster (CC), Configuration Interaction (CI), Density Functional Theory (DFT), and Møller-Plesset (MP) perturbation theory. Detailed explanations of these methods can be found in references: [34] and [78].

1.2.2 Forcefields

Obtaining the energy of a system by solving the SE is expensive when compared to using classical, molecular mechanics methods. For quantum-based methods, the atoms are broken down into electrons and nuclei (as point masses), whereas in classical methods the atoms are represented as charged point particles. In most cases, a representative set of conformers can be found by approximating the U (potential energy) term from the electronic SE (Equation 1.4). Therefore, the PES can be more cheaply constructed using a classical, molecular mechanics forcefield to calculate the energy of a given set of coordinates. This approximation of the PES will henceforth be referred to as the energy landscape.

A forcefield is used to estimate the potential energy of a system such that geometries of the same molecule might be compared - e.g. is conformation A less strained than conformation B? The energy resulting from a quantum chemistry calculation is essentially always negative, whereas forcefield energies can be negative or positive (depending on which term dominates in the final summation). It is more useful to compare the energy difference rather than the absolute energy for both types of calculations.

The experiments presented in this thesis use the Merck Molecular Force Field

(MMFF94) [27][28][29][30][31]. MMFF94 was designed to accurately locate conformational minima and to reasonably describe energy barriers revolving around torsion angles. This forcefield was parametrised using a combination of high-level quantum chemistry calculations and crystal structures. It was chosen based on its suitability for torsion driven conformer searching (see Section 1.3), as well as for practical reasons; the forcefield was established in 1996, giving it more than 20 years of use. Multiple implementations of MMFF94 are also freely available, and new implementations can be validated using the test-suite (located here http://www.ccl.net/cca/data/MMFF94/). The forcefield is not expected to be perfect for all molecules, but its limitations are at least known and quantified.

A variant of the MMFF94 forcefield, called MMFF94s [39] - where the 's' stands for static, was made to account for the puckering of unstrained, delocalised, trigonal nitrogen atoms that occurs when using the MMFF94 forcefield. This variant is recommended in cases where structures optimised using a forcefield are to be compared to crystal structure geometries, since these nitrogens should appear planar. The results from this thesis are compared with computational data and not crystal structures; therefore, the MMFF94 variant was considered to be a suitable forcefield for conformer searching and subsequent analysis. The MMFF94 forcefield was compared to other forcefields (specifically, MFF94s, CFF95, CVFF, MSI CHARMm, AMBER, OPLS, MM2, and MM3) in this [40] review paper, where it was found to give erroneous energies for halocyclohexanes and in condensed-phase simulations.

The MMFF94 energy expression can be broken down into 7 terms, which represent the energies with respect to bond stretching, angle bending, stretch-bend interactions, out-of-plane (OOP) bending at tricoordinate centres, torsion interactions, van der Waals (vdW) interactions, and electrostatic interactions, respectively. Of the 7 summations shown in Equation 1.5, the first 5 terms result from bonded interactions, and the last 2 terms represent pairwise, non-bonded interactions. The units for energy are kilocalories per mole (kcal/mol), and angstroms (Å) and degrees (°)

are used for distance and angles, respectively.

$$E_{\text{MMFF}} = \sum EB_{ij} + \sum EA_{ijk} + \sum EBA_{ijk} + \sum EOOP_{ijk;l} + \sum ET_{ijkl} + \sum EvdW_{ij} + \sum EQ_{ij} \tag{1.5}$$

The terms in the MMFF94 forcefield will now be explained. Subscript format follows the same conventions as from the original paper [27], where lower-case is used to indicate specific atoms (e.g. i, j, k) and upper-case is used to identify numeric MMFF atom types (e.g. I, J, K). For the bonded terms, ij implies i and j are bonded atoms (which can be expanded for angles ijk - j is the angle vertex - and torsion angles $ijkl$). The notation $ijk;l$ denotes the Wilson angle, which is the angle between the j-l bond and the i-j-k plane. Any k terms are used to represent force constants. For the non-bonded terms - i.e. van der Waals and electrostatic interactions - i and j atom pairs are only included for atoms that are separated by 3 or more bonds.

Bond Stretching

The bond stretching term is a quartic function that is a product of a spring-like term and a quadratic anharmonic correction. This expression matches Equation (2) from [27].

$$EB_{ij} = 143.9325 \frac{kb_{IJ}}{2} \Delta r_{ij}^2 \cdot \left(1 + cs\Delta r_{ij} + \frac{7cs^2 \Delta r_{ij}^2}{12}\right) \tag{1.6}$$

Where $\Delta r_{ij} = r_{ij} - r_{IJ}^0$ specifies the difference in distance when comparing the actual bond length for the i^{th} and j^{th} atoms (r_{ij}) versus a reference bond length (r_{IJ}^0). $cs = -2\text{Å}^{-1}$ represents the cubic stretch constant, and kb_{IJ} represents the force constant of the spring term.

Angle Bending

The angle bending term is dependent on the types of atoms involved; angles involving delocalised single bonds and/or small rings have different parameters than, for

instance, linear bond angles. This expression matches Equation (3) from [27]. The general expression for angle bending is:

$$EA_{ijk} = 0.043844\frac{ka_{IJK}}{2}\Delta\theta_{ijk}^2 \cdot (1 + cb\Delta\theta_{ijk}) \tag{1.7}$$

As with the DREIDING and UFF forcefields, MMFF94 applies the following equation for linear and near-linear angles. This expression matches Equation (4) from [27].

$$EA'_{ijk} = 143.9325ka_{IJK}(1 + cos\theta_{ijk}) \tag{1.8}$$

where ka_{IJK} is the force constant, $\Delta\theta_{ijk} = \theta_{ijk} - \theta^0_{IJK}$ is the change in bond angle when comparing the actual bond (θ_{ijk}) to the reference bond angle (θ^0_{IJK}), and $cb = -0.007deg^{-1} = -0.4rad^{-1}$ is the cubic bend constant.

Stretch-Bend Interactions

The stretch-bend interactions apply to the relationship between the *i*-*j* and *k*-*j* stretches and the *i*-*j*-*k* bend, and include two force constants, kba_{IJK} and kba_{KJI}. This expression matches Equation (5) from [27]. This term is not needed for linear or near-linear angles.

$$EBA_{ijk} = 2.51210 \cdot (kba_{IJK}\Delta r_{ij} + kba_{KJI}\Delta r_{kj})\Delta\theta_{ijk} \tag{1.9}$$

Out-of-plane Bending at Tricoordinate Centres

This term describes the interaction at the Wilson angle, which is the angle between the *j*-*l* bond and the *i*-*j*-*k* plane. $koop_{IJK:L}$ is the out-of-plane force constant and $\chi_{ijk;l}$ is the Wilson angle. The other Wilson angles that share the same centre atom (j) also share the same force constant. This expression matches Equation (6) from [27].

$$EOOP_{ijk;l} = 0.043844\frac{koop_{IJK:L}}{2}\chi^2_{ijk;l} \tag{1.10}$$

Torsion Interactions

Torsion terms, where the torsion angle ϕ applies to *i-j-k-l* connections, have 3 main terms. This expression matches Equation (7) from [27].

$$ET_{ijkl} = \frac{1}{2}\left[V_1(1+cos\phi)+V_2(1-cos2\phi)+V_3(1+cos3\phi)\right] \tag{1.11}$$

V_1, V_2, and V_3 are constants that depend on the type of atoms in the torsion angle, according to the three bonded pairs: *i-j*, *j-k*, and *k-l*. These constants are also dependent on the connectivity to surrounding atoms (i.e. if the torsion angle is within a four-membered ring or a saturated five-membered ring).

Van der Waals Interactions

The van der Waals potential term is known as the Buffered-14-7 form from [23]. This expression matches Equation (8) from [27]. When expanded (and after removing the R^*_{IJ} buffering constants), the 14th and 7th powers denote the attractive and repulsive terms that comprise the potential well. This term is similar to the Lennard-Jones potential, which is represented by a 12-6 interaction. ε_{IJ} is the depth of the potential well, which depends on the atomic polarizabilities.

$$E_{vdW_{ij}} = \varepsilon_{IJ}\left(\frac{1.07R^*_{IJ}}{R_{ij}+0.07R^*_{IJ}}\right)^7\left(\frac{1.12R^{*7}_{IJ}}{R^7_{ij}+0.12R^{*7}_{IJ}}-2\right) \tag{1.12}$$

Electrostatic Interactions

The electrostatic potential energy term matches Equation (13) from [27]. It is based on a buffered version of Coulomb's law, and accounts for atomic charges, as described by:

$$EQ_{ij} = \frac{332.0716q_iq_j}{D(R_{ij}+\delta)^n} \tag{1.13}$$

The partial atomic charges, q_i and q_j are calculated by: $q_i = q^0_I + \sum \omega_{KI}$, where q^0_I is the formal charge of atom i, and ω_{KI} represent the set of contributions from each attached atom k to atom i based on bond polarity. D is the dielectric constant, and the n term is usually 1 (but can be 2 if using a distance-dependent dielectric constant).

R_{ij} is the internuclear distance between atoms i and j, and δ is the electrostatic buffering constant ($\sim$0.05Å).

The δ term from Equation 1.13 must be positive, as it prevents the attractive forces between oppositely charged atoms from ignoring the repulsive term from the van der Waals interactions described in Equation 1.12. If not for the δ term, oppositely charged atoms could have an infinite attractive potential (when $R_{ij} \sim 0$), which the finite term from the van der Waals repulsive forces cannot reasonably counteract.

In MMFF94, the 1,4 electrostatic interactions (i.e. those between atoms 1 and 4, where atoms 1 through 4 are bonded via 1-2-3-4) are scaled by a factor of 0.75, to reduce the error associated with conformational energies from polar compounds containing hydroxycarboxylic acids, esters, and dicarboxylic acids [31]. The 1,4 van der Waals interactions are not scaled.

1.2.3 Geometric Evaluation

One of the important features of a set of conformers is that they be as distinct as possible. A set of conformers is not helpful if they are all slight variations of the same structure, since no little additional information can be gained with regards to reactivity or stability. Therefore, the Root-Mean-Square Deviation (RMSD) is used to calculate the difference between two conformers by comparing their arrangements in 3D coordinate space.

The basic formula for RMSD is, where $\text{RMSD}_{ij} \in \mathbb{R} \mid \text{RMSD}_{ij} \geq 0$:

$$\text{RMSD}_{ij} = \sqrt{\frac{1}{n_a} \sum_{\alpha=1}^{n_a} ((x_{\alpha i} - x_{\alpha j})^2 + (y_{\alpha i} - y_{\alpha j})^2 + (z_{\alpha i} - z_{\alpha j})^2)} \tag{1.14}$$

The RMSD is a floating point value that represents the average distance between the atoms of two conformers, i and j. Both conformers should represent the same molecule. The larger the RMSD, the larger the difference in structure. If the RMSD is zero, then both structures are identical. In Equation 1.14, n_a is the number of atoms in the input molecule, $x_{\alpha i}$, $y_{\alpha i}$, and $z_{\alpha i}$ are the x, y, and z coordinates of the

α^{th} atom from conformer i, and $x_{\alpha j}$, $y_{\alpha j}$, and $z_{\alpha j}$ are the x, y, and z coordinates of the α^{th} atom from conformer j.

Conformers need to be optimally translated and rotated in space to make the RMSD calculation meaningful. Prior to calculation of the RMSD, the conformers should be centred and, to one conformation, a rotation matrix should be applied. These steps ensure that the smallest possible RMSD is calculated between two geometries.

One way to accomplish a minimal RMSD value is to use the Kabsch algorithm [10], which was originally applied to compare crystal structures. The Kabsch algorithm uses a singular value decomposition (SVD) (see Appendix, Section A.1.9) to minimise the RMSD by finding an optimal rotation matrix that can be applied to one of the two geometries.

There are three parts to the Kabsch algorithm:

1. apply the centroid translation (Equation 1.15) to both conformers,
2. calculate a covariance matrix, $\boldsymbol{C}$, and
3. calculate an optimal rotation matrix, $\boldsymbol{R}$, such that the RMSD between the two structures is minimised.

$$\text{centroid} = \left[\frac{1}{n_a} \sum_{\alpha=1}^{n_a} x_\alpha, \frac{1}{n_a} \sum_{\alpha=1}^{n_a} y_\alpha, \frac{1}{n_a} \sum_{\alpha=1}^{n_a} z_\alpha \right] \tag{1.15}$$

To centre a molecule, the centroid of each conformer is calculated and subtracted from each atomic coordinate. The procedure for this operation in `D` dimensions is as follows:

```
input coordinates in D dimensions
for dimension in range(D):
    calculate mean for dimension
    for coordinate in input:
        subtract mean from coordinate
        (at current dimension)
return new coordinates
```

The resulting structure will have an average coordinate of zero for each dimension, meaning it is centred about the origin.

The covariance matrix, $\boldsymbol{C}$, is a square matrix of size D (which is 3 for regular conformers). It represents the extent to which the atom coordinates are varying together when comparing two geometries. Let $\boldsymbol{P}$ represent the coordinates for the i^{th} conformer, and let $\boldsymbol{Q}$ represent the coordinates for the j^{th} conformer. Both $\boldsymbol{P}$ and $\boldsymbol{Q}$ are of the shape n_a by 3. To get the covariance matrix, the dot product is applied: $\boldsymbol{C} = \boldsymbol{P^T} \bullet \boldsymbol{Q}$ ($\boldsymbol{P^T}$ is $\boldsymbol{P}$ transpose).

To calculate the rotation matrix, $\boldsymbol{R}$, first SVD is used to break down the covariance matrix: $\boldsymbol{C} = \boldsymbol{U\Sigma V^T}$. Then, the determinant for each $\boldsymbol{U}$ and $\boldsymbol{V}$ is calculated. If the product of the two determinants is negative (i.e. $|\boldsymbol{U}| * |\boldsymbol{V}| < 0$), then a modification is made to the rotation matrix to preserve the right-handed coordinate system (see Appendix, Section A.1.1). The modification means that the last row of the S matrix and the last column of the U matrix are negated. Finally, the rotation matrix is given by: $\boldsymbol{R} = \boldsymbol{U} \bullet \boldsymbol{V}$.

The RMSD can help to identify geometrically similar conformations, but it does not account for conformations related by symmetry. For example, the two chair structures of cyclohexane in Figure 1.2 are identical, but the RMSD would not be zero. Since the point group for most molecules is C_1 (i.e. no symmetry elements are present), the RMSD is considered to be a suitable measure of geometric diversity for conformational analysis.

1.3 The Torsion Driven Approach

For a conformer search, the locations of interest from the PES are its minima - the places where the partial derivatives are zero with respect to all atoms. Using the set of second derivatives, it is possible to differentiate a minimum from a saddle point or an inflection point, for an added level of computational cost.

To explore the energy landscape (and thus find its minima), the dihedral angles (or torsion angles) are manipulated in what is known as a torsion driven approach.

Usually bond angles and bond lengths are kept constant during a conformer search. It is difficult, if not impossible, to maintain a molecule's connectivity if its bond angles or bond lengths are altered significantly. Such modifications also require energy input, which means making and breaking bonds (studying reactions and not conformations). The bond angles and lengths can be thought of as prior knowledge that is being used, or a restraint on the system, such that the resulting optimal geometries have the same connectivity as the intended structure. Optimising a set of 3D coordinates would return the best arrangement of the input atoms, regardless of the initial structure. Therefore, structural or constitutional isomers would be found, rather than conformers.

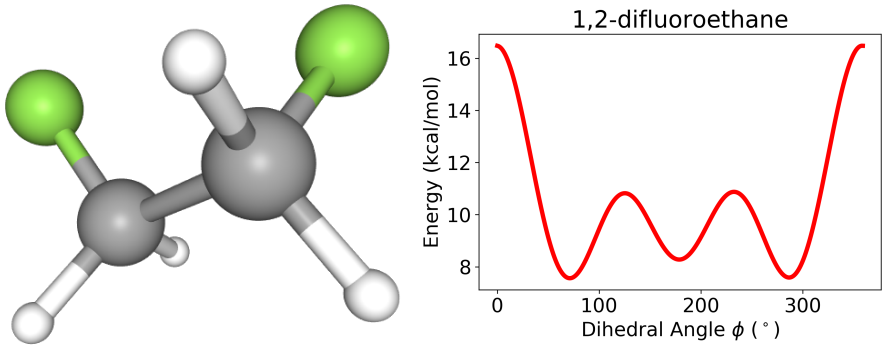
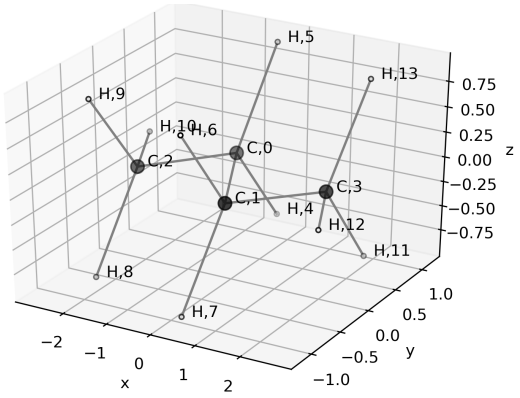


Figure 1.3: The left gives a figure of 1,2-difluoroethane (from *PubChem*[91]). The right shows an example of the energy landscape for 1,2-difluoroethane as a function of one torsion angle.

The value of a dihedral angle is the the angle between the plane formed by atoms a,b,c and the plane formed by atoms b,c,d, if the dihedral is described by a,b,c,d. After setting a molecule's dihedral angles, a local optimisation of the space - using numerical methods (for example steepest descent or conjugate gradients analysis [25]) - can be performed to get the best geometry from the potential well. Consider the 1,2-difluoroethane molecule, shown in Figure 1.3. If the bond lengths and angles are left unchanged, then a 2D energy landscape can be drawn by sampling the F-C-C-F dihedral angle. Changing the F-C-C-F dihedral would be equivalent to changing either of the two H-C-C-H dihedrals (or two H-C-C-F dihedrals).

Kaplan uses 0-based atom indices to represent the four atoms in a dihedral angle. The indices are taken directly from an .xyz file that stores the coordinates for the input molecule. For example, the labelled structure of butane is shown in Figure 1.4. The carbon backbone of this molecule is represented by the (2,0,1,3) dihedral angle. Excluding bonds to hydrogen atoms, butane has 3 rotatable bonds. The atom indices and the dihedral angle values for butane can be found in Table 1.1.



Dihedral Atoms	Angle °	Angle rad
(2, 0, 1, 3)	178.85	3.1215
(1, 0, 2, 10)	179.94	3.1406
(0, 1, 3, 13)	60.205	1.0508

Table 1.1: Left shows a 3D plot of the butane, as generated by *Kaplan*. Right is a table of dihedral angles for butane, with angle sizes in degrees and radians. To convert from radians to degrees, multiply by $180/\pi$.

Using the angles from Table 1.2, butane would appear as in Figure 1.4. The energies have the expected trend - the conformer with the fewest steric interactions ($\phi \simeq \pi$) has the lowest energy, and the conformer with the most steric interactions (blue, $\phi = 0.0$) has the highest energy. The red and green conformers ($\phi = \pi/2$ and $\phi = 3\pi/2$) have similar energies. The energies are not exactly the same, because the other two torsion angles (1,0,2,10) and (0,1,3,13) are still set to their original values (see Table 1.1). Furthermore, the C-H and C-C bond lengths and bond angles, even when related by symmetry, are not guaranteed to be equivalent, and depend on the algorithm or method used to generate the coordinates.

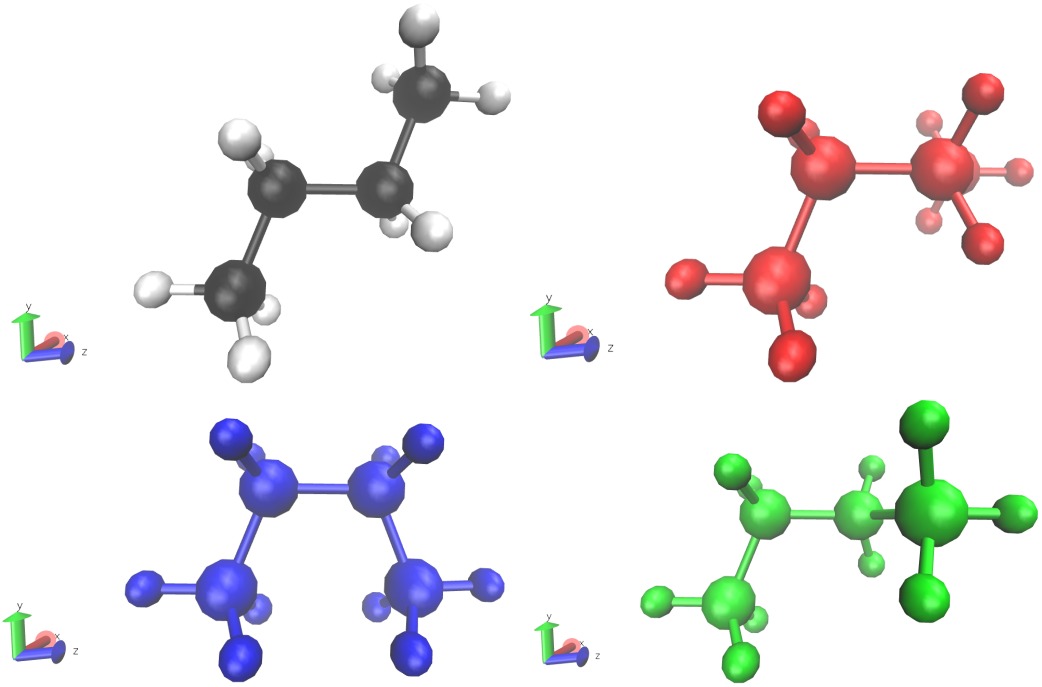


Figure 1.4: The images are (from top left to bottom right): (black and white) the original coordinates $\phi \simeq \pi$, (red) $\phi = \pi/2$, (blue) $\phi = 0.0$, and (green) $\phi = 3\pi/2$, where ϕ is the dihedral angle corresponding to the atom indices (2,0,1,3).

ϕ (°)	ϕ (rad)	**energy** (kcal/mol)
0.0	0.0	5.52746
90.0	$\pi/2$	-2.79188
*178.9	$\sim \pi$	-5.07247
270.0	$3\pi/2$	-2.81389

Table 1.2: Example MMFF94 forcefield energy values as a function of the (2,0,1,3) dihedral angle for butane. The dihedral from the original structure (given in the 3D plot) is indicated by a star (*).

A full plot of the energy profile for butane was generated, sampling every degree from 0 to 360, and it is shown in Figure 1.5. When the two largest groups are 180°

(π radians) away from one another, this is referred to as the *anti* configuration. When the two largest groups are 60° ($\pi/3$ rad) and 300° ($5\pi/3$ rad) away from one another, they are said to be in the *gauche* conformation. If the groups are overlapping in a Newman projection (see Table 1.4), then this conformation is called *eclipsed* – these conformations can be seen in the plot at approximately 120° and 240° ($2\pi/3$ and $4\pi/3$ rad).

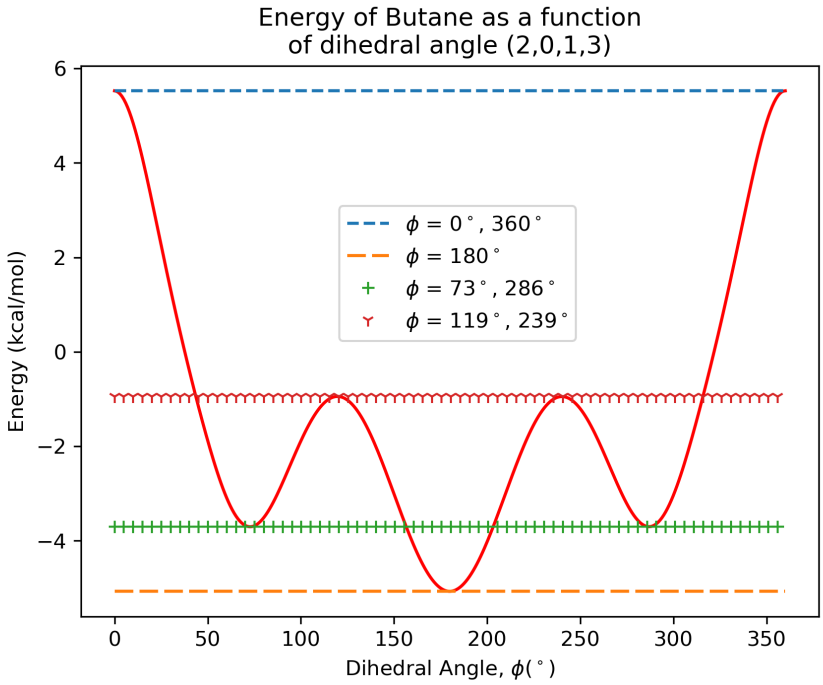


Figure 1.5: Energy profile for butane. The intercepts for the minima and maxima (given by flat lines) are provided in the legend. The other two minimum dihedral angles were set to π and $\pi/3$.

Conformational enantiomers are conformers that are mirror images of one another, but not superimposable. The difference in energy for each conformational enantiomer pair is provided in Table 1.3. If the geometries were exactly symmetrical, the difference in energy for each pair of enantiomers would be zero. Therefore, even when setting all of butane's dihedral angles, some molecular asymmetry remains.

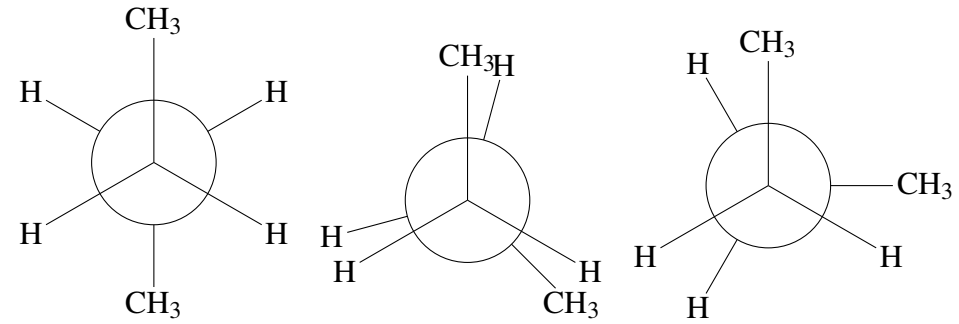
angle index	*angle* (rad)	**energy** (kcal/mol)
0	0.0	5.52814
1	$\pi/4$	-1.15478
2	$\pi/2$	-2.79127
3	$3\pi/4$	-1.55948
4	π	-5.07546
5	$5\pi/4$	-1.54435
6	$3\pi/2$	-2.81415
7	$7\pi/4$	-1.07220
8	2π	5.52814

angle indices	Δ Energy (kcal/mol)
0-8	0
1-7	-0.083
2-6	0.023
3-5	-0.015

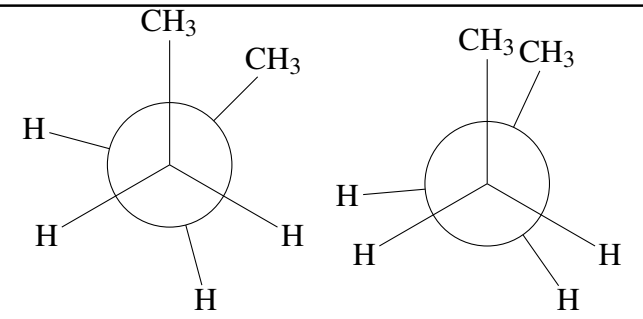
Table 1.3: The value for *angle* as a function of energy, where *angle* corresponds to the torsion angle of butane connecting atoms (2,0,1,3). Dihedral angles (2,0,1,3) and (0,1,3,13) were set to π and $\pi/3$ respectively. The energy difference between conformational enantiomer pairs is given in the table on the right.

To understand the trends in energy for the conformational enantiomer pairs, one can use Newman projections down the C0-C1 bond - shown in Table 1.4. The highest energy conformer is the $\phi = 0$ case, since the methyl groups are directly eclipsed. The next highest energy conformer is $\phi = \pi/4$, followed closely by $\phi = 3\pi/4$. The second lowest energy is at $\phi = \pi/2$, and the lowest energy is at $\phi = \pi$, since the largest groups are *anti* to one another, and the hydrogens are placed optimally at $\pi/3$ rad apart.

As mentioned in the previous section, one way to evaluate conformers is to calculate the root-mean-square deviation (RMSD). First, the structures must be centred about the origin by applying a centroid translation. The centred structures of butane, with the values of *angle* given in Table 1.3, are shown overlapped in Figure 1.6. The carbon backbones for non-equivalent geometries no longer overlap, but the distance between each dihedral set is approximately equal. Only 8 distinct ge-



180° | π rad	135° || $\frac{3\pi}{4}$ rad	90° || $\frac{\pi}{2}$ rad



45° || $\frac{\pi}{4}$ rad	*0° || 0 rad

Table 1.4: Newman projections showing different values of ϕ for butane. When the larger groups are eclipsed, then the energy of the system increases (becomes less negative). Note: * the actual angle here is slightly bigger than zero so the direct overlap of groups does not obscure half of the molecule.

ometries are visible, since the 0 and 8 cases directly overlap. The RMSD between the geometries produced by setting $angle = 0.0$ and $angle = 2\pi$ is essentially zero, since these structures are identical.

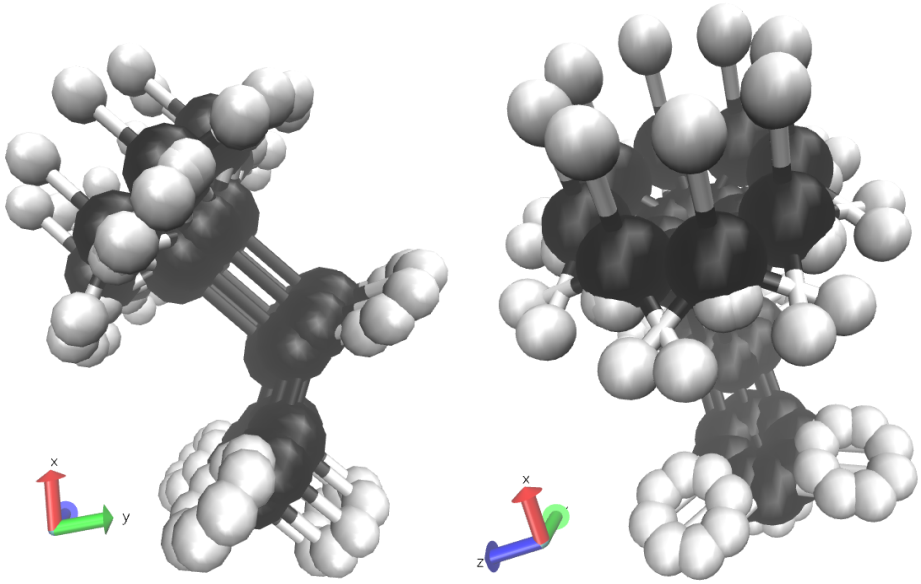


Figure 1.6: Overlapped structures of butane, representing centred structures with 9 torsion angles for the (2,0,1,3) dihedral angle. 2 structures, $angle$ at 0 and 2π, are identical.

The RMSD was also compared between geometries that should be similar by symmetry - the conformational enantiomer pairs. In most cases, the RMSD is calculated based on fixed atom ordering (the atom indices matter). The RMSD can be calculated instead with reordering via three methods: distance (crude, quick), brute-force search (exact, yet costly), and the Hungarian algorithm to optimise the assignments of atoms in one molecule to the other molecule. The RMSD values with reordering for each pair of geometries are shown in Table 1.5 alongside regular RMSD calculations.

The trend in RMSD is as expected; enantiomer sets 1-7 and 3-5 should have the same RMSD, since they both differ in the placement of the methyl group by $\pi/2$ rad (90°). Conformer set 2 and 6 should have the highest RMSD, since the methyl

angle	RMSD	atom reordering methods		
indices	*Kabsch*	*brute*	*Hungarian*	*distance*
0-8	1.14E-15	1.14E-15	1.14E-15	1.14E-15
1-7	0.958	1.291	0.527	1.561
2-6	1.662	1.452	0.553	1.671
3-5	0.949	1.576	0.522	1.773

Table 1.5: Results for comparing RMSD for each pair of conformational enantiomers, as well as the identical set of geometries (0 and 8).

group is rotated π rad (180°). Interestingly, although the brute-force method took much longer to compute than the Hungarian method for RMSD, the brute-force method found a larger RMSD than the Hungarian method for all pairs (except in the identical case, where all RMSD values were essentially zero)[2]. Of the three atom reordering methods, only the Hungarian approach was consistent with the original trend in RMSD values. RMSD calculation in *Kaplan* is performed using the regular procedure given Section 1.2.3 (without any reordering), since the atom indices are kept in the same order throughout the conformational search. Such guarantees cannot be made for other methods, and so in some cases it may be necessary to compare the RMSD and RMSD+reordering results. The smaller of the two values should be considered to be the "true" RMSD between two conformers.

1.4 Conformer Searching Methods

In exploring the energy landscape, it is easy to "get stuck" in a potential well (local minima). Therefore, a combination of exploration (finding new potential wells) and exploitation (descending into potential wells) is required. Some ways in which conformers can be generated include:

[2]The *rmsd* program [106] that was used to calculate these values is probably incorrectly implementing the brute-force algorithm.

- exhaustive/brute-force - explore all possible conformations by a combinatorial algorithm.
- systematic - starting with a given conformation, follow a sequence of steps to produce a "better" geometry. Usually this approach involves sampling the space of conformations according to a grid (e.g. of regularly-spaced torsion angles).
- stochastic - randomly generate conformations and choose the best one from those generated.
- domain-knowledge - use known template structures (i.e. from a crystal structure database) for a molecules sub-domain or substructures and piece these together (similar to homology modelling in protein structure prediction).
- directed-search - starting with a subset of conformers, use information from the current subset to explore nearby structures.

The main method that is used to find conformers in this work is torsion-based evolutionary computation, which is discussed in detail in Section 1.5. Using evolution to find conformers can be thought of as a combination of stochastic and directed-search. A few other methods are discussed in Sections 1.4.1 and 1.4.2, followed by a review of some free software packages for conformer searching in Chapter 2.

1.4.1 Meta-Dynamics

In a paper by Grimme [103], semi-empirical tight-binding quantum chemical methods were combined with root-mean-square-deviation (RMSD)-based meta-dynamics, not only for the purpose of finding conformers, but also as a way to explore products from reactions and find reaction pathways. One of the motivations for his work was to perform reasonable conformer searches in less than a day of computational time on a laptop computer for sizable organic molecules (Grimme defines large, organic

molecules as being between 50 and 176 atoms). In this method, the energy for the system is broken down into three components:

$$E_{tot} = E_{tot}^{el} + E_{bias}^{RMSD} + E_{bias}^{wall} \tag{1.16}$$

The first term in Equation 1.16 represents the total electronic tight-binding quantum chemistry energy, the second term represents the biasing RMSD potential, and the last term is an optional reactor wall cavitation potential. The E_{tol}^{el} term also contains potentials that can fix internal coordinates and enforce other molecular constraints. The E_{bias}^{wall} term is used for reaction space exploration, as part of a nanoreactor, and not for conformer searching. The RMSD for this energy is calculated between a reference structure and the current structure in the simulation.

1.4.2 Distance Geometry Methods

A distance geometry method uses a model to generate a large ensemble of potential geometries, which are then energetically evaluated to ensure correctness [36]. The model is a set of geometric constraints, where it is assumed that such constraints can fully represent the geometric space of a molecule [87]. The constraints can be the set of minimum and maximum possible distances between each pair of atoms in a molecule. Then, the problem becomes finding a way to convert these constraints into sets of Cartesian coordinates for the atoms.

The ETKDG combines ETDG (Experimental-Torsion Distance Geometry) with a set of K terms, representing basic chemical knowledge, such as aromatic rings should be flat, and triple bonds should be linear [87]. ETDG methods use distributions of torsion angles from known crystal structures to limit the conformer search space. For example, the CONFECT (Conformations from an Expert Collection of Torsion Patterns) conformer generator [77] builds conformers from frequency distributions of experimental torsion angles. The molecule is split into sections that are assigned a torsion angle group.

1.5 Evolutionary Computation

Evolutionary computation is a popular tool that can be used to find conformers. In essence, a directed, population-based, stochastic search is implemented, whereby a reasonable solution can be found without having to perform an exhaustive search. An evolutionary algorithm (EA) consists of three main parts:

- Representation - solution to the problem.
- Evaluation - how good is a given solution?
- Regeneration - mutations of old solutions.

The endpoint of an EA can be goal-based; for example, once a certain number of conformers with energies above a given threshold are found, stop the program. The endpoint can also be arbitrary, as in the case of a set number of iterations. In this text, an iteration of the EA is called a mating event, or `mev` for short. For example, *Kaplan* has two termination conditions: (1) stop after a set number of `mevs`, or (2) stop if, after a set number of `mevs`, the best solution does not improve.

In most cases, the important (yet often competing) considerations for an evolutionary algorithm are exploration of the solution space (sometimes called *fitness landscape*) and the exploitation of hills (for a maximisation problem) or wells (for a minimisation problem) within said space. A balance of exploitation and exploration must be achieved to procure good results. If there is not enough exploitation, solutions may be weak and suboptimal. Conversely, if there is too much exploitation, the solutions gathered from the EA may all exist within the same well or on the same hill in the fitness landscape, thereby missing better solutions from other wells and hills (not to mention misrepresenting the solution space).

1.5.1 Representation

In this case, the solution should be a conformer, or set of conformers. Each solution, in this text referred to as a `pmem` (or population member), has a representation that

is chosen by whoever writes the algorithm. For example, the representation could be as simple as a list of numbers, where each item in the list is a dihedral angle for the molecule of interest. It could also be a string that maps to a set of instructions explaining how to construct a solution, or even multiple strings or arrays.

An important design decision is the size and arrangement of the population - the set of `pmems`. The diversity of the initial population can determine the variety of solutions to be explored during the course of evolution. The size of the initial population is also important; using a large population permits more diversity in the beginning, but can prevent exploration of useful parameter space if the algorithm disregards seemingly poor solutions with good potential (i.e. are in need of fine-tuning). Some populations are fixed in size, and others can be dynamic, with a growing and shrinking number of `pmems`.

1.5.2 Evaluation

This step requires the investigator to know the properties of a good solution and how these properties might be combined. For example, conformers should have relatively low energy (such that they are stable) and high geometric diversity (so they are not all small variations of the same conformer). For example, the RMSD can be used to filter-out similar geometries. In an EA, the quality of a solution is called its *fitness*, such that a `pmem` of high fitness should be better than a `pmem` of low fitness [3].

1.5.3 Regeneration

Using the knowledge from previous results, how can the algorithm direct itself towards a better solution? This section of the text will explain an overview of regeneration, which consists of selection and alteration of existing solutions for the purpose of producing new, better solutions.

[3] The directionality/signage of high and low are dependent on the type of optimisation being performed - that of maximisation or minimisation.

One way of regenerating pmems is through a *tournament*. A tournament involves a selection process, producing the new solutions from a chosen subset of said selection, and returning the new solutions to the population. In some cases, a pmem with poor fitness may be one update away from an optimal solution. An example of this could be a conformer whose structure is well defined, except for one dihedral angle that permits atom overlap (resulting in extremely poor energy and thus poor fitness). How does the algorithm ensure that such a pmem has a chance to find its "best self"? The easy answer here is to allow random selection during a mev, rather than exclusively choosing the best pmems.

Tournament selection usually has a parameter called *tournament size*, which indicates how many pmems to choose from the population. Other selection criteria can include location; for instance, if the population is stored as a list, the tournament could be conducted on a contiguous segment of that list, or select pmems randomly. Some EAs also update the structure of the population prior to regeneration, such that pmems with higher fitness have a higher chance of being selected.

Once the members of the tournament have been selected, the pmems that will be used to produce new solutions must then be identified - usually these pmems are referred to as *parents* and the resulting pmems are called their *children* (also offspring). Again, parents can be chosen randomly or, more commonly, by some measure of fitness. The number of parents can be anything from 1 to the number of tournament participants.

The generation of children is done by combining information stored by the parent(s) in the hope that the best combination of information can be found. Going back to Section 1.2, the parents contain domain knowledge that can be updated and passed on to subsequent generations of pmems. If a tournament selects two parents, where one parent has a good conformation for the left half of the molecule and the other parent has a good conformation for the right, when these parents produce offspring, the best outcome is that the child has the better half of both parents (resulting in a good overall conformation). In the worst-case scenario, the child would have the worst half of both parents, thus producing a low fitness solution. This sort of change is called a *crossover* event. There are multiple types of crossover, including

single-point, multi-point, and uniform.

Using only crossover means that there is no potential for the EA to produce solutions with potentially new components. To increase algorithmic exploration, mutation operators are also applied to child `pmems`. An example would be to change the value of a dihedral angle found in a child `pmem`, either to a completely new value or to alter it by some amount. The amount of mutation is usually an input parameter, since too much mutation means favouring exploration, whereas too little favours exploitation. For instance, if the amount of mutation is high, the qualities that made the parent's fitness high may be sacrificed to random changes. If the mutation rate is too low, then the algorithm may become trapped in a well or on a hill.

1.5.4 Genetic Algorithms

Genetic algorithms (GA) are a subset of evolutionary algorithms (EA). It should be mentioned here that, in some of the literature, the use of the term genetic algorithm is a misnomer. GAs use a set of binary strings as their representation [18]; these strings encode solutions to the problem at hand. Mutation of a binary string consists of bit flips, and future solutions are generated by applying crossover to current bit string solutions [24]. When reading the implementation details of some GAs, it becomes clear that the method actually belongs to the more general class of evolutionary algorithms.

1.5.5 Application to Conformer Searching

This section will review the literature on previous applications of EAs to conduct conformer searches. The conformer searching packages *Balloon* and *Openbabel*, which also use EAs, are explained in detail in Sections 2.3.2 and 2.3.3, respectively.

In the case of phenyl glycosides, found in flavonoids, tannins, lignans, and others, a conformational search is necessary to determine the orientation of the aryl group with respect to the sugar portion of the molecule. Wałjko et al. used an

EA to help map the potential energy surface of 6 O-glycosides by manipulating the O1-C1'-O2-C1 and C1'-O2-C1-C2 torsion angles [111]. Specifically, they implemented a Genetic Algorithm-Assisted Grid Search method, where the energy was evaluated with the MMFF94 forcefield (see Section 1.2.2). They found 2-3 minima for each compound analysed, and then used DFT (specifically, the B3LYP hybrid functional with the 6-31+G(d,p) basis set) to further optimise the conformers. The dielectric constant was also altered in the forcefield calculations, but any differences this caused in the resulting geometries disappeared when DFT analysis was performed. The evolutionary part of the algorithm, which was also used in a few other papers [84][90], is described here [44].

As presented in the paper by Adriana Supady [89], the program Fafoom (Flexible algorithm for optimization of molecules) [88] allows the user to combine quantum chemistry energy evaluations (via NWChem [65]) or forcefield calculations (via *RDKit* [80]) with an EA-based search for conformers. In this implementation, the algorithm keeps a "blacklist" of evaluated structures, such that the same energy calculation does not occur twice. A structure is considered *unique* if it has a significantly different RMSD (discounting hydrogens) when compared to all structures in the blacklist. This program was applied to seven dipeptide structures and mycophenolic acid.

1.5.6 Choosing Input Parameters

Default options are difficult to choose for an EA, so much so that meta-algorithms have been developed for the purpose of finding the most efficient input parameters [51][15][66]. Parameter-less algorithms have also been designed; in their paper [41], Harik and Lobo argue that users desire black-box, push-button software that provides the correct answer to their problem, without having to understand the input parameters and their optimisation. The *Balloon* paper [52] did not include a parameter optimisation; the authors stated that such experiments would be worthwhile (as future work).

According to the paper by Brain and Addicoat [66], the initial parameter opti-

misation for a GA used to search molecular conformer space was most impacted (in terms of efficiency) by population size and mutation rate. A smaller population size (25), as tested on 5 organic molecules each with ∼8 rotatable bonds, was found to be better than a larger population size (125). The definition for efficiency used was: "the mean number of conformer evaluations required to locate [the] *a priori* known global minimum conformer of the molecule" [66]. A set of input parameters for a GA was considered to be "unreliable" if the search failed to find the global minimum; a low mutation rate would sometimes result in a suboptimal solution being found. However, the most important parameters may change if a set of conformers are considered (instead of the global minimum), especially since the criteria for fitness includes both geometric diversity and energetic evaluation.

Each molecule may require a different set of inputs to most efficiently use evolution for conformer searching; however, the analysis of EA-based conformer searching methods rarely includes an exploration of the inputs [70]. Instead, the ability of the package to reproduce crystal structure data for a database of compounds (using the default inputs) is assessed. The stochastic nature of an EA also means that multiple runs of the algorithm may be necessary to find the best conformers, especially if the initial population is concentrated in a small area of the search space.

Kaplan has some default inputs - for example, the level of mutation scales with the number of dihedral angles and conformers (per `pmem`) being optimised. These defaults apply to the `swap`, `crossover`, and `mutate` operators. The number of possible dihedral angles to search is also limited to a discrete set such that the `mutate` operator is more effective at searching the available space.

1.6 Why *Kaplan*?

This section discusses the ways in which the implementation in *Kaplan* is different from other algorithms and the augmentation that *Kaplan* will provide to the current set of conformer searching programs.

Most conformer searching programs fall into one of two main categories: those

that are a part of a larger cheminformatics toolkit (examples: *RDKit* and *Openbabel*), and those that are specifically designed to produce conformer geometries (examples: *Frog2* and *Balloon*). *Kaplan* aims to be somewhere in between, by providing not only dedicated conformer searching, but also ways to analyse and interact with the results.

Kaplan is transparent about how it generates conformers. The dihedral angles being manipulated, as well as the other inputs, are written to the output directory. By recording the inputs, experiment reproducibility is greatly improved (i.e. for cases where analyses are performed on years-old data or data collected by another researcher). If there are no dihedral angles to manipulate, *Kaplan* throws an input error, whereas some other programs just silently output the same input structure. *Kaplan* allows the user to specify which dihedrals to use and the values dihedrals can have. Therefore, as was the case for the O-glycosides conformers [111], *Kaplan* can perform a conformer search while keeping known sections of a molecule fixed. No other tested program allows the user to select their dihedral angles of interest.

Due to the "black box" nature of most conformer searching tools, it is hard to diagnose the problem when a conformer search performs poorly. Although it is not fast, the code is in a singular programming language – *Python*, which is rapidly growing in use [96] and easy to learn [43]. A common backend for large cheminformatics software packages is *C++*, but, when wrapped by another language, the function names and syntax can change quite drastically. If the tool breaks or performs sub-optimally, the lack of code transparency means that it takes much longer to debug.

Since *Kaplan* is based on an EA, it works with a population of potential conformers. The author was unable to find an EA-based tool that allowed the user to interact with the population; without the population, the geometries tested and the size and shape of the energy landscape are concealed from the user. For example, if the population contains many duplicates of the same conformer, then a higher mutation rate may improve the results. Furthermore, if the conformer diversity is high and the variation in fitness is low, then the number of conformers to search for may need to be increased. It may also be that the best solutions are contained

within one or more of the population members (`pmems`). Therefore, *Kaplan* returns the population (in the form of a *Python* pickle file) as well as the best result after the algorithm terminates. The user can also analyse the stats file to see whether the EA is stagnating or rapidly improving, which can indicate for how long the program should be run.

Kaplan's default behaviour is to accept a name as input, no coordinate files required. *Kaplan* also handles directory structure such that each conformer search (including its output) is clearly separated and labelled. These features make it easy for a user to generate and analyse a database of conformers.

Kaplan also has the ability to accept conformer data from external programs (as, for example, a starting population). A basic workflow would be to first use *Kaplan* to identify the dihedral angles (and corresponding atom indices) for the conformer (or conformers) of interest. Then, a `pmem` can be constructed with these dihedrals and added to the population prior to running the EA. *Kaplan* can also be re-run using old populations as a starting input (for example, in the case where more evolution is required, or if the user wishes to explore different input parameters). For the other tools tested, the user can only input one structure with which to explore the conformer space.

Algorithmically, *Kaplan*'s approach is different from other EA-based conformer searching packages, because it uses a `ring`-based population. Furthermore, its `pmems` contain multiple conformers rather than just one, which allows for fewer pairwise RMSD calculations while still maintaining distinct geometries. Unlike other methods, `pmems` in *Kaplan* cannot necessarily reproduce with one another – they must be close in proximity. Therefore, *Kaplan* imposes a geography that promotes the growth of small communities of solutions. The `ring` can contain a more diverse set of solutions than might be present in one large population where all `pmems` can mate with one another. During evolution, the `ring`'s population grows to a maximum size, and the `pmems` can be killed using a variety of extinction operators. These operators are designed for situations where the algorithm converges early or the population fails to diversify (i.e. gets stuck in one area of the search space).

Even if the user does not run a *Kaplan* conformer search, they can still use the `ring` to store a set of conformers for a molecule. Only the `ring` and `inputs` objects would be needed to fully reproduce all of the conformer geometries, without having to store hundreds or more output files.

In conclusion, *Kaplan* is a toolkit for conformer searching. The software was designed using a document-driven approach [47], and adheres to code quality measures, such as linting, testing, version control, and continuous integration. Due to its modularity, each piece of the software can be accessed independently. With *Kaplan*, the user can generate and refine conformers, store experiments reproducibly, evaluate and optimise geometries using both quantum and forcefield methods, and view the conformer searching process as opposed to simply the output structures.

1.7 Document Organisation

The rest of the thesis proceeds as follows. Chapter 2 discusses the software tools used in this work, including example output for the amino acids from freely available conformer searching packages: *Openbabel*, *Confab*, *RDKit*, *Frog2*, *Balloon*, and *Kaplan*. Computer and script-friendly molecule identifiers and visualisation techniques are also provided. In Chapter 3, the *Kaplan* conformer searching package is presented in technical detail, with examples on how to perform a conformer search with *Kaplan* and use its individual modules. This chapter provides the reader with sufficient information as to recreate concepts from *Kaplan*. Chapter 4 compares the results of *Kaplan* conformer search with geometries obtained via a detailed, computational-based search on the 20 methyl-capped amino acid structures. Therefore, Chapter 2 is intended to compare the approach and usage of various conformer searching packages, whereas Chapter 4 compares the output of a *Kaplan* conformer search to a set of known conformers. Chapter 5 summarises the observations made from the other chapters and suggests possible improvements to *Kaplan*. The Appendix contains miscellaneous technical information and scripts for conformer searching.

Chapter 2

Current Conformer Searching Techniques

The basic purpose of this chapter is to provide support for researchers, computational or otherwise, who might want to perform a conformer search. This chapter will identify some common free software packages for conformer searching, as well as some tools that can be used to generate, evaluate, and visualise molecular geometries. For a review of commercial conformer searching programs, please see [93].

Section 2.1 provides a table that contains all of the version numbers for all software packages and modules that were used for this thesis. The reader can easily determine whether the conformer searching package has a new version by comparing Table 2.1 to the latest version online. Following Section 2.1 (Software Tools), Section 2.2 discusses the optimisation of the amino acid structures from *PubChem*, which were used as test inputs to the conformer searching packages discussed in Section 2.3. Each subsection introduces a program and how it works. Some tables are given in Section 2.3 to compare the qualities of the programs. The results of the conformer searches are summarised in the last section.

2.1 Software Tools

This section will briefly cover some of the tools that were used to generate data found in the later sections. Table 2.1 provides a list of programs and their respective version numbers that were used in this research, either as as a part of the *Kaplan* conformer searching package, or as analysis tools. The main use cases for each software package are given in this chapter. More details about *GOpt*, *NumPy*, and *Kaplan* can be found in Sections 3.5.1, A.1.7, and 3, respectively.

2.1.1 Structure Determination

PubChem [105] is an online database hosted by the National Institutes of Health (NIH). Using its PUG (Power User Gateway)-REST (Representational State Transfer) API (Application Programming Interface) [97], the database can be programmatically queried for information on specific molecules, including 2D and 3D coordinates [75][91], properties (such as charge, molecular formula, molar mass, etc.), and identifiers (SMILES strings, InChI, and synonyms). Each molecule in the *PubChem* database also has an associated integer called a CID (Compound Identifier).

SMILES stands for Simplified Molecular-Input Line-Entry System [17][19][21]. It is a method, based on graph theory, that can be used to specify a molecule's connectivity and (in the case of isometric SMILES) stereochemistry. A community-based specification for SMILES is hosted at `http://opensmiles.org/`, and the original specification can be found at `https://www.daylight.com/dayhtml/doc/theory/theory.smiles.html`. A simple example of a SMILES string would be `C1CCCCC1`, which represents the cyclohexane molecule (hydrogens are implied). Multiple SMILES strings can be made for the same molecule, depending on which atom is used as the starting index. Canonical SMILES (which includes rules for numbering atoms in a molecule as to make the SMILES unique for each molecule) are not always consistent, since conflicting algorithms to specify these rules have been written by commercial and open-source software packages.

Package Name	Version	Relevant Links/References
*Avogadro**	1.2.0	[98][72]
Balloon	1.6.8	[110][52][64]
Confab	Used from *Openbabel*	[67]
*Frog2**	2.14	[101][74][63]
GOpt	dev - db698db	private repository
Kaplan	dev	[102]
Matplotlib	3.1.0	[50]
NumPy	1.16.4	[99]
Openbabel	2.4.1	[107][55][68]
Python	3.7.3	python.org
Psi4	1.3.2	[100][95]
PubChem	N/A	[75][91][97][105]
PubChemPy	1.0.4	[109]
RDKit	2019.03.3	[87][108]
rmsd	1.3.2	[106]
Vetee	dev - f92e5ef	private repository
VMD	1.9.3	[32]

Table 2.1: Software packages used in this thesis. The dev version means that the package is in development, and the version number is the latest git commit hash. *With the exception of *Avogadro* (which was run on Windows 10) and *Frog2* (run via webapp), all software was run on Ubuntu 19.04.

A subset of the SMILES specification is SMARTS (SMILES arbitrary target specification). The original specification can be found at https://www.daylight.com/dayhtml/doc/theory/theory.smarts.html. SMARTS are substructure specifications that can be used to query structures for similar functional groups. Furthermore, molecules can be given fingerprints based on how many SMARTS they contain (e.g. from a reference database of SMARTS). Examples of SMARTS can be found in Figure 2.1.

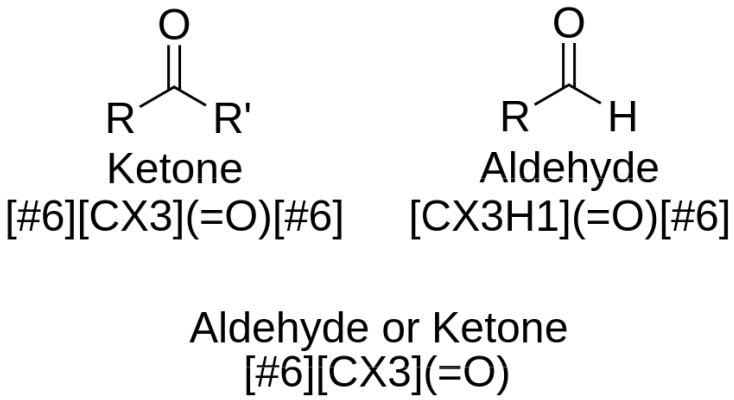


Figure 2.1: Examples of SMARTS, with their matching line structure drawings where R and R' represent aliphatic groups.

InChI [86] (IUPAC International Chemical Identifier, where IUPAC stands for the International Union of Pure and Applied Chemistry) refers to a line notation that can be used to construct a molecule following a structure-based approach, similar to SMILES. InChI was devised to be unique for each substance, with the requirement that the software be non-proprietary and open-source (to prevent other, competing implementations from being written). InChI can be converted to a hash (called an InChI key) that is also unique to the molecule. An example of an InChI for cyclohexane is: `InChI=1S/C6H12/c1-2-4-6-5-3-1/h1-6H2`, and the corresponding InChI key is `XDTMQSROBMDMFD-UHFFFAOYSA-N`.

A *Python* wrapper for PUG-REST is provided in the *PubChemPy* [109] software package. *Vetee* is a database generation package (work in progress) based on work by Kumru Dikmenli and the author. It combines features from *Openbabel*

and *PubChemPy* to organise relevant chemical information needed to setup and run quantum chemistry jobs. It is used by *Kaplan* to get initial geometries for conformer searching (from the aforementioned identifiers) and to determine the overall molecular charge and multiplicity.

2.1.2 Structure & Data Visualisation

PubChem has a 3D conformer generation tool [75], which also allows the user to view and download a single 3D depiction of a molecule. Some examples are shown in the previous chapter (see Figures 1.1 and 1.3). *VMD* (Visual Molecular Dynamics) [32][38] can be used to view a variety of chemical structure formats, including the xyz files that are generated by *Kaplan*. Examples can be found throughout this document (see Figures 1.2, 2.6, and 4.12). *Matplotlib* [50], a *Python* library, was used to make most of the plots in this thesis, and is also interfaced by *Kaplan* to produce automated plots. *Matplotlib* boxplots are used to compare the energy and RMSD distributions for the amino acid conformers. See Appendix, Section A.3, for details on how the boxplots are constructed in *Matplotlib*, as well as an example plot (Figure A.3).

Avogadro [98][72] is a free, open-source, GUI-based chemical toolkit that can be used to view 3D molecular structures. The author cannot recommend the Windows version of *Avogadro* for conformer searching; the MMFF94 energies it returned did not agree with those calculated by *RDKit* or *Openbabel* (whose energies agree exactly despite having separate implementations of the forcefield). For example, *Avogadro* MMFF94 energies for the amino acids did not follow the same trend, were off by multiple kcal/mol, and in some cases had the opposite sign, when compared with *Openbabel*/*RDKit* energies. Furthermore, conformer searching with *Avogadro* does not return more than one structure (all the user is able to do is download what is currently on-screen). Based on the types of conformer searching options available and the default values for the parameters, *Openbabel* is being used as a backend to do the conformer searching. Therefore, the author recommends *Openbabel* for conformer searching, until *Avogadro*'s issues are fixed.

2.1.3 RMSD Calculation

RMSD calculation is achieved in most cases using the *Python rmsd* package [106]. It has a command-line tool option (via `calculate_rmsd`) and an API to use in scripting. By default, the Kabsch algorithm [10] is used to solve for the optimal rotation matrix, but the Quaternion algorithm [22][48] is also available. The package supports centroid translation, and atom reordering, which accommodates input files describing the same molecule with potentially mismatched atom indices. The atom reordering is implemented using the Hungarian algorithm by default [3], but brute-force and distance methods are also available.

Openbabel provides an RMSD calculation function in its `obutil.cpp` source file, as well as a SMARTS-alignment-based superimposition (via Quaternion algorithm) executable, `obfit`. *RDKit* also has the ability to calculate the RMSD between molecules - for an example, see the **RMSD Calculation between N molecules** Section of the RDKit Cookbook https://rdkit.readthedocs.io/en/latest/Cookbook.html.

2.1.4 Energy Evaluation

Energy evaluation can be accomplished through *Openbabel* for the following forcefields: `gaff`, `ghemical`, `mmff94`, `mmff94s`, and `uff`. *Openbabel* makes it easy to calculate the energy of a molecule by providing the `obenergy` executable, which can be run from the terminal and takes a filename as input. *RDKit* allows for forcefield energy calculations using `UFF`, `MMFF94`, and `MMFF94s` [80]. For quantum chemical energy evaluations, the *Psi4* package can be used [100][95]. The latter two packages must be scripted to be used.

2.2 Initial Structure Optimisation

This section describes how *Openbabel* was used to improve *PubChem* structures for the 20 amino acids. Visual and numerical descriptions will be presented, showing how a geometry changes following a local forcefield optimisation. During a conformer search, *Kaplan* implements local forcefield optimisation to prevent atom clashes, optimise final conformations, and more efficiently explore the energy landscape. The number of iterations needed to converge the energy was also used to parametrise some *Kaplan* inputs. Since *PubChem* is commonly used to generate initial coordinates for *Kaplan* conformer searches, the amount of optimisation needed to reach the local minima for the *PubChem* structures is useful information.

The initial structures for the amino acids were downloaded from the *PubChem* website in sdf format. A *Python* script was written to analyse the structures, using the `localopt` function from *Pybel* [55], which is a *Python* wrapper for *Openbabel*. This function uses steepest descent to optimise a given input structure, for a maximum number of iterations. The MMFF94 forcefield was used to score the energy.

During the experiments, the author independently discovered a bug (see Issue #1366) that had also been posted on the *Openbabel* Github repository. The outcome of this bug was that the convergence criteria, as specified by an energy difference, was ignored; therefore, the two methods for optimisation, steepest descent and conjugate gradients, would complete the maximum iterations regardless of whether the energy difference was within the input tolerance.

To fix this issue, the author and Justin Schonfeld debugged the C++ code in `forcefield.cpp` and found that the gradients were not being properly updated between iterations, thereby making it impossible for the function to converge before reaching the maximum number of iterations. The appropriate changes were made and the author made a pull request, which was accepted (see PR #2017). Therefore, future versions of *Openbabel* will not have these convergence issues.

As a workaround to the bug, the author modified the analysis script to check the energy every 100 iterations. If the energy did not change within 10 such checks,

then the structure was considered to be converged. The energy was plotted as a function of number of iterations for each amino acid; examples of such plots can be found in Figure 2.2. The change in energy that each structure underwent as a result of the optimisation is given in Table 2.2.

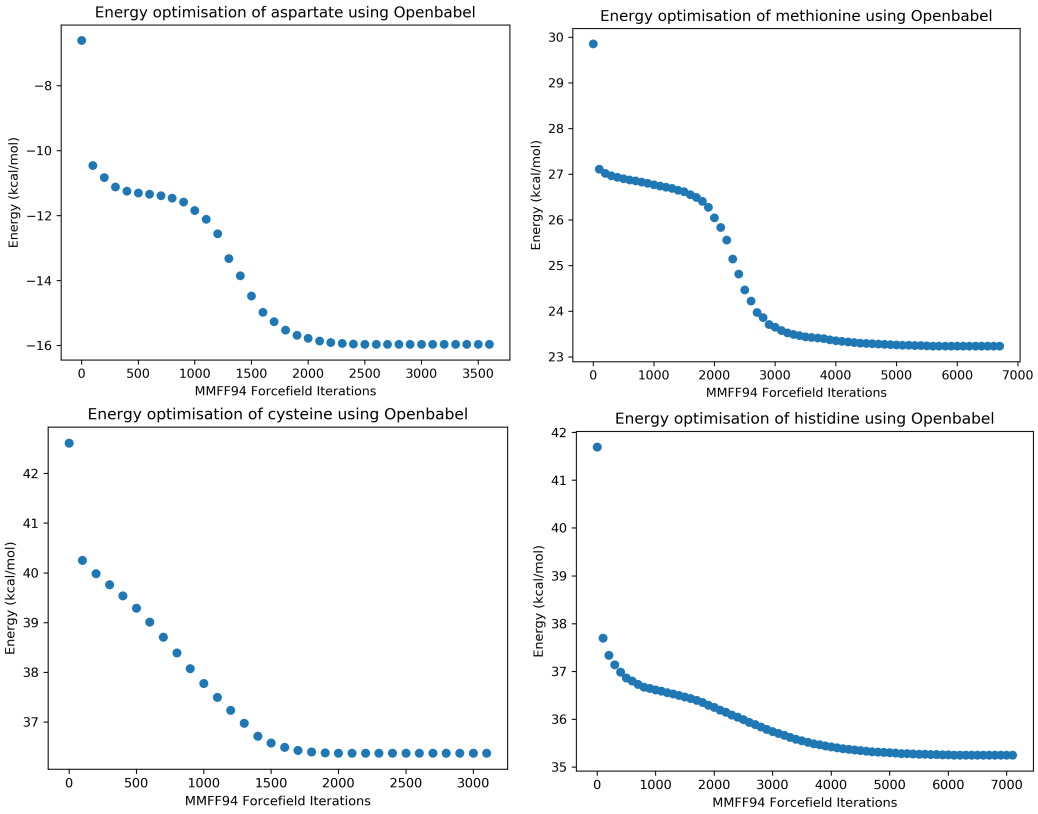


Figure 2.2: Energy as a function of steepest descent iterations for the optimisation of four amino acids. The left two plots (aspartate and cysteine) underwent a large structural change within fewer iterations than the right two plots (methionine and histidine).

From the plots in Figure 2.2, inflection points can be seen for some of the molecules. Inflection points make it harder for some numerical optimisers to find the local minimum, since they must somehow pass by a location where the local derivatives are zero (or close to zero). For example, using conjugate gradients to op-

timise aspartate (with the debugged *Openbabel* code) returns a minimum energy of -11.27 kcal/mol, which is higher in energy than the local minimum of -15.96 kcal/-mol shown in the plot (top left of Figure 2.2). The RMSD for aspartate, comparing the final and initial structures, was 0.1697 from conjugate gradients and 0.7237 for the results from the modified analysis script.

Figure 2.3 shows the structure of aspartate before and after optimisation, where the coordinates were aligned using the *rmsd* package. Using conjugate gradients, the molecule is optimised without any significant changes to the overall structure. However, when aspartate is optimised with strict rules for energy convergence (i.e. having 10 equivalent values over 1000 steepest descent iterations), the structure changes to better facilitate favourable electrostatic interactions - shown in the right image in Figure 2.3. During a *Kaplan* conformer search, only small structural changes need to be made during the optimisation step, since it is easier to facilitate larger structural changes via mutation operators.

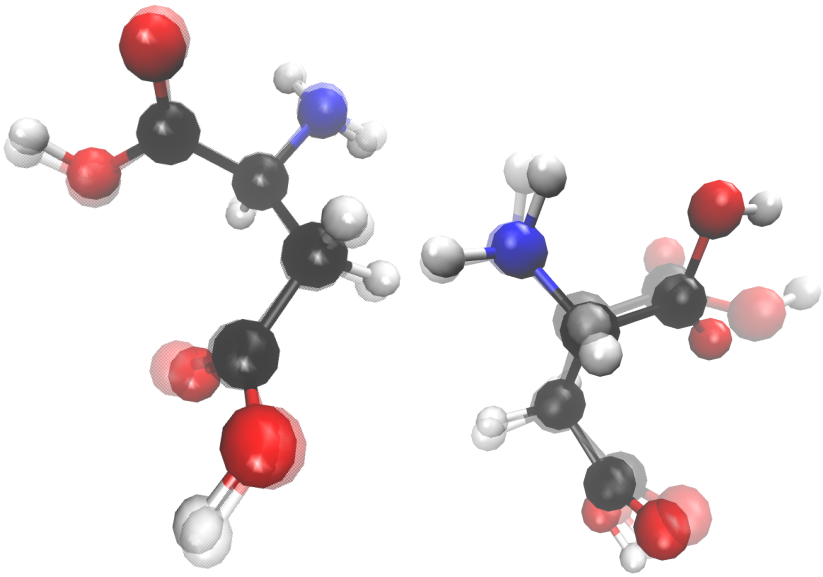


Figure 2.3: The coordinates before and after optimisation for aspartate are transparent and opaque, respectively. Left shows optimisation using conjugate gradients (with the fixed *Openbabel* code), whereas right shows the optimisation after energetic convergence was reached.

The first 100 iterations of steepest descent account for at least one third of the energy difference caused by the optimisation. The change in energy at 100 iterations is also quantified in Table 2.2; the percent converged at 100 iterations was calculated using:

$$\%\text{Conv @ } E_{100} = \frac{E_{100}-Ei}{Ef-Ei} * 100\%$$

where E_{100}, E_i, and E_f are the energies at 100 steps, initially, and after convergence, respectively. Therefore, for the amino acids, some local optimisation significantly improves the energy of the *PubChem* structures.

The RMSD was calculated for each amino acid to compare the initial *PubChem* coordinates and the final coordinates after optimisation. The number of iteration steps was then plotted as a function of the RMSD, with labels corresponding to the amino acid letter code, as shown in Figure 2.5.

Glutamine, phenylalanine, and tyrosine took longer to reach convergence for a moderate structural change. Their plots are shown in Figure 2.4, where it is clear that the high number of iterations was not worthwhile, as they neither significantly changed the molecule's structure or energy. Of the 20 amino acids, aspartate and cysteine benefitted most from the optimisation, as these molecules underwent a larger structural change within relatively few iterations - their plots are shown on the left of Figure 2.2. Methionine and histidine (also shown in Figure 2.2), took more iterations to achieve a comparable RMSD to aspartate, which may have been due to their larger size (20 atoms versus aspartate's 16 atoms).

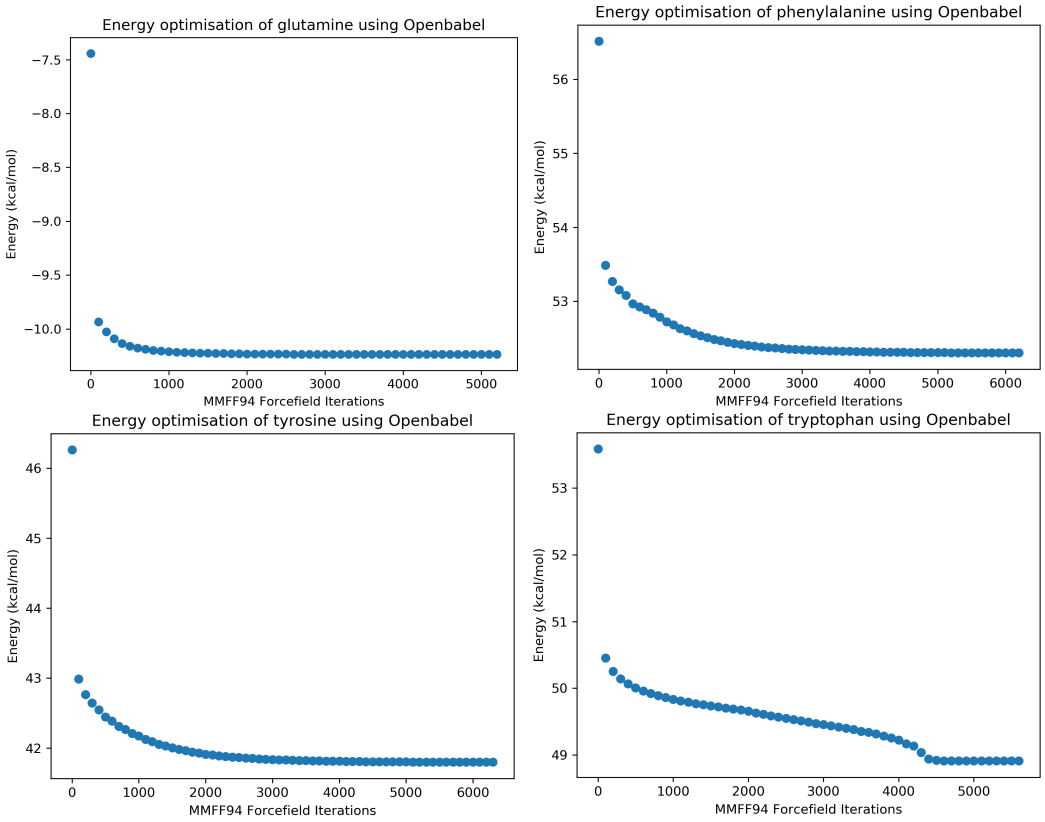


Figure 2.4: Plots showing optimisation of four amino acids, measuring change in energy, which was small relative to the number of iterations to achieve 10 consecutive, identical energies. The RMSD change for these molecules, glutamine, phenylalanine, tyrosine, and tryptophan, was small relative to the number of steepest descent iterations completed.

Amino Acid	E_i	E_f	$E_f - E_i$	E_{100}	**%Conv**	**Steps until Converged**
	kcal/mol				$@E_{100}$	
asparagine	-9.28	-12.90	-3.62	-12.00	75	3100
glutamine	-7.44	-10.24	-2.79	-9.93	89	4300
aspartate	-6.60	-15.96	-9.36	-10.45	41	2700
glycine	23.02	20.93	-2.09	20.93	100	400
tryptophan	53.59	48.91	-4.68	50.45	67	4700
cysteine	42.61	36.38	-6.24	40.25	38	2200
threonine	54.84	50.70	-4.15	51.19	88	1000
alanine	26.06	22.84	-3.22	23.58	77	1200
isoleucine	29.37	26.38	-2.99	26.59	93	2100
leucine	36.21	30.33	-5.88	31.16	86	1600
tyrosine	46.26	41.80	-4.46	42.99	73	5400
glutamate	-2.61	-7.66	-5.05	-6.81	83	1200
proline	29.27	24.50	-4.77	25.29	83	2000
histidine	41.69	35.25	-6.44	37.69	62	6200
lysine	21.19	18.73	-2.45	18.80	97	2100
serine	46.83	43.93	-2.90	44.48	81	2300
arginine	-101.22	-113.00	-11.78	-106.96	49	1300
valine	32.34	28.95	-3.38	29.42	86	3700
methionine	29.85	23.23	-6.62	27.11	41	5800
phenylalanine	56.52	52.30	-4.21	53.49	72	5300

Table 2.2: E_i, E_f, and E_{100} represent the MMFF94 energies initially (from *PubChem*), after optimisation, and after 100 iterations of steepest descent, respectively. The number of steps until convergence is based on the first time the minimum energy appeared. The total energy change after optimisation is given in the fourth column, and the percent of the total energy change achieved after 100 iterations is given in the 6th column.

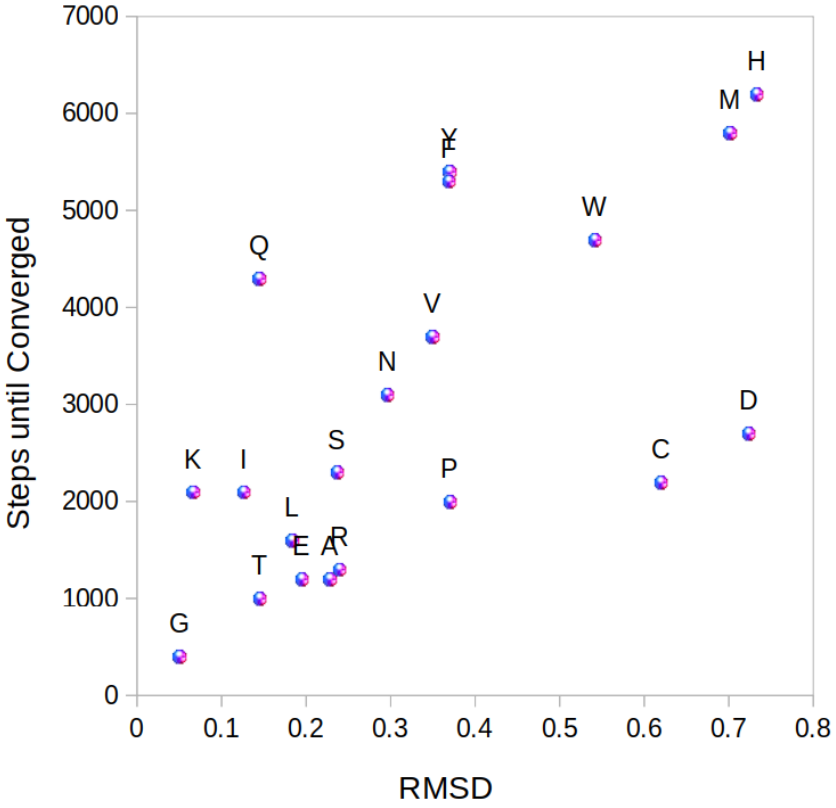


Figure 2.5: RMSD between the initial *PubChem* coordinates and the coordinates after optimising with *Openbabel*. The plot shows the RMSD as it relates to the number of steps until convergence was reached for each amino acid (indicated by letter code).

In conclusion, all amino acid structures from *PubChem* benefitted from local forcefield optimisation. The average energy decrease was 4.85 kcal/mol. For 16 of the 20 amino acids, 100 steps of steepest descent accounted for at least 60% of the overall energy improvement. To obtain the last 40% of the energy improvement, the number of steps needed was between 400 and 6200 steps. Running thousands of steps is impractical for a local optimisation, but can be performed on the final structures from a conformer search. Sometimes, as was the case for aspartate and histidine, a numerical optimisation causes a torsional change (thus finding a significantly different structure), rather than refining the input structure.

Package	Method	How to run	Conformer eval?
Frog2	Monte Carlo	Webapp	N
Balloon	**MOGA** or DG	Terminal	N
Openbabel	**GA** or systematic	Terminal	Y
Confab	Systematic	via *Openbabel*	-
RDKit	ETKDG	*Python* Script	Y
Kaplan	EA	*Python* Script	Y

Table 2.3: A table to compare free conformer searching packages. Bold indicates a method that was tested, for the packages that offer more than one way to conformer search. Y means "yes" and N means "no".

2.3 Free Conformer Searching Packages

This section will cover some free conformer searching packages, including how they work, what their differences are, and the types of output the user can expect from their usage. Specifically, the packages covered are: *Frog2*, *Balloon*, *Openbabel*, *Confab*, *RDKit*, and *Kaplan*. The main differences between these packages are given in Table 2.3.

Most of the packages can be run using a simple terminal command. *Confab* does not have a separate codebase, and is instead distributed as part of *Openbabel*. Both *Kaplan* and *RDKit* require a script; however, the programming knowledge required to write an *RDKit* script is much higher than for *Kaplan*. The *Frog2* source code can also be scripted; the user would have to compile the source code and run `www_iMolecule.py` with *Python* version 2.

The oldest package by far is *Frog2*, since its source code has not been updated in 5 years, and the last release was in 2011. It is the only conformer searching package that has a web-based interface, which makes it accessible to the widest range of users. The other conformer searching packages in Table 2.3 have all been updated within the last year, although *Openbabel* (and by extension *Confab*) has not had a stable release since 2016. The more often that a package is updated, the

more likely a user can get support (e.g. for new features or when bugs are found in the software).

From a conformer searching perspective, *Openbabel* is the least-documented, because the only information about how the genetic algorithm (GA) works is contained within the source code. The default logging information is also unhelpful, and looks more like debugging output. Comparatively, *Frog2*, *Confab*, *RDKit*, and *Balloon* all have papers explaining how their algorithm generates conformers; of these, the papers for *Balloon* and *RDKit* require payment (i.e. a journal subscription), whereas the *Frog2* and *Confab* papers are free to the public. Alternative sources of documentation, aside from academic papers, are important, so the user is notified in the case of a major feature addition, and can get technical help, such as for installation and program execution.

As mentioned in the introduction, parametrisation ensures the most efficient use of an evolutionary algorithm (EA). Since *Openbabel*'s algorithm is not documented, it is more difficult to know which options to use, even for those with experience with GA. Furthermore, *Balloon*'s Multi-Objective Genetic Algorithm (MOGA) was not parametrised in the initial paper, and no help is provided in its documentation as to which parameters are important for which molecules. These packages therefore require the user to run the conformer search with different inputs, but do not include ways to compare the results of each search. Furthermore, the stochastic nature of some programs means that it is hard to distinguish "algorithmic noise" from significant improvement in the results.

Kaplan allows the user to interact with the conformer searching process and the population, such that it is more obvious when the algorithm requires a parameter change. For example, the two main ways to score the GA conformer search in *Openbabel* are energy and RMSD. *Kaplan* instead weights these qualities using coefficients (which are each 0.5 by default). Therefore, the user can easily increase their consideration for RMSD over energy, and vice versa. Since *Kaplan* accepts previous conformer results as input, it is also possible to refine the same set of geometries after a change in scoring metric.

Comparing running times, *Kaplan* is a lot slower than the other programs (ap-

proximately 3 minutes per amino acid conformer search). Comparatively, running *Frog2* on the webserver takes less than a minute, and the other packages are almost instantaneous for the amino acids. *Frog2* was reported to be 20 times faster than *Frog1* [63], so it would not be unusual to release a second, optimised version of *Kaplan*. EAs are expensive to run, and *Python* is not a fast programming language. *Kaplan* does not implement any sort of parallelism, and was written to prioritise readability instead of speed. Furthermore, *Kaplan* performs local numerical optimisation during each mating event, which contributes greatly to the run-time. The runtime can easily be reduced by decreasing the default number of steps (100) for local optimisation. Another major contributor to the initialisation in *Kaplan* is the http request that is sent to *PubChem* for the initial structure. Therefore, a possible improvement to *Kaplan*'s speed would be to allow an offline mode.

2.3.1 *Frog2*

The *Frog2* web-based software package [63] was used to generate conformers for the amino acids. As of writing, the website can be found here: `http://bioserv.rpbs.univ-paris-diderot.fr/services/Frog2/` the webserver is here: `http://mobyle.rpbs.univ-paris-diderot.fr/cgi-bin/portal.py#forms::Frog2` and the version used was Frog v2.14, where Frog stands for **FR**ee **O**n line dru**G** conformation generation. The source code is also freely available and can be found on github: `https://github.com/tuffery/Frog2`; however, using the code without the webserver is somewhat undocumented.

Improvements were made to *Frog2* over *Frog1* in terms of computational speed and conformer diversity. The update also allows for construction of rings outside of the *Frog* database, which are made using DG-AMMOS [56]. AMMOS stands for Automated Molecular Mechanics Optimization tool for *in silico* Screening. *Frog2* does not yet account for ring flexibility during conformer development. It combines rule- and data-based approaches to construct molecular geometries, and is based on *Frowns* [74]. *Frowns* is a chemistry toolkit (in part based on the *PyDaylight* API by Andrew Dalke) that includes: SMILES parsing, SMARTS substructure search-

ing, sdf file handling, fingerprint generation, aromaticity perception, and molecule depiction. Up to 8 stereocenters can be accounted for when designing an initial molecule, such that all possible stereoisomers have at least one conformation. When building 3D structures, *Frog2* uses *Openbabel* to calculate the necessary hydrogen coordinates.

***Frog2* Conformers Produced**	
3	glycine, proline, alanine
9	valine, threonine, cysteine, serine
22	asparagine
24	isoleucine, tryptophan
25	phenylalanine, tyrosine, leucine
27	aspartate, histidine
50 (max)	lysine, arginine, methionine, glutamine, glutamate

Table 2.4: Number of conformers produced by *Frog2* for each amino acid.

The basis of the *Frog2* conformer search is a two-stage Monte Carlo method. Stage one finds a representative selection of dihedral angles, based on atom types, and explores any resulting geometries. The dihedral selection is weighted such that dihedrals with an equal number of atoms on either side are more likely to be chosen than, for example, dihedral angles at the ends of the molecule. Stage two, which is implemented for high-energy conformers, fine-tunes the geometry by adjusting the dihedrals of the stage one conformers by small amounts. Divisive hierarchical clustering (see Appendix, Section A.3.4, for details) is used to ensure that conformers have a RMSD above a certain threshold.

Rather than computing all of the forcefield energy terms for each conformer, conformers are ranked on van der Waals interactions only. However, the option to minimise the conformers with the AMMP forcefield from AMMOS is available (and was selected for these experiments).

According to the website, the `Disambiguate` and `Minimize` options minimize the conformers using AMMOS. Based on the wording, `Disambiguate` could mean using a distance metric and `Minimize` could mean using an energy metric. When `Produce` is set to `Multi`, multiple conformers are generated per conformers, to a maximum of `#conf`. Single conformer production returns one energetically-reasonable conformer from a set. `E max` is the maximum absolute

energy (van der Waals score) that a conformer can have to be considered "correct", but the energy units are not given. The `#mc steps` parameter is the maximum number of Monte-Carlo steps that are performed to sample the conformer space. The `Energy Window` parameter is stated as being the energy window (referenced from the lowest energy conformer) in which conformers are accepted (again, the units are not given, but it is assumed the units are the same as the `E max` parameter).

Input Field	Default	Options
`Output Format`	mol2	mol2/sdf/pdb
`Disambiguate`	Yes	Y/N
`Minimize`	No	Y/N
`Produce`	Multi	Multi/Single
`#conf`	50	-
`E max`	100.0	-
`#mc steps`	100	10,100,1000
`Energy Window`	50	10,25,50,100

Table 2.5: The calculation options for *Frog2*, with defaults given in the middle column,and fixed options given in the right column.

For the *Frog2* test, `output format` was set to `sdf`, `Disambiguate` and `Minimize` were both set to `Yes`, `#mc steps` was set to 1000, `Energy Window` was set to 100, and, for the rest of the inputs, the defaults from Table 2.5 were used. The sdf files supplied to *Frog2* were downloaded from each amino acid page on the *PubChem* website (under 3D Conformer). With 3D input, *Frog2* does not have to generate an initial conformer. The number of amino acid conformers that were produced can be found in Table 2.4.

2.3.2 *Balloon*

Balloon [52][64] is a conformer searching package that uses a multi-objective genetic algorithm (MOGA) to explore molecular coordinate space. The executable can be downloaded here: `http://users.abo.fi/mivainio/balloon/`. Other than a list of inputs accessed via `./balloon --help`, there is some documentation for *Balloon* on the website at `http://users.abo.fi/mivainio/balloon/faq.htm`.

The MOGA works as follows: first, a set of solutions is randomly generated, and

each solution is represented as an array (or chromosome) of numbers (called genes). Then, each solution is assigned a measure of value per criterion, and can dominate another solution for one or more criteria. The Pareto rank describes the ordering of solutions in terms of domination over other solutions. To create new solutions, a combination of two random individuals is generated using a crossover operator followed by mutation of the resulting offspring. The selection is biased towards solutions with better Pareto rank. The ordering, selection, and reproduction cycle is completed for a given number of generations.

With *Balloon*, non-3D inputs are converted into a 3D template or topology using either stochastic proximity embedding or metric matrix embedding. It is possible to choose the `rebuildGeometry` option such that the initial geometry is built from scratch; otherwise, the 3D inputs are used as the template. There is also a fourth-dimension that is constructed at start-up to account for stereocenters. Energies are calculated using the MMFF94 forcefield; parameters for MMFF94 are distributed with the *Balloon* executable. Unlike some of the other conformer searching packages, *Balloon* allows for the consideration of flexible aliphatic rings in its conformer searching. However, it is not open-source.

The initial structure is important for the balloon software, because all subsequent solutions to the conformer search are produced relative to the starting structure. Each solution is encoded by a set of 6 chromosomes; 4 chromosomes that encode for structural modifications that are applied to the starting template, and 2 chromosomes that encode the order of execution for said modifications. Chromosome one consists of torsion angles, where the torsion angles are floating point numbers in the range $[-\pi, \pi)$ representing any acyclic single bonds connecting non-terminal atoms.

The crossover operator in *Balloon* works by first producing a copy of each of the parent solutions. Then, uniform crossover is applied to the torsion angle and global transformation chromosomes of the offspring. Uniform crossover works by iterating over the genes in one chromosome for two possible solutions. Each gene in each chromosome is swapped between the offspring with a probability between 0.1 and 0.2.

Cyclohexane

Since *Balloon* is one of the few conformer searching packages to offer ring conformational analysis, its ability to find the conformers of cyclohexane was tested. Two specific tests were run: (1) `rebuild`, and (2) `modify`. The starting structure for both tests was downloaded from *PubChem* as an sdf file. For test (1), the cyclohexane molecule was built from scratch by *Balloon* using the `--rebuildGeometry` command-line option. For both tests, the `--nconfs` was set to 3, and the `--nGenerations` was set to 300. The number of conformers selected indicates the size of the initial population (or a GA-based conformer search), not the number of conformers to return. The population size for *Balloon* is dynamic, depending on how many individuals are found at the Pareto front.

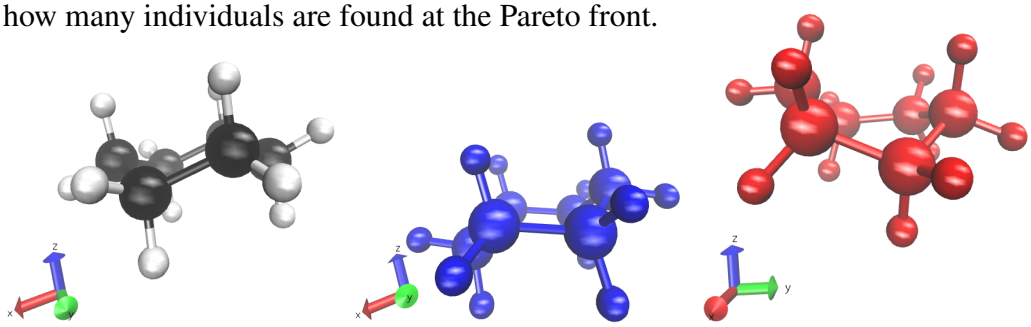


Figure 2.6: Some *VMD* images of cyclohexane from the *Balloon* conformer searches. Left is the *PubChem* input coordinates, centre is the `modify` structure, right is the `rebuild` structure.

Both tests returned one conformer each. The structures are given in Figure 2.6. The `rebuild` structure is a twisted-boat conformer, with an RMSD of 0.928Å compared with the *PubChem* structure. Furthermore, the `modify` structure is the chair flip of the `PubChem` structure, hence the higher RMSD of 1.290Å. Both structures had a higher energy than the input structure, though the difference was small (0.00039 kcal/mol) when comparing the `modify` structure. The energy difference between `rebuild` and the *PubChem* structure was 5.93041 kcal/mol. A breakdown of the energy terms (calculated using `obenergy` from *Openbabel*) can be found in Table 2.6. Therefore, this test found that *Balloon* is suitable for finding ring conformations.

Energy Term	*PubChem*	`rebuild`	`modify`
Bond Stretching	0.64224	0.77519	0.65113
Angle Bending	1.05171	1.86131	1.04050
Stretch Bending	-0.06058	0.00327	-0.06189
Torsional	-11.40905	-8.00099	-11.41037
Out-of-Plan Bending	0	0	0
van der Waals	6.21477	7.73071	6.22010
Electrostatic	0	0	0
Total Energy	-3.56092	2.36949	-3.56053

Table 2.6: The energy breakdown (in kcal/mol) for the original coordinates of cyclohexane (from *PubChem*), the `rebuild` structure, and the `modify` structure.

Amino Acids

The amino acids were also passed to *Balloon* for conformer searching. A script (Appendix, Section A.2.3) was written to automate the conformer search, energy evaluation (with `obenergy`), and RMSD calculation (with `calculate_rmsd`). In the case where only one conformer was produced, the RMSD was calculated relative to the starting geometry. The numbers of conformers found for each amino acid is given in Table 2.7.

Balloon **Conformers**
1:G, W, C, T, A, I, L, Y, P, H, S, V, F
2:N, 3:M, 4:Q, 5:D
6:K, 7:R, 15:E

Table 2.7: Number of conformers produced by *Balloon* for each amino acid.

A low number of conformers were produced by *Balloon*. The input options were: `num_confs = 10`, and `nGenerations = 1000`, which is how many iterations of the GA to complete. *Balloon* discards the conformer with the highest energy if the RMSD between the conformers is less than 0.5Å (default value). Therefore, it is likely that many conformers were pruned in the final stage of the algorithm.

2.3.3 *Openbabel*

Openbabel [68][107] has two main methods for generating conformers: using a GA (default) and using *Confab* [67]. Forcefield-based conformer generation is also supported, but was not tested by the author. *Confab* will be discussed in the next section. *Openbabel* and *Confab* results were generated using the same script as *Balloon*, which can be found in the appendix, Section A.2.3.

The GA-based conformer searching tool has four scoring methods: RMSD, minimum RMSD, energy, and minimum energy. Using the default inputs, each parent generates 5 children, which are mutated 5 times each. The GA is run until 5 identical populations have been made. Two conformer searches were performed for the 20 amino acids, one was scored with RMSD and one with energy. 10 conformers were requested for each amino acid.

2.3.4 *Confab*

Confab [67] is a systematic conformer searching package. It requires a 3D input structure with reasonable bond lengths and angles, so it is different from other packages (like *Frog2* and *Balloon*) that can generate their own 3D coordinates if needed. Ring conformations are not yet supported, and so the rings in the molecule must also have reasonable structures.

In *Confab*, rotatable bonds are defined as: "all acyclic single bonds where both atoms of the bond are connected to at least two non-hydrogen atoms, but neither atom of the bond is sp-hybridised." Therefore (in *Confab*), 1,2-difluoroethane has one rotatable bond, whereas 1,1-difluoroethane, ethane and propane have no rotatable bonds. Based on matching molecular substructure with SMARTS strings, *Confab* assigns a set of allowed torsion values to each rotatable bond. Redundant torsion angles (related by symmetry) are excluded from the calculations.

Energies are calculated using the MMFF94 forcefield, where the van der Waals, electrostatic, and torsion terms are regularly updated (other terms are constant). Torsions are optimised using a greedy algorithm, starting with the central-most torsion angle. The four most central torsion angles are used as a starting point for

the torsion optimisation, thus negating the need to exhaustively search all torsional combinations. The lowest energy conformer from this search is used as the global minimum.

After finding an estimate of the global energy minimum, *Confab* generates n possible conformers using the allowed torsion angles, where $n = 1$ million by default. A Linear Feedback Shift Register (LFSR) is implemented to randomly sample possible torsion angle combinations; therefore, its search is not limited to a particular part of the space. If the energy of a given random conformer is within the user-specified energy cutoff (default of 50.0 kcal/mol), then the geometry is stored.

The QCP algorithm [48] – which is called the Quaternion algorithm in Charnley's RMSD package [106] – is implemented instead of the Kabsch algorithm in *Confab*, since the authors found the former to be twice as fast in the alignment of molecules. To limit the number of pairwise RMSD calculations, an RMSD-based tree is constructed, where structures are only kept if they have a greater RMSD than the RMSD cutoff (default 0.5Å) when compared to existing structures in the tree. The first RMSD tree can retain similar conformers if they are in different branches. A second RMSD tree is built, storing conformers in order of increasing energy, where RMSD is calculated between conformers at the same energy level (sibling nodes). The RMSD calculations that are performed on sibling nodes also consider possible automorphisms - permutations of the atoms that preserve the connectivity.

A *Confab* conformer search was conducted for each amino acid using *Openbabel* with the default cutoffs and sampling size. Compared to other methods, this systematic exploration generated many more conformations, making pairwise RMSD a more costly analysis. The number of conformers generated per amino acid for *Confab* can be found in Table 2.10.

2.3.5 *RDKit*

Similar to *Openbabel*, *RDKit* [87] is an open-source cheminformatics software package that can be used for finding conformers. Two separate scripts were written to find 10 conformers for each amino acid with *RDKit*. Using the first script,

a bare-bones conformer search was performed with the *PubChem* sdf structures as input. For the second script, *RDKit* was used to generate the amino acid geometries from scratch, using SMILES strings as input, and then optimise each conformer using the MMFF94 forcefield for 1000 steps.[1] This number of iterations was sufficient to converge the energy for all structures tested. Finally, the energy for each conformer and the RMSD for each conformer pair was calculated from within the *RDKit* framework. The *RDKit* scripts can be found in the Appendix; see Section A.2.2 for the non-optimised conformer generation script and Section A.2.1 for the optimised version.

2.4 Comparison of Packages

This section will discuss some results from the conformer searches of the amino acids. As a general rule, it is important to optimise the geometries of any final conformers to ensure their geometries represent the local minima. In the cases where the user cannot visually verify the correctness of their conformers (e.g. too many structures to individually check), the RMSD should be calculated with a couple of methods to determine the smallest RMSD (for example, with and without reordering). A parameter search is recommended for molecules where no reference structure is available.

Friedrich et al. found *RDKit* and its ETKDG algorithm to be the best out of *Balloon*, *Confab*, *Frog2*, and Multiconf-DOCK [94], and also found its performance comparable to mid-ranked commercial programs [93]. The biggest difference that Friedrich et al. found between the free and commercial conformer searching programs was the level of robustness, since commercial programs can handle at least 99% of input molecules. *Confab* was found to have the lowest success rate in terms of number of processed molecules from their database, followed by *Frog2*. The differences in performance of the commercial programs was also found to be much smaller than for the free programs [93].

[1] Note that RDKit has its own implementation of forcefield optimisation.

In general, Ebejer et al. [70] found the RMSD values for *RDKit* to be lower than for other programs. In their paper, 708 molecules from the Astex Diverse Set were tested on *Balloon*, *RDKit*, *Frog2*, and *Confab* and compared the results to those from a commercial conformer searching toolkit, OMEGA [61]. Ebejer at al. also stated that *Confab* performed better than *RDKit* for molecules with more rotatable bonds (10+). Their speed rankings were *Frog2*, then *RDKit*, followed by a large separation for the other free programs. The author compared the usability of each program, and the results are summarised in Tables 2.8 and 2.9.

To run the conformer searches for each program, the author wrote a script that covers *Balloon*, *Openbabel*, *Confab*, and *RDKit*, where *Confab* was run using *Openbabel* and *RDKit* was run by calling two versions of an external *Python* script. The scripts used to run the conformer searches can be found in the Appendix, Section A.2.3.

Except for the optimised version of the *RDKit* script, the only constant parameter for these searches was the input file. Amino acid geometries were downloaded as sdf files from the *PubChem* database; these geometries are the amino acid structures without any special caps (i.e. COOH and NH_2 ends).[2] To fairly compare the conformers produced by the programs, an exhaustive parameter search would need to be performed for each method for each input structure, which is outside the scope of this work.

Kaplan was run via the following script:

```
from kaplan.control import run_kaplan
from kaplan.tools import amino_acids
for aa in amino_acids:
    run_kaplan({
        "struct_input": aa,
        "num_geoms": 10,
        "stop_at_conv": 50,
    })
```

[2] Aspartate and glutamate actually represent their carboxylic acid counterparts, but the shorter names reference the same *PubChem* compound. Therefore, all structures have a default charge of 0.

Package	**Time to Solution**	**Documentation**	**Code Accessibility**	**Latest Stable Release**	**Conformers Generated**
Frog2	Medium*	Free paper, good online documentation	Python/C required	8 years	Moderate
Balloon	Fast	Paid paper, minimal online documentation	Closed-source	<1 month	Low
Openbabel	Fast	Minimal online documentation (for GA conformer searching method)	C++ required	3 Years	Exact
Confab	Fast	Free paper, minimal online documentation	C++ required	3 Years	High
RDKit	Fast	Paid paper, good online documentation	Python required	<1 month	Exact
Kaplan	Slow	Thesis, minimal online documentation	Python required	<1 month	Exact

Table 2.8: Some comparisons for the free conformer searching packages, including program time to solution, level of documentation available, programming skills required to read the code, when the last stable release was made, and how many conformers each package generates for small molecules. Exact conformers generated means that an input parameter to the program is how many conformers to produce. *if the webapp is not used, the code must be compiled, whereas the other tools have pre-compiled versions for most operating systems.

Package	**Energy Eval**	**Energy Opt**	**RMSD Eval**	**Select Dihedrals**	**Accepted Inputs**
Frog2	N	FF	N	N	SMILES, sdf, mol2
Balloon	N	N	N	N	sdf, mol2, SMILES, Gaussian98/03.log, pdb, vbf, xyz
Openbabel/ Confab	FF	FF	Y	N	Most structure file types
RDKit	FF	FF	Y	N	SMILES, sdf, RDKit mol object
Kaplan	FF + quantum	FF + quantum	Y	Y	Name, SMILES, InChI, CID (PubChem ID), xyz, com (Gaussian Input)

Table 2.9: Some comparisons for the free conformer searching packages, including availability of energy evaluation (FF=forcefield), RMSD calculation, and energy optimisation (opt), accepted inputs, and if package supports selecting specific dihedrals for conformer searching. *Openbabel* and *Confab* are in the same row, because analysis tools and input processing for *Confab* is directly available through *Openbabel*.

The `obenergy` terminal command (part of the *Openbabel* software package), which uses the MMFF94 forcefield by default, was used to calculate final conformer energies for each package. Figures 2.8 and 2.9 show boxplots for each amino acid, where the energy distributions are given relative to the minimum energy (for each molecule) across all experiments. For 12 of the 20 amino acids, *Kaplan* clearly found the lowest-energy conformer when compared to the fully-optimised *RDKit* conformers. For 3 amino acids, *RDKit* found the lowest-energy conformer. For the remaining 5 cases, the lowest-energy conformer was similar when comparing *Kaplan* and *RDKit* results.

RDKit uses the BFGS (Broyden-Fletcher-Goldfarb-Shanno) algorithm [5][6][7][9] to optimise the conformers, which is a iterative, numerical, quasi-Newton optimisation method. The *Openbabel* optimisation is done with either conjugate gradients or steepest descent. The optimised *RDKit* conformers were tested for convergence, and required a maximum of 1000 iterations of BFGS. The *Kaplan* conformers were not tested for convergence, and received a maximum of 2500 iterations of conjugate gradients. Therefore, in the cases where the *Kaplan* conformers are higher in energy, the conformers may need more iterations to reach convergence. To improve *Kaplan* results, an option to exhaustively optimise the final conformers could be added, as well as the option to choose the optimisation method.

The *Frog2* conformers were optimised with the AMMOS forcefield, but the energies of the conformers were still significantly worse than for the optimised *RDKit* and *Kaplan* results. *Openbabel* conformers, scored by energy, have smaller energy variance than for all other packages tested. In most cases, the *Openbabel* conformers, scored by RMSD, are most similar to the energies produced by *Confab*.

Balloon could not find more than one conformer for 13 of the amino acids; however, it is the only free program that allows for the exploration of ring flexibility. Therefore, it is recommended for larger molecules, or cases where the user wants to specifically look for ring conformations. *Balloon* has an option `fullforce`, which means the user can optimise their final geometries using the full forcefield (by default the torsion gradient and electrostatics are ignored). Therefore, the poor energies of the *Balloon* conformers could be improved by enabling this option.

Amino Acid	# of Conformers Returned		
	Confab	*Frog2*	*Balloon*
asparagine	68	22	2
glutamine	151	50	4
aspartate	65	27	5
glycine	3	3	1
tryptophan	51	24	1
cysteine	14	9	1
threonine	14	9	1
alanine	4	3	1
isoleucine	39	24	1
leucine	25	25	1
tyrosine	24	25	1
glutamate	146	50	15
proline	3	3	1
histidine	31	27	1
lysine	169	50	6
serine	13	9	1
arginine	295	50	7
valine	15	9	1
methionine	88	50	3
phenylalanine	25	25	1

Table 2.10: The number of conformers returned per amino acid, for the conformer searching programs that evaluate conformers based on cutoff metrics.

The pairwise RMSD values were calculated for each set of conformers returned by each of the programs. This test was done to see whether the packages returned duplicate structures for the same input molecule. No duplicates were found in the *Balloon* conformers (for the structures with more than one conformer). *Frog2* found duplicate conformers for arginine, tryptophan, threonine, isoleucine, and leucine. There is an obvious cutoff for the *Openbabel* RMSD-scored conformers of 0.25 Å. For the energy-scored *Openbabel* conformers, the cutoff is even more pronounced, showing that optimising energy tends to decrease RMSD to its lower limit. *Confab* appears similar to RMSD-scored *Openbabel* results, where there is also a cutoff at 0.25 Å.

Sometimes, the reordering option can find a lower RMSD due to symmetry relations. One interesting result from these tests was that the *Kaplan* and *RDKit* RMSD values increased on average with reordering selected. Since *Kaplan* guarantees the atomic ordering to be the same for all output files, it is concluded that the *rmsd* package may not be correctly considering the case where no atom shuffling is necessary, and instead tries to overlap the wrong atoms.

Kaplan returned duplicate conformers for 11 amino acids. Therefore, it may be useful to add a penalty to the fitness function for `pmems` containing duplicate structures. The user can easily see which conformers are duplicates in *Kaplan* by viewing the automated plot for the best `pmem`, an example of which is given in Figure 2.7. From this knowledge, the user can select another `pmem` from the `ring` and combine the outputs to produce a larger, more distinct set of conformers.

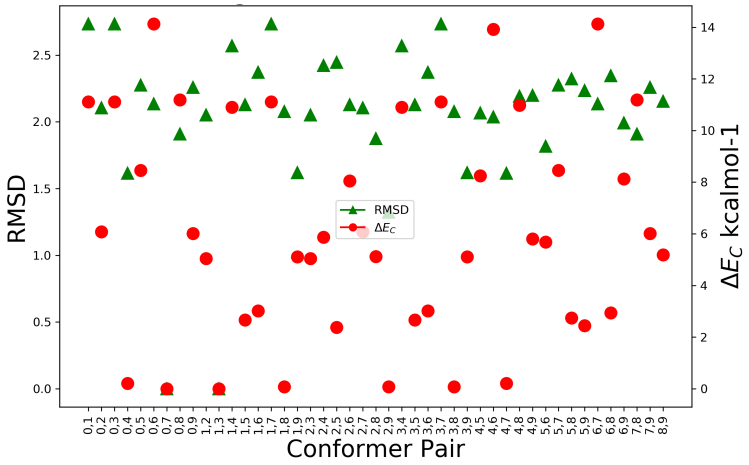


Figure 2.7: An example plot for glutamine auto-generated by *Kaplan*, so the user can quickly identify which conformers are duplicates. In this plot, conformers 0 and 7 and conformers 1 and 3 are likely identical.

In conclusion, this chapter has compared some conformer searching tools and packages, including an energy parametrisation of initial conformer geometries. *Kaplan* was able to find the lowest-energy conformer for 12 of the 20 amino acids, compared with optimised *RDKit* conformers. *RDKit* is generally considered to be the best free tool for conformer searching in the literature. *Kaplan* got the same result for lowest-energy conformer as *RDKit* in 5 cases, whereas *RDKit* was able to find a better structure in the remaining 3 cases. *Kaplan* has a tendency to return identical structures, as determined by RMSD. To improve *Kaplan* results, a penalty should be given to `pmems` containing identical conformers. As an easy workaround, duplicate structures produced in the best `pmem` can be replaced with conformers from other `pmems` in the `ring`.

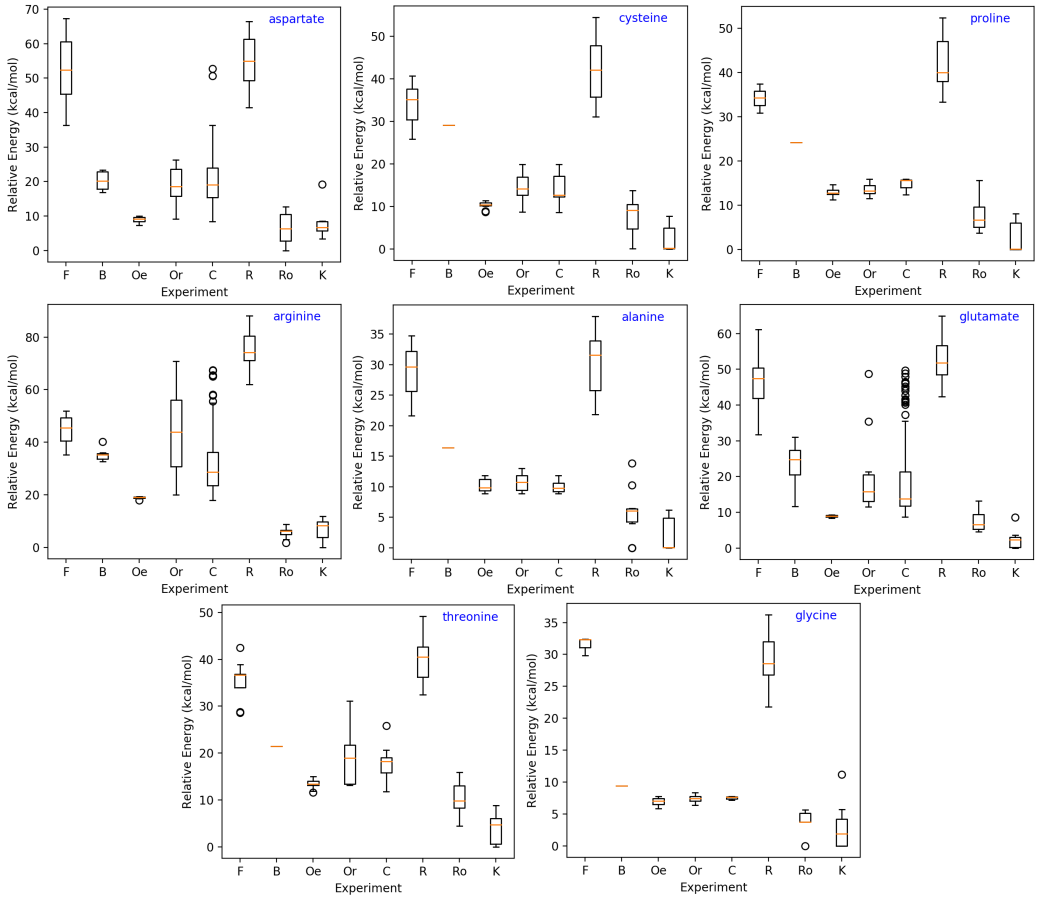


Figure 2.8: Boxplots showing energy distributions for 8 of the 20 amino acids. The experiment labels are: F= *Frog2*, B=*Balloon*, Oe=*Openbabel* GA scored by energy, Or=*Openbabel* GA scored by RMSD, R=*RDKit*, *Ro=RDKit* with optimisation, and K=*Kaplan*.

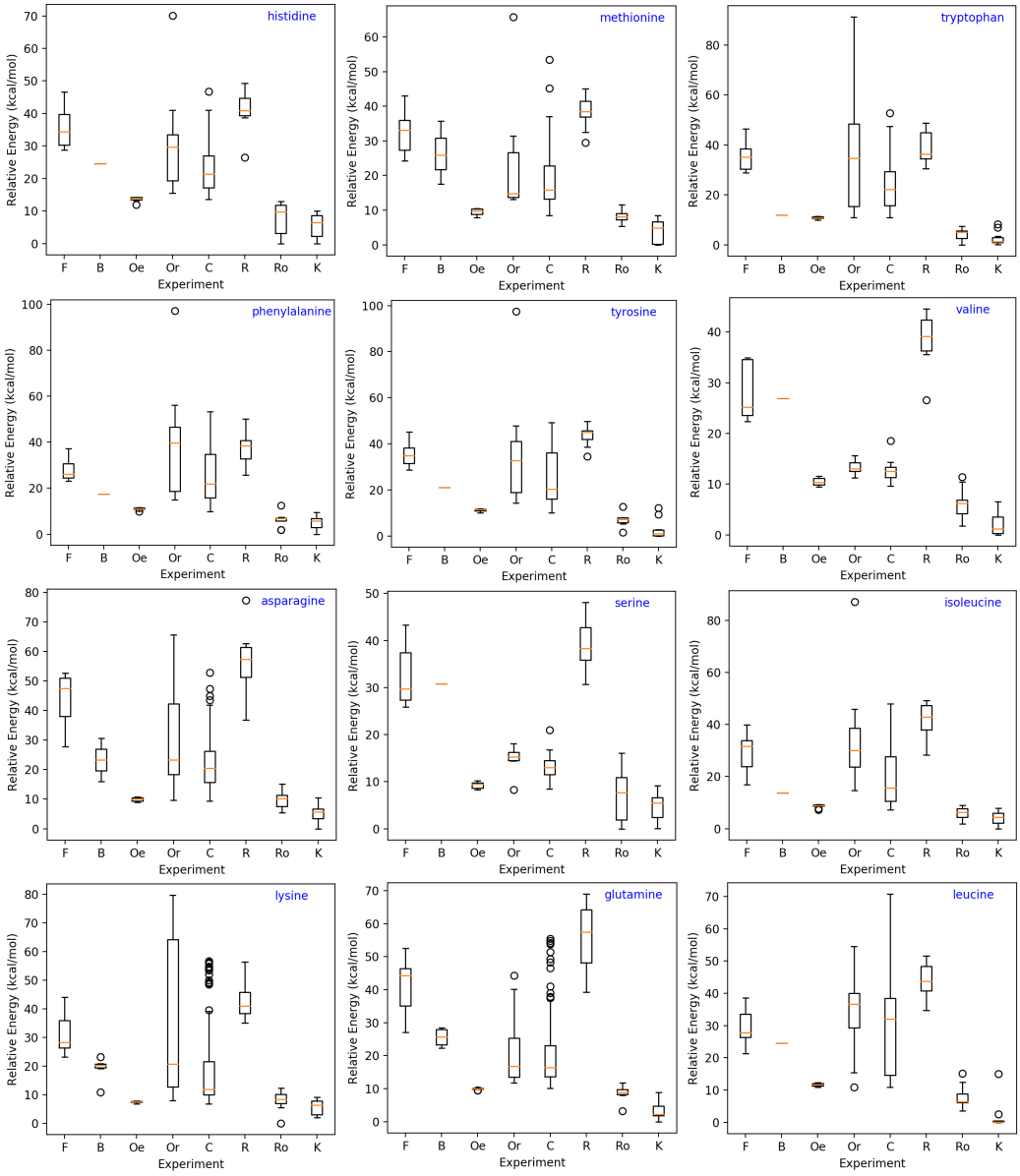


Figure 2.9: Boxplots showing energy distributions for 12 of the 20 amino acids. Experiment labels follow Figure 2.8.

Chapter 3

The *Kaplan* Conformer Searching Program

This chapter will introduce the author's program for conformer searching: *Kaplan*. The first section will cover the basic components of *Kaplan*. A breakdown of its modules and examples of how to use the code will be provided.

3.1 Algorithm Overview

Kaplan uses an evolutionary algorithm (EA) to search for a set of conformers. It is intended for use with small molecules, up to a couple of hundred atoms in size. Each population member (`pmem`) represents a set of conformer geometries, which are encoded using dihedral angles. The dihedral angles to manipulate can be user-specified, with built-in options to select (1) all dihedral angles or (2) a minimum set (default). How the latter selection works is described in detail in Section 3.5.1. Briefly, a connectivity matrix is generated for the input molecule. Each bond connecting two non-identical atoms, *b* and *c* is considered to be the centre of a dihedral angle (*abcd*) if *b* and *c* are: connected to more than 1 atom each, connected to at least 4 unique atoms, and not within a ring. *a* and *d* are chosen based on the largest substituent (in terms of number of atoms) connected to *b* and *c*, respectively.

The population of pmems are stored in a ring. Ring species is a biological concept that relates to the growth of a population around a large physical barrier, such as a lake or mountain. Usually, there is a point at which a member of the population cannot breed with another population member due to evolutionary divergence. The implementation of the ring in *Kaplan* was inspired by work from Daniel Ashlock and his group [60][76][92][83][54][53]. The important ring features are: its initial population size (how many pmems to randomly instantiate), the total size (which should be at least as large as the initial population), and the mating radius. The ring consists of slots; a slot may be empty or filled with a pmem.

Initially, the ring is filled with a contiguous block of pmems, as shown in Figure 3.1. Each initial pmem is generated by sampling, with replacement, uniformly at random from a np.array of allowed dihedral values. By default, this array contains 16 values at uniform increments in the half-open interval $[-pi, pi)$; this selection can also be user-specified. The initial geometry may also be optimised using the aforementioned tools in Section 2.2.

A mating event (mev) in *Kaplan* consists of the following items: (1) ring-based tournament selection, (2) mutation, (3) new pmem fitness evaluation, (4) extinction, and (5) reporting.

The general procedure for tournament selection, discussed in detail in Section 3.7.1, is to first randomly choose an existing pmem from the ring, which is then the first parent. The mating radius then determines how many other pmems are in the tournament, and the best of these pmems is selected to be the second parent. For example, for a mating radius of 3, the pmems 3 slots to the left and right of parent1 would be part of the tournament.

To compare pmems (and choose parent2), each pmem is assigned a fitness (see Section 3.6). A pmem stores the energy of each of its conformers, as well as its pairwise RMSD values. These properties are combined to afford a fitness for each pmem. There are two methods to calculate fitness: with or without normalisation. Using normalisation (default), each energy and RMSD value is assigned a z-score (see Equation 3.3) based on the mean and standard deviations of said values in the entire ring. Fitness is correlated with low conformer energy and high RMSD.

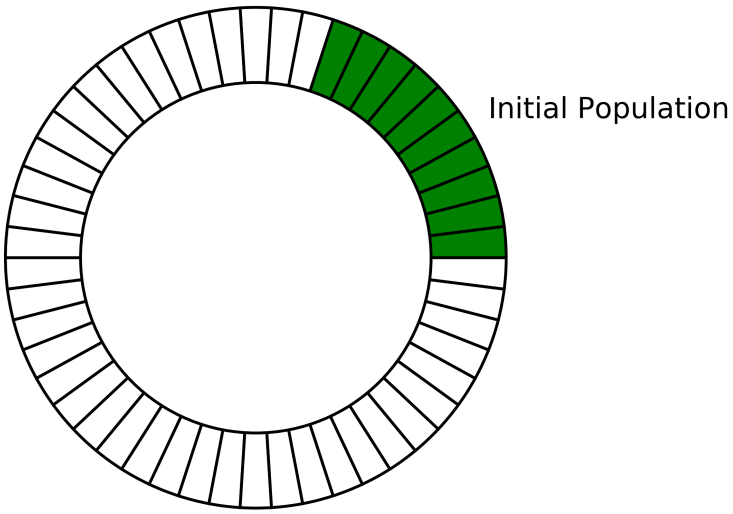


Figure 3.1: The `ring` at the start of the evolutionary algorithm has an initial population that fills a contiguous segment of slots. As evolution progresses, the slots will become filled. This particular `ring` has 50 slots, where white is empty and green is filled, with an initial population size of 10.

To obtain the geometry needed to perform an energy calculation, first the dihedral angles are applied to the initial geometry (which may or may not be optimised). Then, a small optimisation is performed (to account for clashing atoms), after which the energy is calculated, using either quantum chemical methods or a forcefield.

Using a mixture of `swap`, `crossover`, and `mutate` (see Section 3.7.2), two new `pmems` are generated from the parent `pmems`. If any slots in the tournament are empty, then the new `pmems` fill these slots; otherwise, the new `pmems` replace the slots of the worst tournament members (assuming their fitness is not worse than the current occupant).

Eventually, given enough `mevs`, the `ring` will become filled with `pmems`. The purpose of the `ring` is to initially allow for exploration of the solution space - there are empty slots and so the likelihood that a `pmem` will be added to the population is higher. When the `ring` is filled, the algorithm becomes more exploitative, as new `pmems` must have fitness at least as good as the inhabitant of their potential slot to take over that slot.

To reset the `ring` back to a more exploratory state, extinction operators are implemented in *Kaplan*. These operators destroy a variable selection of `pmems`; an extinction event is activated, once per `mev`, if a random number, selected uniformly between 0 and 1, is above the threshold value set by the user. The four available extinction operators, `asteroid`, `deluge`, `agathic`, and `plague`, are explained in Section 3.5.4.

Two types of stopping criteria are implemented in *Kaplan*: a set number of `mevs` have occurred (default 100), or the best `pmem` (the `pmem` with the highest fitness in the `ring`) has not changed after a set number of `mevs` (default 25). How quickly a conformer search fails to improve the best `pmem` can indicate that the search has prematurely stagnated (i.e. has not found a good set of conformers), and could benefit from a higher mutation rate or an extinction event.

At the end of the algorithm, the output module is called to perform a more rigorous optimisation of the best `pmem` before the final structures are written to output files. The progress of the evolution can also be viewed in the stats file, which is updated according to a user-specified interval (default 25 `mevs`). Example output is provided in Figure 3.3.

3.2 Module Guide

There are 14 modules in *Kaplan*. The general purpose of each module can be gleaned from its name. Here is a brief description for each module:

- **Control** - ties together all the other modules to perform a conformer search. The main point of access for *Kaplan*.
- **Inputs** - handles input verification.
- **Output** - the procedure that is called when the EA has finished running. Responsible for returning the results to the user. It may also be used to collect intermittent summary statistics.

- **Ring** - stores the population and handles `pmem` instantiation and population updates.
- **Pmem** - stores solutions (and their properties) to the conformer searching problem.
- **Energy** - runs energy calculations for given geometries.
- **Optimise** - optimises geometries using either a forcefield or a quantum chemistry method.
- **RMSD** - calculates the RMSD for pairs of geometries.
- **Extinction** - applies extinction operators to the `ring`.
- **Fitness** - calculates the fitness of a `pmem`. Knows how to assemble `pmem` components to evaluate a solution.
- **Tournament** - selects parent1 for a tournament, and calls the mutations module to generate new children.
- **Mutations** - applies mutation operators to the solution instances (i.e. sets of dihedral angles).
- **Tools** - contains general purpose tools for analysing output, including generating informative graphs and performing profiling.
- **Geometry** - allows for the manipulation and evaluation of geometries, including finding and updating dihedral angles.

A uses hierarchy - shown in Figure 3.2 - is a diagram that shows how each module uses, or is used by, the other modules. The concept of a module guide was originally conceived by Britton and Parnas and aided in their design of aviation software [12][16][14]. In the module guide, a cloud (software decision) module requires an external software package to meet its requirements. The boxed (behaviour hiding) modules have features written by the author that are meant to remain specific to said module.

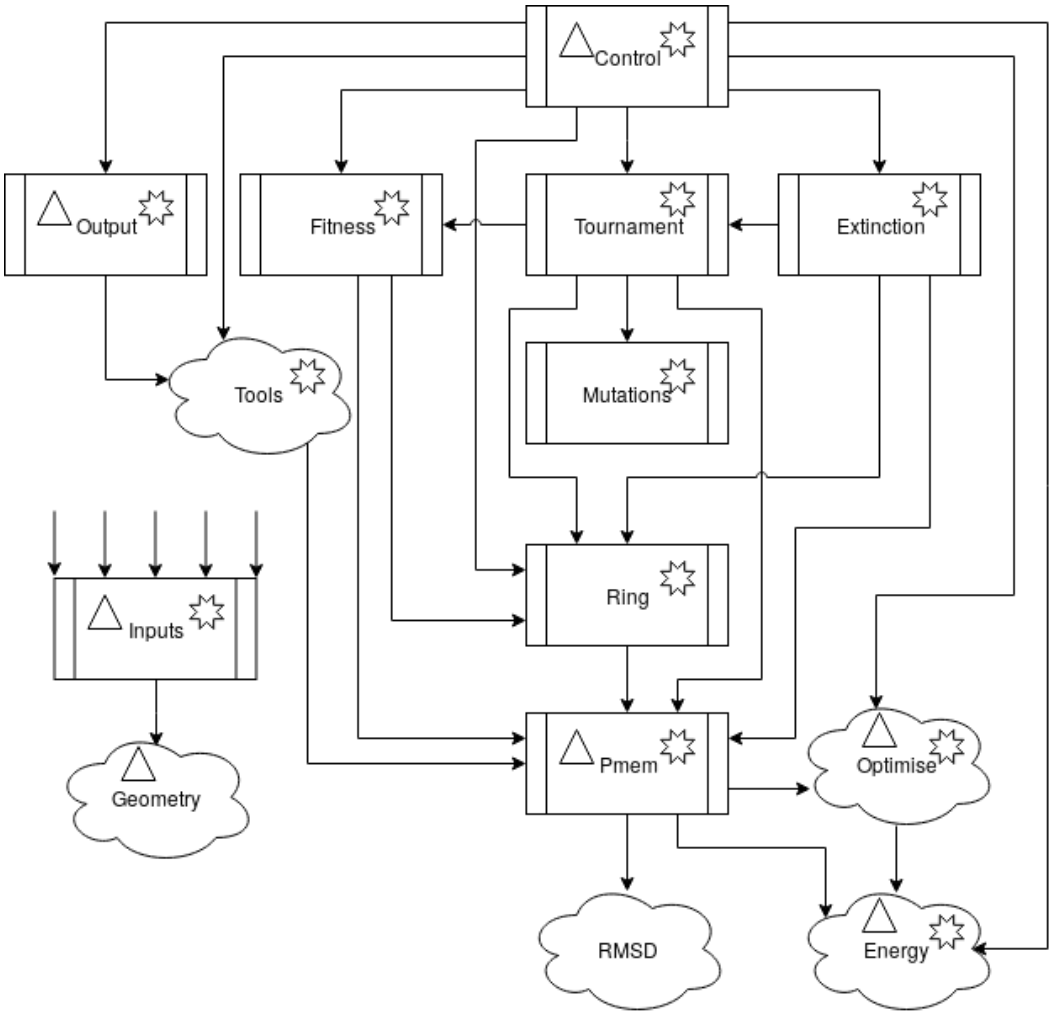


Figure 3.2: The module guide for *Kaplan* shows how each module is related. The arrows show the use cases. An arrow from module A to module B means that module A uses module B. Clouds indicate modules where the majority of the code is written as a wrapper to another package. Boxes indicate behaviour hiding modules. Stars and triangles are used to label modules that use the Inputs and Geometry modules, respectively.

3.3 Running *Kaplan*

The Control module is the module that most users will interact with to run the *Kaplan* program. It has one function, `run_kaplan`, which takes in 6 arguments: `job_inputs`, `ring`, `save`, `new_dir`, `save_every`, and `recalc_fit`. The only required argument is the `job_inputs`, which specifies how the program will be run and for which molecule to find conformers. `job_inputs` can be a string that specifies the path and file name of a previously pickled `inputs` object, an `inputs` object, or a *Python* dictionary, which will be used to construct an `inputs` object. Details on the inputs available can be found in Section 3.4. The most basic execution of *Kaplan* is as follows:

```
from kaplan.control import run_kaplan
run_kaplan({"struct_input": "histidine"})
```

The `ring` argument is `None` by default, but allows the user to provide the path and file name of a pickled `ring` object that can be used at the start of evolution. The purpose of this input is to allow the user to restart a job from a previous state. The `save` argument is `True` by default, which means that the `ring` and `inputs` objects will be saved to the output directory as pickle files; `False` will result in no pickle files being written. The next argument – `new_dir` – defaults to `True`, which means that, in the event that an `inputs` or `ring` object is input, a new output directory will be generated. The `save_every` argument defaults to 25, and specifies the number of mating events to complete before updating stats-file.txt (located in the output directory). The last argument – `recalc_fit` – is a boolean that defaults to `True`. This parameter only impacts program execution if a `ring` pickle file was provided as input to `run_kaplan`. If `True`, then the `pmems` in the `ring` object have their energies and RMSD values recalculated. If `False`, these values are left unchanged.

The basic steps in the `run_kaplan` function are as follows:

1. Initialise and verify inputs (via Inputs module).

2. Optimise the initial geometry (via Optimise module), or (if `opt_init_geom`

is `False`) run an energy calculation to check for convergence (via Energy module).

3. Using the Tools module, generate 2D and 3D visualisations of the molecule, including a diagram that labels the atoms according to their appearance in the input_coords.xyz file.

4. Initialise `ring` object, or reload it from previous state (i.e. when `ring` is not `None` (via Ring module). The `ring` size (`num_slots` parameter) must be the same between runs.

5. Iterate over mating events, starting with 0 (if `ring` is `None`) or the age of the oldest `pmem` from the existing `ring` object. The number of iterations is equal to the input parameter `num_mevs`, unless `stop_at_conv` input parameter is `True` (see Section 3.4.1).

6. Perform a tournament event for each mating event (via Tournament module).

7. Apply an extinction operator (see Section 3.5.4) if the random chance chosen at the current `mev` is $\leq$ the percent chance for said extinction operator, as specified in the inputs.

8. Run the Output module. The output module is called when the current mating event is a multiple of the `save_every` input parameter. It is also called after the final mating event has taken place.

The Control module calls the Fitness module to update the fitness values in the `ring` if a `plague` or `deluge` is performed, or if the Output module is called.

3.4 Program Inputs

The `inputs` module is responsible for accepting and verifying all user inputs, as well as formatting some input parameters from existing values. There are 2 types of

inputs: those generated by *Kaplan* (internal inputs) and those that the user can specify. Technically, all inputs can be changed by the user, but in some cases this may break the program. All inputs are subject to the `_check_input` method of the `kaplan.inputs.Inputs` class, which is called at the start of `run_kaplan`. In the lists below, each input is given with its default value (if applicable) in square brackets [`default`]. If the default value is a string, then the surrounding quotations (i.e. "`default`") are implied.

The only required input to `run_kaplan` is the `struct_input` parameter, which (if given as the only argument) should be a molecule name. The molecule should be in the *PubChem* database if the coordinates are to be generated by *Kaplan*. It is also possible to generate coordinates from other formats, e.g. SMILES, even if there is no specific *PubChem* entry. In this case, the `struct_type` parameter must be set to `smiles`.

The inputs module has one variable (`hardware_inputs`), which is a dictionary. The keys represent program names and the values are a dictionary. This sub-dictionary is intended to be used for hardware parameters. Currently, the only use for this variable is the *Psi4* memory, which is set to `4GB` by default.

3.4.1 User Inputs

These inputs are expected to be modified by the user of *Kaplan*. There are several categories of user inputs, which usually correspond to a module.

Output

The `output_dir` input parameter defaults to `pwd`, which stands for present working directory. If the user wants the output to be in a specific location, they can provide a directory using this input. The other option is `home`, which (for linux users) will put the output in the `/home/username` directory. Each time *Kaplan* is run from the same output directory, a new job directory is created. All of the output is stored under a directory called `kaplan_output`. For example, if the user ran 3 jobs, one for butane, one for propane, and the other for caffeine with the

`output_dir` input set to `/home/user/experiment1`, then the directory tree would look (from the `/home/user` directory) as in Figure 3.3.

- `name [None]` the string of characters to use when generating an output directory. This `name` will appear on output plots for the molecule. If `None`, then *Kaplan* will devise a `name` for the input molecule using the `get_name` method from the `kaplan.inputs.Inputs` class. The `name` parameter should not include any spaces or special characters, to avoid problems with directory navigation and file creation.

- `output_dir [pwd]` where to store the output. Defaults to the directory from where the code is run. If specified, the directory should exist, as *Kaplan* will not generate new directories outside of `kaplan_output` and `job_x_name`.

Geometry

If the user does not provide the charge and multiplicity of the input molecule, these values can be estimated by *Openbabel* during input parsing; however, the user must ensure their correctness.

- `struct_input` the value of the structural input to use. Examples include the file name and path, a chemical identifier (CID - from *PubChem* database), a SMILES string, an InChIKey, or a molecule name.

- `struct_type [name] {com, xyz, smiles, inchi, inchikey, name}` the type of structural input.

- `charge` $\{x \mid x \in \mathbb{Z}\}$ the molecule's total charge.

- `multip` $\{2S+1 \mid S \in \{0, x, x/2 \mid x \in \mathbb{N}\}\}$ the molecule's multiplicity, where S is the total spin angular momentum.

- `num_geoms [5]` $\{x \mid x \in \mathbb{N}\}$ the number of conformers to search for; given as n_G in the text. This parameter changes the size of each `pmem` in terms of number of geometries each one contains.

Directory Tree

```
experiment1/
├── kaplan_output
│   ├── job_0_butane
│   │   ├── conf0.xyz
│   │   ├── conf1.xyz
│   │   ├── conf2.xyz
│   │   ├── conf3.xyz
│   │   ├── conf4.xyz
│   │   ├── heatmap.png
│   │   ├── input_coords.xyz
│   │   ├── inputs.pickle
│   │   ├── obmol2d.png
│   │   ├── plot2d.png
│   │   ├── pmem10-delta-es-rmsds.png
│   │   ├── pmem10_energies.png
│   │   ├── ring.pickle
│   │   └── stats-file.txt
│   ├── job_1_propane
│   │   ├── conf0.xyz
│   │   ├── conf1.xyz
│   │   ├── conf2.xyz
│   │   ├── conf3.xyz
│   │   ├── conf4.xyz
│   │   ├── heatmap.png
│   │   ├── input_coords.xyz
│   │   ├── inputs.pickle
│   │   ├── obmol2d.png
│   │   ├── plot2d.png
│   │   ├── pmem42-delta-es-rmsds.png
│   │   ├── pmem42_energies.png
│   │   ├── ring.pickle
│   │   └── stats-file.txt
│   └── job_2_caffeine
│       ├── conf0.xyz
│       ├── conf1.xyz
│       ├── conf2.xyz
│       ├── conf3.xyz
│       ├── conf4.xyz
│       ├── heatmap.png
│       ├── input_coords.xyz
│       ├── inputs.pickle
│       ├── obmol2d.png
│       ├── plot2d.png
│       ├── pmem13-delta-es-rmsds.png
│       ├── pmem13_energies.png
│       ├── ring.pickle
│       └── stats-file.txt
├── kaplan_script.py
└── timer.dat


4 directories, 44 files
```

tree v1.8.0 © 1996 - 2018 by Steve Baker and Thomas Moore
HTML output hacked and copyleft © 1998 by Francesc Rocher
JSON output hacked and copyleft © 2014 by Florian Sesser
Charsets / OS/2 support © 2001 by Kyosuke Tokoro

Figure 3.3: Output directory tree structure, including example output file names, for *Kaplan* jobs run with `output_dir` set to `/home/user/experiment1`. This tree is shown assuming the top level directory is `user`.

- `no_ring_dihed [True]` if set to `True`, then any dihedral angles (*abcd*) where *b* and *c* are within a ring are removed from the list of dihedral angles for manipulation. This option should be kept as `True` until *Kaplan* has a procedure for manipulating rings.

- `min_dihed [True]` if set to `True`, a minimum set of dihedral angles are chosen for optimisation. If `False`, all possible dihedrals are chosen (excluding those from rings).

- `opt_init_geom [True]` optimise the initial geometry. If set to `False`, the initial geometry is not optimised. Optimisation falls into two categories: major and minor. This counts as a major optimisation.

- `avail_diheds [np.array([`$-\pi, -7\pi/8, -6\pi/8, \ldots, 7\pi/8$`])]` a *NumPy* array representing the possible dihedral values for any instantiated or mutated `pmem`s. Defaults to an array of 16 values in the half-open interval $[-\pi, \pi)$, as generated using the `np.linspace` function.

- `use_GOpt [False]` by default, *Openbabel*'s `SetTorsion` method from the `OBMol` class is used to apply torsion angles. If this input is set to `True`, then *GOpt* will instead be used to apply the dihedral angles. The difference here is that *Openbabel* applies the torsion angles one at a time, whereas *GOpt* applies the torsion angles all at once and tries to find a good geometry with such internal coordinates.

Convergence

These inputs change the stopping condition for *Kaplan*'s control module. Section 3.3 explains the `run_kaplan` function that is the main access point of *Kaplan*, which has its own inputs.

- `num_mevs [100]` $\{x \mid x \in \mathbb{N}\}$ how many mating events to complete. Each mating event represents one round of tournament selection, as well as possible application of extinction operators and a call to the output module.

- `stop_at_conv [False]` $\{x \mid x \in \{$`False, 0, 1, 2, ...`$\}\}$ if `False` (default) or 0, then the algorithm will run for a set number of `mevs`, as specified by the `num_mevs` input parameter. If `True`, then the algorithm checks the best `pmem` every *x* `mevs`. If there are no changes to the best `pmem`'s energies or RMSDs, then the algorithm will stop.

Energy and Optimisation

The Energy and Optimise modules can be expanded to include other programs. Availability for basis set and method is subject to the program in use. If the `prog` is set to *Psi4*, then *Openbabel* is still used to perform minor geometry optimisations with the MMFF94 forcefield.

- `prog [openbabel]` the program to use for energy calculations and major geometry optimisations, which are performed on the initial and final coordinates. Currently, the only available options are *Psi4* and *Openbabel*.

- `basis [STO-3G]` the basis set to use for the quantum chemistry energy evaluation. This option only applies if `prog` is *Psi4*. STO-3G stands for Slater-Type Orbitals

- `method [HF or MMFF94]` the quantum chemical method to use for energy evaluation (if `prog` is *Psi4*) or the forcefield to use (if `prog` is *Openbabel*).

- `major_tolerance [1e-6]` $\{x \mid x \in \mathbb{R}\}$ the maximum difference in energy that is permitted before a major optimisation is considered to have converged. If set to a negative value, then the optimisation is completed for `maxsteps` without consideration for the energy delta.

- `major_maxsteps [2500]` $\{x \mid x \in \mathbb{N}\}$ if the `tolerance` condition is never satisfied, represents how many iterations to complete in total for a major optimisation.

- `major_sampling [100]` $\{x \mid x \in \mathbb{N}\}$ how often to check the energy during a major optimisation for the `tolerance` in terms of number of conjugate gradient iterations.
- `minor_tolerance [0.1]` $\{x \mid x \in \mathbb{R}\}$ same description as `major_tolerance`, except for minor optimisations (i.e. those performed during evolution).
- `minor_maxsteps [100]` $\{x \mid x \in \mathbb{N}\}$ same description as `major_maxsteps`, except for minor optimisations.
- `minor_sampling [10]` $\{x \mid x \in \mathbb{N}\}$ same description as `major_sampling`, except for minor optimisations.

Ring

See Section 3.5.3 for more information.

- `num_slots [50]` $\{x \mid x \in \mathbb{N},\ x \geq$ `init_popsize`$\}$ how many slots to make for the `ring`.
- `init_popsize [10]` $\{x \mid x \in \mathbb{N},\ x \leq$ `num_slots`$\}$ how many slots in the `ring` that will be filled with one `pmem` at the start of evolution. Represents a contiguous block of slots.
- `mating_rad [3]` $\{x \mid x \in \mathbb{N}, x \leq$ `num_slots`$//2\}$ the mating radius determines how far away (in number of slots) from one `pmem` another `pmem` can be chosen for a mating event. Furthermore, the potential slots in which to place a child `pmem` are represented by the slots in the mating radius of the first chosen `pmem` (also called `parent1`).

Mutations

See Mutations module in Section 3.7.2 for more information.

- `num_muts [(num_geoms*num_diheds)//3]` $\{x \mid x \in \mathbb{Z}, 0 \leq x \leq$ `num_geoms*num_diheds`$\}$ the maximum number of point mutations for perform when generating children.
- `num_swaps [num_geoms//2]` $\{x \mid x \in \mathbb{Z}, 0 \leq x \leq$ `num_geoms`$\}$ the maximum number of swaps to perform when generating children.
- `num_cross [num_geoms//2]` $\{x \mid x \in \mathbb{Z}, 0 \leq x \leq$ `num_geoms`$\}$ the maximum number of n-point crossovers to perform when generating children, where n is chosen uniformly at random during crossover from the interval [1, `max_cross_points`]. If the `num_diheds` parameter is equal to 1, then `num_cross` is set to zero.
- `max_cross_points [num_diheds//3]` $\{x \mid x \in \mathbb{N}, 1 \leq x \leq$ `num_diheds-1`$\}$ the maximum number of crossover points that can be selected from a conformer. This value defaults to a third of the number of dihedral angles, except when there are fewer than three dihedral angles (in which case this value is set to 1).
- `cross_points [None]` where to split a list of dihedral angles during n-point crossover - should be a list of integers from the half-open interval [1, `num_diheds`). If this value is `None` (default), then the crossover points are chosen without replacement from the half-open interval [1, `num_diheds`) uniformly at random during each crossover event. This input parameter can be used to ensure that certain groups of dihedral angles are not split up during crossover.

Fitness

See Section 3.6 for more information.

- `fitg [0]` the fitness formula to use when evaluating `pmems`.
- `coef_energy [0.5]` $\{x \mid x \in \mathbb{R}, x \geq 0\}$ the constant coefficient value, C_E, for the energetic term in the fitness function.

- `coef_rmsd [0.5]` $\{x \mid x \in \mathbb{R}, x \geq 0\}$ the constant coefficient value for the root-mean-square deviation (RMSD) term, C_{RMSD}, in the fitness function.
- `normalise [True]` if this is set to `True`, then the fitness of each `pmem` is calculated using a z-score metric that normalises energies and RMSD values based on the current `ring` population. If `False`, then a `pmem`'s fitness is a static value that combines energy and RMSD according to the fitness formula (set by `fitg`).
- `exclude_from_rmsd [None]` if the user wishes to exclude some atom types from the RMSD calculation, then this input should be given as a list of atomic numbers to exclude. For example, to exclude hydrogen atoms, this variable should be set to `[1]`. This input is helpful in excluding high-energy conformers with high RMSD where the only alteration is the placement of the hydrogens.

Extinction

See Section 3.5.4 for more information. By default, no extinction operators are used during *Kaplan* execution.

- `asteroid [0.0]` the percent chance that an asteroid operator is applied to the ring, per `mev`.
- `plague [0.0]` the percent chance that a plague operator is applied to the ring, per `mev`.
- `deluge [0.0]` the percent chance that a deluge operator is applied to the ring, per `mev`.
- `agathic [0.0]` the percent chance that an agathic operator is applied to the ring, per `mev`.

3.4.2 Internal Inputs

Internal inputs are those generated by *Kaplan*, which implies that the user should not change them or input them as arguments. However, if the user wishes to alter the set of dihedrals being optimised, they can be changed after calling `update_inputs` - see Section 3.4.3.

- `atomic_nums` the atomic numbers for each atom in the molecule. The atoms appear in the same order as the input_coords.xyz file.
- `coords` the coordinates for the original geometry (in input_coords.xyz) as an `np.array((n,3), float)`, where `n` is the number of atoms in the molecule.
- `num_diheds` $\{x \mid x \in \mathbb{N}\}$ the number of dihedral angles that will be optimised during the conformer search. If *Kaplan* cannot find any dihedral angles to optimise, then `kaplan.inputs.InputError` will be raised.
- `diheds` $\{(\texttt{side\_i}, i, j, \texttt{side\_j})\}$ a list of length `num_diheds`, which represents a selection of dihedral angles, comprising of atom indices (indices correspond to the same order as found in input_coords.xyz) - the selection process is explained in Section 3.5.1.
- `obmol` an *Openbabel* molecule object that is used internally to apply torsion angle updates and run energy calculations and geometry optimisations. This object is generated by reading the input_coords.xyz file from the output directory.

3.4.3 Working with *Kaplan* Inputs

The `kaplan.inputs.Inputs` class is used by most modules in *Kaplan* (see stars in Figure 3.2). If the user passes a dictionary to the `run_kaplan` function to run *Kaplan*, then the `inputs` object is created by default. To create an `inputs` object, the following steps apply:

```
from kaplan.inputs import Inputs
inputs = Inputs()
inputs.update_inputs({
    "struct_input": "aspartate",
    # add any other inputs here
})
print(inputs)
```

Because the `inputs` object follows a `Borg` procedure (also called Monostate pattern - see this link for details https://github.com/faif/python-patterns/blob/master/patterns/creational/borg.py), any `inputs` object that is created in the same *Python* script will share all of the same input parameters. Furthermore, if any one of the `inputs` objects are modified, then the other `inputs` objects will be modified accordingly. Any time that the `update_inputs` method is called, all of the input parameters are reset to their default values (including resetting the geometry, re-selecting dihedral angles, and recreating an `obmol` object). If the user has modified the `inputs` object and wishes to verify the input parameters, then it is sufficient to call `inputs._check_input()`.

By default, `run_kaplan` will write an inputs.pickle file to the output directory after completing a *Kaplan* job. The user may wish to view the set of inputs used to run *Kaplan*, especially the dihedral angles selected for optimisation. The `inputs` object can be loaded in one of two ways. First, by using the `pickle` library directly:

```
import pickle
with open("inputs.pickle", "rb") as f:
    inputs = pickle.load(f)
print(inputs)
```

Or, second, using the inputs function `read_input`, which also re-generates the `obmol` *Openbabel* molecule object[1]. In this manner, the user can dynamically change the inputs and start a new job.

```
from kaplan.inputs import read_input
```

[1] These objects cannot be pickled.

```
inputs = read_input("inputs.pickle", False)
print(inputs)
```

Kaplan also writes the contents of inputs.pickle to the output directory as a plain text file called `inputs.txt` for convenience.

3.5 Representation in *Kaplan*

Kaplan uses an evolutionary algorithm (EA) to determine the best sets of dihedral angles for a molecule such that a set of low-energy geometries are procured. A population member (`pmem`) is a potential solution to the conformer searching problem that lives in a `ring`. A `pmem` object contains a set of geometries representing conformers. Each conformer is described using dihedral angles, which are combined with the original geometry to find the corresponding Cartesian coordinates.

3.5.1 Extracting Dihedral Angles from Cartesian Coordinates

The program *GOpt*, which was written by Xiaotian (Derrick) Yang based on the Ph.D. thesis by Sandra Rabi [79], is used to select dihedral angles for the input molecule. There are two options for the selection, controlled by the `min_dihed` input variable. If `min_dihed` is set to `False`, then there is a possibility of having more than one dihedral angle per rotatable bond.

First, the Cartesian coordinates are converted into internal coordinates (bond lengths and bond angles). From the internal coordinates, a connectivity matrix A_{n_a} (which is square and symmetric) can be produced. The elements of this square matrix, x_{ij}, indicate whether atoms i and j are:

-1. $i == j$, i.e. along the diagonal,

0. not connected,

1. connected via covalent bond,

2. connected via hydrogen bond,

3. connected via inter-fragment bond,

4. connected via auxiliary bond, or

5. connected via linear chain

Below is the pseudocode for selecting a minimal set of dihedral angles from an existing geometry. The connectivity matrix is given by `cmat`. The variable `num_connect` is a list of length `num_atoms`, where each element `c` represents how many other atoms in the molecule are connected to atom `c`. The function `num_unique_connect(i, j)` takes two atom indices `i` and `j` as input, and returns the number of distinct atoms that are connected to the input atoms (counting the input atoms), discounting connections of type 4. The function `biggest_branch(i, j)` takes two atom indices `i` and `j` as input, and returns a tuple of atom indices, representing the two largest (in terms of number of connected atoms) groups attached to atoms `i` and `j` respectively. Note that `biggest_branch` cannot return either input atom as output.

```
given cmat, num_connect
let dihedrals = list()
for i in range(num_atoms-1):
    for j in range(i+1, num_atoms):
        if any(
            cmat[i][j] in [-1, 0, 4],
            num_connect[i] < 2,
            num_connect[j] < 2,
            num_unique_connect(i, j) < 4
        ):
            continue
        side_i, side_j = biggest_branch(i, j)
        dihedrals.append([side_i, i, j, side_j])
return dihedrals
```

In other words, the selection algorithm considers all possible pairs of atoms (the upper diagonal of the connectivity matrix). Pairs that are disconnected, have fewer than 2 connections to either atom, consist of the same atom twice, or whose

set of connections is smaller than 4 atoms are excluded. Consider the example of given in the introduction, Figure 1.3, of 1,2-difluoroethane. None of the hydrogen or fluorine atoms have more than 1 connected atom; therefore, the selected dihedral angles will not contain hydrogens or fluorine atoms in the `i` or `j` positions. By only selecting the upper diagonal of the connectivity matrix, the dihedrals list will not contain the reversed version of any dihedrals found so far (i.e. `[side_j, j, i, side_i]`).

The molecule ethyne would contain one dihedral as per this algorithm, since the total number of unique connections between the two carbon atoms would be 4. Water would not contain any dihedrals, since $\nexists$ 2 connected atoms with at least two connections each.

The `biggest_branch` function sorts the connected atoms by how many other atoms are connected, but it doesn't distinguish between groups. For example, given the molecule 4-ethylheptane (shown in 3.4), a choice will be made as to whether to consider the ethyl or n-propyl group as part of a dihedral. Since the atoms branching from the centre carbon atom both have four connections, it will depend on the input geometry as to how the `side_j` is chosen.

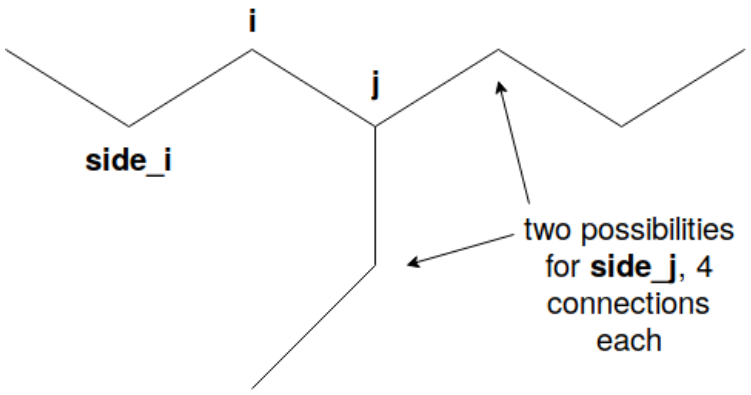


Figure 3.4: The last part of the dihedral `side_j` will be chosen based on the order that the input atoms are read from the xyz file.

By choosing a minimal set of dihedrals, the code will only have to optimise a given subset of the total possibilities, many of which are redundant. Furthermore,

by restricting the number of possible dihedral values (via the `avail_diheds` input parameter), the search space that the `pmems` explore for the best solution is contracted.

3.5.2 Example `pmem`

The `pmem` module is responsible for generating initial solution instances to the conformer searching problem. Each `pmem` is an instance of the `kaplan.pmem.Pmem` class. The `pmem.setup` method calculates the energies and RMSDs of a `pmem`'s conformers; this method takes one argument: `major`, which is a boolean value that specifies if the conformer optimisations (performed prior to energy calculation) are to be major or minor. The return value for this method is a list of geometries for the optimised conformers (although these geometries are not stored in the `pmem` object). The `pmem` object has the following attributes:

- `ring_loc`: $\{x \mid x \in \mathbb{Z},\ 0 \leq x < \texttt{num\_slots}\}$ the slot number where the `pmem` is located in the `ring`.
- `num_geoms`: $\{x \mid x \in \mathbb{N}\}$ the number of conformers in the `pmem`.
- `num_diheds`: $\{x \mid x \in \mathbb{N}\}$ the number of dihedral angles per conformer.
- `dihedrals`: `np.array((num_geoms, num_diheds), "float")`. Each element in the array is a value from `avail_diheds`. The units are radians.[2].
- `energies`: `list` $\{x \mid x \in \mathbb{R}\}$ a list of length `num_geoms`, where each element is an energy value for the corresponding, optimised conformer geometry. The units are Hartrees for *Psi4* and kcal/mol for *Openbabel*.
- `rmsds`: `list` $\{i, j, RMSD\}$ is a list of length `num_pairs`, where each element is a list of length three containing the indices for the two geometries and their RMSD value.

[2]When printing the dihedral angles of a `pmem` (i.e. `print(pmem.dihedrals)`), the values are given in radians. When printing a `pmem` object, the dihedrals are given in degrees.

- `fitness`: $\{x \mid x \in \mathbb{R}\}$ the `pmem`'s fitness.
- `birthday`: $\{x \in \mathbb{Z} \mid x \geq 0\}$ the `mev` during which the `pmem` was created.

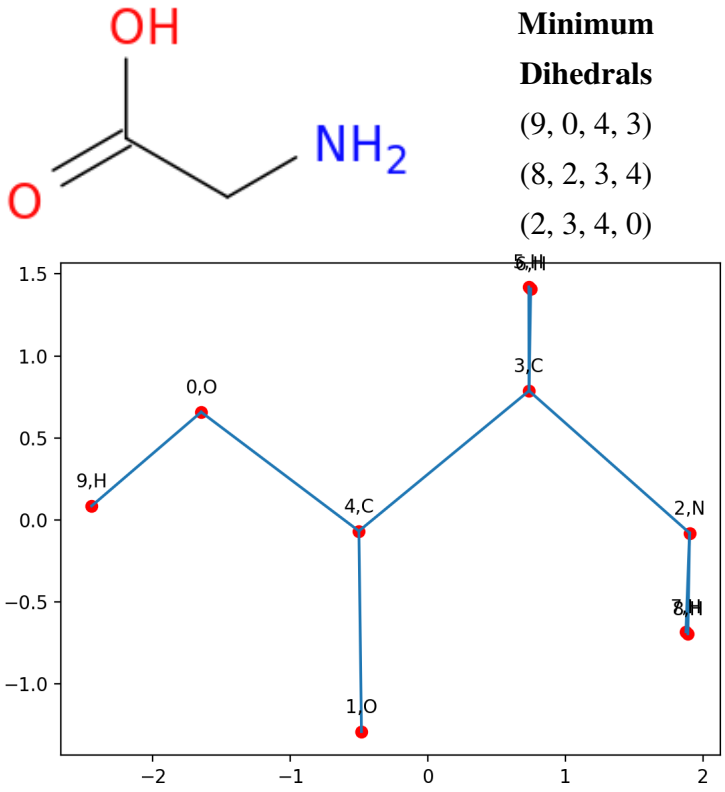


Figure 3.5: The 2D representation of glycine. The top left figure is generated using *Openbabel*. The top right is a list of the dihedral angles by their atom indices, which correspond to the labelled plot (bottom). The atom labels indicate the type of atom and the atom index.

Each `pmem` has `num_geoms` sets of dihedrals of length `num_diheds`. Each dihedral is selected uniformly at random from `avail_diheds`. To evaluate the energy of each conformer, each set of dihedrals is applied to the original starting geometry to produce a new set of Cartesian coordinates, which are subsequently optimised. The original geometry is stored in the form of an *Openbabel* `obmol`

object, which also contains internal coordinates.[3] The geometry module contains some functions that allow the user to interact with the `obmol` object.

The `pmem` object is iterable - it supports the *Python* syntax:

```
for conformer in pmem:
    print(conformer)
```

where `conformer` is a set of dihedral angles. The output of such code would be a list of dihedral angles in radians for each conformer. Example print output for an entire `pmem` (i.e. `print (pmem)`) is given in the sample output below.

Glycine is the simplest amino acid, given that its R-group is a hydrogen atom. In *Kaplan*, glycine is represented by 3 dihedral angles, consisting of the following atom indices: (9, 0, 4, 3), (8, 2, 3, 4), and (2, 3, 4, 0). The xyz file used to form these dihedral angles can be found in the appendix, Section A.2.4. The `pmem` for glycine, whose `birthday` and `ring_loc` are both set to zero, would look like:

```
from kaplan.inputs import Inputs
from kaplan.pmem import Pmem
from kaplan.fitness import set_absolute_fitness

inputs = Inputs()
inputs.update_inputs({
    "struct_input": "glycine",
    "normalise": False,
})

pmem = Pmem(0, 0, inputs.num_geoms, inputs.num_diheds)
pmem.setup(major=False)
set_absolute_fitness(
    pmem, inputs.fit_form,
    inputs.coef_energy, inputs.coef_rmsd
)
print(pmem)
```

[3] Due to the nature of the `obmol` object, this object must be regenerated for each new *Python* session using the inputs module.

which would output something similar to the following:

```
Slot #: 0
    Dihedrals 0: 90.00 157.50 -90.00
    Dihedrals 1: 157.50 45.00 -22.50
    Dihedrals 2: -180.00 0.00 22.50
    Dihedrals 3: -112.50 -67.50 135.00
    Dihedrals 4: 112.50 112.50 -157.50
    Birthday: 0
    Fitness: -55.98832767305205
    Energies: [
        26.941099234730288,
        22.921684404313204,
        23.071323079347184,
        22.592396424851092,
        26.221170205432937
    ]
    RMSDs: [
        [0, 1, 0.8787568346659916],
        [0, 2, 1.1583605382017812],
        [0, 3, 1.2222862904592282],
        [0, 4, 0.7581811868541476],
        [1, 2, 0.4503826385564213],
        [1, 3, 1.1406345834614158],
        [1, 4, 1.0116758128187684],
        [2, 3, 0.8734795100131609],
        [2, 4, 1.2654522303924076],
        [3, 4, 1.011808377147288]
    ]
```

Kaplan also has the ability to make figures (examples in Figure 3.5 and Figure 3.6) where the atoms are labelled. These figures are crude representations so that the dihedrals may be identified. The numbering is a direct result of the atom ordering from the xyz input file. Regardless of the type of input provided to *Kaplan*, an xyz

file is generated as to keep a record of the atom ordering. For larger molecules, it is likely necessary to view the structure in, for example, *VMD*, where the user can colour atoms by their index. Note that, in the example 2D figure for glycine, the labels are overlapped for hydrogens 5 and 6 (connected to 3C) and hydrogens 7 and 8 (connected to 2N), which is a direct consequence of converting the plot from 3D to 2D coordinates by simply removing the z-coordinates.

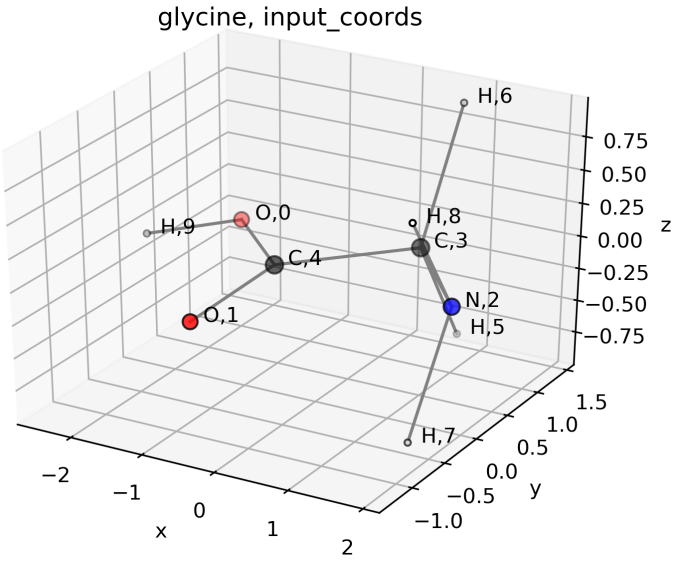


Figure 3.6: A 3D plot for glycine, generated using *Kaplan*'s Tools module.

3.5.3 Ring Representation

In *Kaplan*, the `ring` is an object. One of its methods is called `update`, which describes the protocol for adding potential solutions to the existing population. `ring.update` takes 2 arguments: the `child pmem`, and the `potential_slot`. Before sending a `pmem` to the `ring`, it must have its energies and RMSD values calculated (via `pmem.setup`) and its fitness set. The Tournament module is responsible for preparing `child pmems` prior to a `ring.update`, as well as ensuring the `pmem` at `potential_slot` (if it exists) has an updated fitness. Fitness updates are necessary when the input parameter `normalise` is `True`, since the mean and standard deviations of energies and RMSD values may change from `mev`

to `mev`.

Consider a `child` `pmem` with fitness 100 and a `potential_slot` of 5. If the `ring` at slot 5 is empty, the child is placed there (`ring[5] = child`). Otherwise, the `pmem` at slot 5 must have a fitness $\leq$ 100 in order for the `child` to occupy the slot (`ring[5] = child if ring[5].fitness` $\leq$ `100`).

The `ring` object is iterable, which means that it supports the *Python* syntax `for pmem in ring`. It also has support for printing methods, such that `print(ring)` results in each slot being reported in human-readable format.

3.5.4 Extinction Operators

Another addition to the `ring` is to add extinction operators [82]. As of writing, there are four extinction operators in *Kaplan*: `asteroid`, `deluge`, `plague`, and `agathic`. These operators are intended to reset the `ring`'s population to a more exploratory state, and encourage new growth by deleting segments of the `ring`. Each operator has its own criteria for how to remove existing `pmems`. In *Kaplan*, the operators are designed such that a minimum of one `pmem` remains after an extinction event.

The application of each extinction event is based on a random chance, once per `mev`. For example, consider a mating event where the `asteroid`, `deluge`, `agathic`, and `plague` inputs are set to 0.01, 0.05, 0.1, and 0.4, respectively. Then, during the `mev`, the following random numbers (from the range 0-1) are chosen: 0.6, 0.4, 0.5, 0.2. For this `mev`, only the `plague` extinction will occur, since $0.2 < 0.4$.

Asteroid

The `asteroid` operator is a fitness-agnostic extinction event that deletes a contiguous segment of the ring of random size. The size, in number of slots, is chosen uniformly at random between 10-90% (inclusive) of the total number of slots. The `asteroid` location is chosen by randomly picking a slot (empty or filled) until the resulting `asteroid` would not kill the entire population. For example, in a `ring`

of size 20, with 5 filled slots (indexed 0-4, inclusive), an `asteroid` of size 5+ slots could not start at index 0, as this would kill the entire population in the `ring`.

Deluge

The `deluge` operator kills individual `pmem`s based on the `ring`'s current maximum fitness value. First, a water-level is chosen uniformly at random from 10-90% (inclusive) of the maximum fitness. Then, all of the `pmem`s whose fitness value is lower than the water level are killed.

Plague

The `plague` operator is similar to the `deluge` operator in that it is elitist with respect to fitness. The `plague` operator kills a fraction of the population based on the number of slots, which avoids the problem (i.e. in a deluge extinction event) whereby a large fraction of the population is sitting just above the water-level. To apply a plague extinction event, the population is first sorted according to fitness. A fraction of the number of slots, chosen uniformly at random between 10-90% (inclusive), is killed that represents the lowest-fitness `pmem`s.

Agathic

The `agathic` operator requires that each `pmem` have an age. In *Kaplan*, the age of a `pmem` is the number of `mev`s that have passed since the `pmem` was added to the ring. It is tracked by giving each `pmem` a `birthday` upon instantiation. Similar to the `plague`, this operator kills a fraction of the population, except the sorting occurs based on age. The fraction of the total size is chosen uniformly at random between 10-90% (inclusive), and then the oldest `pmem`s are killed.

3.6 Evaluation in *Kaplan*

The Fitness module is responsible for calculating a `pmem`'s fitness; the fitness can be based on the population (when `normalise` is `True`), or based on the individual

`pmem` (`normalise` is `False`). During a *Kaplan* conformer search, the Fitness module is accessed by the Control and Tournament modules. Fitness is evaluated for all `pmem`s prior to `plague` and `deluge` extinction events, and `pmem`s selected during a tournament have their fitness values updated.

Only one fitness function is implemented in *Kaplan* (represented by `fitg = 0`). If adding new fitness functions to *Kaplan*, the normalised and absolute cases must be considered and implemented in the Fitness module.

The evaluation of `pmem`s in *Kaplan* is based on the following fitness function:

$$fit_G = C_E \sum_{i=1}^{n_G} -E_i + C_{RMSD} \sum_{i=1}^{n_G} \sum_{j>i} RMSD_{ij} \tag{3.1}$$

The fitness is a maximisation function; therefore, the energy of each conformer geometry E_i is negated, which accounts for the cases where the energy evaluation is positive (as occurs in forcefield evaluations for some molecules) or negative (quantum chemical energies are essentially always negative). $RMSD_{ij} \in \mathbb{R}$ is the root-mean-square deviation between conformers i and j. The sum of conformer energies and the sum of RMSD values are subject to coefficients C_E and C_{RMSD}, respectively. These coefficients can be specified by the user to control the importance of energy versus RMSD in the evaluation of conformer geometries. There are n_G conformers in each `pmem`. The number of RMSD terms is equal to the number of possible ways that two `pmem`s can be chosen:

$$n_{pairs} = \binom{n_G}{2} = \frac{n_G!}{2!(n_G-2)!} = \frac{n_G(n_G-1)(n_G-2)!}{2(n_G-2)!} = \frac{n_G^2 - n_G}{2} \tag{3.2}$$

One of the options for *Kaplan* is to use normalisation as to make the coefficients more intuitive. Turning on normalisation changes the form of the fitness function to consider all current population members in the `ring`. Each energy and RMSD value is normalised according to its z-score, where *stdev* refers to standard deviation:

$$E_{iN} = \frac{E_i - E_{mean}}{E_{stdev}} \quad ; \quad RMSD_{ijN} = \frac{RMSD_{ij} - RMSD_{mean}}{RMSD_{stdev}} \mid E_i, RMSD_{ij} \neq \texttt{None} \tag{3.3}$$

If the energy evaluation fails to converge (usually during a quantum chemical evaluation), the energy of the offending conformer is set to `None`. In future work, it may be necessary to handle cases where the geometry of a given conformer cannot be constructed, but the author has yet to encounter any of such cases. Without a geometry, any RMSD calculations related to said geometry should also be set to `None`. The E_i and $RMSD_{ij}$ values that are considered in the E_{mean}, E_{stdev}, $RMSD_{mean}$, and $RMSD_{stdev}$ therefore follow the rules that they are (1) part of the current `ring` population, and (2) that they are not equal to `None`.

A penalty is applied to the fitness for each bad geometry possessed by a `pmem` (i.e. one that has its energy set to `None`). When `normalise` is `False`, the maximum energy of the `pmem` is considered. If this maximum ≤ 0, then the `None` value is ignored (essentially setting E_i to 0). If the maximum is positive, then E_i is set to four times the `pmem`'s maximum energy value. When `normalise` is `True`, an energy of `None` is replaced by negative four times the standard deviation (essentially placing it at the far left of the normal distribution). Any RMSD values that are `None` are simply ignored from the fitness (since their values would then contribute 0 to the fitness, the worst possible RMSD).

The fitness function with normalisation becomes:

$$fit_{GN} = \frac{C_E}{n_G} \sum_{i=1}^{n_G} -E_{iN} + \frac{C_{RMSD}}{n_{pairs}} \sum_{i=1}^{n_G} \sum_{j>i} RMSD_{ijN} \tag{3.4}$$

3.6.1 Energy Module

The *Kaplan* Energy module is responsible for determining the energy of a set of input coordinates. The external packages *Psi4*[95][100] and *Openbabel*[68][107] are used, as well as *Vetee*'s `periodic_table` function.

There are two exception classes in the Energy module: `BasisSetError` and `MethodError`. The Inputs module does not check the basis set or method; during

the execution of the control module, an energy calculation is attempted with the current inputs, and error handling is used to determine whether said basis set and/or method exist in the chosen program.

Access to this module is accomplished through the `run_energy_calc` function, which takes a set of coordinates (`np.array((n,3), float)`) as input. The `hardware_inputs` dictionary from the Inputs module is used to set the amount of RAM available on the computer running *Kaplan*. Currently, this only impacts *Psi4* energy calculations.

A *Psi4* energy calculation invokes two functions; first, `prep_psi4_geom` is called to generate an input string for the *Psi4* program. Then, `psi4_energy_calc` is called to actually calculate the energy. The default energy unit for *Psi4* calculations is Hartrees. The *Psi4* energy calculation requires that the following input parameters are set: `atomic_nums`, `method`, `basis`, `multip`, and `charge`.

An *Openbabel* energy calculation first checks that the input coordinates match those of the `obmol` object from the Inputs module. If there is a discrepancy, then the `obmol` object is updated to the input coordinates. Then, the `obabel_energy` function is called, which evaluates the energy of the `obmol` object. Therefore, it is important that the `obmol` object has its charge, multiplicity, and coordinates set. The `method` is also required to be set for the *Openbabel* energy calculation to work. The method here represents the forcefield to use, and not a quantum chemistry method. The default energy units for *Openbabel* are kcal/mol.

If the `prog` variable is not *Psi4* or *Openbabel*, then a `NotImplementedError` will be raised. If a user wishes to expand this module, then arguments can be added to the `inputs.extra` dictionary and handled within the `run_energy_calc` function. It is recommended that each `inputs.prog` option get at least one function for calculating energy.

3.6.2 Optimise Module

The Optimise module is responsible for optimising molecular geometries. A major optimisation is performed for the input coordinates (unless `opt_init_coords`

is set to `False`), and for the final best `pmem` geometries before they are written to output files. Major optimisations are performed using conjugate gradients (if `prog` is *Openbabel*) or the `OptKing` program (part of *Psi4*). A minor optimisation is performed for each conformer in new `pmem`s that are generated during evolution or during the initial `ring` filling. Regardless of the `prog`, minor optimisations are performed with a forcefield; if `prog` is *Psi4*, then the forcefield is MMFF94 by default. Energies for *Psi4* jobs are still calculated using the Energy module.

As with the Energy module, the Optimise module has a main function called `optimise_coords`, which takes an array of coordinates as input and a boolean value, `major`. If `major` is True, a major optimisation is performed, and vice versa. If the `prog` is anything other than `Openbabel` or `Psi4`, then a `NotImplementedError` will be raised. The `obmol` object is manipulated as in the Energy module to carry-out the forcefield optimisation. For *Psi4* optimisations, the `prep_psi4_geom` function is used from the Energy module to format the coordinates, charge, and multiplicity.

From Section 3.4.1, there are three input parameters that change the optimisations: `tolerance`, `maxsteps`, and `sampling`. Note that the major optimisations with *Psi4* do not use these input parameters. For all other cases, the `tolerance` is the difference in energy that is compared against every `sampling` iterations of the conjugate gradients optimisation. If the energy delta is smaller than `tolerance`, then the geometry is said to have converged. Conjugate gradients will run for a maximum of `maxsteps` before the function returns the coordinates. The exact implementation is given in the `obabel_geometry_opt` function from the Optimise module. This function will require an update when the next version ($>$ 2.4.1) of *Openbabel* is released, as the `sampling` parameter will no longer be a necessary input.

3.6.3 RMSD Module

The *Kaplan* RMSD module has two functions: `calc_rmsd`, and `apply_centroid`. The `calc_rmsd` function takes two sets of coordinates as input (both are

`np.array((num_atoms,3), float)`), whereas the `apply_centroid` function applies to one set of coordinates. This module uses *NumPy*[69][99] and the *rmsd* module from Github user *charnley* [106]. The Kabsch algorithm, as mentioned in Section 1.2.3, is used to calculate the RMSD between two geometries.

3.7 Regeneration in *Kaplan*

There are two components to regeneration in *Kaplan*: selection and mutation. A version of tournament selection is implemented in the tournament module, and new child `pmems` are made from mutation operators in the mutations module.

3.7.1 Tournament Module

Tournament selection in *Kaplan* works as follows:

1. Choose `parent1` at random from the currently occupied slots. Let this slot be slot i.

2. If i is the only occupied slot, perform `single_parent_mutation` on `parent1`, generate 1 child `pmem`, and call `ring.update` using an empty slot in the mating radius of `parent1`. The tournament is now terminated.

3. Otherwise, using the mating radius (r_m), determine the slots that are now in the tournament. These slots are represented by $\{i-r_m, i-r_m+1, ..., i-1, i+1, ..., i+r_m-1, i+r_m\}$. The slots will wrap around the ring if necessary.

4. Order the tournament members by fitness from lowest to highest. If any slots in the tournament are empty, then these slots are placed at the start of the sorted list.

5. Set `parent2` to be the slot with highest fitness in the tournament. Note: the tournament does not include i (`parent1`), which means that lower-fitness `pmems` have a chance to be chosen as a parent.

6. Set the first two slots in the sorted tournament (the slots that are empty or have lowest fitness) to be the "worst" slots.

7. From `parent1` and `parent2`, generate children using the mutations module (see Section 3.7.2).

8. Call the `ring.update` method with the two new child `pmems` and the worst slots as inputs.

3.7.2 Mutations Module

The `mutations` module has two main functions: `generate_children` and `single_parent_mutation`, the latter of which is only called when one `pmem` is left in the `ring` (perhaps, for example, as a result of an extinction event, which can destroy all but one of the `ring`'s population). The `generate_children` function takes seven arguments: `parent1, parent2, num_muts, num_swaps, num_cross, max_cross_points`, and `cross_points`. The `parent1` and `parent2` inputs are numpy arrays with shape `(num_geoms, num_diheds)` - essentially representing the dihedral angles for each conformer in the parent. The other inputs are the same as those described in Section 3.4.1. `cross_points` defaults to `None`, but the other inputs are required.

The generate_children function proceeds as follows:

1. Make a deepcopy (see appendix, Section A.1.5) of `parent1` called `child1`.

2. Make a deepcopy of `parent2` called `child2`.

3. Randomly choose how many swaps to perform. This value (`swaps`) is between 0 and `num_swaps`, inclusive.

4. Randomly choose how many *n*-point crossovers to perform. This value (`crosses`) is between 0 and `num_cross`, inclusive. For each crossover event, the number of points to crossover (the value of *n*) is randomly selected from the interval [1, `max_cross_points`].

5. Twice (once per child), choose a number of point mutations to perform. These values (`muts1` and `muts2`) are random integers between 0 and `num_muts`, inclusive.

6. Apply swap operator `swaps` times to the child `pmems`.

7. Apply the crossover operator `crosses` times to the child `pmems`.

8. Apply the mutate operator `muts1` times to `child1`.

9. Apply the mutate operator `muts2` times to `child2`.

10. Return the mutated copies of `parent1` and `parent2`, `child1` and `child2`.

If all three operators are inactive (i.e. set to 0 in the inputs module), then no changes are made to `parent1` and `parent2`.

The notation in Figures 3.7, 3.8, and 3.9 have 1-based indexing. Each rectangle denotes a set of dihedral angles - based on $D_{i,j}$, where i is the `pmem` number (in this case representing `child1` or `child2`) and j is the conformer number (in these `pmems`, the number of conformers is 5). Each square represents a dihedral angle - in these figures, there are 6 dihedral angles per geometry.

In *Kaplan*, the three types of mutation operators are performed without replacement, which means that once a conformer (`swap`), set of dihedrals (`crossover`), or dihedral angle (`mutate`) has been altered, it does not get altered again during the tournament process.

Swap Operator

The `swap` operator takes two `pmems` and two indices as input. A single swap can be seen in Figure 3.7, where conformer 2 from pmem1 and conformer 2 from pmem2 swap places. The swap can be thought of as two solutions trading individual conformers. Note that the swap location does not have to be the same for both `pmems` - see the pseudocode below.

```
given pmem1, pmem2, swaps
let indices1 = range(num_geoms)
let indices2 = range(num_geoms)
for i in range(swaps):
    randomly choose choice1 from indices1
    randomly choose choice2 from indices2
    exchange conformers at pmem1[choice1]
    and pmem2[choice2]
    remove choice1 from indices1
    remove choice2 from indices2
return pmem1, pmem2
```

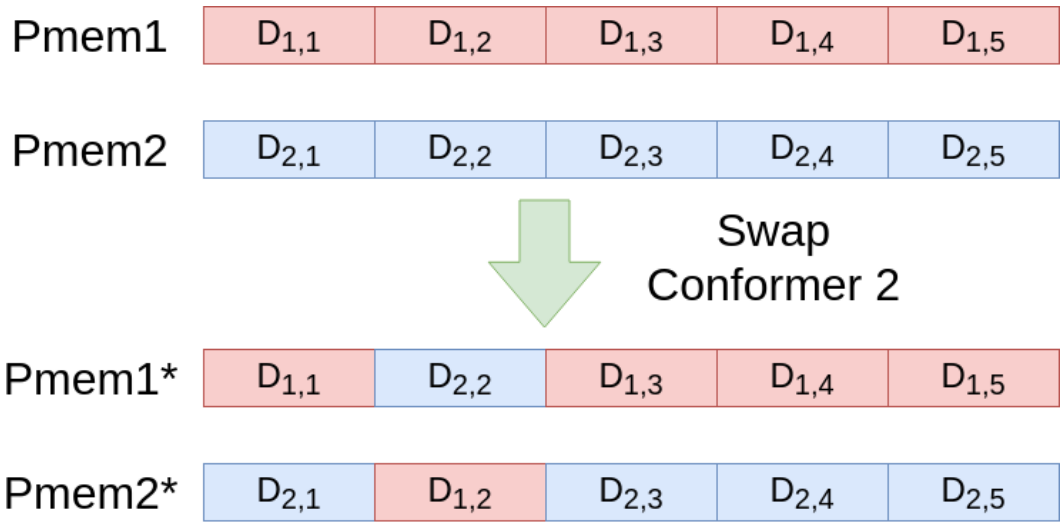


Figure 3.7: The swap mutation exchanges two geometries, where one geometry from `parent1` is swapped for another geometry in `parent2`.

Crossover Operator

The `crossover` operator is similar to `swap`, except the dihedral angles themselves are being exchanged between `pmems`. After this operator is applied, the `child pmems` will have a combination of dihedral angles from both `parent pmems` for one or more geometries, depending on how many times crossover is applied (`crosses`).

```
given pmem1, pmem2, crosses
let indices1 = range(num_geoms)
let indices2 = range(num_geoms)
for i in range(crosses):
    randomly choose choice1 from indices1
    randomly choose choice2 from indices2
    randomly choose index_s from range(1, num_dihed-1)
    combine1 pmem1[choice1] until index_s-1 and
             pmem2[choice2] from index_s
    combine2 pmem2[choice2] until index_s-1 and
             pmem1[choice1] from index_s
    pmem1[choice1] = combine1
    pmem2[choice2] = combine2
    remove choice1 from indices1
    remove choice2 from indices2
return pmem1, pmem2
```

Kaplan implements *n*-point crossover, which means that *n* crossover indices `index_s` are chosen. The above pseudocode applies to single-point crossover, but this process can be easily repeated for a new value of `index_s`. For example, two-point crossover would require 2 crossover indices to be chosen. Then, the pieces would be reconstructed as $R_1 - B_2 - R_3$ and $B_1 - R_2 - B_3$, where $R_1 - R_2 - R_3$ represents the chosen set of dihedrals from `pmem1` and $B_1 - B_2 - B_3$ represents the chosen set of dihedrals from `pmem2`. Both sets of dihedrals in this example are split in two arbitrary places to afford 3 subsection each. Uniform crossover occurs when each `pmem` is split at every dihedral angle (where *n* is equal to `num_diheds`-1).

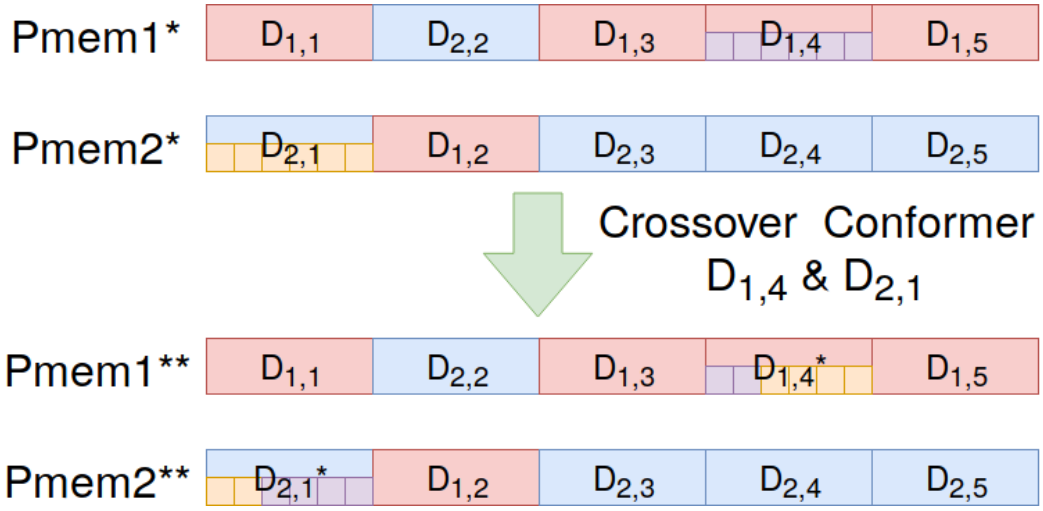


Figure 3.8: The crossover mutation changes two geometries. In this example, one geometry from `parent1` undergoes single-point crossover with another geometry in `parent2`, and `index_s` cuts the set of dihedral angles between the second and third angles.

The purpose of the `cross_points` input parameter is to allow for groups of dihedral angles to move together. For example, if the first, second, and fourth dihedral angles were found to be correlated (i.e. facilitate a beneficial electrostatic connection, or prevent a poor interaction), a `pmem`'s fitness may benefit from keeping them together. Therefore, (for a 6-dihedral geometry) the `cross_points` parameter could be set to `[2, 3, 4]`, `[2, 3, 5]`, `[2, 3]`, `[4]`, `[5]`, or `[4, 5]` to keep those dihedral angles together.

Mutate Operator

The `mutate` operator is the simplest of the three mutation operators; a dihedral angle is given a new value within one `pmem`. In the `single_parent_mutation` function, the mutate operator is applied `muts1` times to `parent1`. Mutate consists of setting a dihedral angle to a random value, as selected from the array, `avail_diheds`, from the Inputs module. This operator can therefore be a `null` operator, since there is a non-zero probability that the new dihedral angle will be

the same as the current angle.

In the example in Figure 3.9, `pmem1` is mutated at conformer 1, dihedral angle 4, whereas `pmem2` is mutated at conformer 2, dihedral angle 6.

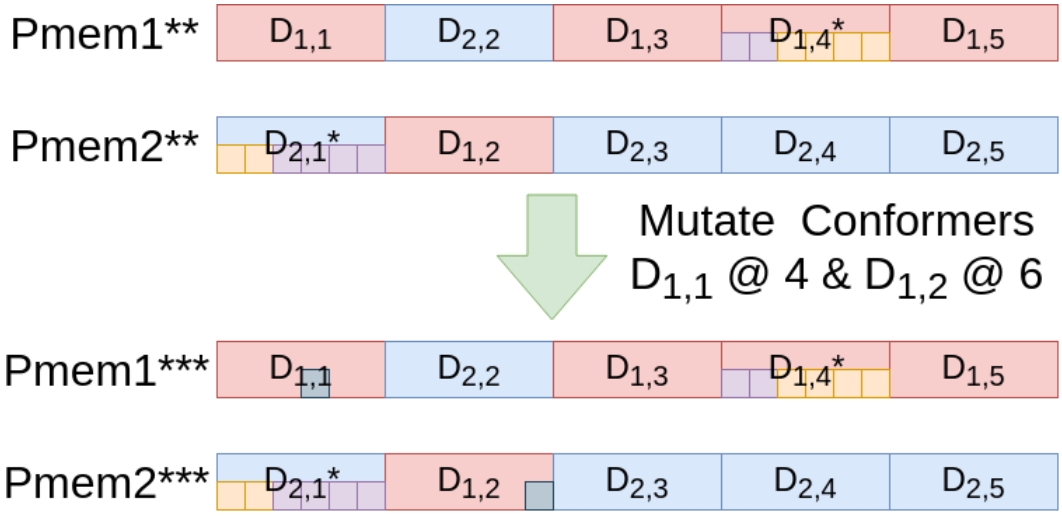


Figure 3.9: The mutate mutation applies to a single geometry, where one random dihedral angle is changed for a new value. The new value is in radians, selected from `avail_diheds`.

3.8 Rings in *Kaplan*

During the development of *Kaplan*, molecules with rings were found to have no flexibility and conformer searches involving rings were disproportionately difficult. This section will discuss the author's investigation of cyclobutane and cyclohexane, and explain why the current approach does not work for sampling ring conformations. Important note: the version of *Kaplan* used to perform the tests from this section did not yet include any form of optimisation.

3.8.1 Cyclobutane

Cyclobutane is a good test molecule, because it is a small constrained ring that has a dihedral without hydrogens. The structure of cyclobutane, on average, is non-

Dihedral Atoms	Angle (°)	Angle (rad)
(2, 0, 1, 3)	-25.086	-0.43783
(1, 0, 2, 3)	25.088	0.43787
(0, 1, 3, 2)	25.088	0.43786
(0, 2, 3, 1)	-25.090	-0.43791

Table 3.1: The set of minimum dihedral angles for cyclobutane's original structure (from *PubChem*), with values rounded to five significant figures.

planar (puckered), with an intermediate planar conformation [2][4][8]. A 3D plot of cyclobutane can be found in Figure 3.10.

Openbabel's `SetTorsion` method (see the `obmol.cpp` source code file) was used to apply torsion angle changes to cyclobutane. This function is used by *Kaplan* to generate new geometries based on changes in dihedral angles. For these tests, torsional changes were applied to a structure a-b-c-d, where a and d are connected. Minimum dihedrals were selected according to the minimum selection procedure from *Kaplan*.

The starting coordinates for cyclobutane were taken from *PubChem*. The dihedral angle for cyclobutane in the literature ranges from 20°-37° [2][4] using electron diffraction and 27° using proton NMR. The *PubChem* dihedral angle is approximately 25°. From the minimum set of dihedral angles in Table 3.1, each pair of attached carbon atoms is a centre for a dihedral angle. No hydrogens are within the minimum set of dihedral angles, since the carbon atom will always be the more substituted atom at any point in the ring.

The (2,0,1,3) dihedral angle was negated (set to approximately 25°). Then, the other minimum dihedrals were set to their original values. After the test, the torsion angles were recalculated, but they were found to be the same as the initial values from Table 3.1.

The geometry (when directly overlapped with the original coordinates) was rotated. Interestingly, the energy of the resulting geometry was approximately six

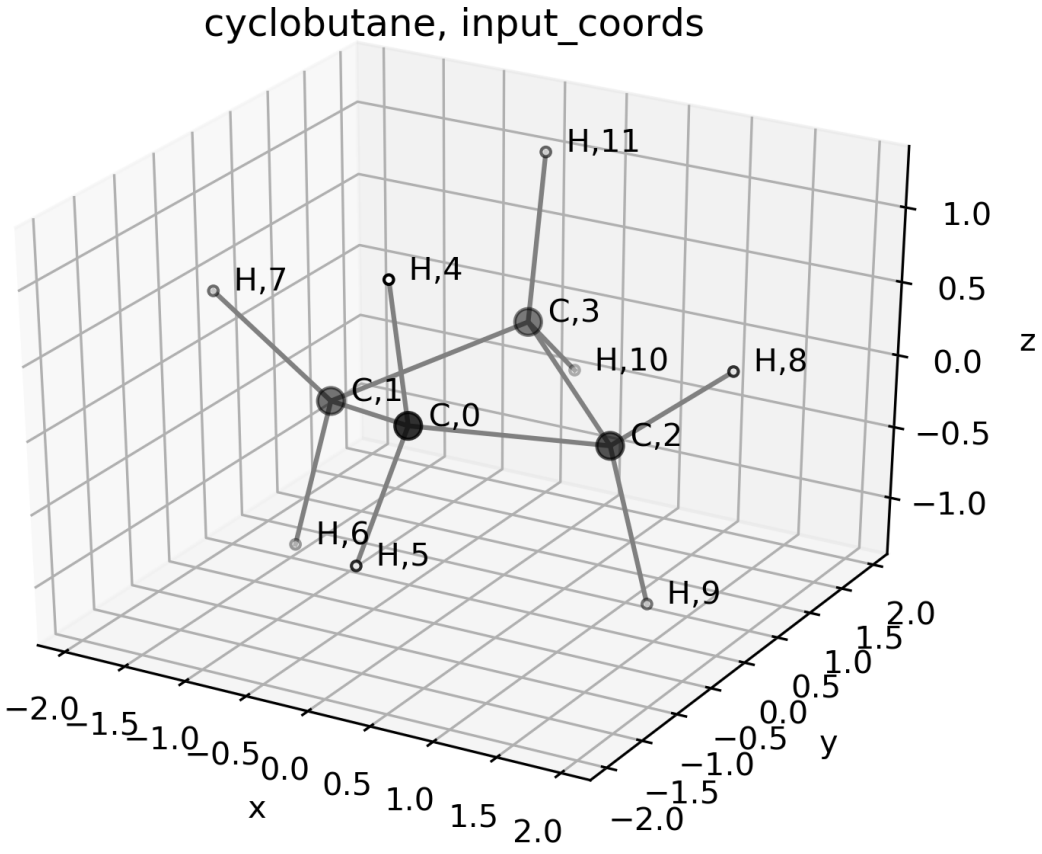


Figure 3.10: The 3D plot of cyclobutane.

Energy Term	**Original Coords**	**Change Dihedral (2,0,1,3) Test**
Bond Stretching	0.30486	0.30486
Angle Bending	1.37408	49.30815
Stretch Bending	-0.25608	-0.22145
Torsional	6.39727	10.35889
Out-of-Plan Bending	0	0
van der Waals	2.46319	2.33709
Electrostatic	0	0
Total Energy	10.28331	62.08753

Table 3.2: The energy breakdown for cyclohexane's original coordinates versus the new coordinates generated by changing the dihedral angles. The terms for which there was a significant increase in energy over the original coordinates have been highlighted in red.

times the energy of the initial geometry. The breakdown of energy terms was compared for both structures by looking at the output from the `obenergy` calculation. From Table 3.2, the terms most impacted by the dihedral change were angle bending, and (less significantly) torsional.

The original and new coordinates were overlapped and viewed in VMD - see Figure 3.11. Evidently, some of the blue hydrogen atoms are not at their original angles relative the carbon backbone, hence the increased energy from angle bending. Even though the minimum torsion angles were measured and found to be the same as the initial minimum torsion angles, of the 24 (a,b,c) angles, 8 had changed after the torsion angle modifications - the exact angle values are given in Table 3.3.

From a previous test performed on butane, angles were found to be stable with respect to dihedral angle changes. Therefore, the cause of the angle change had to be found in case any other types of systems could undergo unintended angle changes during a conformer search. From Table 3.3, only angles connected to car-

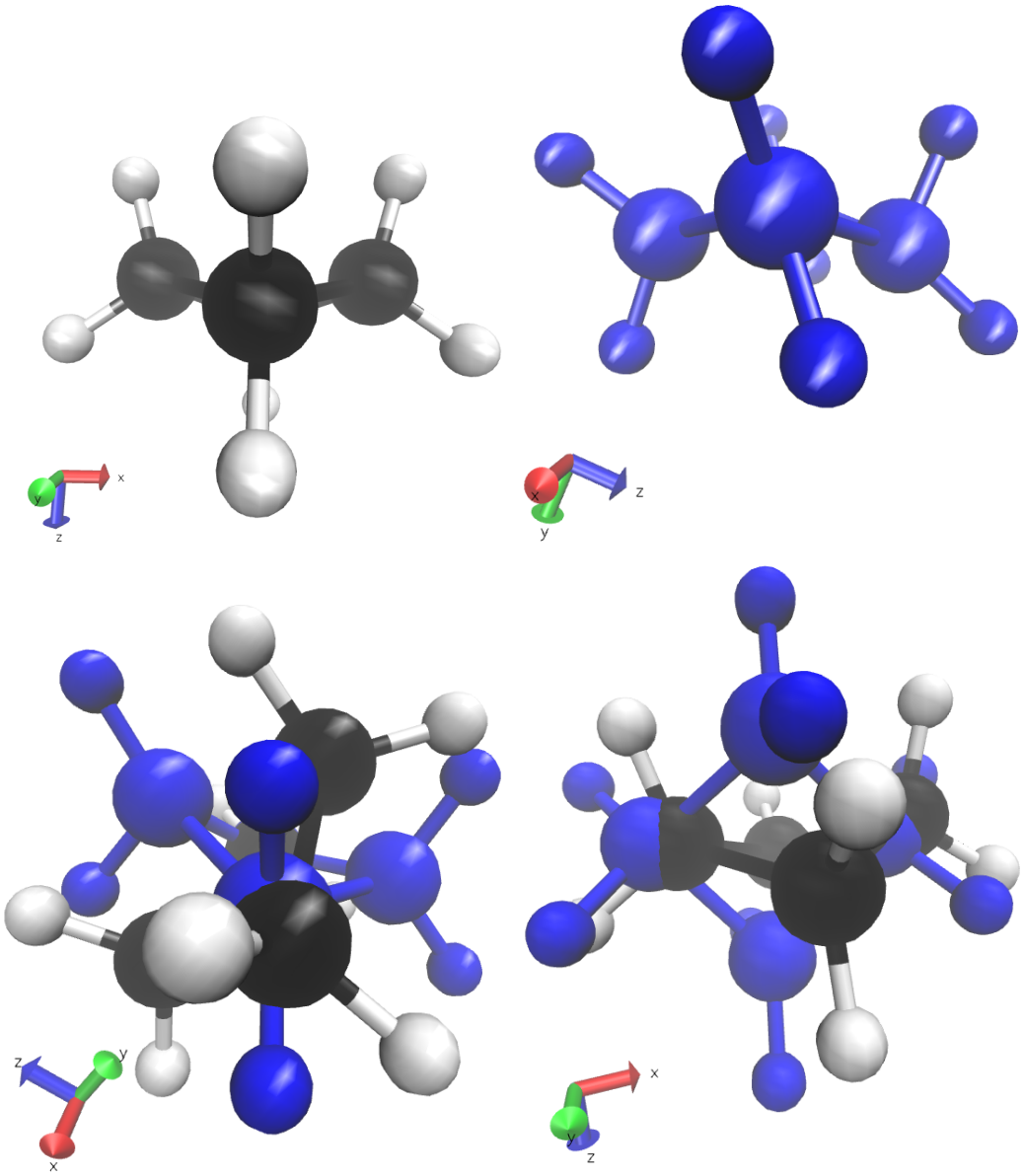


Figure 3.11: Individual and overlapped structures of cyclobutane, where black and white represent the original coordinates and blue represents the coordinates after changing the minimum dihedral angle (2,0,1,3) to approximately 25°, and setting the other minimum dihedral angles at their original values.

a	b	c	θ_{input} (rad)	θ_{new} (rad)	$\Delta\theta$ (rad)	$\Delta\theta$ (°)
2	0	5	2.0222	1.6723	-0.3499	-20.05
1	0	5	2.0223	1.5425	-0.4798	-27.49
2	0	4	1.9858	2.3328	0.3469	19.88
1	0	4	1.9857	2.2775	0.2918	16.72
0	1	6	1.9857	2.4108	0.4251	24.36
0	1	7	2.0222	1.5894	-0.4329	-24.80
0	2	9	1.9858	2.4086	0.4228	24.22
0	2	8	2.0222	1.5917	-0.4304	-24.66

Table 3.3: Results from the cyclobutane angle test. Each angle is given by (a,b,c), where b is the vertex, and $\Delta\theta = \theta_{new} - \theta_{input}$.

bons C0, C1, and C2 were affected - the carbon backbone angles were unaffected. The affected angles had the form (C_a, C_b, H_c), where C_a and $C_b \in \{0,1,2\}$ $(C_a \neq C_b)$ and $H_c \in \{4,5,6,7,8,9\}$ - the hydrogen atoms connected to (C1,C0,C2). None of the HCH angles were changed.

For the next test, only the (2,0,1,3) dihedral was set to $\pi/4$. For internal consistency[4], (3,1,0,2) was also set to $\pi/4$ (starting from the initial geometry). In both cases, two angles changed after the torsion was set. As shown in the results in Table 3.4, the selection of dihedral angle changed the location of the 2 angle modifications, but not the amount of angle change. Therefore, the results were consistent regardless of whether an `obmol` dihedral was chosen for modification.

[4]The `obmol` object has its own internal coordinates, and (3,1,0,2) was listed as a dihedral, but not (2,0,1,3).

By reading the *Openbabel* documentation and looking at the source code, the angle changing issue was a result of the `FindChildren` function that is called from within `SetTorsion`. The documentation for `FindChildren` is as follows:

> "Locates all atoms for which there exists a path to 'c' without going through 'b'. Children must not include 'c'."

This function takes three arguments: an empty vector the function fills with child atoms, b, and c, where b and c are from the dihedral angle (a,b,c,d). In cyclobutane, the hydrogens attached to b do not get altered when `SetTorsion` is called, because there isn't a path to c that isn't through b. This behaviour was confirmed by saving the cyclobutane coordinates before and after the torsion angle was applied. It is clear that the hydrogens at indices 4 and 5 did not have their xyz values altered in any way when dihedral $(2,0,1,3)$ was changed. Similarly, hydrogens at indices 6 and 7 did not change when dihedral $(3,1,0,2)$ was modified.

Going back to the original change torsion test, where the four minimum dihedrals were set (and only one was modified), 8 angle changes were detected. Therefore, the angle deviations are compounded when multiple torsion angles are set within ring systems. The same pattern occurred, in that only hydrogens attached to carbon atoms in the b position failed to undergo necessary rotation to maintain the molecule's original angles.

Considering cases where larger rings are modified, this error would still occur. Take for example the case of cyclohexane, as shown in Figure 3.12. Modification of the dihedral (a,b,c,d), (a',b,c,d), or (a',b,c,d') would fix the Hx and a' positions, while rotating the other atoms. Therefore, the `SetTorsion` method, as implemented, is not suitable for any dihedral angle changes to (a,b,c,d), where b and c are contained within a ring

During the single-application `SetTorsion` tests, it was noticed that the torsion angle didn't actually get set to the value $\pi/4$ for dihedral $(2,0,1,3)$. Instead, the final torsion angle was measured as -0.437826 - the original value, whereas $\pi/4 \approx 0.785398$. The result was the same when the new torsion angle was set to

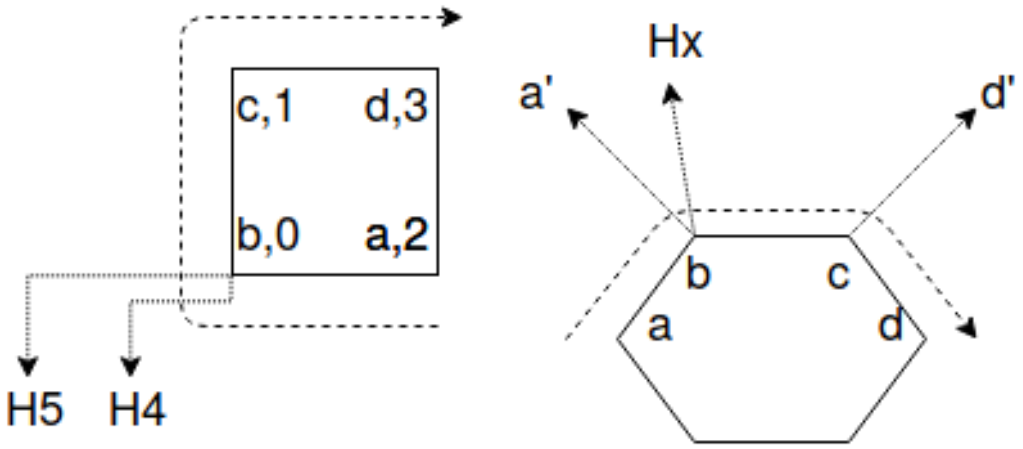


Figure 3.12: Left shows the (2,0,1,3) dihedral for cyclobutane, along with the (a,b,c,d) labels, where the arrow indicates the dihedral angle that was changed. The affected hydrogen atoms, attached to $b,0$ are also shown. Right expands the problem to a cyclohexane ring, but in reality this problem could be any ring containing b and c.

Set (2,0,1,3) to $\pi/4$						
a	b	c	θ_{input} (rad)	θ_{new} rad	$\Delta\theta$ rad	$\Delta\theta$ °
2	0	5	2.0222	0.9437	-1.0785	-61.794
2	0	4	1.9858	2.7671	0.7813	44.765
Set (3,1,0,2) to $\pi/4$						
3	1	6	1.9858	2.7672	0.7814	44.769
3	1	7	2.0223	0.9437	-1.0785	-61.796

Table 3.4: Results from changing the indicated torsion angle in cyclobutane, without setting any other torsion angle. The angles that changed are given by (a,b,c), where b is the vertex.

the negated version of the original torsion angle. Hydrogens attached to carbon b aside, it seems as if setting any constrained dihedrals containing only carbon atoms is not possible while still maintaining the original angles. The only case where the `SetTorsion` worked was for dihedrals of the form (a,b,c,d), where a was not in the ring - for example $(4,0,1,3)$ was changed from 1.565623 to 0.785398 successfully. This observation makes sense when compared to previous results, as the free a atom in this case is not considered for the rotation matrix during `SetTorsion`.

3.8.2 Cyclohexane

Cyclohexane was tested originally to search for the boat and chair conformations. However, these tests were mostly failures, and the failures were due to the inappropriate use of the `SetTorsion` method. *Kaplan* was able to find conformations close to the starting geometry - the chair - but no other viable conformation, as no amount of torsion modification would actually change the internal coordinates of the carbon backbone. A 3D plot for the input coordinates, which were taken from *PubChem*, is given in Figure 3.13.

From the failed experiments, any consideration in fitness for RMSD favoured misplaced hydrogens, as it was impossible to change the relative positioning of the carbon atoms through `SetTorsion`. It was due to these failed experiments that the author decided to add the option in *Kaplan* to remove hydrogens from RMSD calculations. When the hydrogens were removed from RMSD consideration, the resulting conformers were all the same chair conformation with slightly different dihedral angles - an example is provided in Figure 3.14. An example of the test inputs are given below:

```
run_kaplan({
    "struct_input": "cyclohexane",
    "no_ring_dihed": False,
    "exclude_from_rmsd": [1],
})
```

This type of conformer search would need to locate the original dihedral angles,

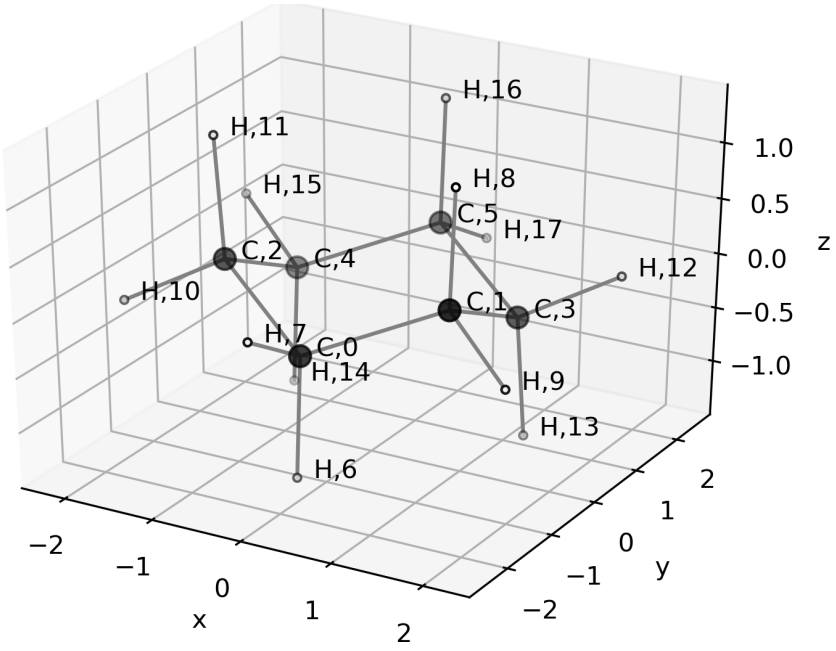


Figure 3.13: The 3D plot of cyclohexane.

because deviation from those angles would cause the same problems as with cyclobutane - the hydrogens would become misplaced and the energy would increase dramatically. The dihedrals from the best `pmem` from the cyclohexane conformer search were not equivalent for all 5 conformers, since the RMSD term of the fitness function meant that it was favourable for the conformers to be misaligned. Exact dihedral angles are shown in Table 3.5. The difference in energy between conf0 and conf2 is high, even though they only differ by 1 dihedral angle. The plot of energy differences and RMSDs can be found in Figure 3.15.

The *GOpt* code, written by Xiaotian (Derrick) Yang, was used for one of the tests. Instead of applying the `SetTorsion` method for each dihedral, a geometry is constructed by calling `optimize_to_target_ic`, where `ic` stands for internal coordinates. In this method, the code tries to find a set of coordinates that match the given internal coordinates. In some cases for cyclohexane, the search fails, because the internal coordinates don't make physical sense. For example, if two dihedral angles are modified in cyclohexane - (a,b,c,d) to 0 rad and (a,b',c',d)

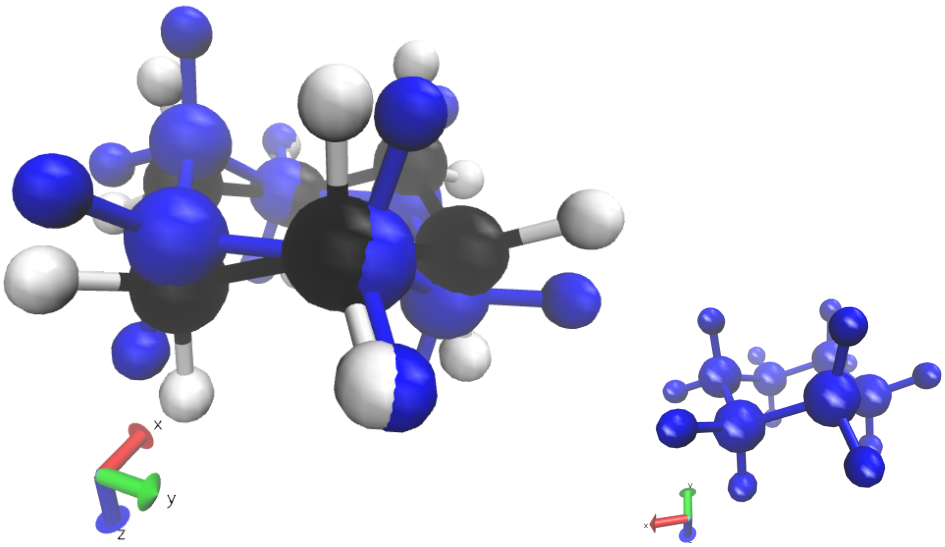


Figure 3.14: The lowest-energy conformer (in blue) from the best `pmem` shown overlapping the original input coordinates (black and white) from the conformer search for cyclohexane. The original chair conformation was recovered, despite the inappropriate use of `SetTorsion`.

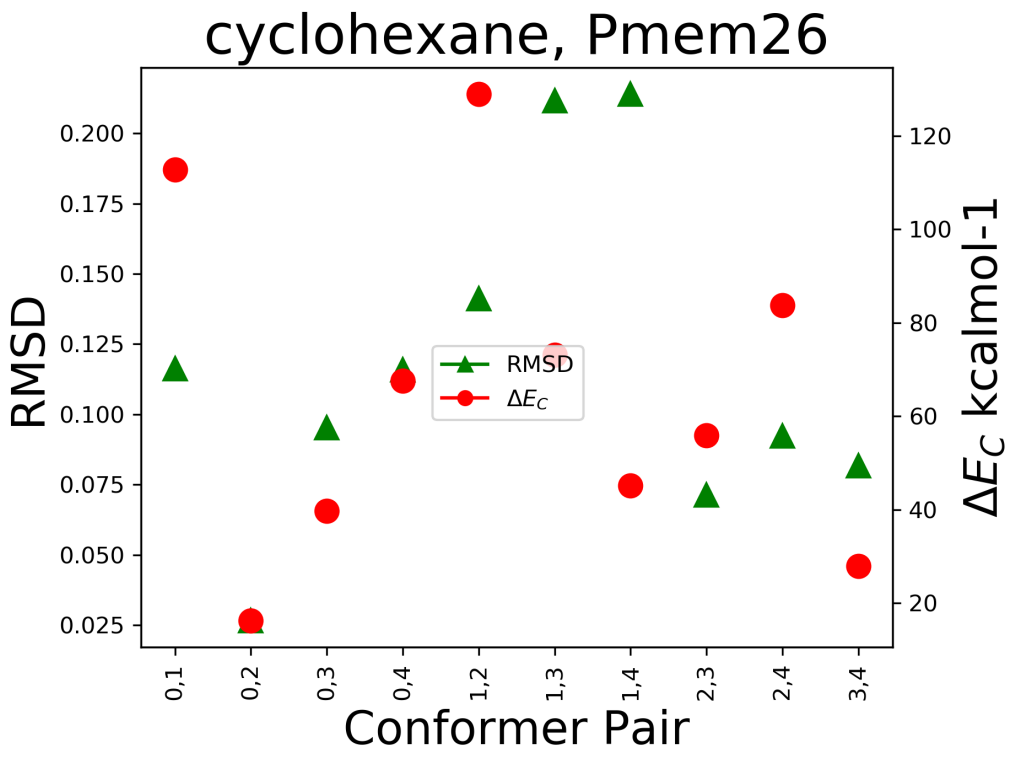


Figure 3.15: A plot of energy differences and RMSD values for the best `pmem` from the cyclohexane conformer search.

		Dihedrals					
Conf#	Energy (kcal/mol)	2013	1024	0135	0245	1354	2453
0	16.87673	5.803	0.739	0.928	5.305	5.233	1.008
1	129.63404	6.276	0.361	0.946	5.291	5.771	0.329
2	0.69020	5.358	0.739	0.928	5.305	5.233	1.008
3	56.57709	5.358	0.739	0.159	5.764	5.233	1.008
4	84.46931	4.814	1.674	0.928	5.764	5.233	1.008
input coords	-3.56092	-0.947	0.947	0.947	-0.947	-0.947	0.947

Table 3.5: Dihedrals for the best `pmem` from the cyclohexane conformer search, with the dihedrals given for the starting coordinates. Note that $-0.947 \approx 5.336$ when converted to the $0-2\pi$ range. All dihedrals are in radians.

to π rad - a set of coordinates that still satisfies the original connectivity matrix does not exist, since it would not be possible to facilitate the $c-d$ and $c-d'$ bonds concurrently with these dihedral angles.

Here is an example of the inputs for a *GOpt*-based conformer search for cyclohexane:

```
run_kaplan({
    "struct_input": "cyclohexane",
    "no_ring_dihed": False,
    "use_gopt": True,
    "exclude_from_rmsd": [1],
})
```

There are a few issues that need resolving for using *GOpt*. First, the code is significantly slower than when using *Openbabel*, since *Openbabel* is written in C++ and *GOpt* is written in *Python*. As an example, when run on the same laptop, 2000 `mevs` for a cyclohexane conformer search took 29 hours when using *GOpt* and less than 2 minutes for *Openbabel* (where both experiments used the same inputs).

Another issue is that the bond lengths are not conserved from the original geometry - as shown in Figure 3.16. When using VMD to view the resulting conformers from the best `pmem`, no bonds are found. Furthermore, none of the conformers were hexagonal from the best `pmem`; instead they adopted rectangular shapes.

When `obenergy` was called on the final conformers, the energy calculation was unsuccessful - giving the error that the forcefield could not be setup for the input molecule. Therefore, the energies reported in the stats file are only accessible by first constructing the `obmol` object, constructing the `internal` object (for *GOpt*), inputting the dihedral angles to the `internal` object, and finally setting the coordinates of the `obmol` object to those of the `internal` object and running the energy calculation. The energy values reported in the stats file are also way too high - they are in the 14,000 kcal/mol range, whereas the input coordinates have an energy of -3.56092 kcal/mol.

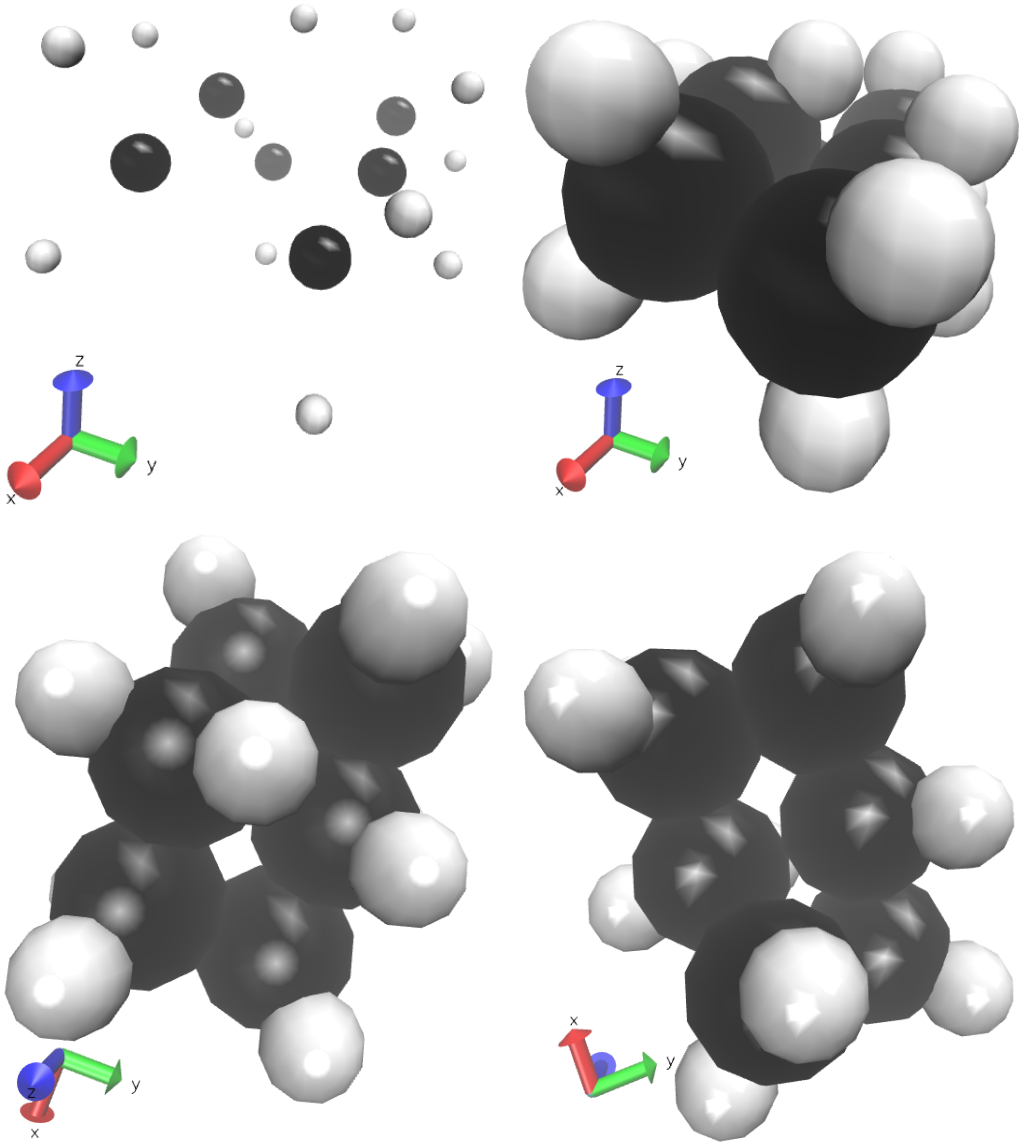


Figure 3.16: Shows one of the conformers that resulted from the GOpt conformer search for cyclohexane. The top left shows a CPK view in VMD, and the other three images show the same conformer using a van der Waals view in VMD.

Chapter 4

Finding Known Conformers

This chapter will compare sets of known conformers, as calculated from more exhaustive quantum chemical methods, to the conformers produced by *Kaplan*. The purpose is to verify *Kaplan* can find the correct conformers. Two datasets are used: one that contains individual amino acid conformers, and another that has conformers for the Penta-Alanine (PA) peptide. A description of the dataset is given, followed by energy and RMSD evaluations.

The amino acids are used as test cases, because they will be structurally familiar to most chemists. They all have the same backbone structure with varying R-groups (side-chains). Furthermore, their conformations can be useful in polypeptide structure determination. Their structures are also well-characterised using computational and experimental methods [11][26][42][85].

4.1 Amino Acid Dataset

By studying the conformations of amino acids, information with regards to the local structure of proteins can be gained. The number and placement of energy minima for the amino acids are known to fluctuate with varying levels of theory [81]. Therefore, it is hard to confirm that a set of conformers represent the true minima from the potential energy surface (PES). To test *Kaplan*'s conformer searching abilities,

an external set of geometries was used to represent the target energy minima for the amino acids.

The data for this test was taken from the supplementary material from [81]. The pdf containing structure files was scraped using a *Python* script, and the corresponding xyz files were sorted by amino acid and placed into two subdirectories based on the method and basis set used to optimise the structure.[1] The amino acids used for this paper contain methyl caps (via a peptide bond on either end), and so follow the main structure:

$$CH_3{-}C(=O){-}N(H){-}(H)C_\alpha(R){-}C(=O){-}N(H){-}CH_3$$

These methyl caps are intended to represent the alpha carbons (C_α) of neighbouring amino acids (as if each structure was a part of a larger protein). An example is shown in Figure 4.1 for the structure of alanine.

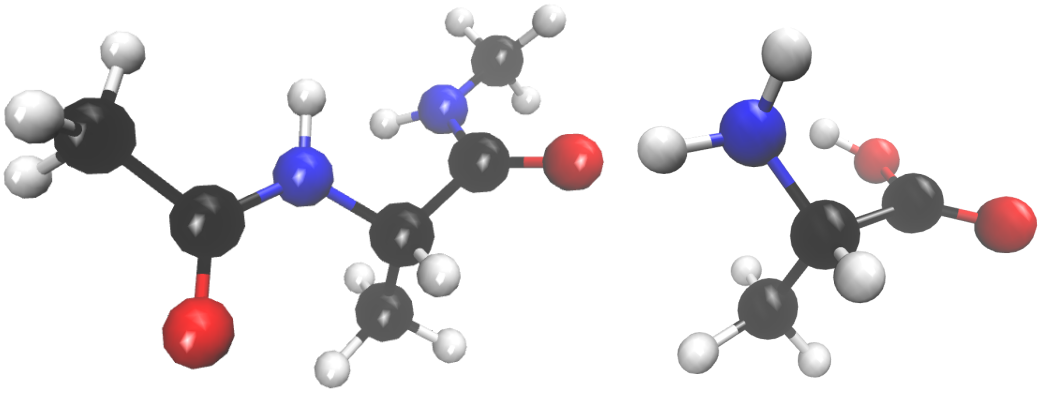


Figure 4.1: The structure of alanine, whose R-group is a methyl group (CH_3). The left image shows the structural variant with peptide bonds on the carbon backbone, as was used for testing the amino acids in this chapter. The right image shows the alanine with NH_2 and COOH caps.

To find the energy minima for the amino acids (and the corresponding conformers), Yuan et al. first used *MarvinView* (a ChemAxon product) to produce a set of

[1] Density functional theory (DFT) using B3LYP and the apc-1 basis set was used in one optimisation. The second optimisation method used Møller-Plesset (MP) perturbation theory (specifically, MP2) with the cc-pVDZ basis set.

geometries by sampling all internal single bonds from within a specified diversity limit. The version of *MarvinView* that was mentioned in their paper uses the Dreiding forcefield [20]. Duplicate geometries were filtered by optimising the conformers with Hartree-Fock (HF) and the 6-31G(d,p) basis set from *GAUSSIAN09* [58], with the "tight" input parameter. Another filter was applied to the remaining structures by comparing their torsion-based root-mean-square (rms_{ij}), given in Equation 4.1. Geometries were considered separable if their rms_{ij} was above 40°.

$$rms_{ij} = \sqrt{\frac{\left(\phi_i - \phi_j\right)^2 + \left(\psi_i - \psi_j\right)^2 + \sum_{k=0}^{m-1}\left(\chi_{ki} - \chi_{kj}\right)^2}{n}} \tag{4.1}$$

In Equation 4.1, i and j represent two conformer geometries from the same amino acid. Three important dihedral angles are indicated in the structure of amino acids: ϕ, ψ, and χ_k, where χ_k is the k^{th} rotatable bond from the R-group. The number of χ dihedral angles varies depending on the amino acid. For example, glycine has $m = 0$ and aspartate has $m = 3$. Rotatable bonds containing hydrogen atoms are not considered in this calculation. In Equation 4.1, the denominator n is the number of dihedral angles being considered, where $n = m + 2$. Each structure in the amino acid dataset follows the same numbering convention, where the labelled dihedrals correspond to the atom indices given in Table 4.1.

Dihedral Label	Atoms Involved
ϕ	$C_{17}-N_0-C_1-C_2$
ψ	$N_0-C_1-C_2-N_{11}$
χ_0	$N_0-C_1-C_4-O_5$

Table 4.1: Atom indices representing the three important dihedral angles for this analysis.

The ϕ and ψ angles are well known in protein chemistry, because these angles are important in characterising the backbone. The χ dihedral angle is therefore

useful in describing the behaviour of the R-group that is unique to each amino acid.[2] An example for the serine molecule is given in Figure 4.2.

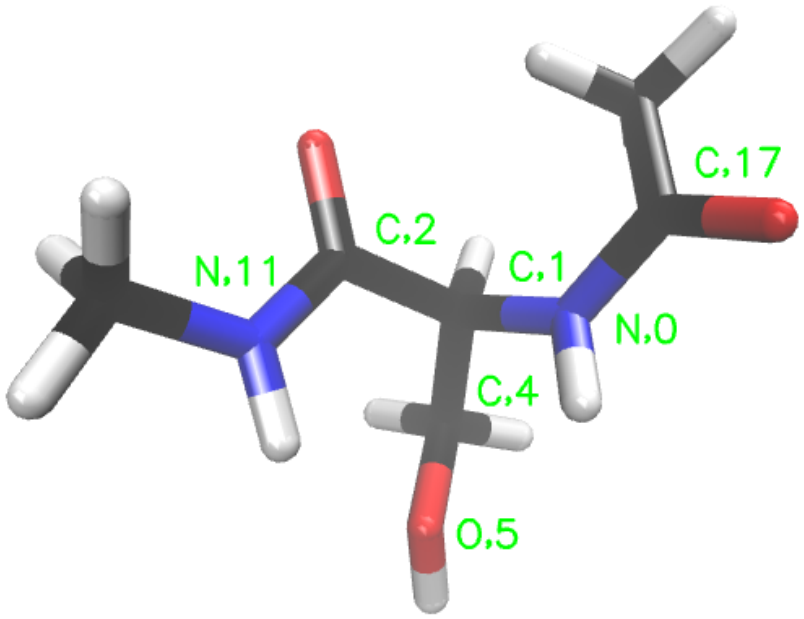


Figure 4.2: The structure of serine (specifically, conformer 0 from the B3LYP optimisation) with labels to indicate atom indices forming ϕ, ψ, and χ dihedral angles. Atoms 17-0-1-2 represent ϕ, atoms 0-1-2-11 represent ψ, and atoms 0-1-4-5 represent χ_0 (the first R-group dihedral angle). Serine only has one R-group dihedral angle ($m = 1$), since dihedral angles containing hydrogen atoms are not counted.

A data table showing the mean and standard deviation (stdev) for the energies is provided in Table 4.2. From this table, the amino acids with the most energy minima were arginine (61) and methionine (57). The amino acids with the fewest energy minima were glycine (9) and proline (5). The conformer energies have a low level of deviation, indicating a high amount of optimisation. The B3LYP/apc-1 energies are consistently lower than the MP2/cc-pVDZ energies for all amino acids, but the energies follow the same trend for both method/basis set combinations.

[2]This work uses 0-based indexing. The numbers are one less than those found in the paper.

		B3LYP/apc-1		MP2/cc-pVDZ	
Amino Acid	**Count**	*mean*	*stdev*	*mean*	*stdev*
asparagine	12	-664.5055	0.0070	-660.6946	0.0074
glutamine	21	-703.8126	0.0060	-699.7296	0.0063
aspartate	36	-684.3650	0.0081	-680.5220	0.0090
glycine	9	-456.4971	0.0067	-453.8512	0.0071
tryptophan	26	-858.3849	0.0064	-853.2240	0.0069
cysteine	24	-894.0160	0.0074	-890.4195	0.0074
threonine	17	-610.3258	0.0082	-606.7867	0.0095
alanine	11	-495.8057	0.0069	-492.8884	0.0071
isoleucine	25	-613.7328	0.0070	-609.9985	0.0075
leucine	28	-613.7350	0.0063	-610.0003	0.0069
tyrosine	17	-802.0515	0.0057	-797.3190	0.0062
glutamate	36	-723.6746	0.0063	-719.5605	0.0068
proline	5	-573.2147	0.0039	-569.7985	0.0035
histidine	24	-720.8108	0.0085	-716.5704	0.0090
lysine	39	-669.0801	0.0047	-665.0296	0.0056
serine	26	-571.0218	0.0063	-567.7542	0.0066
arginine	61	-778.5755	0.0062	-773.9359	0.0074
valine	15	-574.4246	0.0086	-570.9625	0.0098
methionine	57	-972.6313	0.0064	-968.4886	0.0071
phenylalanine	30	-726.8298	0.0069	-722.4500	0.0073

Table 4.2: Energy data (in Hartrees) from the amino acids dataset gathered from [81]. Two sets of method and basis set are shown. Note: *stdev* is standard deviation. The counts indicate how many energy minima were included in the calculations, and the counts were equivalent for both sets of data.

A pair of boxplots were produced, using the scraped geometries as input, for the pairwise RMSD values for each amino acid – see Figure 4.3. Some outliers at

approximately 0 RMSD were found for about a third of the amino acids, meaning that some of the conformers from the dataset were geometrically redundant.

An example of an identical set of conformers is conformers 6 and 24 from the MP2/cc-pVDZ data of isoleucine. The RMSD between these conformers was calculated to be 0.000209 Å. The images appear identical after applying the Kabsch-derived rotation matrix. The RMSD boxplot for isoleucine optimised with B3LYP/apc-1 does not show an outlying point at 0; therefore, two of the MP2/cc-pVDZ conformers may have converged to the same place in the potential energy surface (PES), while remaining separated in the B3LYP/apc-1 optimisation.

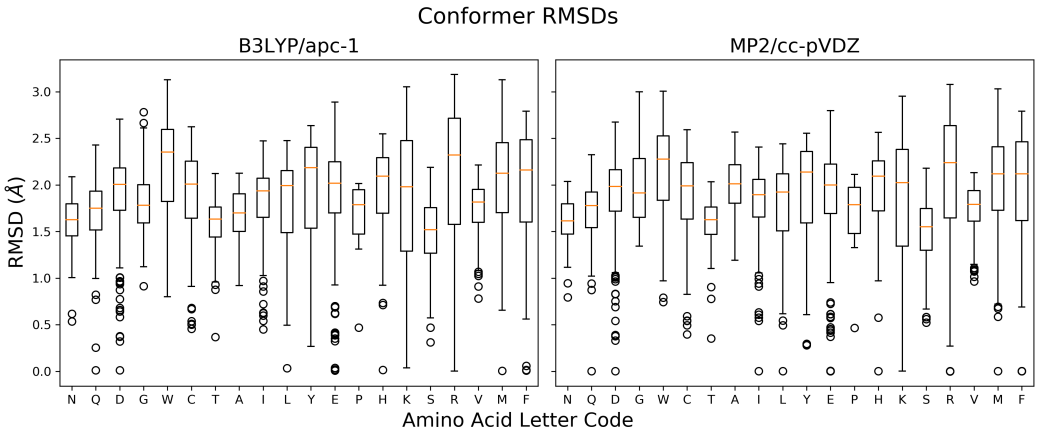


Figure 4.3: The pairwise RMSD values (with reordering) for each amino acid in the scraped data set, as calculated using the *rmsd* package. The left and right boxplots show the RMSD results for the B3LYP/apc-1 and MP2/cc-pVDZ optimised conformers, respectively.

4.2 *Kaplan* Conformer Search

The first B3LYP/apc-1 conformer for each amino acid was used as an input file to the *Kaplan* conformer searching program. The script to run the conformer search is given below. The complete runtime for this conformer search was approximately 90 minutes (using an Intel Core i7-8550U processor) for the 20 amino acids.

```
from kaplan.control import run_kaplan
from kaplan.tools import amino_acids
for aa in amino_acids:
    run_kaplan({
        "struct_input": f"input_xyzfiles/{aa}.xyz",
        "struct_type": "xyz",
        "charge": 0,
        "multip": 1,
        "num_geoms": 10,
        "stop_at_conv": 50,
        "opt_init_geom": False,
    })
```

Compared to the first conformer search for the amino acids in Chapter 2, *Kaplan* found fewer duplicate conformers. The larger size of the molecules may have allowed *Kaplan* to find more distinct conformers. Boxplots showing the distributions of pairwise RMSD values (calculated with `reorder`) are given in Figure 4.4. *Kaplan* found identical conformers for glutamine (Q), cysteine (C), proline (P), and arginine (R). The RMSD distribution for proline is lower than for the other amino acids; this result may reflect the inability of *Kaplan* to optimise ring dihedrals. Furthermore, *Kaplan* was tasked with finding 10 conformers, but the quantum-based search only found 5 minima for proline.

The best `pmem` from each of the *Kaplan* conformer searches had its energies calculated using the MMFF94 forcefield. The same calculation was performed on the structures from the amino acid dataset. The relative energy distributions (determined from the lowest energy conformer for each amino acid, across the 3 sources) are given in Figure 4.5. Since the *Kaplan* conformers were partially optimised at the end of the conformer search with MMFF94, the energies are, as expected, lower than those from the quantum chemistry dataset. The number of `mevs` needed to reach convergence (i.e. no improvement to the best `pmem` after 50 `mevs`), the energy ranges, the maximum RMSD values, and duplicate conformer ids are given in Table 4.3. From these results, methionine took the most `mevs` to converge (550), whereas asparagine, tyrosine, histidine, and phenylalanine took the fewest (100).

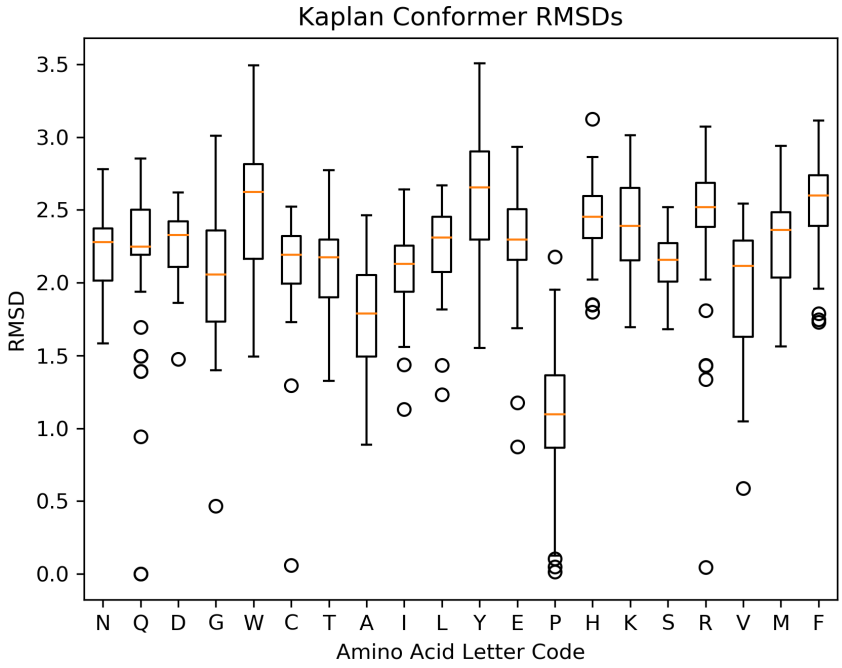


Figure 4.4: For each methyl-capped amino acid, the boxplot of pairwise RMSD calculations from the best `pmem` produced by *Kaplan* (45 values per boxplot).

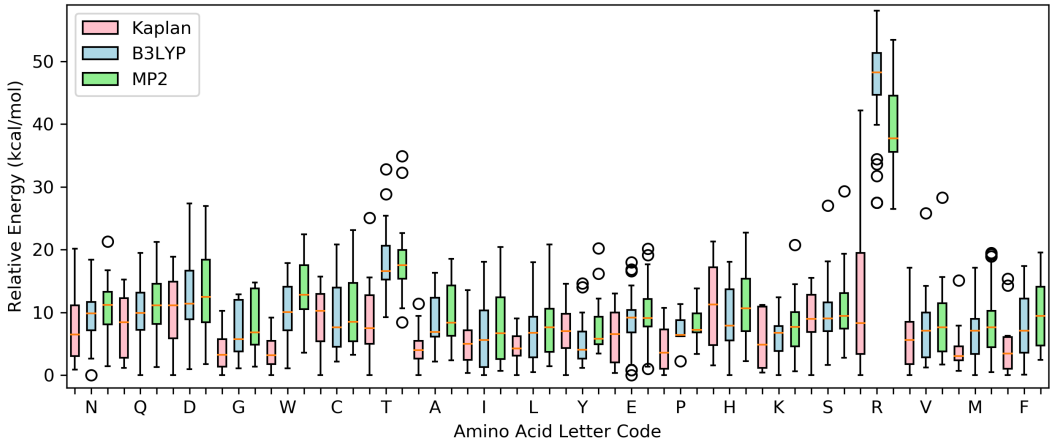


Figure 4.5: For each conformer group: *Kaplan*, B3LYP/apc-1, and MP2/cc-pVDZ, the minimum energy was calculated for each amino acid using the MMFF94 force-field. Then the minimum energy values were subtracted from each conformer energy. The boxplots show the set of relative conformer energies.

To compare the structures of the *Kaplan* conformer search directly with those found by Yuan et al., a script was written to run 3 RMSD calculations – (1) `reorder`, (2) no `redorder`, and (3) no `reorder` without hydrogens – to compare each *Kaplan* conformer with the conformers from the B3LYP/apc-1 and MP2/cc-pVDZ optimised conformers. The conformers from both the *Kaplan* and the external data set have the same atomic order in the xyz files; however, the RMSD with `reorder` was found to be lower and higher than the RMSD value without `reorder`, depending on the conformers being compared. Therefore, the algorithm may not be considering the case where there is no atom reordering for the instances where the RMSD increases upon enabling `reorder`.

From Figure 4.6, each boxplot contains 10 values representing the lowest RMSDs (i.e. no hydrogens and no reordering) collected after comparing *Kaplan* conformers 0-9 with conformers 0-*b* from B3LYP/apc-1 and 0-*b* from MP2/cc-pVDZ. The values for *b* are given in Table 4.2, and represent the number of conformers (distinct energy minima) found per amino acid. Overall, the geometric differences between the conformers produced by *Kaplan* and those from the quantum dataset are less than 1.5Å, with the majority of RMSD values being below 1.0Å. These differences are probably a result of comparing structures that were optimised with two different methods, symmetry, and/or slight variances in torsion angles.

A problem was found with the starting coordinates for the arginine. The input multiplicity for the structure was 1; however, *Openbabel* calculated a multiplicity of 2. Therefore, the input coordinates from conformer 0 from the B3LYP/apc-1 dataset may not have been a good starting structure. Conformer 0 from the MP2/cc-pVDZ dataset for arginine did not produce this error at startup, and the *Openbabel* 2D structures are different for the two input files (see Figure 4.7). In Figure 4.5, the arginine energy distributions for the quantum datasets are much higher than for *Kaplan*. Because of the multiplicity problem, *Kaplan* recorded erroneous energies for arginine. Therefore, the high RMSD values when comparing *Kaplan* conformers with the other dataset are also erroneous. Despite the issues with multiplicity, two important hydrogen bonds were found in the arginine conformers from *Kaplan*, shown in Figure 4.8.

Amino Acid	`Mevs`	E_{mean}	E_{stdev}	R_{mean}	R_{stdev}	E_{min}	E_{max}	R_{max}	E_{min}, E_{max} **id**	R_{max} **id**
asparagine	100	-35.701	9.506	2.003	0.376	-58.441	-39.147	2.284	8,7	1,5
glutamine	250	-29.658	8.383	2.193	0.413	-52.562	-38.435	2.734	0,6	*0,1
aspartate	200	-33.404	8.450	2.004	0.354	-59.452	-40.603	2.496	7,4	6,8
glycine	150	-10.447	5.400	1.701	0.348	-19.260	-9.013	2.123	9,8	5,8
tryptophan	200	19.765	6.511	2.446	0.584	5.057	14.237	3.149	2,5	6,8
cysteine	150	12.696	8.103	1.938	0.347	-8.189	7.547	2.571	4,3	1,8
threonine	350	18.331	8.019	1.919	0.409	-3.417	21.655	2.472	7,8	6,8
alanine	250	-5.667	5.305	1.758	0.365	-15.994	-4.622	2.290	9,1	1,6
isoleucine	200	10.428	8.824	2.250	0.454	-9.567	3.615	2.783	0,5	1,3
leucine	300	4.400	6.438	2.112	0.369	-11.506	-2.449	2.635	3,6	2,5
tyrosine	100	17.920	7.724	2.294	0.492	-1.429	13.168	3.278	2,0	2,7
glutamate	350	-30.735	7.132	2.099	0.380	-47.070	-34.407	2.712	2,9	5,8
proline	150	3.479	4.552	1.427	0.366	-6.410	4.302	1.853	5,7	4,8
histidine	100	17.469	11.937	2.177	0.387	-9.265	10.483	2.905	4,0	0,9
lysine	450	1.490	7.381	2.288	0.486	-16.828	-6.056	2.962	9,6	4,9
serine	200	13.176	5.969	1.844	0.398	-4.674	10.861	2.320	1,7	1,8
arginine	150	-156.957	13.251	2.500	0.470	-188.844	-169.026	3.462	8,2	0,6
valine	150	6.444	8.950	2.006	0.373	-12.278	4.868	2.552	0,1	4,8
methionine	550	-0.652	7.838	2.145	0.441	-14.813	-0.460	2.915	3,1	1,9
phenylalanine	100	27.313	7.821	2.269	0.437	9.815	25.181	3.064	1,3	3,7

Table 4.3: Overall results for the *Kaplan* conformer search for each amino acid, including number of `mevs` taken to reach convergence, mean and standard deviation (stdev) energies (E) and RMSD values (R) from the final `ring`, as well as the best RMSD (R_{max}) and energy range ($E_{min} - E_{max}$) for the best `pmem`, with associated conformer numbers (id). * multiple identical conformers with maximum RMSD were found for glutamine.

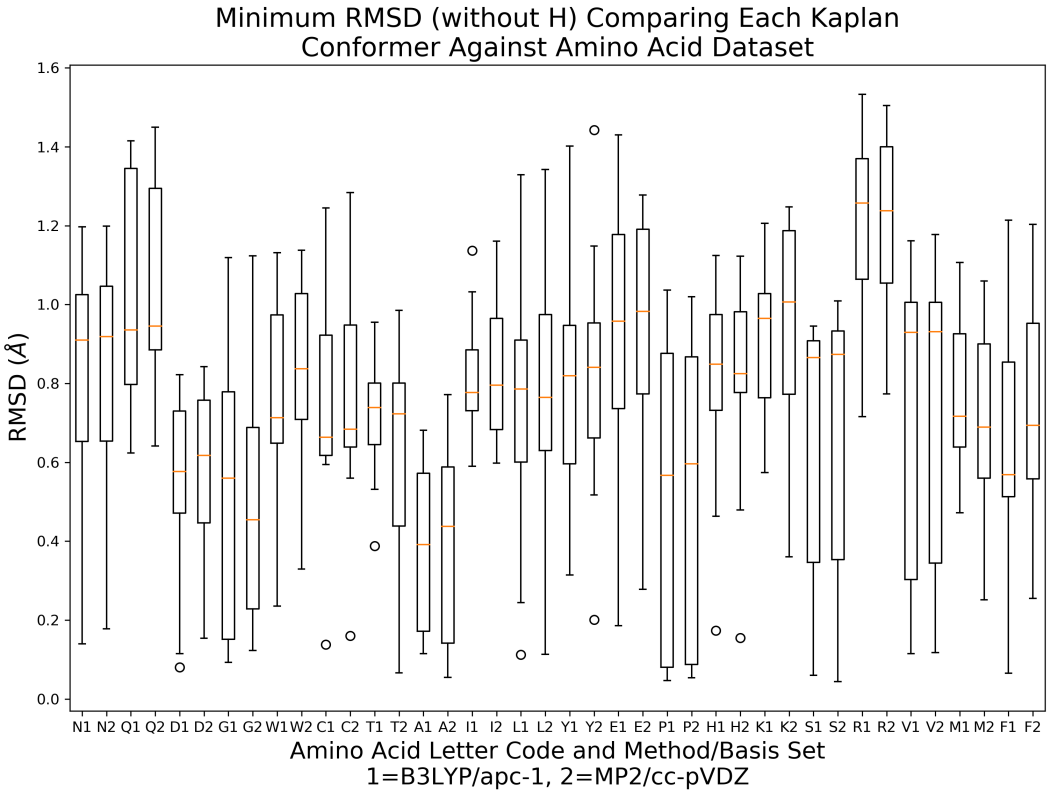


Figure 4.6: Minimum RMSD values as calculated (without hydrogens) between each *Kaplan* conformer and (1) the set of B3LYP/apc-1 conformers and (2) the set of MP2/cc-pVDZ conformers. Lower values indicate that the structures were more similar to the energy minima found from the quantum-based conformer search.

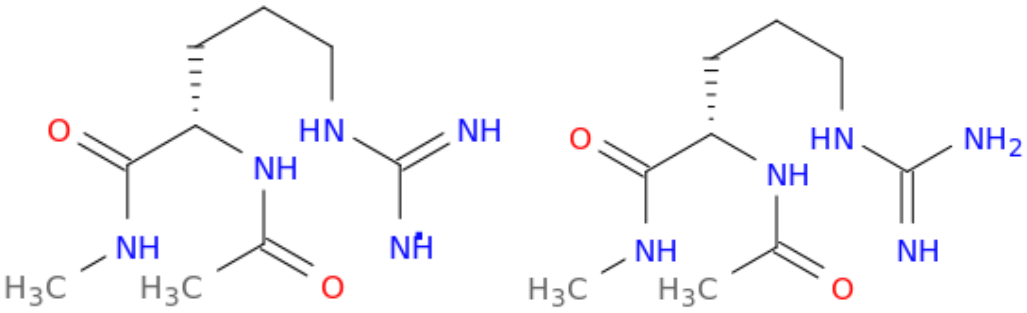


Figure 4.7: 2D structures generated by *Openbabel* as part of the *Kaplan* startup routine. The multiplicity of the left structure was calculated to be 2, and the diagram is missing a hydrogen. This structure was taken from the B3LYP/apc-1 dataset (specifically, conformer 0). The right image had a multiplicity of 1 and does not show any missing hydrogens. This input structure was taken from the MP2/cc-pVDZ dataset (again, conformer 0).

The conformer searching results for glutamine were interesting due to the coupled nature of the RMSD and the energy. In the final best `pmem`, conformers 1 and 3 were identical, as well as conformers 0 and 7. The maximum RMSD was found between conformers (0,1), (0,3), (1,7), and (3,7), and the minimum energy was found for conformers 0 and 7. Therefore, rather than finding new solutions, the evolutionary algorithm was exploiting the fitness function to afford itself fitness by finding an optimal combination of fewer geometries. In future work, it may be beneficial to add a penalty to the fitness function for `pmems` containing identical conformers, thus eradicating this algorithmic "loophole". Other than glutamine, conformers with low RMSD (as calculated without reordering) included cysteine 2,6 (0.0610Å), proline 1,5 (0.608Å), and arginine 1,4 (0.725Å).

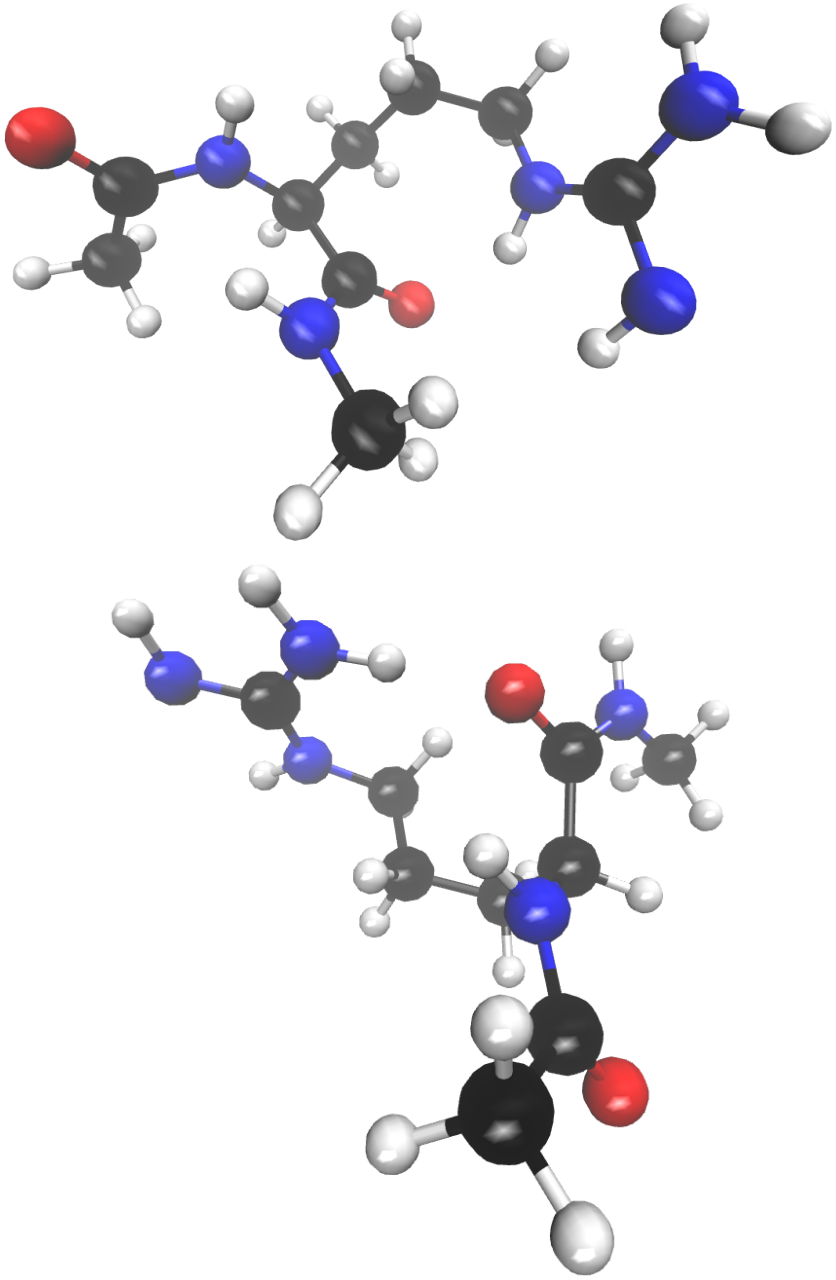


Figure 4.8: Two conformers (4 and 8) from the *Kaplan* conformer search for arginine, showing hydrogen bonding.

4.3 Conformers of Methionine

To reduce the number of differences between the quantum-based conformers and the *Kaplan* conformers, the conformers can be exhaustively optimised using the same method. Therefore, structures from *Kaplan* and the external dataset will converge to the same location in the energy landscape, if they represent the same conformer.

Methionine was chosen as a test molecule for this purpose, because it had the second highest number of conformers found (57) in the computational dataset (indicating energy surface complexity). Furthermore, arginine (at 61 conformers) could not be tested, because its multiplicity was incorrectly guessed by *Openbabel* for some input structures. Optimising arginine structures proved to be erroneous, since the energy increased after optimisation (due to the multiplicity issues). *Kaplan* warns the user when the calculated multiplicity is not the same as the input multiplicity for a molecule. Therefore, results for arginine conformers should not be considered until this issue has been resolved in *Openbabel*. Methionine was also deemed a difficult test molecule from Section 2.2, since, after local optimisation, its RMSD changed significantly and the steps until converged was relatively high (see Figure 2.5).

To perform this test, the 57 B3LYP/apc-1 methionine conformers (henceforth shortened to `b3lyp` conformers) were optimised using *Openbabel*'s MMFF94 forcefield with steepest descent. The same procedure as in Section 2.2 was followed, such that the conformers were considered to have converged in energy if, sampling every 100 steps, the energy did not change after 10 such steps (i.e. 1000 steps overall). The 6 best `pmems`, sorted by fitness, from the *Kaplan* `ring` each contributed 10 conformers, for a total of 60 conformers. These structures were also optimised using the same method as for the `b3lyp` conformers. This step was performed to ensure that all structures were at the lowest possible point of their potential energy well in the MMFF94 forcefield landscape.

Of the 60 *Kaplan* conformers for methionine (henceforth shortened to `kaplan` conformers), 44 did not change in energy after the additional forcefield optimisa-

tion, indicating that they were already at the bottom of their respective potential wells. Of the non-zero energy changes (16 conformers), the average energy change was -1.05 kcal/mol ($\pm$1 stdev). Therefore, using the parametrisation from Section 2.2 sufficiently minimised most of the final conformers for methionine. By comparison, the average energy change was -3.45 kcal/mol ($\pm$1 stdev) for the `b3lyp` conformers.

The lowest conformers from `kaplan` and `b3lyp` had final energies of -18.887 kcal/mol and -18.884 kcal/mol, respectively. The RMSD between these conformers was 1.10Å, and the RMSD discounting hydrogen atoms was 0.0459. The global minimum conformer for *Kaplan* did require minimisation in this case (6500 steps for -1.65 kcal/mol change in energy); the RMSD change after applying the second optimisation was 0.434Å and 0.313Å with and without hydrogens, respectively. Therefore, *Kaplan* was able to identify the global minimum conformer for methionine. This result was confirmed by visual inspection in VMD.

The second-lowest conformer from the `b3lyp` set was a duplicate structure of the lowest energy conformer. A potential energy well may have collapsed when converting from the quantum to the forcefield energy landscape, or this structure could be the duplicate identified in the initial dataset (see Figure 4.3). There were some duplicates in the `kaplan` set, which was expected due to the use of multiple `pmems`. Specifically, 28 of the pairwise calculations (within the `kaplan` set) were 0 (1770 total pairwise for 60 structures).

The highest energy conformer from the `kaplan` set was 0.144 kcal/mol, whereas the highest energy conformer from the `b3lyp` set was -1.54 kcal/mol. Since the optimisation did not change the structure of the highest energy `kaplan` conformer, it is likely that *Kaplan* was able to identify a high energy conformer on forcefield energy surface that was not in the quantum dataset.

Comparing the data as a whole, to identify all 57 methionine conformers from the `b3lyp` set using the *Kaplan* conformers (while noting that some of those conformers might be geometrically equivalent), the maximum backbone RMSD (not including hydrogens) between any two conformers (comparing across data sets) would be 1.28 Å for conformers `kaplan6` and `b3lyp45`. Considering all atoms,

the RMSD would be 1.65 Å between these conformers.

To identify all but one of the `b3lyp` conformers, the largest backbone RMSD would be 1.16 Å, between `kaplan28` and `b3lyp49`, where the RMSD would be 1.62 Å. To identify the 5 lowest-energy `b3lyp` conformers, the maximum backbone RMSD would be 0.0828 Å, and the maximum RMSD would be 1.11 Å. These tests have shown that it is important to include the *Kaplan* population when considering the final set of conformers, since the top `pmem` alone would not account for the best overall conformer (scored by energy). In fact, the best conformer came from the 5th ranked `pmem` in the ring.

4.4 Penta-Alanine Peptide Dataset

This section will cover the tests related to a conformational search of Penta-Alanine (PA) peptide – essentially five connected alanines, as would occur in a protein fragment. The structure of PA, shown in Figure 4.9 was taken from Toon Verstraelen's[3] set of conformers, where each structure was optimised with *Gaussian* using B3LYP/6-311+G(2df,p). The first conformer was used, from the set of 103 unique structures, as input to *Kaplan*. Note that this version has methyl caps, whereas the structure found in the *PubChem* database for the Penta-L-alanine has COOH and NH_2 caps. As with the amino acid structures, the methyl cap is intended to mimic the alpha carbon of an attached amino acid.

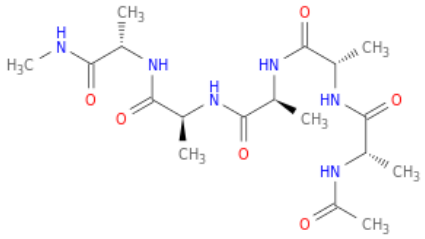


Figure 4.9: A 2D representation of penta-alanine peptide.

As before, each structure from the external dataset and the final structures from the *Kaplan* datasets were optimised via *Openbabel* with the MMFF94 forcefield until 10 identical energies were achieved, sampling every 100 iterations. Compared with the amino acids, this molecule is more difficult to optimise. In *Kaplan*, PA is represented by 36 dihedral angles, whereas the largest number of dihedral angles for an amino acid structure was 15 for arginine and asparagine. Five *Kaplan* experiments were performed; their inputs are summarised in Table 4.4.

A boxplot of the optimised energies per experiment was generated and is shown in Figure 4.10. The experiment codes are available in Table 4.4. From these results, *Kaplan* was not able to find the lowest energy PA structure from the B3LYP dataset, but increasing the number of mating events and lowering the number of conformers per `pmem` significantly lowered the conformer energies. Therefore, the lowest-energy conformers may be found through further evolution. Changing the extinction operator did not significantly impact the results for only 1000 `mevs`.

[3]Toon is a professor from Ghent University in Belgium who collaborates with the Ayers lab.

Expt. (abbr.)	num mevs	opt init geom	num geoms	confs optimised	No opt needed	identical pairs (RMSD < 0.1)	lowest energy (kcal/mol)
b3lyp	-	-	-	103	0	0	4.57
kaplan (k)	200*	N	50	2x50	16	0	15.06
k-asteroid (kas)	1000	N	50	2×50	13	5	15.95
k-plague (kp)	1000	N	50	2×50	15	2	10.46
k-agathic (kag)	1000	N	50	2×50	10	14	15.16
k-deluge (kd)	1000	N	50	2×50	11	13	16.51
k-plague2 (kp2)	10000	Y	20	5×20	27	74	7.72

Table 4.4: Summary of experiments performed on the PA peptide. The extinction probability was 0.005 (∼1/200 `mevs`) for the indicated extinction operator in the experiment name; otherwise, no extinction events were applied. In the num geoms column, the first digit indicates how many `pmems` were taken from the ring for optimisation. The last three columns indicate: the number of conformers for which no local optimisation was needed, how many identical pairs were found in the final set of conformers, and the lowest energy found from the selected `pmems`. *this experiment used the `stop_at_conv = 50` input parameter.

Another set of boxplots (Figure 4.11) were made to determine the number of iterations of the forcefield needed to fully optimise each conformer. The boxplot ranges were similar across the experiments, although the size of the Interquartile Range (IQR) grew for the kp2 experiment. Therefore, it may be beneficial for the experiment to use more than 100 iterations in the intermediary steps. Using *RDKit*'s implementation of the optimiser may take fewer iterations, as 1000 iterations was found to be sufficient for *RDKit* conformers, compared to 2500 for *Kaplan* conformers for the amino acids experiment.

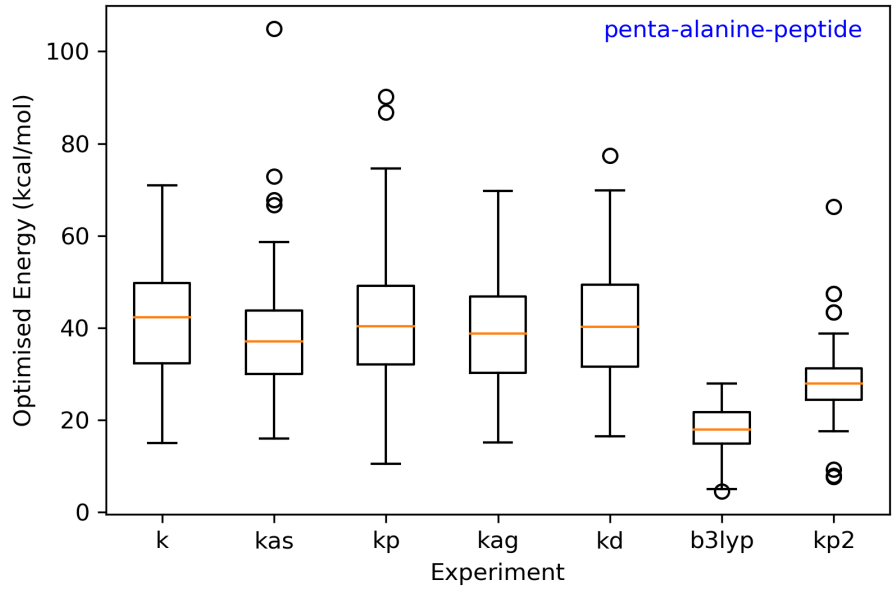


Figure 4.10: Boxplots to show the optimised energy distributions for the PA peptide.

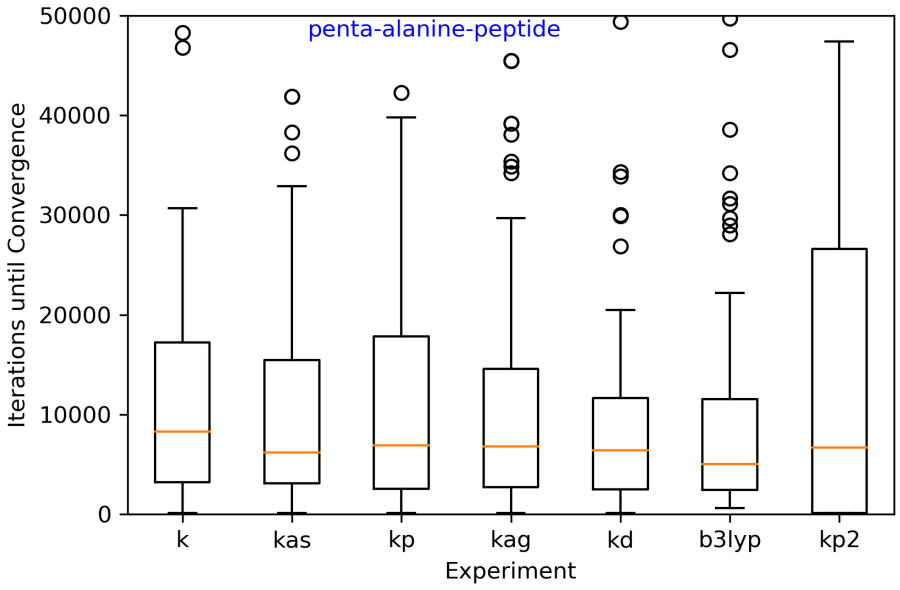


Figure 4.11: Boxplots to show the number of MMFF94 iterations needed to converge the energy for each experiment on the PA peptide. Some outliers were cropped as to make the IQRs more visible.

Chapter 5

Summary

This chapter will revisit the major points from the previous chapters. Possible future work and ideas for improvements are also discussed here.

5.1 Conclusions

This section will recount the main results from Chapters 2, 3, and 4.

5.1.1 Comparing Conformer Searching Methods

In Chapter 2, freely-available software tools used for conformer searching were discussed. The author presented ways in which to view, analyse, and compare conformers, mostly centred around root-mean-square-deviation (RMSD) and energy calculations. Examples of SMILES and InChI strings were provided as a means to convey molecular structure through line notation. *PubChem* can be used get 3D structures for most input molecules, provided they are calculable with the MMFF94 forcefield, and have fewer than 51 non-hydrogen atoms, 16 rotatable bonds, and 6 stereocenters. In the cases where a 3D *PubChem* structure is not available, any of the conformer searching packages discussed can generate a valid initial structure using a SMILES string as input.

Openbabel's tools for locally optimising conformers were used on the *PubChem* amino acid structures (with NH_2 and COOH caps). The author's contribution to fixing a software bug was also mentioned. For 16 of the amino acids, 60% of the overall energy gain from optimisation was achieved in the first 100 steps; attaining the last 40% meant 400-6200 more steps. None of the *PubChem* structures were fully optimised, and underwent an average energy decrease of 4.85 kcal/mol. Therefore, using a small amount of optimisation significantly improved the energies of the starting structures.

Avogadro was used to perform a conformer search on the amino acids, but the results were discarded, since the conformer energies were not correct (based on calculations on the same structure using *RDKit* and *Openbabel*). Avogadro could not return more than one structure, and was only able to find a single rotatable bond for most of the amino acids.

An amino acid conformer search, with *PubChem* structures as input, was performed using *Frog2*, *Balloon*, *Openbabel* - scored using RMSD and energy, *Confab*, *RDKit* (with and without optimisation), and *Kaplan*. Though the results are not directly comparable, the author instead described the ease of use for each package, including available documentation and required programming skills.

The main benefits of *Kaplan* is that it is an accessible conformer searching package that can also be used to perform analysis, including RMSD, energy, and optimisation. It has features such as choosing which dihedral angles to modify, so the user doesn't have to search the conformational space of a whole molecule unless it is required. Since *Kaplan* returns a `ring`, the user has access to many conformers. The population can be constructed randomly, or using other conformers as input.

Of the techniques, *Openbabel*, *RDKit*, and *Kaplan* allow for a specific number of conformers to be requested as output, whereas the other programs output a variable number of geometries. *Balloon*, for example, only returned one geometry for 7 of the 20 amino acids (supposedly the best structure); it is a more aggressive program when it comes to detecting and throwing away identical conformers. *Balloon* also has a distance-geometry (DG) based conformer search, but this was not tested in favour of using *RDKit*'s DG techniques. In contrast to *Balloon*, *Confab* output a

large number of conformers for each amino acid, whereas *Frog2* output a variable number of conformers up until a maximum value (set to 50).

Of the techniques tested, *Confab* was, in the author's opinion, the best explained, whereas the documentation for *Openbabel*'s conformer searching genetic-algorithm was essentially non-existant. *Balloon* has a paper to explain its techniques, but is closed source and does not report much during its conformer search. *RDKit* has papers to explain its DG techniques, as well as examples online as to how to use the software. *Frog2* is easy to use since it is available online, but the state of the package from a scripting point of view (also in terms of code quality) is poor.

Rings are a point of difficulty in conformer searching, since ring dihedrals cannot be easily manipulated without causing structural damage (i.e. breaking bonds). *Frog2* generates rings by keeping a structural database from which to sample ring structures (based on matching SMARTS queries), but it cannot explore ring conformations. *Balloon* can handle ring generation, and, when a conformer search was performed for cyclohexane, a twisted boat conformer was found alongside the chair flip of the input structure. If the user wishes to sample conformations of larger rings or multiple rings, it may be useful to implement a *Balloon* conformer search. For *Confab* and *Openbabel*, it is difficult to determine whether ring conformations can be generated, because the documentation for such procedures is not readily available. Two attempts were made to implement ring conformations in *Kaplan*, but neither succeeded in the task.

The results from *Kaplan* and *RDKit* (with optimisation) were similar in terms of the conformer energies. In these experiments, *RDKit* conformers were fully optimised with the MMFF94 forcefield via BFGS. *Kaplan* conformers were optimised with conjugate gradients, and the number of steps was based on the parametrisation performed for the *PubChem* structures. *Kaplan* was able to find the best energy for 17 of the 20 amino acids, where 5 of those energies were also found by *RDKit*. *RDKit* found the best conformer for 3 of the amino acids.

5.1.2 *Kaplan* Technical Details

In Chapter 3, the author's conformer searching program *Kaplan* was explained in detail. Rather than designing input structures from scratch, *Kaplan* uses *PubChem* and *Openbabel* to generate initial coordinates. To improve upon the typical torsion-driven approach, *Kaplan* applies intermittent local geometry optimisation using conjugate gradients via the MMFF94 forcefield. Therefore, the potential wells in the energy landscape can be exploited through numerical optimisation and by means of dihedral mutations. Enabling the local optimisation makes sure that each structure generated is valid (no atom clashes). If the user only runs the algorithm for a few mating events, then the structures returned will still have the correct connectivity (which was not the case for previous versions of *Kaplan*).

There are many options in *Kaplan* that can help a user to improve upon their conformer searching results, such as removal of hydrogens from RMSD calculations, modifications to the coefficients to promote energy over RMSD or vice versa, selection of preferred dihedral angle values and atom indices, convergence criteria (based on number of mating events or on condition of no improvements being made), and multiple energy-calculation options from both quantum chemistry and classical forcefields. Furthermore, extinction operators are available to promote new growth around the `ring` in the event that a conformer search stagnates. *Kaplan* handles output directory structure such that multiple jobs can be run in an organised way. Furthermore, it is possible to rerun old conformer searches using old solutions as input. All of the input parameters are presented in a clear manner via inputs.txt, and the course of evolution can be tracked via the stats output file. Aside from the evolutionary algorithm, there are a number of structure-based tools available that allow the user to interact with and investigate conformer geometries.

5.1.3 *Kaplan* Conformer Searching

Chapter 4 conducted an analysis of an amino acid conformer dataset, taken from the supplementary material of Yuan et al. [81]. The structures from this dataset were used as inputs to *Kaplan* to perform a conformer search for each methyl-

capped amino acid structure. The energies of the conformers, as calculated using the MMFF94 forcefield, were compared between the B3LYP, MP2, and *Kaplan* conformers. The relative energy distributions for *Kaplan* were significantly different from the other datasets for the following amino acids: glycine, tryptophan, tyrosine, alanine, proline, arginine, and methionine. The rest of the amino acids had overlapping IQR in the relative energy boxplots. The energies for the *Kaplan* conformers should be lower in most cases, because the *Kaplan* structures underwent some forcefield optimisation, whereas the B3LYP and MP2 conformers were only optimised with quantum chemical methods.

The results for arginine were wrong, because *Openbabel* incorrectly parsed the input structure, and determined the multiplicity to be 2 instead of 1. Therefore, any energies and RMSD values for arginine in this chapter should be ignored. Despite these issues, *Kaplan* was able to identify important conformers for arginine, including those with hydrogen bonding between the side chain and the peptide backbone.

To further verify that *Kaplan* had found the correct conformers, the quantum and *Kaplan* conformers for methionine were exhaustively optimised using the MMFF94 forcefield. Then, the RMSD was calculated between the two datasets. *Kaplan* was able to identify the same global minimum as the quantum dataset for methionine. Furthermore, the top 5 conformers, ranked by energy, were matched with *Kaplan* conformers by a maximum RMSD of 0.0828 Å (not considering hydrogens). To identify all 57 of the methionine conformers, *Kaplan* would have a maximum RMSD from one of its structures of 1.28Å or 1.65Å, ignoring and considering hydrogens respectively.

Some of the conformers returned by *Kaplan* were identical, because the conformers were taken from multiple `pmems` (whose fitness does not count RMSD calculated between `pmems`). Surprisingly, identical conformers were found in the quantum chemical dataset, implying that not all of the structures are unique for some amino acids.

In the case of one amino acid, the fitness function was being exploited, such that the best `pmem` did not find new structures; instead, the structures with high RMSD were duplicated. Therefore, a modification should be made to penalise identical

geometries within one `pmem`. For some cases, when the RMSD was calculated with `reorder`, the RMSD was higher than without `reorder`. The author believes this to be a problem with the *rmsd* package, since it does not consider the case where no reordering should be performed.

Lastly, the Penta-Alanine (PA) peptide dataset was briefly introduced. This molecule has more than twice the number of rotatable bonds than the largest amino acid, making it a much tougher problem. A start was made in determining the best parameters to use in a conformer search. For only 1000 mating events, the extinction operators did not have enough impact to significantly improve the conformer energies. However, by reducing the number of conformers per `pmem` from 50 to 20, and increasing the number of mating events by a factor of 10, the energy distribution approached the energies of the optimal quantum-chemistry dataset. The energy difference between the lowest quantum conformer and the lowest *Kaplan* conformer was about 3 kcal/mol. The number of MMFF94 iterations required to fully optimise the PA conformers was between 5,000 and 20,000; therefore, a more efficient optimiser may need to be used for larger molecules, such as BFGS from *RDKit*.

5.2 Improvements to *Kaplan*

This section will explain how *Kaplan* can be improved upon, from both a useability and an efficiency standpoint.

5.2.1 Enable Ring-Conformations

Future versions of *Kaplan* should be able to induce movement in rings, such as boat-chair flips for cyclohexane and buckling in cyclobutane, while maintaining reasonable angles and bond lengths. Rather than having these as dihedral angles in the representation, it might make sense to have a binary decision modifier - for example to cause a chair flip in cyclohexane, as this places the axial groups equatorial and vice-versa. In some cases where the cyclohexane ring has bulky substituents,

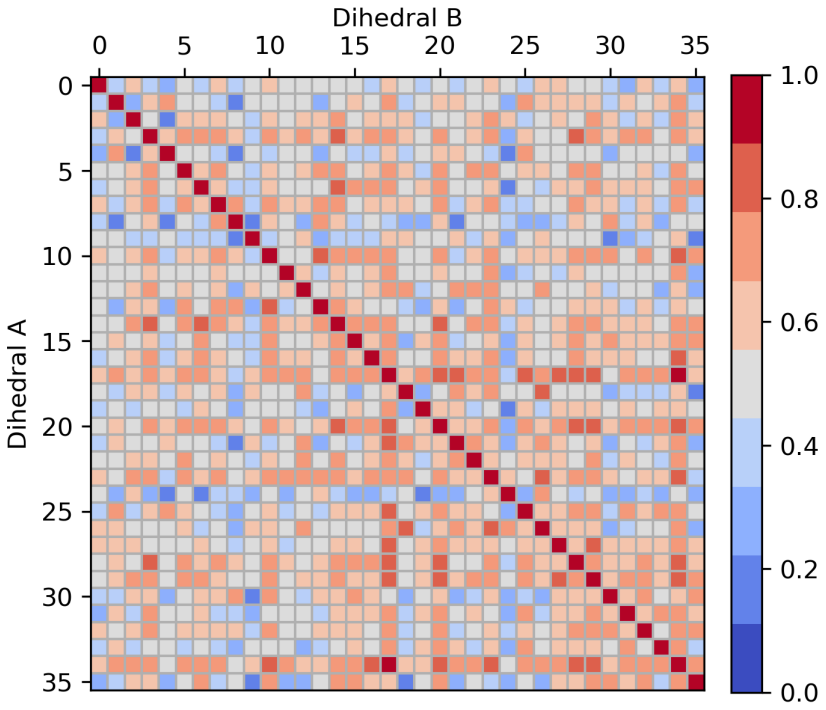


Figure 5.1: Example heatmap for the penta-alanine peptide structure, which is the structure for alanine except in a 5-amino acid chain. Each square represents a pair of dihedral angles, and the colour indicates that the dihedral angles have a high R^2 value (red) or a low R^2 value.

placing the larger groups equatorial to the ring can decrease the energy significantly [1].

5.2.2 Correlated Dihedral Angles

The original reason for testing ring conformers in *Kaplan* was to ensure the dihedral angles within a ring could be identified as correlated - i.e. when one dihedral in a ring changes, so do the other dihedral angles. For reasons discussed previously, the exploration of ring conformations is not yet possible in *Kaplan*. Early work into investigating dihedral angle correlations was done in the form of heatmap generation - an example of which is given in Figure 5.1.

Another consideration might be: given a molecule and a set of dihedral angles, is there a priority sequence for the dihedrals (i.e. order them in terms of importance to the energy). The question to answer here would be if the molar mass is the defining factor for dihedral rank - does a larger substituent always dominate a smaller substituent when looking at the conformational energy change?

5.2.3 Failed Test Case

From random tests, a problem was found for *PubChem* compound: 1,4-Diazabicyclo [2.2.2] octane (CID=9237). Here is the 2D plot of the compound:

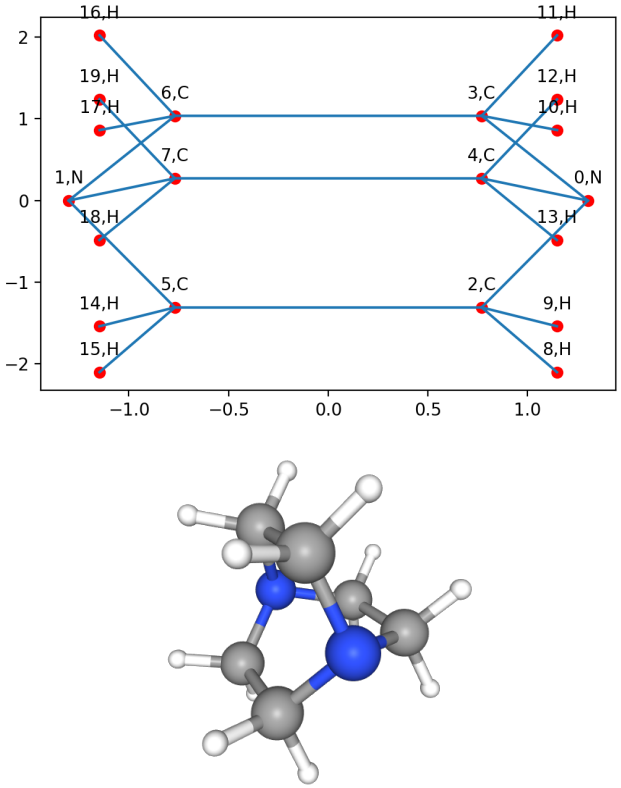


Figure 5.2: Failed test case CID=9237, which has three fused rings.

This compound has fused rings, and, since some rings share the same sets of atoms, the `remove_ring_dihed` function from the geometry module failed to

remove all ring dihedrals. The `remove_ring_dihed` function relies on *Openbabel*'s `OBMolRingIter` iterator, which iterates over all the rings in an `obmol` object. The rings[1] for compound CID=9237 are:

```
from kaplan.inputs import Inputs
from kaplan.geometry import get_rings
inputs = Inputs()
inputs.update_inputs({
    "struct_input": 9237,
    "struct_type": "cid",
})
rings = get_rings(inputs.obmol)
print(rings)
print(inputs.diheds)
# gives the following output
# [(6, False, '', [1, 5, 2, 0, 3, 6]),
#  (6, False, '', [4, 7, 1, 5, 2, 0])]
# [(4, 0, 3, 6), (3, 0, 4, 7), (7, 1, 6, 3), (6, 1, 7, 4)]
```

The `rings` list contains four items: the size of the ring (in atoms), a boolean where `True` means the ring is aromatic and `False` otherwise, the type of ring (here empty, meaning not recognised by *Openbabel*), and lastly the atoms in the ring (as they are attached in order around the ring). Notice that *Openbabel* only recognises two rings, but there are in fact three possible rings from the given coordinates. The missing ring would be:

```
[(6, False, '', [1, 7, 4, 0, 3, 6])]
```

Therefore, if this ring was to be identified, then the last four dihedral angles for compound CID=9237 would be removed. The *Openbabel* function that identifies rings is probably meant to identify the rings such that all atoms in the molecule that are in a ring are in at least one of the returned rings. The *Openbabel* function does not identify "redundant" rings. Future work for *Kaplan* would be adding a feature

[1]Note, rings as discussed here mean structural rings that are part of a molecular geometry, not the `ring` for the *Kaplan* conformer population.

to exhaustively identify all rings.

5.2.4 Stereochemistry

Stereochemistry is an important consideration for molecule design, since some stereoisomers are the preferred metabolites or catalysts or substrates under biological conditions. Questions for *Kaplan* to answer would be: does *Kaplan* preserve stereochemistry? Given a molecule that has a stereocenter, is there a way to generate both stereoisomers, or, conversely, can an input parameter be enabled such that only one stereoisomer is produced by the conformer search?

5.2.5 Miscellaneous Improvements

- 2D/3D images could use some tweaking to prevent atom overlap
- there are several terms in the MMF94 forcefield; it may be beneficial to only compute those that are changing during the course of evolution (however, the computational benefit would be small and the potential risk of incorrectly applying the calculations is high) - this idea comes from the *Frog2* design decision to only account for van der Waals changes in their energy calculations when ranking conformers
- the Inputs module was not designed thoroughly, and may benefit from some refactoring.
- the program would appeal to a wider range of users if the barrier to access it was lower; for example: making a web-based interface to run the program (or to configure input files) would be helpful to users with a lower knowledge of computer programming
- distance geometry techniques could be used to evaluate a conformer - for instance, the fitness could be proportional to how well a set of Cartesian coordinates fit in the constraints (usually it is the other way around - trying to guess Cartesian coordinates from constraints).

- mutations weighted based on a conformer's energy, so poor structures get mutated more often.

5.2.6 Volume- & Area-based Comparisons

Other, more-expensive calculations can be done to compare conformers geometrically. One such method is to calculate the solvent-accessible surface area. In this method, a solvent molecule (such as water) is approximated as a sphere of radius r_i, and the molecule is a collection of atoms, where each atom is represented by a van der Waals sphere. Then, the solvent molecule is rolled over the surfaces of the atoms to map out the molecular surface [59]. From this type of analysis, it might be possible to compare a closed, hydrophobic form of a conformer with an open, hydrophilic form of a conformer or vice-versa, depending on the ratio of polar to non-polar surface atoms.

Another calculation that could be performed would be volume of the conformer - again, each atom would be approximated as a van der Waals sphere. The volume would be an indicator of how tightly-packed the atoms were, and it could indicate the amount of overlap - either positive or negative (i.e. molecule stability). Good overlap here could refer to beneficial electrostatic interactions, for example.

5.2.7 Expansion to Proteins

Kaplan is designed as a small-molecule conformation search tool. It would be an interesting challenge to approximate the conformers of proteins - essentially tackling the protein-folding problem. If a large molecule is passed to *Kaplan* (for example, this was tried with the PDB structure 6dvw, which has about 21,000 atoms), the *GOpt* program will return a `MemoryError`. Smaller protein chunks (for example, the PDB structure 1r48, containing a coiled-coil, of approximately 1000 atoms) also run out of memory, even when using minimal size for the population (see sample script below). A system with more than 16GB RAM may be able to process a small protein request, but this requirement would make *Kaplan* an unreasonable choice

for most users. Ideally, users should be able to run this conformer search on their own machine. To process larger molecules, the number of dihedral angles would have to be lowered considerably; it may also be required that the amino acids be approximated as globular entities instead of explicit molecules.

```
from kaplan.control import run_kaplan
run_kaplan({
    "struct_input": "1r48.xyz",
    "struct_type": "xyz",
    "charge": 0,
    "multip": 1,
    "num_geoms": 2,
    "num_mevs": 5,
    "num_slots": 10,
    "init_popsize": 5,
    "normalise": False,
    "mating_rad": 2,
})
```

Appendix

A.1 Mathematical & Coding Conventions

Symbol	Description	Representation
$\mathbb{N}$	Set of natural numbers	{1, 2, 3, ...}
$\mathbb{Z}$	Set of integers	{..., -2, -1, 0, 1, 2, ...}
$\mathbb{Q}$	Set of rational numbers	$\{p/q \mid (p,q) \in \mathbb{Z}, q \neq 0\}$
$\mathbb{I}$	Set of irrational numbers	$\{x \in \mathbb{R} \mid x \notin \mathbb{Q}\}$
$\mathbb{R}$	Set of real numbers	$\mathbb{I} \cup \mathbb{Q}$
π	transcendental number pi	3.14159...
e	transcendental number e	2.71828...
$\boldsymbol{I}_n$	identity matrix of size $n \times n$	$\begin{bmatrix} 1 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 1 \end{bmatrix}$
$\exists$	there exists	–
$\nexists$	there does not exist	–
$\in$	is in (a set)	–
$\notin$	is not in (a set)	–
$\cup$	union (of sets)	–
$\cap$	intersection (of sets)	–
$>$	greater than	–
$<$	less than	–
$\neq$	not equal to	–

Table A.1: Common mathematical symbols used in the text.

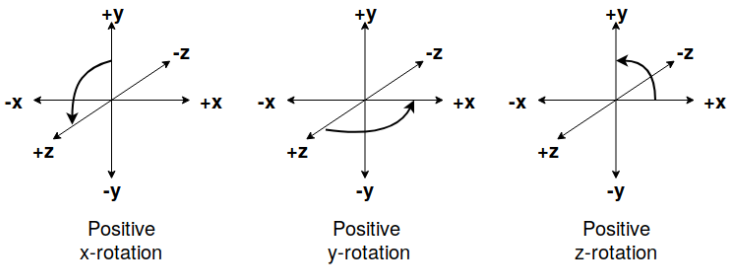


Figure A.1: The right-handed coordinate system follows counterclockwise rotation for the positive direction.

A.1.1 Coordinate System

By default, this work uses the right-handed coordinate system. In 3D space, the positive directions for x, y, and z are right, up, and towards the observer, respectively. Positive rotation occurs counterclockwise about the axis of rotation, as indicated in Figure A.1.

A.1.2 Cartesian versus Internal Coordinates

The cartesian coordinate system is a method for describing molecules in 3D space, whereby each atom has 3 pieces of information: an x coordinate, a y coordinate, and a z coordinate. Molecules can also be represented in 2D space, although the third dimension is required to properly compare conformers. A 2D representation is helpful for describing how all of the atoms are connected and in displaying stereochemistry.

Since bonds are not explicit in cartesian space, chemistry programs usually have threshold distances (which are dependent on atomic number) that designate whether atoms are connected via single, double, triple, etc. bonds.

An internal coordinate system is based on a reference atom (e.g. placed at the origin). The components of an internal coordinate system are: bond lengths, bond angles, and torsion (or dihedral) angles.

A.1.3 Sets

A **set** is a distinct collection of items or elements, whether of fixed or infinite size. An item a is denoted to be in, or an element of, set A by $a \in A$. If b is not in the set A, then $b \notin A$ is used. In combining two sets, A and B to generate a new set C, the **union** of these two sets is written as: $C = A \cup B$. To find elements common to both sets A and B as a new set D, the **intersection** is written as: $D = A \cap B$. The empty set, which contains no elements, is denoted by $\emptyset$. Members of a set are given in curly braces (i.e. $A = \{1,2,3\}$). Conditions for set membership follow the $|$ (should be read as, "such that" or "given"). For example, $A = \{x \in \mathbb{N} \mid x > 100\}$ contains all of the natural numbers that are greater than 100.

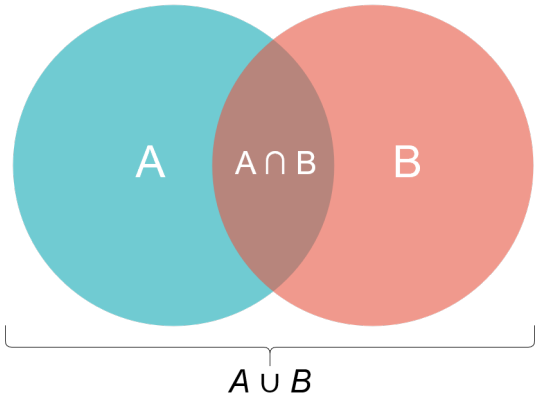


Figure A.2: The union and intersection for two sets A and B.

A.1.4 The `range` Keyword

Since *Python* was used to write many of the projects for this work, the pseudocode will follow a similar style to said language. For loops will often use the `range` designation, which has 1-3 inputs: starting index, ending index, and increment size, with the required argument being the ending index. The ending index is not included in the loop. For example: `range(1,11,2)` would return 1, 3, 5, 7, 9. Indexing in general will also start at 0, so `range(5)` would return 0, 1, 2, 3, 4.

A.1.5 Deepcopy

By default, *Python* passes objects (including lists, dictionaries, and other items) by reference. Furthermore, creating copies of the list by using `list2 = list1` notation does not create a new list in memory. Rather, `list2` will now point to the same place as `list1`. Consider the scenario where the centroid is subtracted from a *NumPy* array of coordinates:

```
import numpy as np

# function that subtracts the mean value of
# the corresponding dimension from each coordinate
def centroid(coords):
    coords -= coords.mean(axis=0)

# randomly generate molecule coordinates
mol_coords = np.random.uniform(
    low=0.0, high=10.0, size=(10,3)
)

print(mol_coords) # original coordinates

centroid(mol_coords)

print(mol_coords) # centroid coordinates
```

The final version of `mol_coords` is the centred version of the coordinates. If the user wishes to keep the original coordinates and still call the `centroid` function, then `deepcopy` from the standard library `copy` can be used:

```
from copy import deepcopy
import numpy as np

# function that subtracts the mean value of
# the corresponding dimension from each coordinate
```

```
def centroid(coords):
    coords -= coords.mean(axis=0)

# randomly generate molecule coordinates
mol_coords = np.random.uniform(
    low=0.0, high=10.0, size=(10,3)
)

# make a copy of the original coordinates
centred_coords = deepcopy(mol_coords)

centroid(centred_coords)

# mol_coords is still the same, only
# centred_coords has changed
print(mol_coords)
```

A.1.6 Mathematical Operators

To check for value equality, the == operator is used. To check for value inequality, the != operator is used. To determine if the memory location of two objects is equal, the `is` keyword is used. For example, two identical lists may be stored in separate locations and are thus not identical using `is`. When dividing values, the // operator represents floor division (i.e. $5//3 = 1$), and the modulo operator `%` gives the remainder (ex: $10 \% 3 = 1$).

A.1.7 Arrays and Matrices

The *Python* library *NumPy* (shortened to `np`, which stands for numerical *Python*) is used to generate array structures for some of *Kaplan*. It is also used extensively in the *rmsd Python* package. Rather than explicitly writing the form of each array, a shorthand notation will be used that also represents the type of the variable as it appears in the code.

```
integer_array = np.array(shape=(n,m), dtype="int")
              = np.array((n,m), "int")
  float_array = np.array((n,m), "float")
```

The above code is used to represent the following *m*x*n* matrix $\boldsymbol{A}$, where m is the number of rows and n is the number of columns:

$$\boldsymbol{A}_{m,n} = \begin{bmatrix} x_{00} & x_{01} & x_{02} & \cdots & x_{0(n-1)} \\ x_{10} & x_{11} & x_{12} & \cdots & x_{1(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ x_{(m-1)0} & x_{(m-1)1} & x_{(m-1)2} & \cdots & x_{(m-1)(n-1)} \end{bmatrix}$$

Where $x_{ij} \in \mathbb{R}$ for floating point arrays, and $x_{ij} \in \mathbb{Z}$ for integer arrays. To get the element at the i^{th} row and j^{th} column of the matrix $\boldsymbol{A}$, the following is written: $\boldsymbol{A}[i][j] = x_{ij}$. Indexing starts at zero.

The transpose of a matrix is given by $\boldsymbol{A}^{\boldsymbol{T}}$, and it represents the operation whereby all elements have their row and column swapped.

$\forall x_{ij} \in$ matrix $\boldsymbol{A}_{m,n}$, $\boldsymbol{A}[i][j] = \boldsymbol{A}^{\boldsymbol{T}}[j][i]$, where $0 < i \leq m$, $0 < j \leq n$

For example:

$$\boldsymbol{A}_{3,2} = \begin{bmatrix} 4 & 5 \\ 2 & 3 \\ 1 & 0 \end{bmatrix}, \qquad \boldsymbol{A}^{\boldsymbol{T}}_{2,3} = \begin{bmatrix} 4 & 2 & 1 \\ 5 & 3 & 0 \end{bmatrix}$$

The dot product between two matrices, $\boldsymbol{A}$ and $\boldsymbol{B}$ (written $\boldsymbol{A} \bullet \boldsymbol{B}$), can only be computed when the number of columns for the first matrix equals the number of rows for the second matrix (i.e. $\boldsymbol{A}_{m,n} \bullet \boldsymbol{B}_{n,k} \mid m,n,k \in \{0,\mathbb{N}\}$). The dot product affords a new matrix, $\boldsymbol{C}_{m,k}$. The procedure for calculating the dot product is as follows:

```
given mxn matrix A
given nxk matrix B
empty mxk matrix C
for row in range(m):
```

```
    for column in range(k):
        C[row][column] = sum(
            A[row][i]*B[i][column] for i in range(n)
        )
return C
```

The determinant of a square matrix $\boldsymbol{A}_{m,m}$ affords a scalar value and is denoted using vertical bars $|\boldsymbol{A}|$ or as $\det(\boldsymbol{A})$. For any matrix $\boldsymbol{A}_{m,m}$ of size $m > 3$, the determinant can be calculated by recursively dividing the matrix into $(m-1),(m-1)$ pieces called *minors*.

Determinant of 1x1 Matrix

$$|\boldsymbol{A}_{1,1}| = x_{00} = \boldsymbol{A}[0][0]$$

Determinant of 2x2 Matrix

$$|\boldsymbol{A}_{2,2}| = \begin{vmatrix} a & b \\ c & d \end{vmatrix} = ad - bc$$

Determinant of 3x3 Matrix

$$|\boldsymbol{A}_{3,3}| = \begin{vmatrix} a & b & c \\ d & e & f \\ g & h & i \end{vmatrix} = a\begin{vmatrix} e & f \\ h & i \end{vmatrix} - b\begin{vmatrix} d & f \\ g & i \end{vmatrix} + c\begin{vmatrix} d & e \\ g & h \end{vmatrix}$$

An **orthogonal matrix**, $\boldsymbol{A}$, is a square matrix whose transpose is equivalent to its inverse: $\boldsymbol{A}^{-1} = \boldsymbol{A}^{\boldsymbol{T}}$. A matrix is invertible if $\exists$ a square matrix $\boldsymbol{B}$ such that: $\boldsymbol{AB} = \boldsymbol{BA} = \boldsymbol{I}_n$.

The **complex conjugate** of an imaginary number of the form $a + ib$ is $a - ib$. In this relation, a and b are real numbers, and $i = \sqrt{-1}$. In the text, the complex conjugate is denoted using a * - for example $f^*(x)$.

A.1.8 Lists and Dictionaries

The type of a *Kaplan* variable may also be a list or a dictionary. A list is similar to an `np` array, except it supports a different set of operations and is stored differently

in memory. If the variable is a list, `list()` will be used. A dictionary in *Python* is a set of key-value pairs. If a variable is a dictionary, `dict()` will be used. Below are examples of lists and dictionaries.

```
 list_n = list([0,1,2,3,...,n-1])
list_33 = list([[1,2,4],
                [2,3,5],
                [2,4,1]])
 dict_n = dict({"key1": "value1",
                "key2": "value2",
                 ...,
                "keyn": "valuen"})
```

The notation for accessing elements of a list (line 1) or values in a dictionary (line 2) is:

```
1 list_n[5]       # 5
2 dict_n["key1"] # "value1"
```

where the evaluation of the line is given after the hash (#) symbol.

A.1.9 Singular Value Decomposition

The SVD theorem states that a real matrix, $\boldsymbol{A}$, may be decomposed into $\boldsymbol{A} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^{\boldsymbol{T}}$, where:

- $\boldsymbol{U}, \boldsymbol{\Sigma}, \boldsymbol{V}$ are unique.
- $\boldsymbol{U}$ and $\boldsymbol{V}$ are column orthonormal, meaning the columns of U and V are orthogonal unit vectors and $\boldsymbol{U}^{\boldsymbol{T}}\boldsymbol{U} = \boldsymbol{I}$ and $\boldsymbol{V}^{\boldsymbol{T}}\boldsymbol{V} = \boldsymbol{I}$ ($\boldsymbol{I}$ is the identity matrix)
- $\boldsymbol{\Sigma}$ is diagonal, containing positive singular values that are ordered from largest to smallest along the diagonal.

A.2 Scripts & Input Files

This section contains scripts and input files used to run tests or external programs.

A.2.1 *RDKit* Script with Optimisation

This script was written to generate optimised conformers using *RDKit* starting from SMILES strings. It is a standalone script that generates csv and sdf output files.

```
import csv
from kaplan.tools import amino_acids, all_pairs_gen
from vetee.coordinates import pubchem_inchi_smiles
from rdkit import Chem
from rdkit.Chem import AllChem


for i, mol in enumerate(amino_acids):

    # number of conformers to make with RDKit
    n = 10

    # get the isomeric smiles for each amino acid using vetee
    j = pubchem_inchi_smiles("name", mol)
    smiles_iso = j["_smiles_iso"]

    # create RDKit molecule from smiles string
    m = Chem.MolFromSmiles(smiles_iso)
    # returns None upon failure
    assert m is not None
    # set the name property so it appears in the sdf
    m.SetProp("_Name", mol)

    # add hydrogens
    mwh = Chem.AddHs(m)

    # generate multiple conformers at once
    # this runs ETKDG n times
    conf_nums = AllChem.EmbedMultipleConfs(mwh, numConfs=n)

    # optimise conformers and calculate the forcefield energy
    # energies is now a list of tuples
    # each tuple has the form (int, float), where int should
```

```
# be zero if the calculation converged
# the float is the energy is units of kcal/mol (assumed)
energies = AllChem.MMFFOptimizeMoleculeConfs(
    mwh, maxIters=1000, mmffVariant="MMFF94"
)

# make sure all of the energies have converged
assert all(energies[i][0] == 0 for i in range(n))

# align conformers to one another and calculate
# RMS (assuming this is RMSD)
mol_RMSD = []
for i, j in all_pairs_gen(n):
    rmsd = AllChem.GetConformerRMS(mwh, i, j, prealigned=False
        )
    mol_RMSD.append((i, j, rmsd))

# fieldnames are the header for the csv
energy_fn = ["conf_num", "energy␣(kcal/mol)"]
rmsd_fn = ["confa", "confb", "rmsd"]

with open(f"{mol}_energies.csv", "w") as f:
    fcsv = csv.DictWriter(f, fieldnames=energy_fn)
    fcsv.writeheader()
    for i, energy in enumerate(energies):
        fcsv.writerow({
            energy_fn[0]: i,
            energy_fn[1]: energies[i][1],
        })

with open(f"{mol}_rmsds.csv", "w") as f:
    fcsv = csv.DictWriter(f, fieldnames=rmsd_fn)
    fcsv.writeheader()
    for i, j, rmsd in mol_RMSD:
        fcsv.writerow({
            rmsd_fn[0]: i,
            rmsd_fn[1]: j,
```

```
                rmsd_fn[2]: rmsd,
            })

    # write conformers to a file (as to save their coordinates)
    for i in range(n):
        with open(f"{mol}_{i}.sdf", "w+") as f:
            print(Chem.MolToMolBlock(mwh, confId=i), file=f)
```

A.2.2 Basic *RDKit* Script

This script was used to generate conformers (without any optimisation) using *RDKit*. It was invoked from the bash script found in Section A.2.3.

```
from sys import argv
from rdkit import Chem
from rdkit.Chem import AllChem


def rdkit_conf_search(sdf_file, mol_name, num_confs, RNS):
    m = Chem.rdmolfiles.SDMolSupplier(sdf_file, removeHs=False)
    m = m.__next__()
    # returns None upon failure
    assert m is not None
    # set the name property so it appears in the sdf
    m.SetProp("_Name", mol_name)

    # generate multiple conformers at once
    # this runs ETKDG n times
    conf_ids = AllChem.EmbedMultipleConfs(
        m, numConfs=num_confs, randomSeed=RNS
    )

    # write conformers to a file
    writer = Chem.rdmolfiles.SDWriter("output.sdf")
    for conf in conf_ids:
        writer.write(m, confId=conf)
```

```
if __name__ == "__main__":
    # required args: sdf file, molecule name,
    # number of conformers, and random number seed
    assert len(argv) == 5
    rdkit_conf_search(argv[1], argv[2], int(argv[3]), int(argv[4])
        )
```

A.2.3 Conformer Search Script

This script was used to generate conformers using *RDKit* (via script in Section A.2.2), *Balloon*, and *Openbabel* (including *Confab*). The script was run from a directory containing a subdirectory for each amino acid (subdirectory format example: 0-asparagine). Each amino acid sdf input file was downloaded from *PubChem*, and placed in each subdirectory.

```
#!/bin/bash

# bash script to run a conformer search for each amino acid
# four conformer searches can be performed:
# GA: energy-scored, rmsd-scored
# Confab
# Balloon
# RDKit (via external Python script)

search_type=rdkit       # should be one of: rmsd, energy, confab,
    balloon, rdkit

# for Balloon search, current_dir should contain MMFF94.mff and
    balloon executable
# for RDKit search, current_dir should contain rdkit_conformer.py
current_dir=`pwd`       # where to place output files

num_confs=10            # number of conformers for Balloon/rmsd/
    energy/RDKit
RNS=500                 # random number seed for Balloon/RDKit
```

```
num_generations=1000    # nGenerations for Balloon (iterations of
    GA)

# Confab parameters (defaults are given here)
rmsd_cutoff=0.5         # RMSD cutoff is 0.5 Angstroms
energy_cutoff=50.0      # energy cutoff is 50kcal/mol
num_samples=1000000     # number of conformers to test is 1
    million

energyfile=$current_dir/energies.txt
rmsdfile=$current_dir/rmsds.txt
logfile=$current_dir/conformer.log

# clean up old files if rerunning
rm $current_dir/*/output.sdf
rm $current_dir/*/inputcoords.xyz
rm $current_dir/*/output*.xyz
rm $energyfile
rm $rmsdfile
rm $logfile

# for each amino acid directory
for d in `ls -d */`; do
    echo Running $d analysis
    echo $d >> $logfile
    echo $d >> $energyfile
    echo $d >> $rmsdfile
    cd "$current_dir/$d"
    infile=`echo *.sdf`

    # use Openbabel to run a Confab conformer search on Pubchem
       sdf file
    if [ "$search_type" == "confab" ]; then
        obabel $infile -O output.sdf --confab --verbose --rcutoff
           $rmsd_cutoff --ecutoff $energy_cutoff --conf
           $num_samples >> $logfile
```

```
# run the balloon executable on the Pubchem sdf file
elif [ "$search_type" == "balloon" ]; then
    $current_dir/balloon -f $current_dir/MMFF94.mff --
        randomSeed $RNS --nconfs $num_confs --nGenerations
        $num_generations $infile output.sdf >> $logfile

# use RDKit script to run conformer search on Pubchem sdf file
elif [ "$search_type" == "rdkit" ]; then
    python ../rdkit_conformer.py $infile $d $num_confs $RNS

# run Openbabel GA conformer search on Pubchem sdf file
else
    obabel $infile -O output.sdf --conformer --nconf
        $num_confs --writeconformers --score $search_type >>
        $logfile
fi

# run obenergy on the output file
obenergy output.sdf | grep "TOTAL␣ENERGY␣=" >> $energyfile

# convert to xyz and calculate RMSD using reorder argument
# the -m argument means do multiple output files, starting
   with output1.xyz
obabel output.sdf -O output.xyz -m

# determine how many xyz files were made
xyzfiles=`find . -name "output*.xyz"`
count=0
for f in $xyzfiles; do
    count=$((count+1))
done

# if there is only 1 conformer produced,
# calculate the RMSD between it and the
# input file
if [ "$count" -eq 1 ]; then
    echo $count conformer >> $rmsdfile
```

```
        obabel $infile -O inputcoords.xyz
        calculate_rmsd --reorder output1.xyz inputcoords.xyz >>
           $rmsdfile
    else
        echo $count conformers >> $rmsdfile
        # iterate over all pairwise xyz files
        for ((i=1; i<=$((count-1)); i++)); do
            for ((j=$((i+1)); j<=$count; j++)); do
                calculate_rmsd --reorder output$i.xyz output$j.xyz
                    >> $rmsdfile
            done
        done
    fi

    # add trailing whitespace to separate each molecule
    echo >> $logfile
    echo >> $energyfile
    echo >> $rmsdfile

done
```

A.2.4 Glycine xyz file

```
10
xyz coordinate file for glycine
O      -1.6487       0.6571      -0.0104
O      -0.4837      -1.2934      -0.0005
N       1.9006      -0.0812       -0.009
C       0.7341       0.7867       0.0079
C      -0.5023      -0.0691        0.012
H       0.7326       1.4215      -0.8824
H       0.7464       1.4088       0.9069
H       1.8743      -0.6844      -0.8301
H       1.8887      -0.6969       0.8031
H      -2.4447       0.0839       -0.026
```

A.3 Statistics

This section of the Appendix covers statistical concepts that are needed to understand the analyses performed in the thesis. Examples of how to generate boxplots using *Python* are provided.

A.3.1 Median

The median is the middle value of an ordered set of values. If the number of elements in the set is odd, then the median is the value at `list[len(list)//2]`. If a set has an even number of elements, then the median is the average of the values at `list[len(list)//2 - 1]` and `list[len(list)//2]`. The `median` function can be imported from the standard *Python* library, `statistics`. An example is shown below for a list called `my_data`, where `>>>` indicates the *Python* prompt:

```
>>> from statistics import median
>>> my_data = [3, 4, 5, 7, 4, 3, 6, 7, 8, -3]
>>> sorted(my_data)
[-3, 3, 3, 4, 4, 5, 6, 7, 7, 8]
>>> len(my_data)
10
>>> median(my_data)
4.5
```

A.3.2 Percentiles

Percentiles indicate the value below which a certain percentage of elements from a list belong. The quartiles represent the values at which to cut an ordered list such that approximately 25% of the data points are between the lowest point and Q1, Q1 and Q2 (the median), Q2 and Q3, and Q3 and the largest point.

The Interquartile Range (IQR) is defined as the difference between the third and first quartiles, as given in Equation 1:

$$\text{IQR} = \text{Q3} - \text{Q1} \tag{1}$$

To obtain the Q1 and Q3 values, the `np.percentile` function can be used as follows:

```
my_data = [3, 4, 5, 7, 4, 3, 6, 7, 8, -3]
Q1, Q3 = np.percentile(my_data, [25, 75])  # 3.25, 6.75
```

To calculate the p^{th} percentile (using 0-based indexing):

1. Order values from smallest to largest.
2. Calculate the index, which is the percentile p times the number of elements minus 1. Example: for the 25^{th} percentile of 10 numbers, index $= 0.25 \times (10-1) = 2.25$.
3. If the index is a whole number, the p^{th} percentile is equal to the value at that index.
4. In the case where the index is not a whole number, both indices surrounding the index are considered. Using the same example from Step 2, indices 2 and 3 would be evaluated. Then, two values, i and j where $i < j$, are defined as the elements at these two indices. The returned percentile depends on the interpolation method used; *NumPy* uses `linear` interpolation by default. The percentile is then $i + (j-i) \times$ fraction, where fraction is equal to the decimal component between the two indices (e.g. 0.25 in the given example). Another option is `midpoint`, where the p^{th} percentile is the average value of the i and j.

Below are some examples of input lists and their corresponding index, Q1, and Q3 values, as calculated using the `linear` interpolation method:

```
Input: [3, 3, 4, 4, 5, 6, 7, 7, 8]
Length (N): 9
Percentile (p): 0.25
p*(N-1), Input[p*(N-1)]: 2.0, 4
```

```
Percentile (p): 0.75
p*(N-1), Input[p*(N-1)]: 6.0, 7
Q1: 4.0
Q3: 7.0

Input: [-20, -17, -4, 0, 3, 6, 30]
Length (N): 7
Percentile (p): 0.25
p*(N-1), Input[p*(N-1)]: 1.5, -17
Percentile (p): 0.75
p*(N-1), Input[p*(N-1)]: 4.5, 3
Q1: -10.5
Q3: 4.5
```

A.3.3 Boxplots

A boxplot is a type of plot that shows the distribution of a set of values. Boxplots have four main components: a line to indicate the median of the data (presented in orange in Figure A.3), a rectangle (the "box") that spans the Interquartile Range (IQR) from Q1 to Q3, whiskers (vertical lines above and below the box), and outliers (indicated in Figure A.3 as circles).

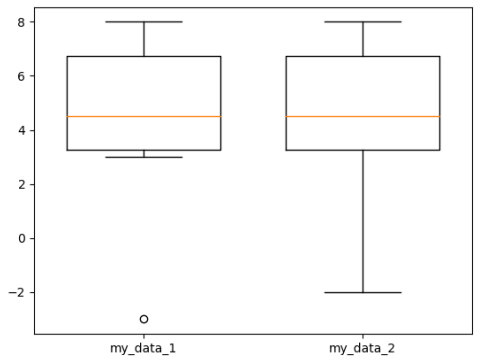


Figure A.3: Example boxplots for the data given in the text. This boxplot was generated with the command `plt.boxplot([my_data_1, my_data_2], labels=["my_data_1", "my_data_2"], widths=0.7)`.

In this thesis, the *Python* library *Matplotlib* is used to generate boxplots. The default plot inputs are used for all figures generated in this work. Example code showing how to obtain the limits for the box and the whiskers is given below.

```
>>> import matplotlib.pyplot as plt
>>> my_data = [3, 4, 5, 7, 4, 3, 6, 7, 8, -3]
>>> box = plt.boxplot(my_data)
>>> box["boxes"][0].get_xydata()
array([[0.925, 3.25 ],
       [1.075, 3.25 ],
       [1.075, 6.75 ],
       [0.925, 6.75 ],
       [0.925, 3.25 ]])
>>> box["whiskers"][0].get_xydata()
array([[1.  , 3.25],
       [1.  , 3.  ]])
>>> box["whiskers"][1].get_xydata()
array([[1.  , 6.75],
       [1.  , 8.  ]])
>>> plt.show()
```

Outliers ("fliers") in *Matplotlib* are points that are smaller than $\text{Q1} - 1.5 \times \text{IQR}$ or larger than $\text{Q3} + 1.5 \times \text{IQR}$. The set for outliers is given by Equation 2. The lower whisker extends to the first value from the ordered dataset that is greater than or equal to $\text{Q1} - 1.5 \times \text{IQR}$. Similarly, the upper whisker extends to the first value that is smaller than or equal to $\text{Q3} + 1.5 \times \text{IQR}$.

$$\text{outlier} = \{x \mid x \in \mathbb{R}, x < Q1 - 1.5 \times IQR, x > Q3 + 1.5 \times IQR\} \tag{2}$$

In the example given below, the values for Q1 and Q3 are equal for `my_data_1` and `my_data_2`; for `my_data_1`, -3 is an outlier, whereas for `my_data_2`, -2 determines the lower whisker. These two datasets are displayed as boxplots in Figure A.3.

```
my_data_1 = [-3, 3, 3, 4, 4, 5, 6, 7, 7, 8]
my_data_2 = [-2, 3, 3, 4, 4, 5, 6, 7, 7, 8]
# 3.25, 6.75
```

```
Q1, Q3 = np.percentile(my_data_1, [25, 75])
# -2.0
lower_whisker_limit = Q1 - 1.5 * (Q3 - Q1)
```

A.3.4 Divisive Hierarchical Clustering

Divisive hierarchical clustering is a technique that can be used to find groups of data. The basic idea is that all data points start in one large group, or cluster. Then, using a distance metric (for example Euclidean distance) and a linkage (either the maximum distance, the minimum distance, an average distance, etc.), this cluster is split into two groups such that the distance between the clusters is maximised. This type of clustering is the opposite of agglomerative hierarchical clustering, where each data point starts in its own cluster, and is grouped into successively larger clusters according to the chosen linkage and distance criteria. The results of the clustering can be visualised in a dendogram, which is a type of tree where the leaves represent the data points and the heights of their connections indicate how far away or how close two data points are from one another. Depending on where the user decides to cut the tree – i.e. the chosen distance threshold – each data point can be assigned to a cluster.

Bibliography

[1] D. H. R. Barton. “The Conformation of the Steroid Nucleus”. In: *Experientia* 6.8 (1950), pp. 316–320. ISSN: 00144754. DOI: 10.1007/BF02170915.

[2] J. D. Dunitz and Verner Schomaker. “The Molecular Structure of Cyclobutane”. In: *The Journal of Chemical Physics* 20.11 (1952), pp. 1703–1707. ISSN: 00219606. DOI: 10.1063/1.1700271.

[3] H. W. Kuhn. “The Hungarian method for the assignment problem”. In: *Naval Research Logistics Quarterly* 2.1-2 (Mar. 1955), pp. 83–97. ISSN: 00281441. DOI: 10.1002/nav.3800020109. URL: http://doi.wiley.com/10.1002/nav.3800020109.

[4] David A. Dows and Nathan Rich. “Dihedral Angle in Cyclobutane”. In: *The Journal of Chemical Physics* 47 (1967), pp. 333–334. ISSN: 00219606. DOI: 10.1063/1.1711869.

[5] C. G. Broyden. “The convergence of a Class of Double-rank Minimization Algorithms 1. General Considerations”. In: *J. Inst. Maths Applics* 6.1 (1970), pp. 76–90. ISSN: 02724960. DOI: 10.1093/imamat/6.1.76.

[6] R. Fletcher. “A new approach to variable metric algorithms”. In: *The Computer Journal* 13.3 (1970), pp. 317–322. URL: https://doi.org/10.1093/comjnl/13.3.317.

[7] Donald Goldfarb. “A Family of Variable-Metric Methods Derived by Variational Means”. In: *Mathematics of Computation* 24.109 (1970), pp. 23–26. ISSN: 00255718. DOI: 10.2307/2004873.

[8] Saul Meiboom and Lawrence C Snyder. “Molecular Structure of Cyclobutane from Its Proton NMR in a Nematic Solvent”. In: *The Journal of Chemical Physics* 52.8 (1970), pp. 3857–3863. DOI: 10.1063/1.1673583.

[9] D. F. Shanno. “Conditioning of Quasi-Newton Methods for Function Minimization”. In: *Mathematics of Computation* 24.111 (1970), pp. 647–656. ISSN: 00255718. DOI: 10.2307/2004840.

[10] Wolfgang Kabsch. “A solution for the best rotation to relate two sets of vectors”. In: *Acta Cryst.* A 32 (1976), pp. 922–923.

[11] Joël Janin et al. “Conformation of Amino Acid Side-chains in Proteins”. In: *Journal of Molecular Biology* 125.3 (1978), pp. 357–386. ISSN: 00222836. DOI: 10.1016/0022-2836(78)90408-4.

[12] K. H. Britton and D. L. Parnas. *A-7E Software Module Guide*. Tech. rep. Washington, DC: Naval Research Laboratory, 1981, pp. 6–30. URL: https://apps.dtic.mil/dtic/tr/fulltext/u2/a108649.pdf%20https://apps.dtic.mil/docs/citations/ADA108649.

[13] Irwin D Kuntz et al. “A Geometric Approach to Macromolecule-Ligand Interactions”. In: *J. Mol. Biol.* 161 (1982), pp. 269–288.

[14] David Lorge Parnas, Paul C. Clements, and David M. Weiss. “The Modular Structure of Complex Systems”. In: *IEEE Transactions on Software Engineering* SE-11.3 (1985), pp. 259–266.

[15] John J. Grefenstette. “Optimization of Control Parameters for Genetic Algorithms”. In: *IEEE Transactions on Systems, Man, and Cybernetics* SMC-16.1 (1986), pp. 122–128. ISSN: 21682909. DOI: 10.1109/TSMC.1986.289288.

[16] David Lorge Parnas and Paul C. Clements. “A Rational Design Process: How and Why to Fake It”. In: *IEEE Transactions on Software Engineering* SE-12.2 (1986), pp. 251–257. ISSN: 16113349. DOI: 10.1007/3-540-15199-0_6.

[17] David Weininger. "SMILES, a Chemical Language and Information System. 1. Introduction to Methodology and Encoding Rules". In: *J. Chem. Inf. Comput. Sci.* 28 (1988), pp. 31–36. ISSN: 00952338. DOI: 10.1021/ci00057a005.

[18] David E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley Longman Inc., 1989.

[19] David Weininger, Arthur Weininger, and Joseph L. Weininger. "SMILES. 2. Algorithm for Generation of Unique SMILES Notation". In: *J. Chem. Inf. Comput. Sci.* 29 (1989), pp. 97–101. ISSN: 00952338. DOI: 10.1021/ci00062a008.

[20] Stephen L. Mayo, Barry D. Olafson, and William A. Goddard III. "DREIDING: A Generic Force Field for Molecular Simulations". In: *Journal of Physical Chemistry* 94.26 (1990), pp. 8897–8909. ISSN: 00223654. DOI: 10.1021/j100389a010.

[21] David Weininger. "Smiles. 3. Depict. Graphical Depiction of Chemical Structures". In: *J. Chem. Inf. Comput. Sci.* 30 (1990), pp. 237–243. ISSN: 00952338. DOI: 10.1021/ci00067a005.

[22] Michael W. Walker, Lejun Shao, and Richard A. Volz. "Estimating 3-D Location Parameters Using Dual Number Quaternions". In: *CVGIP: Image Understanding* 54.3 (1991), pp. 358–367. ISSN: 10499660. DOI: 10.1016/1049-9660(91)90036-O.

[23] Thomas A. Halgren. "Representation of van der Waals (vdW) Interactions in Molecular Mechanics Force Fields: Potential Form, Combination Rules, and vdW Parameters". In: *Journal of the American Chemical Society* 114.20 (1992), pp. 7827–7843. ISSN: 15205126. DOI: 10.1021/ja00046a032.

[24] John H. Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*. MIT Press, 1992.

[25] Jonathan Richard Shewchuk. *An Introduction to the Conjugate Gradient Method Without the Agonizing Pain Edition 1+1/4*. Pittsburgh, 1994.

[26] Scott Gronert and Richard A.J. O'Hair. "Ab Initio Studies of Amino Acid Conformations. 1. The Conformers of Alanine, Serine, and Cysteine". In: *Journal of the American Chemical Society* 117.7 (1995), pp. 2071–2081. ISSN: 15205126. DOI: 10.1021/ja00112a022.

[27] Thomas A. Halgren. "Merck Molecular Force Field. I. Basis, Form, Scope, Parameterization, and Performance of MMFF94". In: *Journal of Computational Chemistry* 17.5-6 (1996), pp. 490–519. ISSN: 01928651. DOI: 10.1002/(SICI)1096-987X(199604)17:5/6<520::AID-JCC2>3.0.CO;2-W.

[28] Thomas A. Halgren. "Merck Molecular Force Field. II. MMFF94 van der Waals and Electrostatic Parameters for Intermolecular Interactions". In: *Journal of Computational Chemistry* 17.5-6 (1996), pp. 520–552. ISSN: 0192-8651. DOI: 10.1002/(SICI)1096-987X(199604)17:5/6<520::AID-JCC2>3.0.CO;2-W.

[29] Thomas A. Halgren. "Merck Molecular Force Field. III. Molecular Geometries and Vibrational Frequencies for MMFF94". In: *Journal of Computational Chemistry* 17.5-6 (1996), pp. 553–586. ISSN: 01928651. DOI: 10.1002/(SICI)1096-987X(199604)17:5/6<553::AID-JCC3>3.0.CO;2-T.

[30] Thomas A. Halgren. "Merck Molecular Force Field. V. Extension of MMFF94 Using Experimental Data, Additional Computational data, and Empirical Rules". In: *Journal of Computational Chemistry* 17.5-6 (1996), pp. 616–641. ISSN: 01928651. DOI: 10.1002/(SICI)1096-987X(199604)17:5/6<616::AID-JCC5>3.0.CO;2-X.

[31] Thomas A Halgren and Robert B Nachbar. "Merck Molecular Force Field. IV. Conformational Energies and Geometries for MMFF94". In: *Journal of Computational Chemistry* 17.5-6 (1996), pp. 587–615.

[32] William Humphrey, Andrew Dalke, and Klaus Schulten. "VMD: Visual Molecular Dynamics". In: *Journal of Molecular Graphics* 14 (Feb. 1996), pp. 33–38. ISSN: 02637855. DOI: 10.1016/0263-7855(96)00018-5. arXiv: arXiv:1503.05249v1. URL: https://www.tapbiosystems.com/tap/products/index.htm%20https://linkinghub.elsevier.com/retrieve/pii/0263785596000185.

[33] G.P. Moss. "Basic Terminology of Stereochemistry". In: *Pure & Appl. Chem.* 68.12 (1996), pp. 2193–2222. ISSN: 1342-3363. DOI: 10.5363/tits.7.7_96.

[34] Attila Szabo and Neil S. Ostlund. *Modern Quantum Chemistry Introduction to Advanced Electronic Structure Theory*. First Edit. Mineola: Dover Publications, Inc., 1996, pp. ix–466. ISBN: 0486691861.

[35] Gareth Jones et al. "Development and validation of a genetic algorithm for Flexible Docking". In: *J. Mol. Biol.* 267 (1997), pp. 727–748. URL: https://www.ncbi.nlm.nih.gov/pubmed/9126849.

[36] David C. Spellmeyer et al. "Conformational analysis using distance geometry methods". In: *Journal of Molecular Graphics and Modelling* 15 (1997), pp. 18–36. ISSN: 10933263. DOI: 10.1016/S1093-3263(97)00014-4.

[37] Garrett M Morris et al. "Automated Docking Using a Lamarckian Genetic Algorithm and an Empirical Binding Free Energy Function". In: *Journal of Computational Chemistry* 19.14 (1998), pp. 1639–1662.

[38] John Stone. "*An Efficient Library for Parallel Ray Tracing and Animation*". MA thesis. Computer Science Department, University of Missouri-Rolla, Apr. 1998.

[39] Thomas A. Halgren. "MMFF VI. MMFF94s Option for Energy Minimization Studies". In: *Journal of Computational Chemistry* 20.7 (1999), pp. 720–729. ISSN: 01928651. DOI: 10.1002/(SICI)1096-987X(199905)20:7<720::AID-JCC7>3.0.CO;2-X.

[40] Thomas A. Halgren. "MMFF VII. Characterization of MMFF94, MMFF94s, and Other Widely Available Force Fields for Conformational Energies and for Intermolecular- Interaction Energies and Geometries". In: *Journal of Computational Chemistry* 20.7 (1999), pp. 730–748.

[41] Georges R. Harik and Fernando G. Lobo. "A parameter-less genetic algorithm". In: *GECCO'99 Proceedings of the 1st Annual Conference on Genetic and Evolutionary Computation - Volume 1*. Florida: Morgan Kaufmann Publishers Inc., 1999, pp. 258–265. URL: https://dl.acm.org/citation.cfm?id=2933949.

[42] L. C. Snoek et al. "Conformational landscapes in amino acids: Infrared and ultraviolet ion-dip spectroscopy of phenylalanine in the gas phase". In: *Chemical Physics Letters* 321.1-2 (2000), pp. 49–56. ISSN: 00092614. DOI: 10.1016/S0009-2614(00)00320-1.

[43] Frank Stajano. "Python in Education: Raising a Generation of Native Speakers". In: *Proceedings of 8th International Python Conference*. Washington, DC, 2000, pp. 1–7.

[44] Aleksander Wawer, Franciszek Seredynski, and Pascal Bouvry. "Evolutionary algorithms for conformational analysis: Vitamine E case study". In: *Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS'04)*. January. 2004. ISBN: 0769521320. DOI: 10.1109/ipdps.2004.1303159.

[45] Johannes Kirchmair et al. "Comparative analysis of protein-bound ligand conformations with respect to catalyst's conformational space subsampling algorithms". In: *Journal of Chemical Information and Modeling* 45 (2005), pp. 422–430. ISSN: 15499596. DOI: 10.1021/ci0497531.

[46] Rajendra Kristam et al. "Comparison of Conformational Analysis Techniques To Generate Pharmacophore Hypotheses Using Catalyst". In: *Journal of Chemical Information and Modeling* 45 (2005), pp. 461–476. ISSN: 15499596. DOI: 10.1021/ci049731z.

[47] Spencer Smith and Lei Lai. "A New Requirements Template for Scientific Computing". In: *International Workshop on Situational Requirements (SREP'05)*. Paris, 2005, pp. 107–121. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.477.1121%7B%5C&%7Drep=rep1%7B%5C&%7Dtype=pdf.

[48] Douglas L. Theobald. "Rapid calculation of RMSDs using a quaternion-based characteristic polynomial". In: *Acta Cryst.* 61.A (2005), pp. 478–480. ISSN: 01087673. DOI: 10.1107/S0108767305015266.

[49] Gerhard Wolber and Thierry Langer. "LigandScout: 3-D Pharmacophores Derived from Protein-Bound Ligands and Their Use as Virtual Screening Filters". In: *Journal of Chemical Information and Modeling* 45 (2005), pp. 160–169. ISSN: 15499596. DOI: 10.1021/ci049885e.

[50] John D. Hunter. "Matplotlib: A 2D graphics environment". In: *Computing in Science and Engineering* 9.3 (2007), pp. 90–95. ISSN: 15219615. DOI: 10.1109/MCSE.2007.55.

[51] W. A. de Landgraaf, A. E. Eiben, and V. Nannen. "PARAMETER CALIBRATION USING META-ALGORITHMS". In: *2007 IEEE Congress on Evolutionary Computation, CEC 2007*. Singapore: IEEE, 2007, pp. 71–78. ISBN: 1424413400. DOI: 10.1109/CEC.2007.4424456. URL: https://ieeexplore-ieee-org.libaccess.lib.mcmaster.ca/document/4424456.

[52] Mikko J. Vainio and Mark S. Johnson. "Generating Conformer Ensembles Using a Multiobjective Genetic Algorithm". In: *Journal of Chemical Information and Modeling* 47 (2007), pp. 2462–2474. ISSN: 15499596. DOI: 10.1021/Ci6005646.

[53] Daniel Ashlock and Taika Von Königslöw. "Evolution of Artificial Ring Species". In: *2008 IEEE Congress on Evolutionary Computation, CEC 2008*. Hong Kong, 2008, pp. 653–659. ISBN: 9781424418237. DOI: 10.1109/CEC.2008.4630865. URL: http://ieeexplore.ieee.org/stamp/

stamp.jsp?tp=%7B%5C&%7Darnumber=4630865%7B%5C&%7Disnumber=4630767.

[54] Daniel Ashlock et al. "Transience in the simulation of ring species". In: *2008 IEEE Symposium on Computational Intelligence in Bioinformatics and Computational Biology, CIBCB '08*. Sun Valley, 2008, pp. 256–263. ISBN: 9781424417780. DOI: 10.1109/CIBCB.2008.4675788. URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=%7B%5C&%7Darnumber=4675788%7B%5C&%7Disnumber=4675751.

[55] Noel M. O'Boyle, Chris Morley, and Geoffrey R. Hutchison. "Pybel: a Python wrapper for the OpenBabel cheminformatics toolkit". In: *Chemistry Central Journal* 2.5 (2008), pp. 1–7. ISSN: 1752153X. DOI: 10.1186/1752-153X-2-5.

[56] Tania Pencheva et al. "AMMOS: Automated Molecular Mechanics Optimization tool for in silico Screening". In: *BMC Bioinformatics* 9.438 (2008), pp. 1–15. ISSN: 14712105. DOI: 10.1186/1471-2105-9-438.

[57] Lishan Yao et al. "NMR Determination of Amide N-H Equilibrium Bond Length from Concerted Dipolar Coupling Measurements". In: *Journal of the American Chemical Society* 130.49 (2008), pp. 16518–16520. ISSN: 00027863. DOI: 10.1021/ja805654f.

[58] M. J. Frisch et al. *Gaussian09 Revision E.01*. Gaussian Inc. Wallingford CT. 2009.

[59] Lydia E. Kavraki. *Geometric Methods in Structural Computational Biology*. OpenStax CNX, Mar. 2009. URL: http://cnx.org/contents/f5c31f8e-7807-4c76-95f8-657d9251fdfb@6.3.

[60] Daniel Ashlock et al. "Evolution and instability in ring species complexes: An in silico approach to the study of speciation". In: *Journal of Theoretical Biology* 264 (2010), pp. 1202–1213. ISSN: 00225193. DOI: 10.1016/j.jtbi.2010.03.017.

[61] Paul C. D. Hawkins et al. "Conformer Generation with OMEGA: Algorithm and Validation Using High Quality Structures from the Protein Databank and Cambridge Structural Database". In: *J. Chem. Inf. Model.* 50.4 (2010), pp. 572–584. URL: http://pubs.acs.org/doi/abs/10.1021/ci100031x.

[62] René Meier et al. "ParaDockS: A Framework for Molecular Docking with Population-Based Metaheuristics". In: *Journal of Chemical Information and Modeling* 50 (2010), pp. 879–889. ISSN: 15499596. DOI: 10.1021/ci900467x.

[63] Maria A. Miteva, Frederic Guyon, and Pierre Tufféry. "Frog2: Efficient 3D conformation ensemble generator for small compounds". In: *Nucleic Acids Research* 38.SUPPL. 2 (2010), pp. 622–627. ISSN: 13624962. DOI: 10.1093/nar/gkq325.

[64] J. Santeri Puranen, Mikko J. Vainio, and Mark S. Johnson. "Accurate Conformation-Dependent Molecular Electrostatic Potentials for High-Throughput In Silico Drug Discovery". In: *Journal of Computational Chemistry* 31.8 (2010), pp. 1722–1732. DOI: 10.1002/jcc.

[65] M. Valiev et al. "NWChem: A comprehensive and scalable open-source solution for large scale molecular simulations". In: *Computer Physics Communications* 181.9 (2010), pp. 1477–1489. ISSN: 00104655. DOI: 10.1016/j.cpc.2010.04.018. URL: http://dx.doi.org/10.1016/j.cpc.2010.04.018.

[66] Zoe E. Brain and Matthew A. Addicoat. "Optimization of a genetic algorithm for searching molecular conformer space". In: *Journal of Chemical Physics* 135.174106 (2011), pp. 1–10. ISSN: 00219606. DOI: 10.1063/1.3656323. URL: https://aip-scitation-org.libaccess.lib.mcmaster.ca/doi/full/10.1063/1.3656323.

[67] Noel M. O'Boyle et al. "Confab - Systematic generation of diverse low-energy conformers". In: *Journal of Cheminformatics* 3.8 (2011), pp. 1–9.

ISSN: 17582946. DOI: 10.1186/1758-2946-3-8. URL: http://www.jcheminf.com/content/3/1/8.

[68] Noel M O'Boyle et al. "Open Babel: An open chemical toolbox". In: *Journal of Cheminformatics* 3.33 (2011), pp. 1–14. ISSN: 1758-2946. DOI: 10.1186/1758-2946-3-33. URL: http://jcheminf.springeropen.com/articles/10.1186/1758-2946-3-33.

[69] Stéfan Van Der Walt, S. Chris Colbert, and Gaël Varoquaux. "The NumPy array: A structure for efficient numerical computation". In: *Computing in Science and Engineering* 13.2 (2011), pp. 22–30. ISSN: 15219615. DOI: 10.1109/MCSE.2011.37. arXiv: arXiv:1102.1523v1.

[70] Jean Paul Ebejer, Garrett M. Morris, and Charlotte M. Deane. "Freely available conformer generation methods: How good are they?" In: *Journal of Chemical Information and Modeling* 52 (2012), pp. 1146–1158. ISSN: 15499596. DOI: 10.1021/ci2004658.

[71] Saulius Gražulis et al. "Crystallography Open Database (COD): An open-access collection of crystal structures and platform for world-wide collaboration". In: *Nucleic Acids Research* 40.D1 (2012), pp. 420–427. ISSN: 03051048. DOI: 10.1093/nar/gkr900.

[72] Marcus D Hanwell et al. "Avogadro: an advanced semantic chemical editor, visualization, and analysis platform". In: *Journal of Cheminformatics* 4.17 (Dec. 2012), pp. 1–17. ISSN: 1758-2946. DOI: 10.1186/1758-2946-4-17. URL: http://www.jcheminf.com/content/4/1/17.

[73] Mostafa M. Ghorab et al. "Synthesis, antimicrobial evaluation and molecular modelling of novel sulfonamides carrying a biologically active quinazoline nucleus". In: *Archives of Pharmacal Research* 36 (2013), pp. 660–670. ISSN: 02536269. DOI: 10.1007/s12272-013-0094-6.

[74] Brian Kelley. *Frowns ChemoInformatics System v. 0.9a*. http://frowns.sourceforge.net/. 2013.

[75] Sunghwan Kim, Evan E. Bolton, and Stephen H. Bryant. "PubChem3D: conformer ensemble accuracy". In: *Journal of Cheminformatics* 5.1 (2013), pp. 1–17. ISSN: 17582946. DOI: 10.1186/1758-2946-5-1.

[76] Andrew McEachern, Daniel Ashlock, and Justin Schonfeld. "Sequence classification with side effect machines evolved via ring optimization". In: *BioSystems* 113 (2013), pp. 9–27. ISSN: 03032647. DOI: 10.1016/j.biosystems.2013.03.022. URL: http://dx.doi.org/10.1016/j.biosystems.2013.03.022.

[77] Christin Schärfer et al. "CONFECT: Conformations from an Expert Collection of Torsion Patterns". In: *ChemMedChem* 8 (2013), pp. 1690–1700. ISSN: 18607179. DOI: 10.1002/cmdc.201300242.

[78] Ira N. Levine. *Quantum Chemistry*. Ed. by Adam Jaworski et al. 7th Editio. Pearson Education, Inc., 2014, pp. x–700. ISBN: 0-321-80345-0.

[79] Sandra Rabi. "New Transition-State Optimization Methods By Carefully Selecting Appropriate Internal Coordinates". PhD thesis. McMaster University, 2014, pp. 1–263. URL: https://macsphere.mcmaster.ca/bitstream/11375/15450/1/New%20Transition-State%20Optimization%20Methods%20By%20Carefully%20Selecting.pdf.

[80] Paolo Tosco, Nikolaus Stiefl, and Gregory Landrum. "Bringing the MMFF force field to the RDKit: Implementation and validation". In: *Journal of Cheminformatics* 6.37 (2014), pp. 1–4. ISSN: 17582946. DOI: 10.1186/s13321-014-0037-3.

[81] Yongna Yuan et al. "Comprehensive Analysis of Energy Minima of the 20 Natural Amino Acids". In: *Journal of Physical Chemistry A* 118.36 (2014), pp. 7876–7891. ISSN: 15205215. DOI: 10.1021/jp503460m.

[82] Daniel Ashlock, Sierra Gillis, and Gary Fogel. "Ring optimization with extinction". In: *2015 IEEE Congress on Evolutionary Computation, CEC 2015 - Proceedings*. Sendai: IEEE, 2015, pp. 1311–1318. ISBN: 9781479974924. DOI: 10.1109/CEC.2015.7257040. URL: http://ieeexplore.ieee.

org/stamp/stamp.jsp?tp=%7B%5C&%7Darnumber=7257040%7B%5C&%7Disnumber=7256859.

[83] Daniel Ashlock et al. "Evolving DNA classifiers with extinction based ring optimization". In: *2015 IEEE Conference on Computational Intelligence in Bioinformatics and Computational Biology, CIBCB 2015*. Niagara Falls: IEEE, 2015, pp. 1–8. ISBN: 9781479969265. DOI: 10.1109/CIBCB.2015.7300312. URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=%7B%5C&%7Darnumber=7300312%7B%5C&%7Disnumber=7300268.

[84] Jarosław Bukowicki, Aleksander Wawer, and Katarzyna Paradowska. "Conformational Analysis of Gentiobiose Using Genetic Algorithm Search and GIAO DFT Calculations with 13C CPMAS NMR as a Verification Method". In: *Journal of Carbohydrate Chemistry* 34 (2015), pp. 145–162. ISSN: 15322327. DOI: 10.1080/07328303.2015.1016230.

[85] Carl Henrik Görbitz. "Crystal structures of amino acids: From bond lengths in glycine to metal complexes and high-pressure polymorphs". In: *Crystallography Reviews* 21.3 (2015), pp. 160–212. ISSN: 14763508. DOI: 10.1080/0889311X.2014.964229.

[86] Stephen R. Heller et al. "InChI, the IUPAC International Chemical Identifier". In: *Journal of Cheminformatics* 7.23 (Dec. 2015), pp. 1–34. ISSN: 1758-2946. DOI: 10.1186/s13321-015-0068-4. URL: http://dx.doi.org/10.1186/s13321-015-0068-4.

[87] Sereina Riniker and Gregory A. Landrum. "Better Informed Distance Geometry: Using What We Know to Improve Conformation Generation". In: *Journal of Chemical Information and Modeling* 55 (2015), pp. 2562–2574. ISSN: 15205142. DOI: 10.1021/acs.jcim.5b00654.

[88] Adriana Supady. *Fafoom - Flexible algorithm for optimization of molecules*. https://github.com/adrianasupady/fafoom. 2015.

[89] Adriana Supady, Volker Blum, and Carsten Baldauf. "First-Principles Molecular Structure Search with a Genetic Algorithm". In: *Journal of Chemical Information and Modeling* 55.11 (2015), pp. 2338–2348. ISSN: 15205142. DOI: 10.1021/acs.jcim.5b00243. arXiv: arXiv:1505.02521v2.

[90] Piotr Wałejko et al. "Phenyl galactopyranosides - 13C CPMAS NMR and conformational analysis using genetic algorithm". In: *Chemical Physics* 457 (2015), pp. 43–50. ISSN: 03010104. DOI: 10.1016/j.chemphys.2015.05.015. URL: http://dx.doi.org/10.1016/j.chemphys.2015.05.015.

[91] Sunghwan Kim, Evan E. Bolton, and Stephen H. Bryant. "Similar compounds versus similar conformers: complementarity between PubChem 2-D and 3-D neighboring sets". In: *Journal of Cheminformatics* 8.62 (2016), pp. 1–17. ISSN: 17582946. DOI: 10.1186/s13321-016-0163-1.

[92] Daniel Ashlock and Sheridan Houghten. "Hybridization and Ring Optimization for Larger Sets of Embeddable Biomarkers". In: *2017 IEEE Conference on Computational Intelligence in Bioinformatics and Computational Biology, CIBCB 2017*. Manchester, 2017, pp. 1–8. ISBN: 9781467389884. DOI: 10.1109/CIBCB.2017.8058532. URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=%7B%5C&%7Darnumber=8058532%7B%5C&%7Disnumber=8058518.

[93] Nils Ole Friedrich et al. "Benchmarking Commercial Conformer Ensemble Generators". In: *Journal of Chemical Information and Modeling* 57 (2017), pp. 2719–2728. ISSN: 15205142. DOI: 10.1021/acs.jcim.7b00505.

[94] Nils Ole Friedrich et al. "High-Quality Dataset of Protein-Bound Ligand Conformations and Its Application to Benchmarking Conformer Ensemble Generators". In: *Journal of Chemical Information and Modeling* 57 (2017), pp. 529–539. ISSN: 15205142. DOI: 10.1021/acs.jcim.6b00613.

[95] Robert M. Parrish et al. "Psi4 1.1: An Open-Source Electronic Structure Program Emphasizing Automation, Advanced Libraries, and Interoperabil-

ity". In: *Journal of Chemical Theory and Computation* 13 (2017), pp. 3185–3197. ISSN: 15499626. DOI: 10.1021/acs.jctc.7b00174.

[96] David Robinson. *Python*. https://stackoverflow.blog/2017/09/06/incredible-growth-python/?_ga=2.23904250.1572276306.1569196389-1740107556.1569196389. 2017.

[97] Sunghwan Kim et al. "An update on PUG-REST: RESTful interface for programmatic access to PubChem". In: *Nucleic Acids Research* 46.W1 (2018), W563–W570. ISSN: 13624962. DOI: 10.1093/nar/gky294.

[98] *Avogadro: an open-source molecular builder and visualization tool. Version 1.2.0*. http://avogadro.cc/. 2019.

[99] NumPy developers. *NumPy - Scientific computing with Python v1.16.4*. https://github.com/numpy/numpy. May 2019.

[100] The Psi4 Developers. *Psi4 Open-Source Quantum Chemistry v1.3.2*. https://github.com/psi4/psi4. May 2019.

[101] *Frog v2.14 FRee On line druG conformation generation*. http://bioserv.rpbs.univ-paris-diderot.fr/services/Frog2/. 2019.

[102] Jennifer H. Garner. *Kaplan Conformer Searching Package*. https://github.com/PeaWagon/Kaplan. 2019.

[103] Stefan Grimme. "Exploration of Chemical Compound, Conformer, and Reaction Space with Meta-Dynamics Simulations Based on Tight-Binding Quantum Chemical Calculations". In: *Journal of Chemical Theory and Computation* 15 (2019), pp. 2847–2862. ISSN: 15499626. DOI: 10.1021/acs.jctc.9b00143.

[104] Russell D. Johnson III, ed. *NIST Computational Chemistry Comparison and Benchmark Database NIST Standard Reference Database Number 101*. http://cccbdb.nist.gov/. Aug. 2019. DOI: 10.18434/T47C7Z.

[105] Sunghwan Kim et al. "PubChem 2019 update: improved access to chemical data". In: *Nucleic Acids Research* 47.D1 (2019), pp. D1102–D1109. ISSN: 13624962. DOI: 10.1093/nar/gky1033.

[106] Jimmy Charnley Kromann. *RMSD v1.3.2 - Calculate Root-mean-square deviation of Two Molecules Using Rotation*. http://github.com/charnley/rmsd. 2019.

[107] *Open Babel - The Open Source Chemistry Toolbox v2.4.1*. https://github.com/openbabel/openbabel. Jan. 2019.

[108] *RDKit: Open-Source Cheminformatics Software v. 2019.03.3*. https://rdkit.org/. 2019.

[109] Matt Swain and Eka Kurniawan. *PubChemPy v1.0.4 Python wrapper for the PubChem PUG REST API*. https://github.com/mcs07/PubChemPy. 2019.

[110] Mikko Vainio. *Balloon v1.6.8*. http://users.abo.fi/mivainio/balloon/index.php. 2019.

[111] Piotr Wałejko et al. "Phenyl glycosides – Solid-state NMR, X-ray diffraction and conformational analysis using genetic algorithm". In: *Chemical Physics* 519 (2019), pp. 126–136. ISSN: 03010104. DOI: 10.1016/j.chemphys.2018.12.001. URL: https://doi.org/10.1016/j.chemphys.2018.12.001.