DESIGN OF A MODULAR FORTRAN 90 MOLECULAR MECHANICS PACKAGE FOR HYDROCARBONS

MICHELLE SHAW

DESIGN OF A MODULAR FORTRAN 90 MOLECULAR MECHANICS PACKAGE FOR HYDROCARBONS

by

© **Michelle Shaw, B.Sc.(Honours)**

A thesis submitted to the School of Graduate

Studies in partial fulfillment of the

requirements for the degree of

Master of Science

Departments of Chemistry, Physics, and Computer Science

Memorial University of Newfoundland

St. John's, Newfoundland, Canada

August, 2000

St. John's, Newfoundland, Canada

Abstract

Molecular mechanics is a popular method for minimization of energies of large biomolecular structures and much work has been done in creating packages which optimize the execution time and memory requirements. Object-based design is a useful tool in creating packages which are easily updated and clearer than procedural-based design due to their inherent modularity. The current codes in use are Fortran 77 and C, but Fortran 90 may prove to be a more viable option for object-based molecular mechanics. In this work, a molecular mechanics package based on the Merck Molecular Force Field (MMFF94) is designed for hydrocarbons using existing Fortran 90 tools and object-based design techniques. Presented in this work are the analysis, design, and implementation of the molecular mechanics package as well as a report of the numerical results. Included in the numerical results are comparisons with literature values for the conformational differences for the ethane and cyclohexane systems.

Acknowledgements

There are several individuals I would like to thank for their guidance and support while this work was being completed. First, I would like to thank my supervisors Jolanta Lagowski and Raymond Poirier for their advice and assistance which was invaluable in the completion of this work. Also, I would like to thank the departments of chemistry, physics, and computer science for the use of their facilities and resources and my colleagues James Xidos, Tammy Gosse, Sharene Bungay, and Darryl Reid for their continued support and encouragement. Thanks to the school of graduate studies, the computational science committee, the National Sciences and Engineering Research Council of Canada, and Memorial University of Newfoundland for financial support and giving me the opportunity to complete my masters programme here.

I would like to extend a special thanks to my family and friends, especially my parents. Their continued support and encouragement throughout the completion of the project will always be greatly appreciated.

Contents

List of Tables

List of Figures

List of Abbreviations

FF	force field
MM	molecular mechanics
C	Carbon
H	Hydrogen
SVD	singular value decomposition
NR	Newton Raphson
VA05AD	Minimization of Sum of Squares method
BFGS	Broyden-Fletcher-Goldfarb-Shanno
DIIS	Direct Inversion of the Iterative Subspace
OC	Optimally Conditioned
OSIPE	Open Structured Interfaceable Programming Environment
MMFF94	Merck Molecular Force Field, 1994 version

Chapter 1

Introduction

1.1 Overview of Project

Molecular mechanics has proven to be a viable method of obtaining geometries and energies for molecular systems. *Ab initio* and semiempirical methods are also available for completing the same task, however in the case of large molecules, they are not feasible. Although molecular mechanics methods are not as accurate as *ab initio* or semiempirical methods, in the case of large molecules an approximate structure is often sought and molecular mechanics provides a less expensive and time-conserving alternative.

Many different packages are currently available. Each molecular mechanics package is somewhat different from others in the form of the energy expression as well as the parameters it uses and the systems it treats. For example, some molecular mechanics packages are designed to reproduce geometries of large biomolecules, while

others reproduce geometries of organic or inorganic systems.

Molecular mechanics packages contain many functionalities, some of which include geometry optimization and topological methods, as well as menu and printing capabilities. When generating results using molecular mechanics, the input must first be parsed, the topology generated and used to determine the connectivity, the coordinates determined from this connectivity, and the geometry must be optimized to provide the desired output.

Design of a molecular mechanics package can be done using procedure or object-based design. Procedure-based design gives a package which has logical program subunits which are smaller executable parts of the program, such as functions or subprograms. The problem is broken down into executable steps taken from the problem definition to the desired solution. Object-based design, involves modeling the problem using a real-world model where objects within this model are defined and the interaction of these objects produces the desired solution.

1.2 Goals of the Project

In this work, a molecular mechanics package restricted to hydrocarbons will be created based on an existing force field and using the techniques of object-based analysis and design. The desired features of this molecular mechanics package are as follows:

- efficient use of memory and computation time

- reliable
- modular
- easy to modify and update
- capable of evaluating a variety of systems
- dynamic allocation of memory
- dynamic execution
- controlled printing and debugging

The language of implementation will be Fortran 90 as it provides backward compatibility with Fortran 77 as well as limited support for object-based design. Because the molecular mechanics package will be added to a larger *ab initio* package written in Fortran 90 and the existing needed mathematical tools are written in Fortran 77, the Fortran 90 language provided the needed functionality for object-based design with straightforward integration into existing Fortran code.

1.3 Outline

The outline of the thesis is as follows. Chapter 2 outlines the molecular mechanics method. It gives the general description as well as a more detailed description of the force field, and provides the problem description needed for the object-based design.

Chapter 3 gives a description of some of the tools available to the molecular mechanics package. It also outlines the coordinate system used.

Chapter 4 provides a description of the object-based design method as well as giving the results of the application of this method to the molecular mechanics problem.

Chapter 5 provides a description of the use of the Fortran 90 language to implement the ideas in Chapter 3 for the molecular mechanics package, as well as describing some useful features of Fortran 90 which were also used in the implementation. It also contains some examples of the implementation of the molecular mechanics method.

Chapter 6 provides some examples of single point and geometry optimizations as well as giving the results of computations on several hydrocarbons. Some comparison with literature values is also given.

Chapter 2

Molecular Mechanics and Force Fields

2.1 Introduction

Molecular mechanics is a term used to describe the computational method of obtaining molecular energies and structures. It is a result of the application of the Born-Oppenheimer approximation, which allows the electronic and nuclear motions within a molecule to be treated separately. Thus, one can treat the molecular system with respect to nuclear motion without explicitly considering the motions of the electrons. This is the basis for the ideas in molecular mechanics [1, 2, 3, 4, 5, 6].

The idea of treating molecules in this manner dates back to the 1930's, where D. H. Andrews and others used the idea that molecules are made up of assemblies of atoms connected by bonds. These bonds, as well as angles, have *natural* values

and the molecule will, if distorted, always attempt to return itself to the geometry defined by these values. In 1946, T. L. Hill proposed using stretch, bend, and van der Waals interactions to minimize the total energy (steric energy) of a molecule. This information would then give structural and energetic information for congested systems. Later on, Dostrovsky, Hughes, and Ingold used the same method to better understand the S_N2 reaction rates for various halides. Also, Westheimer and Mayer studied the rates of racemization of optically-active halo-substituted biphenyls. The work of Westheimer and Mayer gave the most convincing molecular mechanics results. However, it was not until the 1950's with the introduction of computers that the molecular mechanics methods became widely understood and used [1, 2].

2.2 Molecular Mechanics

In molecular mechanics the energy is expressed as a function of the positions of *hard spheres*, or nuclei. The initial use of molecular mechanics was for reproducing spectra, however its uses have since been extended to energy and structural reproduction. Although the accuracy of molecular mechanics method does not rival the current *ab initio* or semiempirical methods, it has some advantages. Molecular mechanics is valuable in cases where the molecule is too large to be treated computationally by *ab initio* or semiempirical methods. Some examples of groups of systems where molecular mechanics may be the only feasible computational method are large polymers, proteins, and coordination complexes. Most molecular mechanics

packages are relatively general in that they are able to treat two or more groups of molecules [1, 2, 3, 4, 5, 6].

The energy, expressed as a function of the nuclear coordinates, forms a surface which is known as the Born-Oppenheimer or potential energy surface. This surface describes the change in energy for the molecule as a function of the motions of the nuclei. There are a few regions of interest on this potential energy surface, and these regions are rich in structural and energetic information, some of which cannot be determined by experiment [7]. For example, in molecular mechanics it is customary to want the particular point on the potential energy surface which has the lowest energy as this gives the most optimal structural information for the molecular system. Also obtainable from the potential energy surface are the vibrational energy levels for the ground electronic state of the system. This information is very useful to spectroscopists and as a result some molecular mechanics methods involve using potential functions which are designed to reproduce this information accurately. Unfortunately, a molecular mechanics method designed to accurately reproduce spectral data is not generally good for reproducing structural and energetic information. Also, methods which reproduce structural and energetic information usually are a poor choice for reproducing spectra [1, 2, 3, 4, 5, 6].

The results of a molecular mechanics method are a set of interactions between atoms within the molecule as well as the energy contributions with respect to each of these interactions. The sum of these energy contributions is the total or steric energy of the molecule. The functions which define the energy in terms of each of these

interactions constitute the force field. This name was adopted from spectroscopy and it is interesting to note that the first derivatives of these energy contributions give the forces for particular interactions [1, 2, 3, 4, 5, 6].

2.3 Force Fields

2.3.1 Introduction

The force field is the energy expression which is used to describe the energy of the system with respect to the nuclear coordinates. There are two classes of force fields. The harmonic force field only contains contributions to the energy that result from harmonic motion within the molecule and the anharmonic force field contains contributions to the energy resulting from both harmonic and anharmonic motions. The force field has many uses, however it is known for its use in calculating minimum energy structures or vibrational spectra. This energy can be investigated over a set of time intervals to give the evolution of a system's energy with respect to time (molecular dynamics), which has widespread use in simulations in solution or condensed phase. Instead of finding a minimum energy, a random sampling of the conformational space of one or more molecules can be used to determine the Boltzmann distribution of the energies of various conformations of a molecule (Monte Carlo simulations) [1, 2, 3, 4, 5, 6, 8].

There are two major criteria a force field must satisfy. First, it should be able to

predict the equilibrium structure of a molecule. Second, it should be able to predict the stability of the structure at or near the minimum energy. The form of the energy expression is therefore important to satisfying these criteria. Since the force field is empirical, there is no correct expression and as a result, many different force fields are available. For more information on the force field implemented in this work, see appendix A. Depending on the purpose of the force field, some expressions may be more accurate than others. Usually there is a tradeoff between accuracy of the force field and the speed at which it computes the energetic and structural information and this in turn depends on the area of application [1, 2, 3, 4, 5, 6, 8].

The presence of intramolecular and intermolecular forces which compress, stretch, and twist the molecule cause the potential energy of the system to change. This change in the potential energy can be further investigated by looking at the form of the energy expression in more detail. If the potential energy of the molecule is expanded in a Taylor series, the following expression results:

$$V_{potential} = V_0 + \sum_{i=1}^{N} \left(\frac{\partial V}{\partial x_i} \right)_0 \Delta x_i + \sum_{i=1}^{N} \sum_{j=1}^{N} \left(\frac{\partial^2 V}{\partial x_i \partial x_j} \right)_0 \Delta x_i \Delta x_j + \cdots \tag{2.1}$$

When the molecule is at a minimum on the potential energy surface, the first term in the series can be set to zero (initial energy is zero) and the second term vanishes (for a stationary point). Only the second order term is kept in the harmonic approximation. This leaves the following expression for the potential energy [1, 2, 3, 4, 5, 6, 8] at an

extremum:

$$V_{potential} = \sum_{i=1}^{N}\sum_{j=1}^{N}\left(\frac{\partial^2 V}{\partial x_i \partial x_j}\right)_0 \Delta x_i \Delta x_j. \tag{2.2}$$

The energy of a molecule can now be given in terms of the deformations of various interactions (Δx_i and Δx_j) which occur between the atoms. Some of these interactions are stretches, bends, torsions, improper torsions or out-of-plane bends, van der Waals, and electrostatics. In some force fields, energy terms derived from the result of one interaction on another are also considered, also called cross-terms. The energy expression can then be rewritten as [1, 2, 3, 4, 5, 6, 8]:

$$\begin{aligned} V_{potential} = {} & V_{stretch} + V_{bend} + V_{torsion} + V_{oopbend\ or\ imp.\ tors.} + V_{vanderWaals} \\ & + V_{electrostatics} + \text{cross terms.} \end{aligned} \tag{2.3}$$

Each of these energy contributions is composed of a function containing parameters which determine the rate of increase or decrease of the energy depending on the atoms involved, and is obtained by summing the individual energies for each interaction of a particular type. The parameters can be determined from experimental results or *ab initio* calculations. However, this does not solve the problem of computations on large systems unless an important assumption is made. If the parameters can be derived for small molecules for which the computations or experimental results are

straightforward to obtain, then it should be possible to transfer these parameters to large molecules. This is a valid assumption if the energy expression contains bonded and nonbonded interactions. However, the parameters are not transferable between force fields because their development is dependent on the energy expression used to derive them. Parameters are, however, transferable between molecules as long as the environments of the interactions are the same [1, 2, 3, 4, 5, 6, 8].

The most common functions for the energy expressions given in the previous equation will now be discussed. For details on the function used in the molecular mechanics package implemented in this work, the reader is again referred to appendix A.

2.3.2 The Potential Energy Expression

The total steric energy for a molecule can be expressed as a sum of contributions due to stretch, bend, torsion, out-of-plane bends or improper dihedrals, van der Waals, and electrostatic interactions as well as the coupling between these interactions to give [1, 2, 3, 4, 5, 6, 8]:

$$\begin{aligned} V_{total} = & \sum_{stretches} V_{stretch} + \sum_{bends} V_{bend} + \sum_{torsions} V_{torsion} \\ & + \sum_{oopbend\ or\ imp.\ tors.} V_{oopbend\ or\ imp.\ tors.} + \sum_{van\ der\ Waals} V_{van\ der\ Waals} \\ & + \sum_{electrostatic} V_{electrostatic} + \text{cross terms} \ . \end{aligned} \tag{2.4}$$

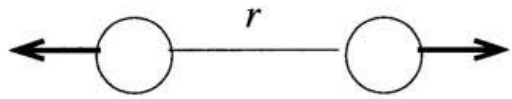

Figure 2.1: Bond stretch.

Bond Stretch

A bond stretch is a deformation within the molecule which occurs along the bond formed between two atoms. This deformation can correspond to a lengthening or shortening of the bond.

Figure 2.1 gives an example of a bond stretch. The potential energy change as the bond is compressed and stretched is given by the potential energy surface of a diatomic molecule and is best described by a Morse potential:

$$V_{stretch} = D_e \left\{ 1 - e^{[-a(r-r_0)]} \right\}^2 \qquad (2.5)$$
$$\text{where } a = \omega \sqrt{\frac{\mu}{2D_e}} \text{ and } \omega = \sqrt{\frac{k}{\mu}}.$$

The Morse potential is the most accurate representation of the diatomic potential energy surface, however it is computationally expensive. For smaller systems this will not be a problem, however, for large molecules this is still not the best option. Since the structure desired is usually the one corresponding to a minimum energy, the Morse potential can be expanded in a Taylor series and truncated after the quadratic term.

The result is Hooke's law for the deformation of a harmonic spring. In the region

around the minimum energy on the potential energy surface, this approximation is adequate, however bonds are not harmonic in nature. In order to account for the anharmonicity in the bonds, the quadratic, cubic, and quartic terms in the Taylor series expansion must be included in the energy expression. This resulting functional form very closely approximates the Morse potential in a larger region about the energy minimum and is less computationally expensive and more accurate than the harmonic approximation. As a result, many force fields use this form of the potential energy or some variation of it:

$$V_{stretch} =^2 K(r - r_0)^2 +^3 K(r - r_0)^3 +^4 K(r - r_0)^4. \tag{2.6}$$

In the above equation, ${}^2K$, ${}^3K$, and ${}^4K$ denote quadratic, cubic, and quartic bond force constants. For molecules with long bonds, such as the bonds between molecules in a transition state, the Morse potential must be used because they are located too far from the minimum on the potential energy surface to be correctly represented by the above equation. The Morse potential also correctly describes bond dissociation [1, 2, 3, 4, 5, 6, 8].

Angle Bend

An angle bend corresponds to a deformation of the angle formed between two atoms bonded to a common third atom.

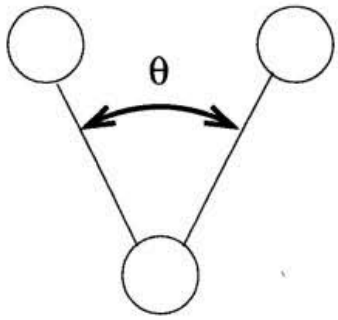

Figure 2.2: Angle bend.

Figure 2.2 gives an example of a bend between three atoms. The angle bends also become harmonic as the angle becomes very close to the equilibrium value. As a result, the angle bend potential energy is often represented by a harmonic potential. The force constant in the harmonic potential prevents the angle bend from deforming too far from the equilibrium value (called a restoring force constant) and is smaller for an angle than for a bond because less force is needed to deform an angle from its equilibrium value. This function is accurate for angle deformations of up to 10 degrees, however if the deformation is larger, the harmonic approximation fails. In these cases, and for better accuracy in general, it is desireable to include the anharmonicity as was done for the stretches. This is again done by adding a cubic and/or quartic term to the harmonic approximation to give the bending energy as:

$$V_{bend} =^2 K(\theta - \theta_0)^2 +^3 K(\theta - \theta_0)^3 +^4 K(\theta - \theta_0)^4. \tag{2.7}$$

In the above equation, ${}^2K$, ${}^3K$, and ${}^4K$ denote quadratic, cubic, and quartic

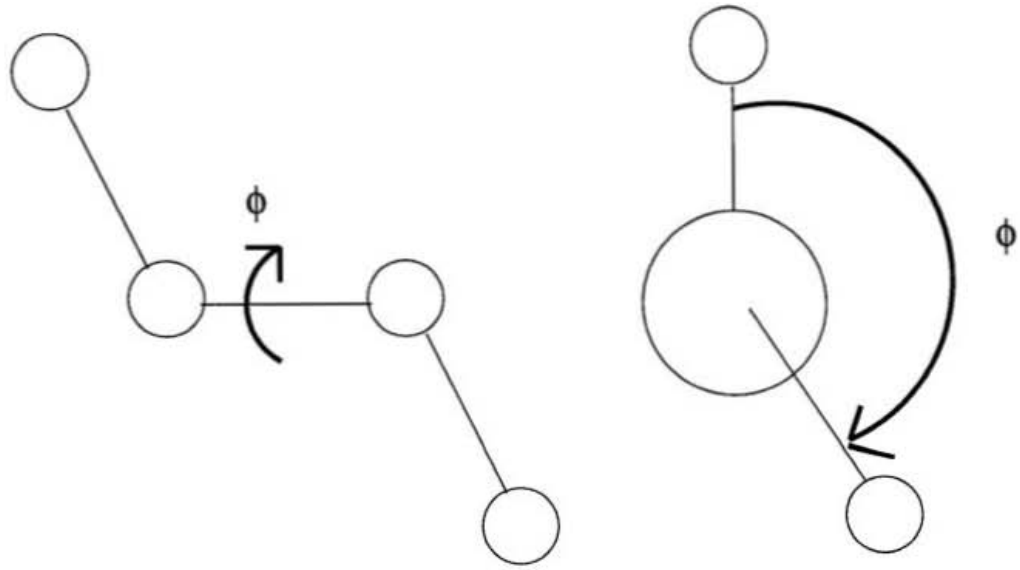

Figure 2.3: Torsion angle.

angular force constants. This equation is sufficient for reproducing most angles, however most force fields will include special atom types to distinguish those angles that must be treated with more accurate force constants and equilibrium values (for example, strained ring systems). For linear bends the angles are not calculated accurately by the above equation so an expression using cosines is used for this special case [1, 2, 3, 4, 5, 6, 8].

Torsion

Torsions describe intermolecular rotations. They occur between atoms separated by three bonds and are defined as the angle formed between two planes, one containing the first and second bond, and the other containing the second and third bond.

Figure 2.3 shows a torsion from both the side view and the perspective when

looking down the central bond. Torsions occur when two atoms separated by three bonds attempt to reduce the antibonding interaction between them. In order to achieve a lower energy the two atoms move in a direction which reduces this repulsive interaction. The energy due to this torsion is represented by a cosine function. Often this potential expression is a Fourier series expansion truncated after the third term, but some force fields include up to six terms. However, for the purposes of obtaining a minimum energy structure, the first three or four terms of the expansion are sufficient, and it turns out only the first six of these terms can be determined by experiment from the overtones. The following functional form is most common in force fields:

$$V_{torsion} = V_1(1 + \cos\theta) + V_2(1 - \cos 2\theta) + V_3(1 + \cos 3\theta). \tag{2.8}$$

The V_1 term in the above expression corresponds to the dipole-dipole interactions between the first and fourth atom in the torsion, the V_2 term corresponds to the conjugation/hyperconjugation effects, and the V_3 term corresponds to steric interactions [1, 2, 3, 4, 5, 6, 8].

Out-of-Plane Bends and Improper Dihedral Angles

Trigonal planar centres within molecules are not adequately represented in force fields which include only stretch, bend, torsion, and nonbonded interactions. As a result, most current force fields now contain an energy contribution to treat these

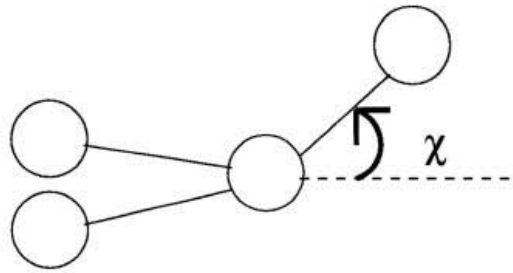

Figure 2.4: Out-of-plane bend.

trigonal planar centres adequately. This is done by including an improper dihedral or out-of-plane bending term in the energy expression.

Some force fields use the out-of-plane bend and others use the improper dihedral angle, and the use of one or the other can offer advantages. Improper dihedrals are not real angles, but the existing torsion energy expression may be used to compute the energy contribution. Out-of-plane bends are actual vibrational modes in spectroscopy, but a new function must be used to compute the energy. If an out-of-plane bend is used, the motion is harmonic, so the energy can be computed using a harmonic potential [1, 2, 3, 4, 5, 6, 8]:

$$V_{oopbend} = \frac{1}{2} K \chi^2. \tag{2.9}$$

The out-of-plane angle represented above by χ can be defined as a *Wilson* angle, shown in figure 2.4, or as a pyramid height [8].

van der Waals Interactions

van der Waals contributions are computed between nonbonded atoms, and most force fields do not include van der Waals contributions resulting from atoms which are bonded to a common third atom. They were first computed by van der Waals and show the deviation of a real gas from ideal gas behavior. This energy term includes the effects of dispersion forces, weak bonds and van der Waals interactions and contains an attractive and repulsive part. The attractive part is long range and is due to induced dipole-dipole interactions. The repulsive part is short range and is a result of the overlap of electron clouds on the interacting atoms. It is common in force fields to represent the van der Waals interaction energy by a Lennard-Jones potential [1, 2, 3, 4, 5, 6, 8]:

$$V_{van\ der\ Waals} = \epsilon\left[\left(\frac{R_0}{R}\right)^{12} - 2\left(\frac{R_0}{R}\right)^{6}\right]. \tag{2.10}$$

The ϵ represents the well depth of the potential function, R represents the internuclear separation, and the R_0 represents the minimum energy separation of the interacting atoms. Some force fields replace the repulsion term in the above expression by an exponential-6 function. The above expression for the van der Waals potential is in general sufficient, although the attractive term is often not good enough to reproduce energies and structures for some organic systems, so some force fields use an R^{-9} term for the repulsive part. However, the R^{-12} term is preferred since it is trivial

to compute it by squaring the R^{-6} term [1, 2, 3, 4, 5, 6, 8].

Electrostatics

Each nonbonded pair also contributes an energy term due to electrostatic interactions. There are several possible ways to model these interactions, although the simple Coulomb interaction between two point charges is popular in most force fields:

$$V_{electrostatic} = \frac{Kq_1q_2}{\epsilon R}. \tag{2.11}$$

Hydrogen bonds are also modeled by electrostatic potentials in some force fields. Also included in most electrostatic potentials is a constant to prevent infinite attractions between opposite charges. The charges q_1 and q_2 are represented by charges at the centre of the nucleus, also called partial atomic charges. These usually consist of the sum of a formal charge determined from Lewis dot structures and bond charge increment contributions from all bonds the atom participates in.

Sometimes it is desireable to model the electrostatic interactions using higher order multipoles, such as dipoles, quadrupoles or octopoles. One reason for this is charge may not actually be located at the nucleus as is depicted by the point charge model, but may be distributed throughout the molecule. In this case a distributed multipole model may be used instead of the point charge model [1, 2, 3, 4, 5, 6, 8].

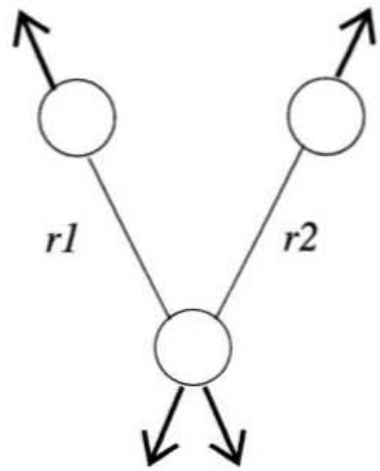

Figure 2.5: The coupling of one bond to a neighbouring bond.

Cross Terms

Because the bond, angle, torsion, and out-of-plane bend (or improper dihedral) interactions are each computed individually, some correction must be added to account for the effect of one interaction on another. These corrections depend on the nearest neighbours of the interacting atoms only. For example, two bonds which are separated from one another by two or more bonds do not affect each other enough to be significant and are not included. However, two bonds sharing a common atom will affect one another and should not be ignored. It was found that the inclusion of energy terms due to coupling of interactions makes the parameters of the force field more transferable to molecules not used to parameterize it. There are many cross terms and only those that have shown significant contributions to the energy expression are included. Two of the most common ones in current force fields that will be discussed are stretch-stretch and stretch-bend [1, 2, 3, 4, 5, 6, 8].

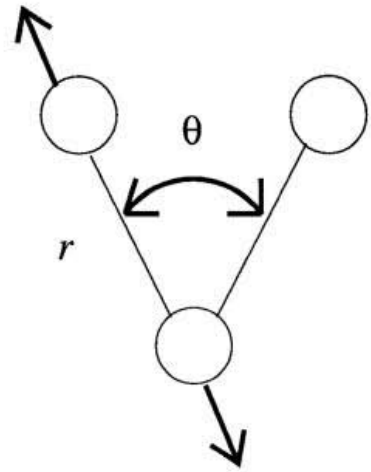

Figure 2.6: The coupling of an angle to the stretch of one of its bonds.

The stretch-stretch coupling interaction is most important in π-bonded systems. They describe the effect of one stretch on a neighboring stretch. Figure 2.5 gives an example of a stretch-stretch interaction. The energy can be expressed using a harmonic potential:

$$V_{stretch-stretch} = K(r - r_0)(r' - r'_0). \tag{2.12}$$

The stretch-bend coupling interaction is included to account for the change in bond length that occurs when angles are changed as well as the change in the angle when the bond lengths are changed.

Figure 2.6 gives an example of a stretch-bend interaction. Often it is expressed in terms of one of two bonds that make up the angle and an energy contribution is given for each bond in the angle. This term is important also if one wants more accurate

vibrational frequencies.

The energy can be expressed by a harmonic potential [1, 2, 3, 4, 5, 6, 8]:

$$V_{stretch-bend} = K(r - r_0)(\theta - \theta_0) \tag{2.13}$$

2.3.3 Parameterization of the Force Field

The constants in the above equations as well as the exact energy expression for a force field must be determined in some manner. Some of these parameters include atom types, force constants, equilibrium values, nonbonded parameters, and scale factors. The parameters must be derived by matching the desired results of the force field to existing data, either from experiment or *ab initio* calculations. In order to accomplish this a systematic method must be used to obtain the parameters from the fits to the existing data. The parameterization step is very important in the development because the accuracy of the force field is very dependent on the energy expressions and their parameters. Due to the use of varying potential functions and parameters in the current force fields, it is very dangerous to use parameters from one force field in another.

Figure 2.7 gives a brief outline of the process of parameterizing a force field. Once the potential energy expression is obtained, a set of initial guesses for the parameters is needed. These can be obtained from existing data or a true *guess.*

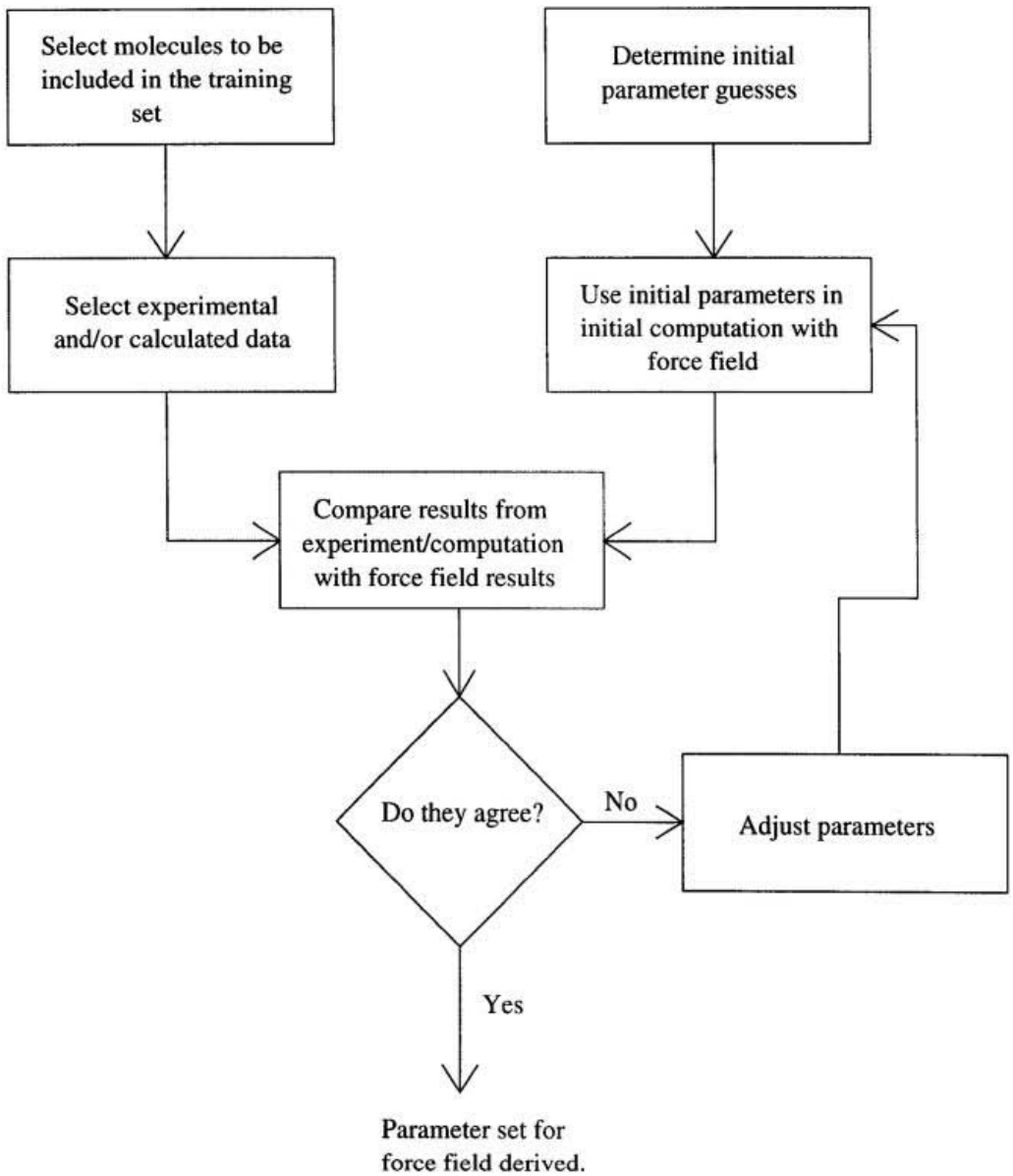

Figure 2.7: The steps in the parameterization process.

These initial guesses are usually very poor so they are not kept. The next step is to obtain the results the force field should produce from either experiment or *ab initio* calculations. These may be energies, spectral, thermodynamic, or some other form of data [1, 3, 4, 8, 9].

A set of training molecules is also needed. The training set is generally representative of the types of systems the force field is designed to handle. A particular subset of the training set is used as *targets* and the data from the force field will be fit to that experimental or computational data using these molecules. Normally training sets are large, often in excess of 500 molecules.

Next, the same results are obtained from the force field using the initial parameters and energy expressions. These two sets of results are compared and necessary changes are then made to the parameters and/or potentials. In the early days of force fields this was manually iterated until the force field results matched the ones from experiment or computation. This process involved a lot of intuition and guess work and was very time-consuming. A linear least-squares fitting procedure is commonly used to iterate the parameter generation to convergence, but even in this case convergence is still slow unless the user makes a few adjustments to attempt to speed convergence [1, 3, 4, 8, 9].

The parameterization process is the most expensive part of force field development. Usually the data used for comparison and fitting is limited to alleviate this expense somewhat. Another problem with parameterization is that optimization of the parameters must be done in the same units. This can be eased by using a

weighting scheme to optimize parameters of differing units. The iterative procedure is usually converged when no improvements are seen in the force field parameters. If a parameter goes to zero during the iterative process, the potential function is eliminated from the energy expression.

Some of the data used for fitting the parameters is listed below along with the type of interaction it is used for in the fitting procedure [1, 3, 4, 8, 9]:

- *Bonds and Angles:*
  - reduced vibrational spectroscopic values
  - optimized geometries of a group of simple (model) compounds
- *Torsions:*
  - barriers to rotation
- *van der Waals:*
  - van der Waals potentials of rare gas atoms from molecular beam experiments
  - van der Waals radii: crystal data or *ab initio*
- *Electrostatics:*
  - experimental dipoles
  - calculated thermodynamic properties

– molecular electrostatic potentials

The partial atomic charges are not observables and cannot be obtained directly from *ab initio* calculations, although this method is often useful for obtaining an initial guess to other properties from which the partial atomic charges can be determined. They can be obtained from molecular electrostatic potentials or other sources [1, 3, 4, 8, 9].

2.3.4 Heats of Formation and Conformations From Molecular Mechanics

The most common uses of molecular mechanics are geometry optimizations, conformational searches, individual energy contributions with respect to a particular interaction, and heats of formation. Techniques for energy minimization will be covered in more detail in the next chapter, so will not be covered here. Generation of individual energy contributions in some force fields usually involves specifying a particular level of printing, with higher printing levels giving more detailed results from the molecular mechanics computation [1, 3, 10].

Generation of conformers has two practical advantages. First, it enables the location of low-energy conformations of a molecule along a potential energy surface with more than one minimum. Second, saddle points along the potential energy surface can also be located, and those saddle points which are first-order often correspond to

transition state structures. Conformations can be generated by fixing some parameters and optimizing with respect to the parameters which are not fixed. Usually, the parameters that are fixed are torsions. Another name for this type of conformer search is torsion driving. In torsion driving using molecular mechanics, a strong torsional potential is added to the force field which fixes one or more of the torsions to a certain value for the duration of the energy minimization. This corresponds to an optimized geometry with the desired torsion angle(s). The strong torsion potential is then removed and a single-point energy is calculated using the original force field, corresponding to the energy of the desired structure [1].

It is also possible to calculate heats of formation using the strain energy calculated from molecular mechanics. The equilibrium electronic energy is a combination of contributions due to the strain energy from molecular mechanics and the bond energies. As a result, it should be possible to approximate the equilibrium electronic energy from a combination of the steric energy of the molecule at the bottom of the potential well and the bond energies. The formation energy for a particular reaction is then the change in this electronic energy which occurs when the products are formed from the reactants [1, 3]:

$$\Delta U^{\circ}_{f,0} = V_{steric} - \text{bond energies.} \tag{2.14}$$

The heat of formation is then expressed as a sum of the internal energy plus the

rotational and translational energy and a PV term added to generate the enthalpy from the energy of the nonlinear molecule [3]:

$$\Delta H_f^\circ = \Delta U_f^\circ + \Delta(PV)^\circ. \tag{2.15}$$

The enthalpy of formation, in general, can thus be computed as a sum of the rotational, vibrational, and PV terms combined with the steric energy from molecular mechanics and the bond energy [1, 3]:

$$\Delta H_f^\circ = 4RT + V_{steric} + E_{bond}. \tag{2.16}$$

In the above expression, the 4RT term represents the contributions to the energy from translation, rotation, and a PV term to convert this energy to enthalpy for a nonlinear molecule. The bond energy increments are usually included in the molecular mechanics package and several options for these values are available [1, 3].

2.4 Summary

Molecular mechanics is a versatile method for obtaining information about a molecular system. It is derived from assuming the nuclear and electronic motions can be treated separately (the Born-Oppenheimer approximation), giving an energy

expression that is dependent on the positions of the nuclei. This expression is a sum of all of the interactions which occur in the molecular system and constitutes the force field. There are a variety of different combinations of energy expressions which gives a variety of different force fields. The parameters used in the energy expressions can be derived using an iterative process of fitting force field results to either *ab initio* or experimental data. The results of molecular mechanics calculations are often structural data and energies although other quantities such as vibrational frequencies and heats of formation can also be obtained.

Chapter 3

Molecular Representation and Geometry Optimization

3.1 Introduction

Two concepts central to any molecular modeling method, including molecular mechanics, are how the molecule is represented and the geometry which the molecule has when its energy is lowest. A molecule can be represented in different ways depending on the coordinate system the method works with. But before the coordinates for the molecule can be built, the positions of the atoms must be determined. This can be done by treating the atoms in a molecule as vertices of a graph and the bonds as the edges which connect the vertices.

Once the coordinates are built and the potential energy function is known, the potential energy surface around the structure can be explored to find points where

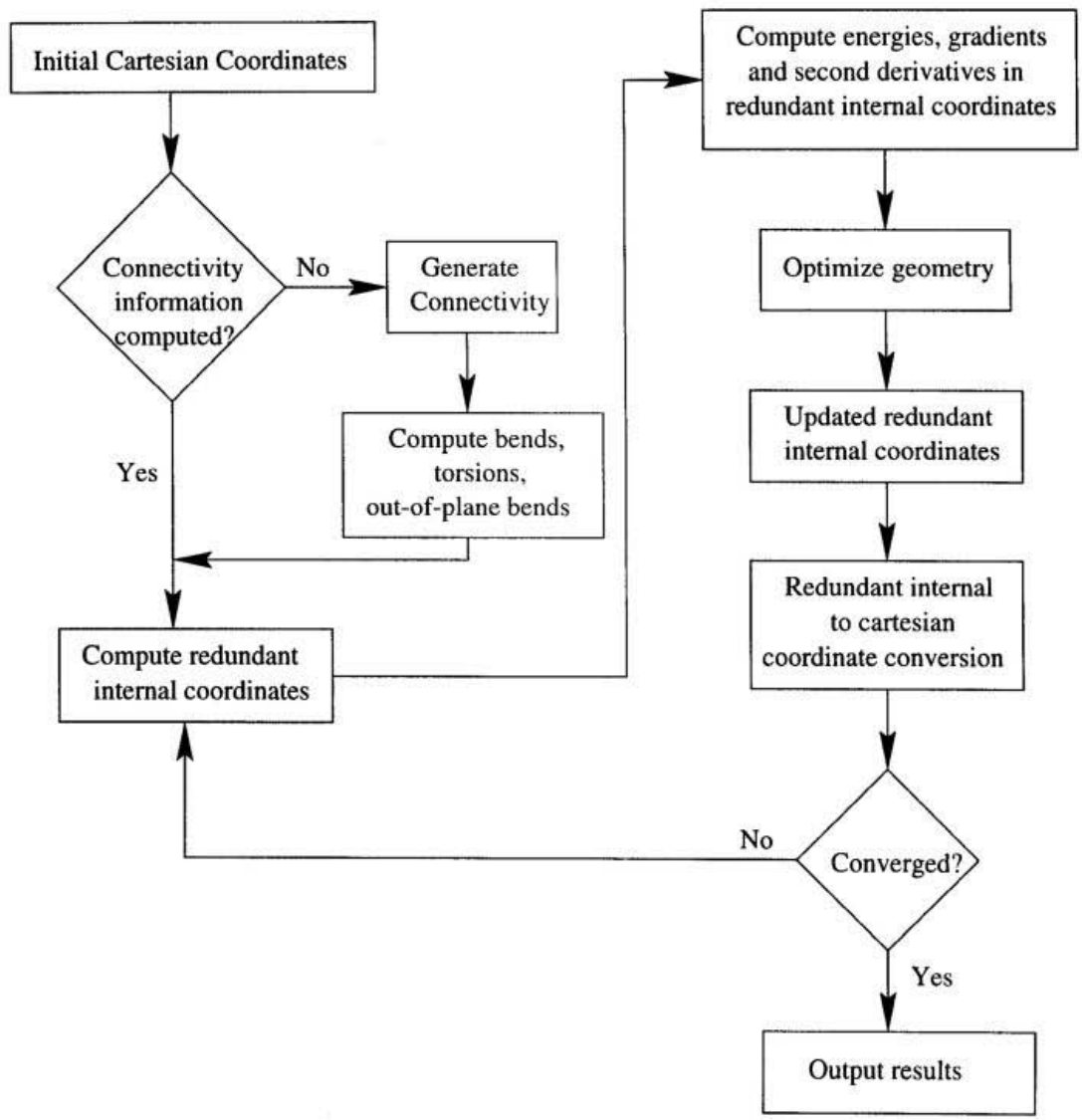

Figure 3.1: Diagram of a geometry optimization after the generation of the initial Cartesian coordinates.

the energy is lowest. Methods are available to do this and are collectively called optimization methods. When these methods are applied to optimize the potential energy of a molecule with respect to the coordinates used, they are collectively called geometry optimization. A diagram of this entire process is given by figure 3.1.

3.2 Topology: Information Regarding 3-Dimensional Molecular Structure

3.2.1 Edges, Vertices, and Graphs

A nondirected graph G can be defined as a collection of vertices and the edges which join them. The collection of vertices form a vertex set, or V_G and the collection of edges form an edge set, or E_G, such that the elements of E_G are unordered pairs of distinct elements of V_G [11, 12].

A graph can be viewed pictorially by representing the vertices by small hollow or filled-in circles which are connected by lines. These lines represent the edges. Often the term graph is used to mean both the pictorial representation and the formal definition.

Graphs are particularly useful in chemistry to represent chemical structures. These structures can be molecules, crystals, reactions, etc., and they contain pieces which are connected to one another. The pieces and their connectivities are analogous to the vertices and edges of a graph, respectively. The vertices in a graph can be used

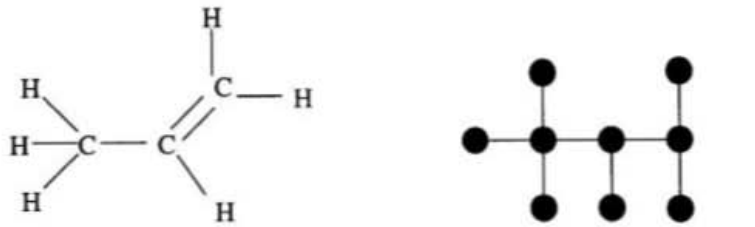

Figure 3.2: Example of a diagram of a molecular graph representing propene.

to represent atoms, molecules, electrons, functional groups, etc. The edges can be used to represent chemical bonds, van der Waals interactions, partially formed bonds, etc [11, 12]. An example of a diagram of a molecular graph is given in figure 3.2.

The information contained in the graph can be used to determine the atoms involved in interactions (for example, torsions), lists of bonded and nonbonded pairs, valencies of the atoms, locations of rings in a molecule, and a variety of other information. Some of the uses of the molecular graph will be discussed in subsequent sections.

3.2.2 Valency, the Adjacency Matrix, and the Distance Matrix

A wide variety of information can be collected from a graph, for example the graph shows which vertices are connected and which ones are not. With the vertices, this connectivity information, and some idea about bonding at centres (represented by the vertices), a molecule can be reconstructed. The connectivity information

is straightforward to compute for a given graph G provided some order is given to the vertices. This can be accomplished with vertex labeling. Once the vertices are labeled, it can be determined if two vertices share an edge if they are adjacent to one another. One must be careful when labeling the graph as the connectivity information depends on the way it is labelled. The matrix formed by a collection of this connectivity information is square symmetric, and is called the adjacency matrix. Its both dimensions are the number of vertices (N), or N X N. In other words, each row (or column) contains information about which other vertices are adjacent to the one represented by the row label. The adjacency matrix is comprised of entries of ones and zeros which are determined as follows [11, 12]:

$$A_{ij} = \begin{cases} 1 & \text{if vertices } v_i \text{ and } v_j \text{ are adjacent,} \\ 0 & \text{otherwise.} \end{cases} \tag{3.1}$$

When building the adjacency matrix for molecular systems, it often does not suffice to determine adjacent vertices, but to define some distance which the vertices must be within to be considered connected.

By summing the entries in a row of the adjacency matrix the number of vertices a vertex is connected to can be determined. This sum is known as the valency of the vertex. If the shortest path between two vertices is placed in the adjacency matrix in place of the entries for each vertex, this matrix is called the distance matrix, and

its entries are denoted by the following relation [11, 12]:

$$D_{ij} = \begin{cases} l_{ij} & \text{if } i \neq j, \\ 0 & \text{if } i = j. \end{cases} \tag{3.2}$$

The l_{ij} in the above equation is the distance of the shortest path between two vertices.

3.2.3 Rings

Rings within a molecule can be found by constructing a spanning tree to determine the number of edges which close a ring. This number of edges is the same as the number of rings within the graph. A spanning tree has the feature that it includes all vertices of a graph, but does not contain cycles. If an edge is found which creates a cycle, that is which closes a ring, then it is removed and counted. Determining the location of rings within a graph involves travelling through the paths of a graph and the paths which are closed, that is the beginning and end vertices coincide, are the rings (or cycles). A path is a sequence of edges and vertices where the edge is incident with the vertices immediately before and after it, but where all vertices are distinct [12, 13].

3.3 Geometry Optimization and Energy Minimization

3.3.1 The Potential Energy Surface and Chemically Interesting Points

The potential energy surface describes the changes in the potential energy of a system with respect to the positions of the nuclei. As a result, this surface can give information on the effect of structural changes on the potential energy.

The structure of a molecule can be determined by investigating the potential energy surface at different geometries. It is possible for the molecule to have minimum and maximum energy structures, and in the case of molecules in a reaction, a transition state structure. These points are referred to as stationary points as the first derivatives of the potential energy with respect to the coordinates is zero [14, 15, 16, 17].

In order to determine the location of these points on the potential energy surface, it is useful to obtain first and second derivatives of the potential energy function as they give information on the location and type of stationary points. Since the negatives of the gradients (or first derivatives) correspond to the forces on the atoms in classical mechanics, these stationary points also are places where the forces on the atoms are zero [14, 15, 16, 17].

The second derivative matrix is the force constant matrix and gives information

about the type of stationary point found. If this matrix is diagonalized, the resulting diagonal values at the stationary points are the vibrational frequencies. For a minimum, all these diagonal values are positive, for a maximum, they are negative, and for a saddle point, some are positive and some are negative. If there is more than one minimum, the one with the lowest energy is the global minimum and all others are called local minima. The order of the saddle point corresponds to the number of negative eigenvalues, or the number of directions for which the energy is a maximum. Transition state structures correspond to first-order saddle points [14, 15, 16, 17].

Figure 3.3 shows an example of a surface for the function given in the caption. It shows examples of maxima, a global and local minimum, and a first-order saddle point.

3.3.2 Energy Minimization

A variety of different forms of energy minimization methods exist, each using varying amounts of information about the potential energy surface. The derivatives employed in these methods can be computed analytically or numerically, depending on the expense of analytical computation. Analytical derivatives are preferred as they greatly speed up the optimization.

The selection of an optimization method depends on the type of problem being solved, the number of independent variables used in the function, and some characteristics of the function used. In the case of molecular mechanics, the function is the

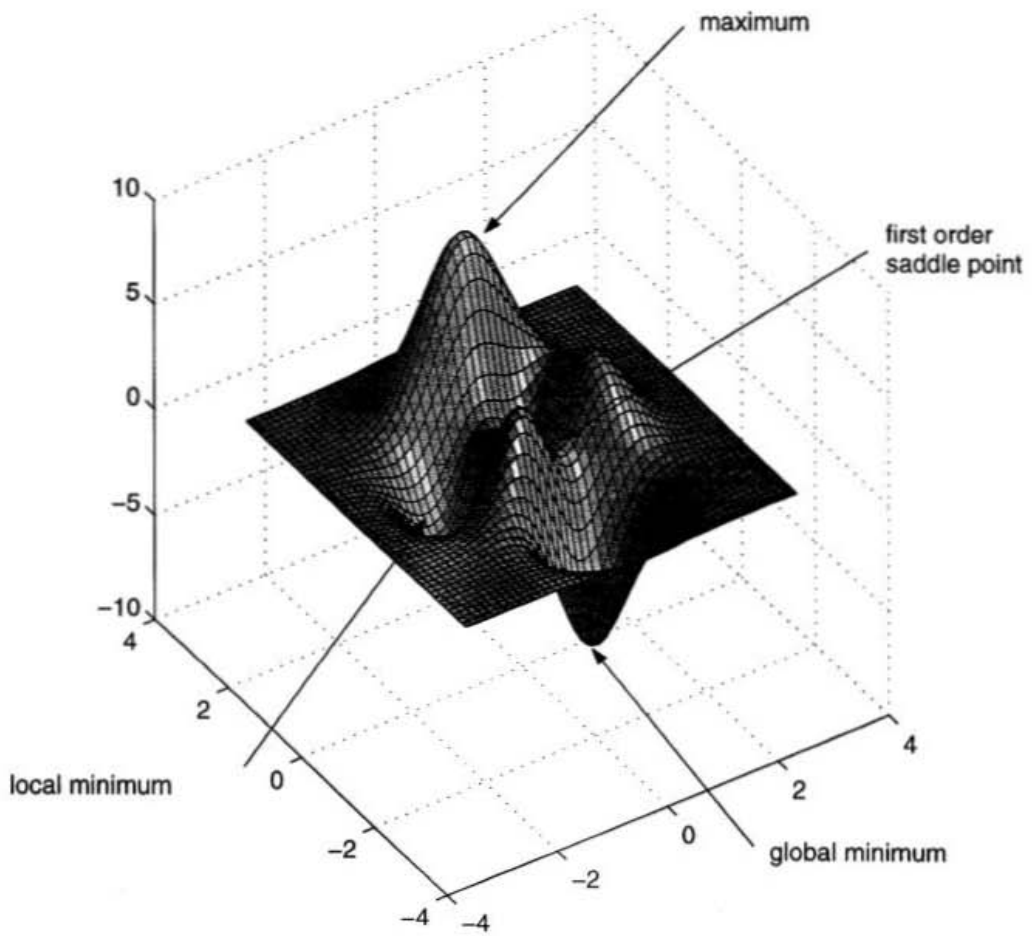

Figure 3.3: Sample plot showing stationary points for $f(x,y) = 3(1-x^2)e^{-x^2} - 10\left(\frac{x}{5} - x^3 - y^5\right)e^{-x^2} - y^2 - \frac{1}{3}e^{-(x+1)^2} - y^2$ [18].

potential energy with respect to either cartesian or internal coordinates. It is a non-linear multivariate function (more than one independent variable) with no imposed constraints. Two possible problems to be solved are minimization and search for saddle points. The methods discussed in this chapter are designed to search for minima. The problem can be formulated using the following equation [4, 15, 16, 17, 19]:

$$\text{minV(q)}, q \in D. \tag{3.3}$$

where the objective function is the potential energy function with respect to the internal coordinates, given as $V(q)$. The potential energy function also contains continuous derivatives up to second order, which is needed to be able to apply the optimization methods.

There are two classes of optimization methods: those that use derivative information and those that do not. The derivative methods are the ones which will be described in this work.

Newton and quasi-Newton methods approximate the potential energy surface with the following quadratic function:

$$V(q_k + p) \approx V(q_k) + g_k^t p + \frac{1}{2} p^t \mathbf{G}_k p. \tag{3.4}$$

where the subscript k is the iteration number, $V(q_k+p)$ and $V(q_k)$ are the current and initial potential energies, p_k is the step vector, g_k is the gradient, and $\mathbf{G}_k$ is the second derivative matrix. The minimum corresponds to a zero gradient, and the descent direction is computed by finding a p which minimizes the following function [15, 16, 19]:

$$\frac{\partial V}{\partial q_k} = g_k(q) = g_k + \mathbf{G}_k \Delta q_k. \tag{3.5}$$

This gives the step size as [15, 16, 19]:

$$\Delta q_k = -\mathbf{G}_k^{-1} g_k. \tag{3.6}$$

The descent direction can be computed by the trust region and line search methods, but will not be described here. The reader is referred to the references [4, 15, 16, 17, 19] for more details.

3.3.3 First Derivative Methods: Steepest Descent

There are several first derivative methods available, with the most common one being Steepest Descent. In steepest descent the second derivative matrix is approximated as unity, giving the step direction and step size as the negative of

the gradient. This is sensible as the gradient is orthogonal to the contours on the potential energy surface where the function value is constant. A negative gradient corresponds to a change in the structure of the molecule to reduce the forces on the atoms which is the direction of steepest descent. Often the step size is taken along the negative gradient, although in some cases a line search is performed [4, 15].

The steepest descent method works well for initial structures which are far from a minimum. If, however, the structure is near a minimum, steepest descent converges slowly. As the optimization is taken close to a minimum, the search vectors become less reliable [4, 15].

3.3.4 Second Derivative Methods

The best known second derivative method is that of Newton and Raphson as it is exact. In the Newton-Raphson (NR) method, the energy is evaluated for the current coordinates, the gradients are computed, a line search is performed to determine the next step direction, the Hessian is recomputed, a step is taken to update the coordinates, and the convergence is tested. In the case of full Newton-Raphson, the Hessian used is the true second derivative matrix.

This method is quick to converge when near a minimum as the quadratic approximation is valid and a true second derivative matrix gives faster convergence than an approximate Hessian. However in places on the potential energy surface where the quadratic approximation is not as good, such as places far from the minimum,

the NR method is slow to converge [15, 16, 19].

Since the inverse Hessian is expensive to compute directly, several methods are available to approximate it and it is the variety in these approximations (update formulas) that gives the different gradient optimization methods.

It is not necessary to compute an exact Hessian and an approximation is much cheaper to compute. An approximate Hessian can be computed by using information about the difference in the computed coordinates and gradients at the current and next steps [15, 19].

Other gradient methods also available to the molecular mechanics package include Broyden, Fletcher, Goldfarb, and Shanno (BFGS) and Newton-Raphson, as well as the optimally conditioned (OC) and direct inversion of the iterative subspace (DIIS) methods. Another method, based on a minimization of sum of squares of the gradients (VA05AD) is also included [4, 15, 16, 19].

The BFGS Method

In the BFGS method, the update to the Hessian is computed by the following relation [4, 15, 16, 19, 20]:

$$H_{k+1} = H_k + \frac{\Delta g_k \Delta g_k^t}{\Delta q_k^t \Delta g_k} - \frac{H_k \Delta q_k \Delta q_k^t H_k}{\Delta q_k^t H_k \Delta q_k} \tag{3.7}$$

The Direct Inversion of the Iterative Subspace Method

DIIS begins by expressing the set of coordinates as a deviation from the expected solution:

$$q_i = q_0 + e_i \text{ where } e_i = -H^{-1}g_i. \tag{3.8}$$

Using the relationship $\sum_i c_i q_i = q_0$, the following system of equations is solved to generate the c_i's:

$$\begin{bmatrix} B_{11} & B_{12} & \cdots & B_{1m} & 1 \\ B_{21} & B_{22} & \cdots & B_{2m} & 1 \\ \cdots & & & & \\ B_{m1} & B_{m2} & \cdots & B_{mm} & 1 \\ 1 & 1 & \cdots & 1 & 0 \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_m \\ -\lambda \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ 1 \end{bmatrix} \tag{3.9}$$

where $B_{ij} =< e_i|e_j >$ and λ is a Lagrangian multiplier. The new gradients and coordinates for the current step are interpolated using $q_{m+1} = \sum_i c_i q_i$ and $g_{m+1} = \sum_i c_i g_i$ and the updated coordinate vector is then given by [21]:

$$q_{m+1} = q'_{m+1} - H^{-1} g'_{m+1} \tag{3.10}$$

The Optimally Conditioned Method

In the OC method, the Hessian is first factored into $H = JJ^T$ and only the matrix J is stored and updated. The change in the coordinates is given as it is for NR and other methods as $-H^{-1}\Delta g$, but the change in the coordinates and the gradients, denoted Δq and Δg, are projections of the actual Δq and Δg. These projections are used to update the Hessian. The optimal conditioning is given as the minimum of the ratio $maximum(\pm\lambda, 1)/minimum(\pm\lambda, 1)$, where the λs are given by the equation $H_+ u = \lambda H u$ [22].

The Minimization of Sum of Squares Method

In the minimization of the sum of squares method (called VA05AD), the sum of the squares of the gradients is used [23]:

$$F(q) = \sum_i [g_i(q)]^2. \tag{3.11}$$

The minimization is then done in terms of F(q), the sum of squares with the goal of giving gradients at the next step which are smaller than those at the current step:

$$F(q_{k+1}) < F(q_k) \tag{3.12}$$

The iteration for the VA method is then given as $x_{k+1} = x_k + \lambda_k \delta_k$ and a suitable λ_k is chosen to satisfy the above relation. If $\lambda_k \delta_k$ is modified to give $q_{k+1} = q_k + \eta_k$, the step size η_k is given by:

$$q_{k+1} = q_k - \frac{1}{2}\left[\frac{\partial F(q)}{\partial q_i}\right]_{q=q_k} / \mu_k, \tag{3.13}$$

where μ_k is a non-negative parameter. This parameter is then optimized to ensure subsequent iterations give smaller values for the sum of squares of the gradients than the previous iterations [23].

3.4 Coordinate Systems

3.4.1 Definition

The potential energy must be expressed with respect to a particular set of coordinates. These define the positions of the atoms with respect to one another. There are several options for coordinate definition and depend on the problem being solved.

In molecular mechanics, the coordinate system most often used is redundant

internal coordinates, where every possible stretch, bend, torsion, and out-of-plane bend is included. These values can be computed using the cartesian coordinates and simple vector relationships, and are the topic of the next subsection [9, 10].

3.4.2 Definition of Redundant Internal Coordinates from Cartesian Coordinates

The definitions of the coordinates are given in chapter 2 and the reader is referred to this chapter for a definition of them. The generation of the coordinates from the cartesians will be given here.

A bond is defined as the magnitude of a vector from atom i to atom j of the bond. It is also the length of the bond, and is described by the relationship [24]:

$$r_{ij} = \sqrt{(x_j - x_i)^2 + (y_j - y_i)^2 + (z_j - z_i)^2}. \tag{3.14}$$

The values x, y, and z are the cartesian coordinates for the atoms.

Let i, j, and k denote three atoms in an angle, where j is the central atom and is bonded to i and k. Also, let $\overrightarrow{A}$ and $\overrightarrow{B}$ be the vectors $\overrightarrow{ij}$ and $\overrightarrow{kj}$, respectively. Then the angle is given as the dot product between the two vectors $\overrightarrow{A}$ and $\overrightarrow{B}$ [24]:

$$\theta_{ijk} = \arccos\left(\frac{\vec{A} \cdot \vec{B}}{|\vec{A}||\vec{B}|}\right). \tag{3.15}$$

Let i, j, k, and l denote four atoms in a torsion, where the central bond is formed between atoms j and k, and j is bonded to i and k is bonded to l. Also, let $\vec{A}$, $\vec{B}$, and $\vec{C}$ denote the vectors $\vec{ij}$, $\vec{jk}$, and $\vec{kl}$, respectively. Then the torsion is given as the angle between two vectors formed from the following cross products [14, 24]:

$$\begin{aligned} \vec{D} &= \vec{A} \times \vec{B} = |\vec{A}||\vec{B}| \sin\phi_1 \\ \vec{E} &= \vec{B} \times \vec{C} = |\vec{B}||\vec{C}| \sin\phi_2, \end{aligned} \tag{3.16}$$

where ϕ_1 is the angle between $\vec{A}$ and $\vec{B}$, ϕ_2 is the angle between $\vec{B}$ and $\vec{C}$, and the vectors $\vec{D}$ and $\vec{E}$ are the resulting vectors of the cross product. The torsion is then given by the following scalar product between $\vec{D}$ and $\vec{E}$ [14, 24] :

$$\phi = \arccos\left(\frac{\vec{D} \cdot \vec{E}}{|\vec{D}||\vec{E}|}\right). \tag{3.17}$$

Let i, j, k, and l be four atoms in an out-of-plane bend where atom j is the central atom and is bonded to only atoms i, k, and l. Also, let $\vec{A}$, $\vec{B}$, and $\vec{C}$ denote the vectors $\vec{ij}$, $\vec{jk}$, and $\vec{jl}$, respectively, and $\vec{D}$ is a vector resulting from the vector

product of $\vec{A}$ and $\vec{B}$. Then $\vec{D}$ and the out-of-plane bend are given by the following relationships [14, 24]:

$$\begin{aligned} \vec{D} &= \vec{A} \times \vec{B} = |\vec{A}||\vec{B}| \sin \phi_1 \\ \chi &= 90^\circ - \left[\arccos\left(\frac{\vec{C} \cdot \vec{D}}{|\vec{C}||\vec{D}|} \right) \right]. \end{aligned} \tag{3.18}$$

3.4.3 Coordinate Conversion: Redundant Internal Coordinates to Cartesians

For the molecular mechanics package, it is possible to do the optimizations without converting from one coordinate system to another, as the optimization package updates the coordinates directly. However, if the cartesian coordinates are needed, they can be converted from the redundant internal coordinates using an iterative process and a transformation matrix called the **B** matrix [25, 26, 27].

In some molecular mechanics packages, the geometry optimization is run using cartesian coordinates, however for ring systems and to speed up convergence, redundant internal coordinates are preferred [25, 26, 27].

The conversion from redundant coordinates to cartesians is curvilinear, and must be done iteratively. The relationship between the two coordinate systems can be written as [25, 26, 27]:

$$\begin{aligned} \Delta q &= \mathbf{B}\Delta x \\ &= \frac{\partial q}{\partial x}\Delta x. \end{aligned} \tag{3.19}$$

The Δx is the difference between x values at successive iterations of the above and the Δq is the difference between the redundant coordinates from the current optimization step and those updated during the iterative process. A matrix $\mathbf{G}$ can then be defined by the relation $\mathbf{G} = \mathbf{B}^t\mathbf{B}$. Because the set of q is redundant, the $\mathbf{B}$ matrix contains linearly dependent rows and $\mathbf{G}$ is singular and it is not possible to compute an ordinary inverse [1]. Ordinary inverses can only be computed for nonsingular square matrices, however it is possible to compute a generalized inverse [28].

The generalized inverse is computed using a diagonalization followed by a back-substitution step, and the most popular method is the singular value decomposition [28].

The update to the cartesian coordinates is given by rearranging the previous equation to give:

$$\Delta x = \mathbf{G}^{-}\mathbf{B}^t\Delta q. \tag{3.20}$$

[1] Only the ordinary inverse gives the identity matrix if multiplied by the original matrix, shown by the relationship $\mathbf{G}\mathbf{G}^{-1} = \mathbf{I}$ [28].

To determine if the iterative conversion process is finished, the change in the cartesians is checked and the iterations stopped when the change is less than a threshold value, usually 10^{-6}. If for some reason the iteration does not converge, the results of the previous iteration are used and the process is stopped [26]. A test for divergence is done by checking at each iteration to make sure the current value is not greater than the previous value of the updated cartesians. This iterative procedure will work well if the step taken by the minimization procedure is not too large [27].

3.5 Summary

Obtaining an optimized geometry is central to molecular mechanics. There are several steps involved, and these were summarized in figure 3.1. From an initial set of Cartesian coordinates, the molecular graph information is obtained. The redundant internal coordinates can then be built from the molecular graph information and the Cartesian coordinates. Geometry optimization can then be completed using the redundant coordinate system and these can be updated by conversion of the new set of redundant coordinates generated at each optimization step to Cartesian coordinates. Because this conversion is curvilinear, it must be done iteratively. The new Cartesian coordinates are used to generate the redundant coordinates for the next optimization step. This process is repeated until convergence is reached.

There are a variety of different methods for geometry optimization, and they fall into two classes. First derivative methods use first derivative information and

include the steepest-descent method. The second derivative methods use both first and second derivative information and include both Newton-Raphson and the family of quasi-Newton methods. Selection of a method depends on the ease of computing the function and its derivatives. Central to coordinate generation is the molecular graph. Before the coordinates can be built, connectivity and valence information is needed, and this is built using relationships derived from topology.

Chapter 4

Object-Based Analysis and Design

4.1 Introduction

There are several details which must be considered when the time comes to start designing code. The analysis and design of any code involves deciding what features the code should have to be considered a good program (see section 4.2) and selecting a method of analysis and design which best models these features in the code. Two types of analysis and design methods often used are called procedural-based and object-based design. Procedural-based design involves organizing the code in terms of the functional steps executed in transforming the statement of the problem one wishes to solve to the desired solution. Object-based design breaks the problem domain (the model of the problem in the real world described by the problem statement) into subunits called objects, and these objects interact to give the solution. There are advantages to using the object-based design over procedural-based design in solving

problems in large codes, and as a result, it will be the method outlined in this chapter.

It should be emphasized to the reader at this point that the term *object-based* is used loosely here. The ideas of this version of the object-based design method are not the same as those used in *object-oriented design*, but may have some similarities.

The object-based design method starts with a description of what features are present in *good* programs. This is followed by a description of the problem in terms of three models: the object, dynamic, and functional models. These models have many similarities with the models in the *object-oriented* design strategy, however the definitions of the parts of the model differ somewhat and the two should not be confused with one another. In this work, the *object-based* design strategy was used and as the analysis and design is given for the molecular mechanics problem described in Chapter 2, the descriptions of the parts of each model and design steps will be given in more detail. After the parts of the model are complete, a language is selected for implementation and the process of implementing the models in the code begins.

4.2 Features of Good Programs

The first step in designing code to solve a problem is deciding on the features it should have which would be desireable to both the programmer and the user. There are several general features all software should have to be considered *good software.* They are correctness, robustness, extendibility, reusability, compatibility, efficiency, portability, verifiability, integrity, ease of use, and proper documentation, and are

defined as follows.

Correctness and Robustness

Correct code does what it was intended to do for all possible known cases. This is the most important feature because if the code does not behave as expected, then the other features mean nothing. However, one cannot possibly know all cases that need to be run. In these cases, should the program fail, it should do so in a clean manner with the proper error message to alert the user or programmer of the problem. This feature is known as *robustness* [29].

Extendibility

Extendible programs are easy to modify or extend, for example when a new feature is needed. For small programs this is not an issue but for large complex programs it is essential. In order to make code more extendible, two things can be done [29]:

1. *Simplify the code:* the program should be designed in a simple manner, with a simple architecture.

2. *Divide program into smaller parts:* The program can be divided into smaller independent subunits of the program.

Reusability

Reusability of the code is also an important feature. The more code is reused, the less code needs to be rewritten and this reduces the cost of development. What parts can be reused is determined by finding parts of the code that are the same or share a common piece [29].

Compatibility

Compatibility is the ability of subprograms to be combined easily without conflicts between each part. This is a necessity for projects which involve more than one contributor. This is important in enabling software parts to interact with one another [29].

Efficiency

The program should be *efficient*, that is it should make optimal use of the hardware and software components of the system it runs on [29].

Portability

Portability, the capability of the program to run on a few different systems, is an important feature. In ensuring this feature, any machine-specific parts should be clearly defined in the documentation [29].

Verifiability

The programmer should be able to prepare test data and procedures to determine if there are any problems with the software. This would be best accomplished if test data was included with the software package along with instructions on running test data and listing of expected output [29].

Integrity

The program should have *integrity*, that is the program components should not be able to corrupt one another. Utilities can be designed to handle security within the program and this should be an essential part of the software design [29].

Ease of Use

The software should be *easy to use* (operate, prepare input, analyze output, handle errors) and should come with documentation to instruct the user what to do if a problem is encountered [29].

User and Programmer Interests

From the user's point of view, the program should be correct, robust, compatible, portable, efficient, easy to use, and be well documented. From the programmer's point of view the program should have all of the above features and also be extendible, reusable (or have reusable parts), verifiable, and have integrity. Ideally for

both the user and the programmer, the program should be optimal in every feature, but this is not always possible. As a result, there are tradeoffs between these features. When designing programs one should try and balance each feature in the best way possible [29].

4.3 Decomposition Approaches

Breaking a problem down into smaller units is central to code design. There are two main ways in which the problem can be viewed. The flow of execution from the problem statement to the desired solution can be examined or the model of the real-world problem (called the *problem domain*) can be subdivided into independent units, called objects, which interact with one another. These two methods of subdividing the problem are called *algorithmic* and *object-based decomposition.* There are advantages and disadvantages to using one or the other decomposition, and this is dependent on both the size and type of problem being examined [30].

4.4 Analysis Steps

The first step in the analysis stage is to completely specify the problem domain using all three models, giving three parts to the analysis. In doing so, no implementation details are to be considered. The goal of the analysis is to completely understand the problem and how to obtain the solution. A good description of the

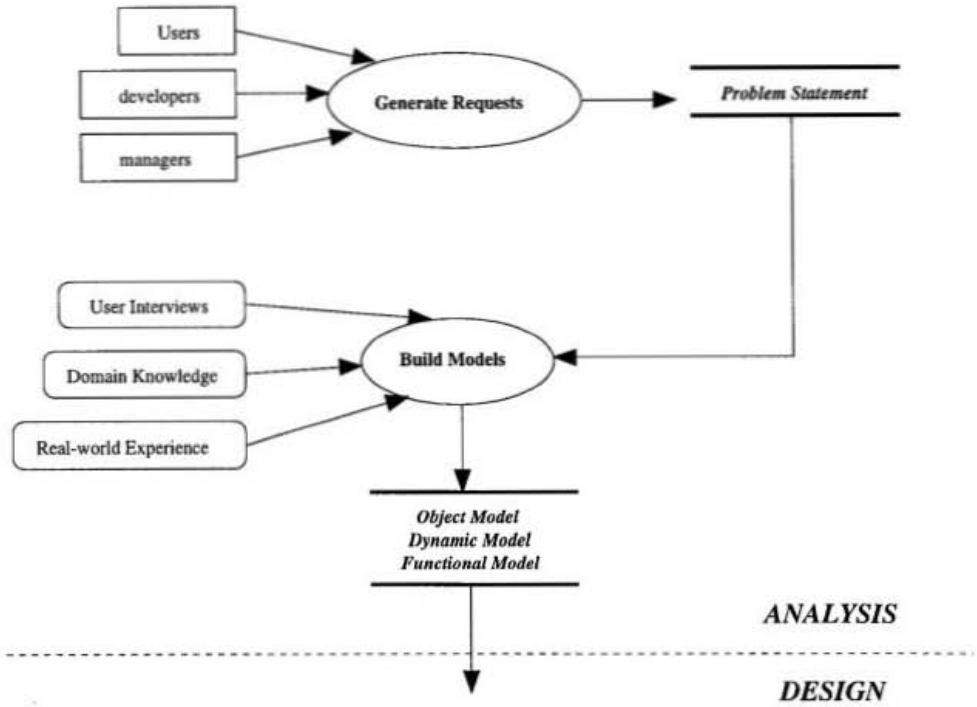

Figure 4.1: Parts of Analysis and Design.

problem enables the designer to obtain a clear picture of it through the three models.

The main parts to analysis and design are shown in figure 4.1 along with their interaction with the real world. The analysis stage is an iterative process that must work to unify all three models. There are four main steps to analysis: the problem is first described in words and the three models are built from both the knowledge of the problem and the words in the problem description [30, 31].

4.4.1 Describing the Problem

The first step to describing the problem was to decide what was needed in the program. This helped to define the necessary parts of the problem. Next, the features

available to users were decided upon as well as which of these were optional and which were not. When formulating the problem statement, no system requirements were considered as they would restrict the analysis. Protocols, without implementation details, were also decided upon and these were to be followed. Also any assumptions which were made in formulating the problem were clearly stated [30, 31]. The problem description is given in Chapter 2.

4.5 The Object Model

Of the three parts to object-based analysis and design, the object model is the largest and encompasses the most concepts. It involves breaking the problem into separate abstractions, defining the attributes and behaviors of these abstractions, defining the communication between them, and relating them to give a completed picture of the problem domain. The four main parts of the object model are abstraction, encapsulation, modularity, and hierarchy. Together they can be used to create programs that have the desired features [29, 32].

4.5.1 Parts of the Object Model

Abstraction

Abstractions are classifications based on the properties of objects that make them different from other objects of different kinds. They are designed to model the real world version of the object and help to reduce the problem domain into

subproblems that can be worked with. They provide clear descriptions of the problem and are based on how their designer interprets them in the real-world model. They are not implementation-dependent and are not complete or accurate, due to the limited ability of people to verbally describe the world around them. Abstractions also provide an external view of an object [29, 32].

The idea behind creating abstractions is to concentrate on those attributes that belong to objects that have some purpose and supress those that do not. They can usually be determined by grouping objects together which share a common purpose and selecting the attributes the objects have in common [29, 32].

Encapsulation

Encapsulation, also known as *information-hiding*, is used to hide the implementation details of the program from the user while creating software which gives enough information to enable the user to easily use it. It also provides barriers between abstraction levels. An interface can be used to achieve the illusion of simplicity by hiding these implementation details. The idea behind encapsulation is twofold: first the details of the implementation of a program are not needed by the user and it promotes independent description of objects so that if one object is changed this will not affect the objects which use it [29, 32].

Modularity

Modularity is a property of a program that is comprised of program pieces, each which have optimal communication and contain a complete set of abstractions. It creates defined partitions within the problem domain and these well-defined sub-problems can be combined to produce the software's architecture. The module can be viewed as a container in which the abstractions can be placed. These modules can be compiled separately and interact with each other through messages. One can place one or many classifications in a module. Communication between modules must be optimal. This enables changes such as additions and deletions to be done easily without the recompilation of entire module interfaces, although this issue also depends on the language used. A good combination of encapsulation and infomation sharing between modules should also be created and one must be careful not to create too many modules [29, 32].

Hierarchy

A *hierarchy* is a logical ordering of abstractions and objects. Two important hierarchies within programs are the structures of both the abstractions and the objects. An important part of hierarchy is *inheritance* and is described as a definition of the relationships between abstractions. Inheritance is defined by *is-a* or *kind-of* relationships, for example an apple *is-a* kind of fruit. There are three types of hierarchies: single inheritance, multiple inheritance, and aggregation.

Single inheritance occurs when one abstraction inherits the attributes and behaviors of another abstraction. *Multiple inheritance* occurs when one or more abstractions, also called *classes*, inherit the attributes and behaviors from more than one abstraction. Usually similar attributes and behaviors become part of superclasses and different attributes and behaviors become part of subclasses. *Aggregation* involves *part-of* relationships and one class is usually contained within another. One important benefit of hierarchies within code is they promote the reuse of common code fragments [29, 32].

4.5.2 Other Concepts in the Object Model

Classes and Objects

An *object* in the real world is a visible thing which has clearly defined boundaries on its definition. It has distinct properties, a state which may change over time, and some well-defined behavior. It is capable of performing actions itself or a user may perform actions on it. Often the objects in a problem to be solved on a computer are defined to model an object in the real world. They also have state, behavior, and some clearly-defined properties. Objects can also be concrete, like something that is visible, or abstract, for example, mathematical formulas. They also have their own identity, which means that every object, whether it has the same attribute values and behaviors as other objects, have some characteristic which makes them different from objects of the same kind [30, 31, 32, 33]. In the description of the problem to

Figure 4.2: A sample class diagram.

be solved, objects are usually the nouns.

A grouping of objects with the same attributes and behaviors can form a *classification*, usually shortened to *class*. The class describes the attributes and behaviors that are similar in the group of objects for which it represents. The objects are different in the values for their attributes and how they relate to other objects is different. What is defined as a class in the problem domain depends on the judgement of the individual performing the abstractions and the definition of the problem. As a result, there is no right or wrong way to define a class, however there are some guidelines that can be followed [30, 31, 32, 33].

A class should provide a very definite abstraction of the objects it represents as determined from the description of the problem domain. It should also contain a set of responsibilities, however small, and it should perform these as expected. They should also be simple in design and easy to understand, although still be built so they may be modified or extended. Finally, they should clearly describe the behavior of the abstraction performed and yet keep the details of their implementation well hidden. Also, they should easily model the group of real world objects for which they were designed.

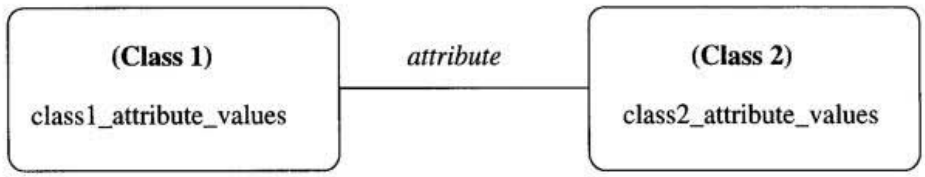

Figure 4.3: A sample object diagram.

There are a couple of diagrams that are useful in describing the object model pictorially. They are object and class diagrams. An example of a class diagram is given in figure 4.2. Usually diagrams are used for abstract modeling and are not useful enough to move directly from them to implementation. The object diagram, however, can be used for both designing the actual program and abstract modeling [30, 31, 32, 33].

An example of an object diagram is given in figure 4.3. In both object and class diagrams, the object or class is defined by a rectangular box, but in the object diagram the corners of these boxes are rounded. In the boxes go the names of either the class or the object. In the object diagram the name of the class is placed in parentheses at the top of the box representing an object that is an instance of it. It is also possible to add attributes to both class and object boxes. Within the class box a line is placed between the class name and the attributes and the attributes are listed with their types with a colon separating the attribute and its type. In the object box the class is listed in boldface and the attribute value is listed in regular print below the class name [30, 31, 32, 33].

Methods (or behaviors) can be added to the class boxes by placing the method below the attribute list. The attributes and methods are separated by a line in the

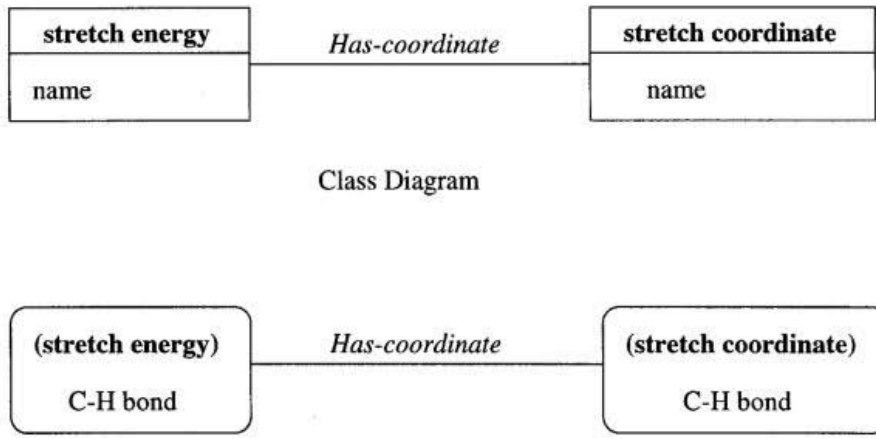

Figure 4.4: Class and instance diagrams showing a one-to-one association between the stretch energy and the stretch coordinate for a C-H bond.

class box. The inputs and their primitive types (for example, integer or character) are also listed. The inputs and types are separated by a colon and placed in parentheses after the methods. Following the inputs but outside the parentheses are the outputs, of which the type of output need only be listed.

The way an object or class relates to another object or class is represented by links and associations. A *link* is much like an object itself, and each link is an example of a particular type of association. *Associations* can be defined as one class (or object) associated with another class (or object) in a very general sense and vice versa. To be more specific, a link is used to indicate relationships between classes or objects in the diagram. An object or class may be linked to one object or class and this is a one-to-one relationship. An example for a one-to-one association is given for the stretch energy and its coordinate in figure 4.4. If more than one object or class is involved in the link it is a one-to-many, many-to-one, or many-to-many relationship.

They are often represented as pointers in a programming language, although this depends on the language being used [30, 31, 32, 33].

In the object and class diagrams, binary relationships are denoted by a line and ternary relationships by a diamond with lines coming from its corners. If a particular end of the relationship corresponds to multiple objects or classes, the line is terminated by a solid circle, if the end corresponds to zero or more classes, the line ends with a hollow circle, and if the end corresponds to one class or object, the line ends with nothing attached [30, 31, 32, 33].

Inheritance

Inheritance is defined by an *is-a* relationship. The main motivation for determining these relationships is code reuse. Often objects may share things in common with their own class and also with objects belonging to other classes. In this case it would be beneficial to extract the attributes and behaviors common to all the classes involved in the inheritance and create a superclass containing these attributes and behaviors. Any attributes and behaviors that are different in the remaining classes can be added to the *subclasses*, which would then inherit the components of the *superclass*. Inheritance also helps to simplify a problem by reducing it to a group of classes, some of which can be conceptually formed from others [31].

The brief outline of inheritance was given to show the complete picture of the object model. However, no inheritance relationships are present and as a result it will not be discussed further.

4.5.3 Building the Object Model

Not all of the objects and classes can be obtained from the list of nouns, although this list is a good first approximation. Some objects not stated as nouns will be implicit in the problem statement. It is also possible some of the classes and objects were poorly stated and in a first set of objects and classes this is quite common. Some possible poorly stated classes and how to fix the problems are:

- If two classes contain the same information then they are redundant and only one should be kept.
- Classes that have no relevance to the problem are irrelevant and should be eliminated. Only keep classes and objects that are necessary in the domain of the problem, no more and no less.
- Classes with an unclear description should be reformulated or removed. A class should be clear and simple.
- Classes which describe one object would be better suited as attributes unless the class makes sense in the context of the problem.
- If a class describes a particular operation and is not operated on itself or it has no features itself, then it is probably an operation and not a class.
- Objects should not be named according to their role in an association.

- There should be no objects or classes which describe some implementation construct. The only objects and classes that should be present should be relevant to the real world model of the problem [30, 31, 32].

In order to assist in choosing classes and objects, it would be helpful to build a data dictionary.

DATA DICTIONARY:

Energy: a quantity computed using the parameters and coordinate values for particular interactions. This quantity must be minimized to get the optimized structure. It contains contributions from each coordinate and can be summed to give a total contribution. The number of energies is given by the number of coordinates.

Gradient: a quantity which is also derived from the coordinate and parameter values, and is equal to the analytical first derivative of the energy with respect to a particular coordinate. This quantity must be zero for minimum structures. It contains contributions corresponding to the coordinates and the number of contributions is given by the number of coordinates.

Second Derivative: another quantity derived from the parameter and coordinate values. It is the analytical second derivative of the energy with respect to the coordinates and contains these second derivative quantities with respect to the coordinates. It also contains the same number of contributions as the number of coordinates squared.

Coordinate: The coordinate is a particular quantity used to describe the geometry of the molecule. Each coordinate object contains a set of contributions which are the coordinate values for a particular type of interaction and include only 1-2, 1-3, and 1-4 bonded interactions. The number of coordinates is given by the number of interactions within the molecule.

Molecular Mechanics Atom: The molecular mechanics atom is described mainly by the atomic number, hybridization, size of ring it belongs to, and whether it belongs to an aromatic system or not. The result is an assigned atom type based on these values. The atom type is then used in the selection of the parameters for a particular interaction (based on the types of atoms involved in the interaction). This quantity is specific to the force field used.

Parameter: The parameter is a set of predefined data values for a particular set of molecular mechanics atoms and interactions. It contains sets of parameters for each interaction type and is used in the energy, gradient, and second derivative computations.

Shown above is an example of a data dictionary for the molecular mechanics problem. In it, the objects are defined, their properties are given, and its uses and responsibilities are defined in the context of the real world problem. This will help to pick out classes, objects, and associations [31]. The interaction class mentioned in several places in the above data dictionary is not defined in the molecular mechanics code. It is defined by the topology code and it is for that reason it is not included

explicitly in the analysis and design of the molecular mechanics program.

Using Chapter 2 and the above data dictionary as guides, the objects were selected and placed in a group according to shared attributes and/or behaviors. The classes in the molecular mechanics package and the corresponding objects are:

- *energy*
  - stretch, bend, torsion, out-of-plane bend, van der Waals, electrostatic, stretch-bend
- *gradient*
  - stretch, bend, torsion, out-of-plane bend
- *second derivative*
  - stretch, bend, torsion, out-of-plane bend, stretch-bend
- *coordinate (redundant)*
  - stretch, bend, torsion, out-of-plane bend
- *molecular mechanics atom*
  - atoms in the molecule
- *parameter*
  - stretch, bend, torsion, out-of-plane bend, van der Waals, electrostatic, stretch-bend

The classes and the list of attributes and behaviors are given by investigating the class descriptions. The data dictionary provides the information needed to form the attribute and behavior list for each class and is given as follows:

- **energy**
  - *attributes:* energies, number of energies, sum of energies
  - *behaviors:* initialize energy, build energy
- **gradient**
  - *attributes:* number of gradients, gradients
  - *behaviors:* initialize gradient, build gradient
- **second derivative**
  - *attributes:* number of second derivatives, second derivatives
  - *behaviors:* initialize second derivative, build second derivative
- **coordinate (redundant)**
  - *attributes:* coordinates, number of coordinates
  - *behaviors:* initialize coordinates, build coordinates
- **molecular mechanics atom**
  - *attributes:* atomic number, hybridization, ring size, atom type, aromatic/not aromatic

- *behaviors:* initialize atom

- **parameter**

  - *attributes:* stretch, bend, torsion, out-of-plane bend, van der Waals, electrostatic, stretch-bend

  - *behaviors:* initialize parameters

In the above list, all energy objects would have a specific value for each attribute depending on the object, all gradient objects would have specific values for the gradient attributes, and so on. In the case of the parameters, the objects all have a parameter set, but do not necessarily have the same type of parameters. An example is given by the stretch and torsional parameters. The stretch parameters have the equilibrium value and force constant as part of the parameter list for each stretch and for each torsion there is no equilibrium value or force constant, but three torsional parameters. As it turns out, the stretch, bend, out-of-plane bend, and stretch-bend parameter sets each have equilibrium values and force constants, the torsional parameter list contains the three torsional parameters, the van der Waals parameter list contains the minimum energy separation and well depth, and the electrostatic parameter list contains the charges on each atom. Therefore, the parameter list is an object which includes all parameter lists as its attributes.

An association is a dependency between two or more classes and one class usually refers to another. Within the problem statement, the verbs generally describe associations between the objects, one object referring to another. From the groupings

of these verbs abstractions can be made on the associations, giving association objects and classes. It is best to write down all the verbs and examine the problem statement for implicit associations [31].

As a start to a list of associations, the verbs or verb phrases in the data dictionary previously given were investigated. These associations correspond to actions in a particular direction, for example data use in a computation. In the molecular mechanics case, the associations correspond to the use of one particular quantity or set of quantities to compute others. For example, the computation of the energy for the stretches requires the stretch coordinates and parameters and the associations between these classes correspond to requests from the stretch energy object to the coordinate and parameter objects for this data. The following table gives each set of objects which share an association and the corresponding association:

Table 4.1: Class Associations

Class 1	**Association**	**Class 2**
energy	uses	parameter
energy	uses	coordinate
gradient	uses	parameter
gradient	uses	coordinate
second derivative	uses	parameter
second derivative	uses	coordinate

continued from previous page		
Class 1	**Association**	**Class 2**
parameter	uses	atom

In the above table 4.1, the class on the left has the association in the middle with the class on the right. For example, in the above table, the entries in the first row state the energy objects use the parameter objects in the computation of the energy contributions and must send a request for this information to the parameter objects.

4.5.4 The Object Model Diagram

The above pieces can now be combined to give a pictorial representation of the object model for the molecular mechanics problem. This diagram is shown in figure 4.5 below.

The parts of this diagram were described in some detail in the previous section (section 4.5.2).

4.6 The Dynamic Model

The object model is a good start for describing a problem domain in object-based terms, but is not a complete description. The object model describes the system from a static point of view, but in many cases the problem domain is dynamic. The *dynamic model* investigates the problem based on the state of the objects and

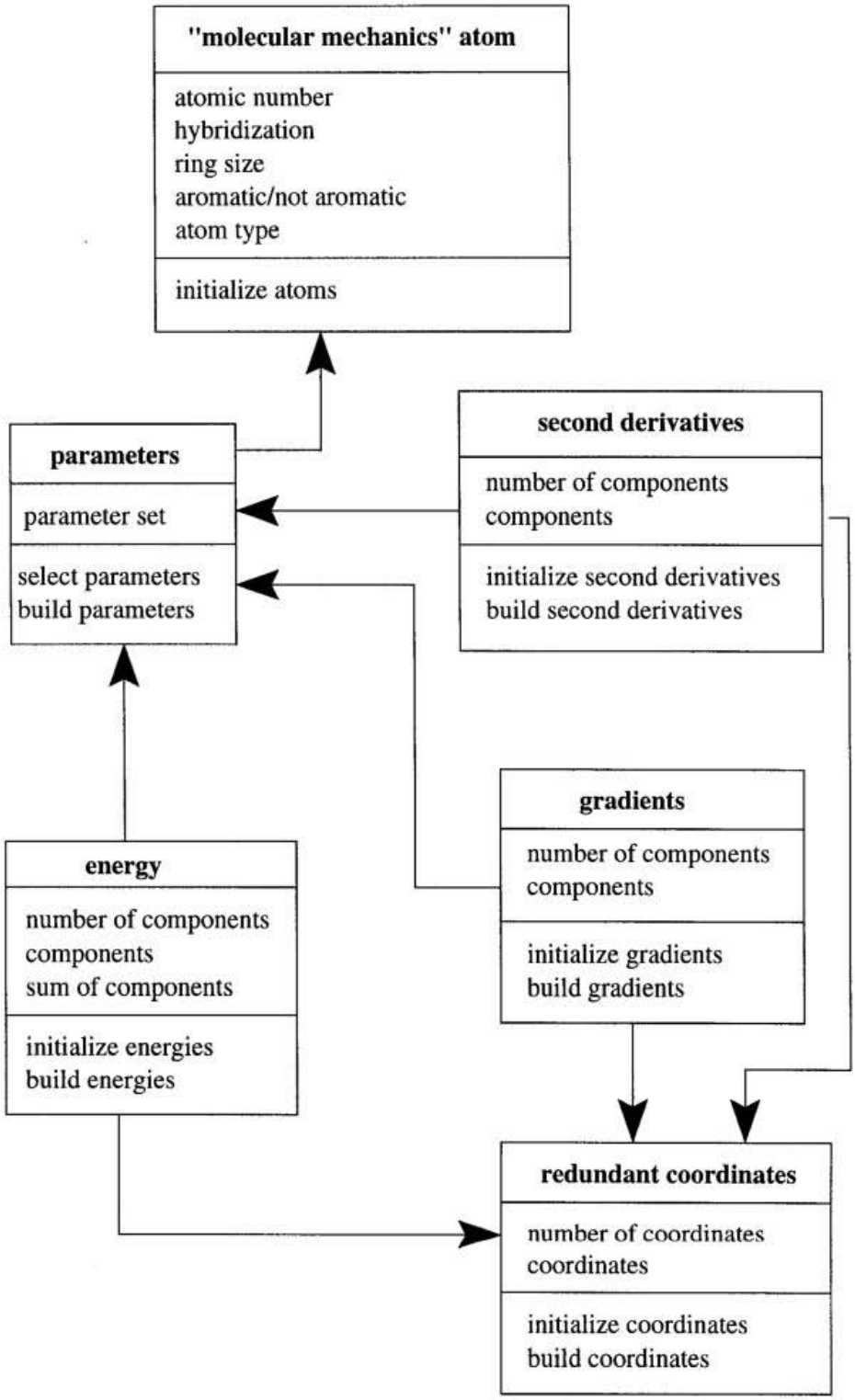

Figure 4.5: The object model for the molecular mechanics package.

events which occur that change some or all of these states. It is a description of the evolution of a system of concurrent objects, describing the actions and reactions of the objects to events (external stimuli) [30, 31]. For the molecular mechanics problem the objects defined above are static. The transformation from input to output is defined in one step and the dynamic model is therefore trivial and will not add additional information to assist in the understanding and breakdown of the problem. For this reason the description of the molecular mechanics problem in terms of the dynamic model will not be covered in this work.

4.7 The Functional Model

The *functional model* represents the flow of data from input to output and shows how the input is transformed to give the output. It represents a sequence of operations (or functions) that operate on the data to achieve the final result. It also includes constraints placed on data or operations and does not include control or object information. It explains more clearly the actions shown in the dynamic model and the operations present in the object model. It is very useful for programs which compute the results of a function [31].

A relationship exists between data values through the functions which act upon it. This is shown with the functional model and is best built after the object and dynamic models are built. The processes in the functional model correspond to the behaviors in the objects and the data flow corresponds to movement and changes in

the objects' attribute values. Building the functional model requires five steps and they will be given below with a description of each step in some detail [31].

4.7.1 Input and Output Values

The design of the functional model begins by making a list of input into the program and the output the program will produce. These usually correspond to the passing of information between the system and external clients such as the users [31].

The molecular mechanics package obtains its inputs from the results of the menu system. If an input file which contains the molecular structure information in the form of a Z-matrix or cartesians, is given to the program, the menu system parses it and obtains molecular information which it converts to cartesian coordinates. This information is used by the topology program to generate a series of interactions within the molecule, at which time the molecular mechanics package obtains, as its input, the cartesian coordinates and atoms involved in the interactions within the molecule.

Output from the molecular mechanics code should consist of the contributions to the energy and the molecular structure in cartesian or internal coordinates.

4.7.2 Data Flow Diagrams

Data flow diagrams specify the flow of data through the program, including operations that use or change this data and any constraints it must satisfy. It also shows relationships between data in a system, the functions used to transform it,

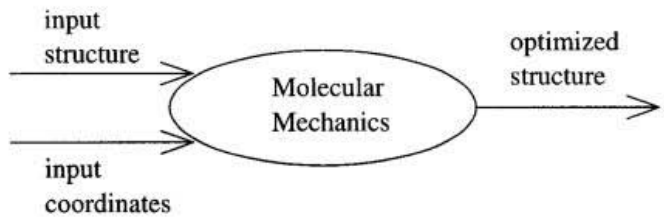

Figure 4.6: A sample high-level data flow diagram.

data stores, input and output values [31].

There is no information in these diagrams which would indicate what path to take through a program or the time the program spends in a certain state, nor does it show how the data is placed into abstractions.

The *processes* are responsible for transforming the information and can be implemented as either high-level or low-level processes. An example of a high-level data flow diagram for the molecular mechanics problem outlined in Chapters 2 and 3 is given in figure 4.6. The high-level processes can be broken down further to give other data flow diagrams and the low-level processes represent a function which cannot be further broken down. In the data flow diagram a process is drawn as an ellipse with the process name in regular print inside. The input data is shown as an arrow leaving from its source (external or another process) and ending at the process. The output data is shown as an arrow leaving the process [31].

Each data flow diagram contains processes, data flows, actor, and data store objects. *Actors* are processes which consume or produce data. They can be the consumers of the eventual output or the producers of the input into the program. They are represented in data flow diagrams by a rectangle with the actor's name

inside it.

Data stores are sources for the storage of data. The data inside a data store is not generated by the data store and is there for some later purpose. The information in the data store may be retrieved in any order, and is usually accessed using some kind of search key. A data store is represented in the data flow diagram as two thick parallel lines with the name of the data store in bold . Arrows come into and out of the data store and represent the input operations that access the data using some key or modify it in some way and the output operation of retrieving the data. Despite the fact a data store is just data storage, it is usually implemented as an object because its usage is different from the other data flow elements. Actors are also implemented as objects [31].

Constructing the Data Flow Diagram

To construct the data flow diagram the method of obtaining the output from the inputs must be determined. The first step is to start by making a high-level data flow diagram with the problem as the method, its inputs, and the desired outputs. Then the method is systematically broken down. Each output is used as a guide and a trace is done back through the diagram to find the function that computed the particular output. The inputs can also be used as a guide and the methods used to generate the final results can be determined. Next all the processes are expanded until they become atomic (they cannot be further broken down). At this point the processes can be described with natural language, an algorithm, or mathematical formula. The data

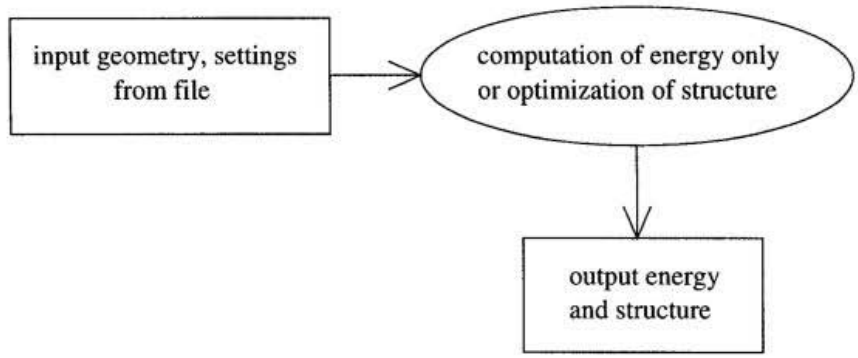

Figure 4.7: High-level diagram (first approximation) for the flow of data through the entire molecular mechanics program.

that is internal but stored is then identified, as are any control flows. These control flows are labeled and the actor objects (sources and sinks for data) are determined. Also data stores must be determined and they usually correspond to external data which is accessed or modified [31].

From the problem description in Chapter 2, the data flows can be determined. A second high level data flow for the molecular mechanics package is given by figure 4.7. It is more descriptive than figure 4.5 and shows another stage in the breakdown process. This does not show the complete data flow through the program, but is an approximation to it. Further breakdown will be shown in a somewhat iterative manner. Once the data is given to the program, the structure and energy are optimized (in the case of an energy minimization) or computed (in the case of a single-point calculation), and the outputs are printed to a file.

The next step in constructing the data flows is to further break down the transformation processes defined at a top level. Based on the input commands, the molecular mechanics program can be executed for just an energy calculation, the geometry

can be optimized (the energy is minimized), or generate one or more of the objects. For example, if one wanted the gradient contributions for each interaction within the molecule, these can easily be printed by themselves.

Since the only concern is the execution of the molecular mechanics code, details of the geometry optimization, menu functions, and generation of topological information will not be given. In the diagram, a representation of these will be in terms of a process. Figure 4.8 shows a diagram of the result of a further breakdown of the data flows. In this diagram, the data is input in the form of a file and it is parsed by the menu. One of the menu commands is MM, which selects the molecular mechanics method as a way of computing the energy and structural information.

Another command is Geom, which selects a geometry optimization. Once the method and type of calculation (single-point, geometry optimization, or output of an object) are determined, the data required is passed to a process which generates the cartesian coordinates. The topology is then generated using these coordinates. It is the cartesian coordinates and topology information which is used in the computation of the molecular mechanics objects. After the molecular mechanics package has computed the total energy and coordinates, they are sent to an output file. If an optimization is required,the optimization is performed before these quantities are output. If an object was selected for computation, then it is sent to the output file.

Figure 4.9 shows the flow of data through the molecular mechanics method. As it shows, the molecular geometry and topology are used in the computation of the parameter and redundant coordinate objects which are sent to the processes which

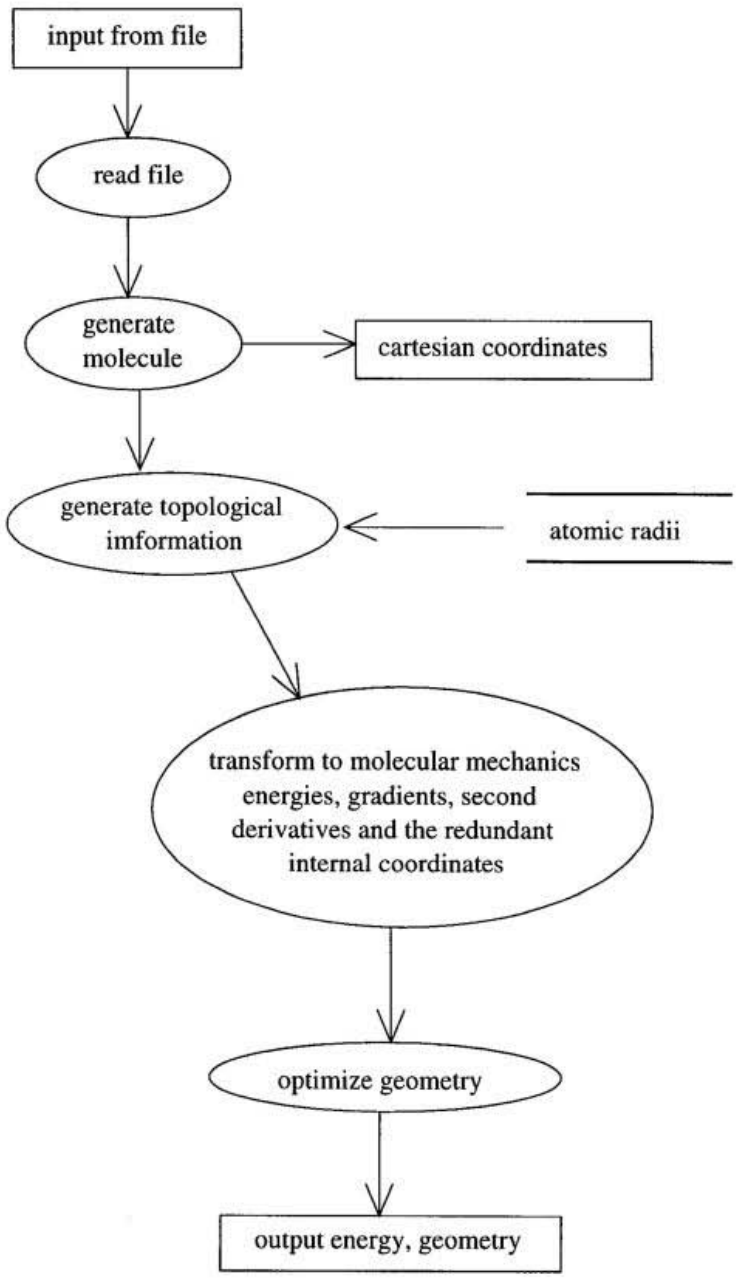

Figure 4.8: Mid-level functional model diagram for the molecular mechanics program.

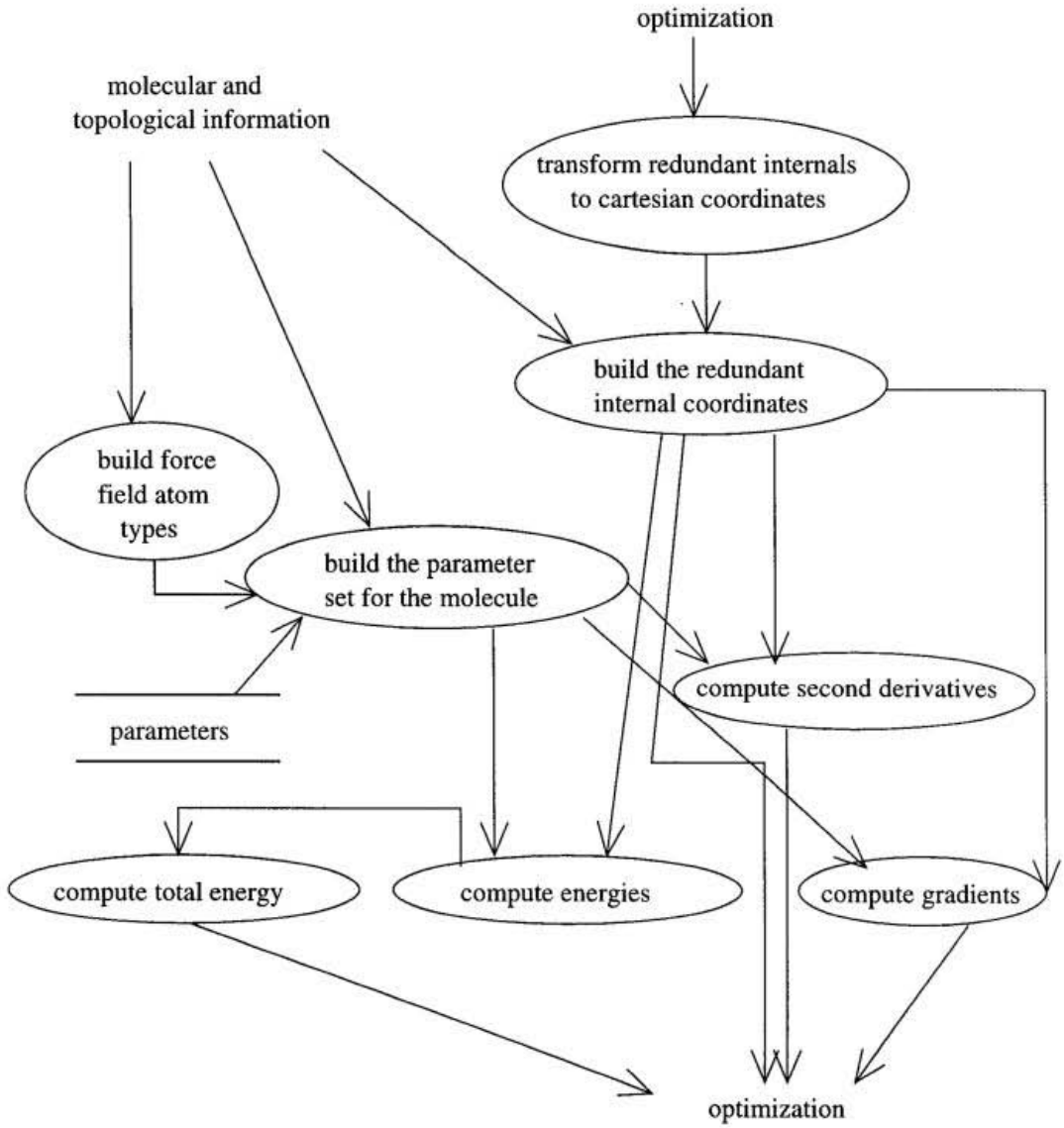

Figure 4.9: Low-level functional model diagram for the molecular mechanics program.

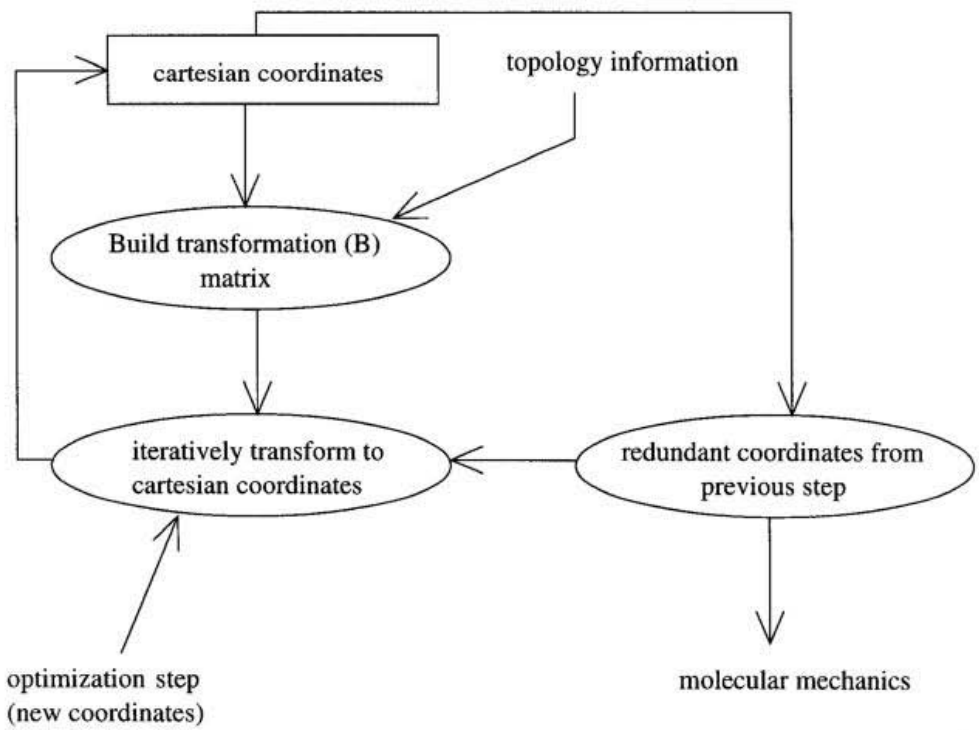

Figure 4.10: Low-level functional model diagram for the conversion of redundant coordinates to cartesians.

compute the energies, gradients, and second derivatives. The force field atom types also obtain the geometry and topology information to generate a list of force field atoms for use in the parameter selection. If an optimization is selected, then at some point the cartesian coordinates may have to be updated, and is represented as a process in figure 4.9.

Figure 4.10 is the lowest level diagram for the data flow through the iterative transformation of redundant internal coordinates to cartesians. In this diagram, the data flow starts with the passing of the cartesians to the process which creates the B matrix. The B matrix is then passed to the iterative coordinate transformation along with the redundant coordinates from both the optimization package and the molecular mechanics code which regenerates the cartesians. This process is repeated iteratively for every optimization step so new coordinate and nonbonded interaction distance values can be recomputed and used in subsequent optimization steps.

4.7.3 Describing Functions

Each process in the data flow diagram can be thought of as a function in which a description using mathematical formulas, natural language, or some other low-level expression should be given. This description should not contain implementation information and should clearly identify the relationships between the function's input and output values.

Functional descriptions are necessary for the complete understanding of any

problem, in this case the molecular mechanics method. The best way to describe a function is by using the description of the transformation it performs. For example, if the function uses mathematical functions, then the functional description should contain these functions. It should also indicate any input and/or output to the function.

compute stretch energies *(coordinates, parameters) − > energy contributions*

for all stretch energies between atoms i and j in a bond, evaluate:

$$V_{ij} = 143.9325\left(\frac{k_{ij}}{2}\right)(r_{ij} - r_{ij}^0)^2 \\ (1 + cs * (r_{ij} - r_{ij}^0) + \left(\frac{7}{12}\right) * cs^2 * (r_{ij} - r_{ij}^0)) \tag{4.1}$$

store energy contributions in energy object attribute CONTRIBUTIONS

number of energies = number of interactions

In the above example, a functional description is given for the computation of the energy contributions for the stretch energy. The formats for the computation of the bend, torsion, out-of-plane bend, van der Waals, electrostatic, and stretch-bend energies are the same, however the energy expressions for each interaction differ. The energy expressions are given in reference [34]. The purpose of this function is to compute the individual energy contributions due to each interaction using the given energy expression. The parameters are input into the equation and the quantities

CONTRIBUTIONS and NENERGIES are computed using the coordinate information which is also passed in. Examples for the remainder of the energy computation functions will not be given here as they are very similar to the above function. Also, the computations of the gradients and second derivatives are similar to the computations of the energies, so the function descriptions will not be covered here.

compute parameter set *(atoms in interaction, parameters) − > parameter set*
for all interactions in the set:
for all atoms involved in an interaction:
determine their atom and interaction type
get the parameters corresponding to that interaction
store the parameters in the object's variables

A description of the method of obtaining parameters of any type is given in the above functional outline. For a particular parameter set the parameters obtained may differ in number and type. For example, the stretch parameters are an equilibrium bond length and force constant, while the torsional parameters are three parameters representing three V parameters in the energy expression. The function obtains as input the atoms involved in the interactions for which the parameters are needed from the topology code as well as the entire parameter set. It outputs the parameter set for the molecule(s) in question. It uses information about the atoms' defined types according to the force field as well as the interaction type, which is usually derived

from the latter information. In the case of bond types, this information is hard-coded, although this will change in future work. The parameters are then selected according to the atom and interaction types (defined by the force field) and this information is stored in the parameter set.

determine atom types *(topology information, molecular information)*

$->$ *atom types*

for all atoms in the molecule:

obtain the atomic number, hybridization, ring it belongs to, and whether it is part of an aromatic system

based on this information and the atom type definitions in the force field, assign an atom type to the atom

A word description is given above for the determination of the force field atom types. These atom types are used to determine subsequent interaction types which are used in parameter selection. As seen above, they are selected based on their atomic number, hybridization, the size of a ring they belong to (zero if they do not belong to a ring), and whether or not they are part of an aromatic system.

compute coordinates *(topology information, cartesian coordinates)* − >
redundant internal coordinates
for all interactions of a particular type:
using vector relationships, the coordinates are computed

The computation of the coordinates of a particular type are computed from the topology information and cartesian coordinates using simple vector relationships. This is noted in the above functional description.

The above is a brief outline of functional descriptions for the processes in the molecular mechanics package. The four of them by no means complete the description and they do not contain any implementation details. This information will be derived in the implementation phase. As a whole, the functions give a good description of the tranformation of object information through the functions that perform these computations.

4.7.4 Identifying Constraints Between Objects

Constraints correspond to object dependencies that are not related to the inputs or outputs, but are related functionally. They can occur between instances of two objects at the same time, at different times, or on two objects at the same time [31].

Constraints do not exist within the molecular mechanics package, however it should be noted that some constraints on the type of calculation and the values of

the energies and gradients are placed either on input or as defaults.

4.7.5 Specifying Optimization Criteria

The values in the functional model which must be optimized should now be determined [31]. The information passed between processes can be optimized by making the information needed by more than one process *global*. If all the processes have access to it, the information need not be passed and multiple copies are not made in memory. However, this should only be done in cases where the data is not sensitive. This should be considered for the case of object data in the molecular mechanics package. In the case of the parameters and coordinates, they are needed by the energy, gradient, and second derivative processes and should therefore kept globally accessible. The data sets are in memory throughout the life of the program and are updated as needed, as in the case of a geometry optimization.

4.8 Combining the Two Models

The two models, now complete for the molecular mechanics problem, give a view of the problem from two perspectives. The object model shows the molecular mechanics method in terms of energies, gradients, second derivatives, force field atoms, parameters, and coordinates. These objects interact through associations due to the need for data. The functional model describes the molecular mechanics method as a set of transformations of the input data through intermediate stages to the output.

This output corresponds to either an energy at the point on the potential energy surface due to the positions of the atoms, or is an optimal geometry with a minimum energy. The objects are the keepers of the data and it is their behaviors which transform it from input to output. The need for data through associations creates the flow of the data seen in the functional model. Combining this information brings the analysis and design phase to a close .

4.9 Summary

Problems can be broken down and the programs designed by structural or object-based analysis. The method chosen is based on the size and type of problem, but for large programs there are more advantages to using object-based analysis and design. Object-based methods also have the features that quality software should exhibit. There are several parts to object-based analysis and design, and each part contributes its own information to the solution and implementation of the problem to form the desired software.

Chapter 5

Implementation of the Molecular Mechanics Method Using Fortran 90

5.1 Introduction

There are many programming languages available for use when designing scientific codes. For many years the language of choice was Fortran 77 due to the support it had for numerical problems. As a result, much of the numerical and scientific code is still only available in Fortran 77.

Object-based programming offers several advantages to the scientific community with respect to code design. Independently executing units of the code are placed in separate modules within the code instead of being spread out over the entire program.

The software is designed in pieces which can *inherit* one another, promoting code reuse. The *objects* within the code are placed together with the routines which define them, forming a new data type. The collective attributes and behaviors of the objects model the objects in the real world problem. Similar operations can also be grouped together into one routine which executes in a similar fashion depending on the type of data it is acting on, preventing multiple copies of the same utility [35].

There are several different languages available for developing object-based code and each one has its own unique features which make it attractive for code development. However, the type of application will dictate which code is ultimately going to be used. It still seems the language of choice for scientific code is Fortran, and the emergence of Fortran 90 has proved beneficial as it has incorporated some additional features for use in object-based design. In the next section, some of these features will be highlighted.

Throughout this Chapter many key features of Fortran 90 for object-based design and their application to the implementation of the molecular mechanics method will be discussed. The molecular mechanics method was covered in detail in Chapter 2.

5.2 Fortran 90 Versus Fortran 77: Some Added Features

There are still some advantages to using Fortran 77 for some scientific code, but there were several key features added when the Fortran 90 standard was developed. Some of these include dynamic memory allocation, the module, pointers, and user-defined types.

5.2.1 Dynamic Memory Allocation

Dynamic memory allocation involves defining, changing and removing memory throughout the life of the program. There are a few benefits to dynamic memory allocation and in some cases it is essential to insure proper behavior of the program. It is most beneficial when the size of the data is not known until the program is executed [36].

In Fortran 90, before an array can be dynamically allocated with a given dimension, it must be tagged as an allocatable array. This is done by giving it the ALLOCATABLE attribute. Arrays with this attribute can be defined at the beginning of a piece of code and allocated once the size is known. When an array is declared with the ALLOCATABLE attribute, its shape must be stated by using a colon to represent a particular dimension. The dimensions are separated by commas. In the molecular mechanics code, many different arrays are declared as allocatable. The most common allocatable arrays are those which are used in a local piece of code

to temporarily store data as they can be allocated, used, and destroyed once they are no longer needed. This helps conserve memory.

It should be noted that an allocated array cannot be allocated again unless it is deallocated first, and the status of an allocatable array can be checked by using the ALLOCATED intrinsic function. The result of this function is a logical variable which is set to true if the array has been previously allocated [36].

An allocatable array can be declared , checked to determine if it was previously allocated elsewhere, and allocated in the following manner:

```
integer, dimension(:), allocatable :: Bond_Type
...
if(.not.allocated(Bond_Type).and.NGBONDS.gt.0) then
  allocate(Bond_Type(NGBONDS))
end if
...
```

In the first line of this example, the array Bond_Type was declared to be an array of variable dimension using the ALLOCATABLE attribute. Once the size (given by NGBONDS) is determined, it can be allocated using the intrinsic function ALLOCATE and giving as arguments the array name and its size.

Once an array is no longer needed, the memory used by an array can be released by deallocating it. For example, if the array in the above code fragment was to be deallocated, this would be done by using the DEALLOCATE intrinsic and giving the

array name as the argument. Use of the deallocate intrinsic function is shown in the following code fragment [36]:

```
...
deallocate(Bond_Type)
...
```

5.2.2 Pointers

Pointers are useful when data must be initialized and used dynamically. This is especially true for very large arrays which if copied could use up unnecessary quantities of space in memory. Arrays in Fortran 90 are created as variables, and as yet it is not possible to create arrays of pointers directly [36, 37].

A pointer is first declared by giving the pointer variable the POINTER attribute. It can then be given memory by allocating it in the same manner as for allocatable arrays [36].

Like an allocatable array, a pointer cannot be associated with (or point to) a part of memory more than once, so it is first tested to check if it is associated with any memory using the ASSOCIATED intrinsic and giving the pointer name as the argument. This intrinsic function returns true if the pointer is associated to memory. Once this test is done, the pointer can be associated to some memory by allocating the memory using the ALLOCATE intrinsic and passing in the pointer name and exact dimension of memory as inputs. The pointer can then be deallocated later

using the DEALLOCATE intrinsic function (not shown in the above example).

A pointer can be in one of three states depending on whether it is defined or associated with any memory. Its three states are defined as follows: it can be defined and associated, undefined and unassociated, or defined and unassociated [36].

A pointer itself can be viewed as an object whose data is hidden from the user's view. The intrinsic functions NULLIFY, ALLOCATE, ASSOCIATE, and DEALLOCATE are used to modify or retrieve this information.

5.2.3 The Module

The module has a variety of uses, but perhaps its most powerful use is to group together similar data and functions into a container which can then be made available to the rest of the code by *using* it. Some other uses include storage of global data and grouping user-defined types (discussed in the last part of this section). The code of the module is stored in a separate file and must be compiled before any code, outside the module, that uses it. More than one module may be contained within a file and this file can be used to group common modules [36, 37].

All of the information to be included in a module is contained between the MODULE and END MODULE lines. Within these lines, the data can be separated from the functions by first defining data, using the CONTAINS statement to separate the data from the functions to follow, and finally defining the functions (or subroutines). This module is then included elsewhere in the code by adding a USE statement to

the parts which require it. The following code fragment is an example of a module which would be in its own file and is part of the molecular mechanics package [36]:

```
MODULE pairlist


* Modules needed...
USE molecule !molecular information
USE constants !global constants, conversion factors
USE topology !topology objects


implicit none      !everything must be declared...


* Data in this module:
integer :: NELEC !number of nonbonded pairs
TYPE pair         !Derived type
  integer :: I !atom i of nonbonded pair
  integer :: J !atom j of nonbonded pair
  integer :: is_vdw !is there a van der Waals contribution?
  integer :: is_elec !is there an electrostatic contribution?
  integer :: is_tors !are i and j separated by three bonds?
END TYPE pair
* nonbonded connectivity matrix
```

```
logical, dimension(:,:), allocatable :: CONN_NB
* pairlist
type (pair), dimension(:), allocatable :: pair_IJ

CONTAINS          !Functions go here...
SUBROUTINE CONN_NBOND
*** initialize nonbonded connectivity here ***
SUBROUTINE MAKE_ELEC
*** initialize pairlist here ***
END MODULE pairlist
```

In the above example, the module is a collection of pairlist data used in the molecular mechanics package. The first part of the module gives any USE statements which indicate other modules needed by this one. The next part in the module is used for declaring variables for the pairlist. Following the CONTAINS statement are any subroutines which are used to initialize data within this module only. Only including the subroutines which initialize this module's data, encapsulates the pairlist into this module. The link between the module and object-based design will be discussed later in this Chapter.

Data and subprograms within a module may be kept private to the module by using the attribute PRIVATE in their definition. If data with this attribute is needed or must be changed outside this module, functions to pass the data and receive the

change must be incorporated as the private data cannot be accessed by anything outside the module, even if it is USEd [35].

5.2.4 Derived Types

User-defined (or derived) types are an addition to Fortran to allow a user to group data together into a new type. This new type has the same properties as an intrinsic type (a name, set of values and operations, and a way to specify constants) and can be used in the same manner as intrinsic types. Derived types are a convenient way of grouping data which is used to describe the same concept or is passed around a program as a group. It is also the only way to define an array of pointers [35, 36].

Derived types are declared by first stating the name using TYPE *name of type*, defining the type's variables, and ending with END TYPE *name of type*. They can then be used anywhere in the program in a similar fashion to intrinsic types [36]. An example of a declaration followed by use of a derived type is given below:

```
* Declaration of Derived Type:
TYPE energy_component
  integer :: NENERGIES !number of energy components
  double precision :: SUM_COMPONENTS !total energy contribution
* Individual energy contributions
  double precision, pointer :: COMPONENTS(:)
END TYPE energy_component
```

```
* Use of Derived Type:
type(energy_component) :: E_STRETCH
type(energy_component) :: E_BEND
type(energy_component) :: E_TORSION
type(energy_component) :: E_OOPBEND
type(energy_component) :: E_VANDERWAALS
type(energy_component) :: E_ELECTROSTATIC
type(energy_component) :: E_STRETCH_BEND
type(energy_component) :: E_TOTAL
```

In the above example, a derived type is used to define an energy component of the total molecular mechanics energy. This energy component contains an integer for the number of energy subcomponents, a double precision variable for the sum of these subcomponents, and a pointer to an array of subcomponents. This type is then used to declare specific instances of this derived type.

One limitation inherent in the derived type is allocatable arrays cannot be declared as part of a derived type [36]. In other words, the following code fragment is illegal:

```
TYPE sd_component
  integer :: N2DERIVATIVES !number of second derivatives
  double precision, dimension(:,:),
```

```
.        allocatable :: COMPONENTS !individual second derivatives
END TYPE sd_component
```

In the above example, the array COMPONENTS was declared in the derived type as allocatable, which is illegal. Instead, the allocatable array can be replaced by a pointer which can be allocated later on in the program:

```
TYPE sd_component
  integer :: N2DERIVATIVES !number of second derivatives
  double precision, pointer :: COMPONENTS(:,:) !individual second derivatives
END TYPE sd_component
...
if(.not.associated(SD_STRETCH%COMPONENTS)) then
  allocate(SD_STRETCH%COMPONENTS(DIM1, DIM2))
end if
...
```

5.2.5 Additional Computational Intrinsic Functions

One additional feature of Fortran 90 that makes it appealing in the design of scientific applications is the addition of matrix operations to the standard. Now, matrices can be added, subtracted, and multiplied provided their dimensions match. Addition and subtraction are both provided by overloading of the '+' and '-' operators and the multiplication of matrices is provided by the intrinsic function MATMUL.

Another very useful function to scientific computation is the dot product intrinsic function DOT_PRODUCT.

```
...
Gmatrix = matmul(Bmatrix, transpose(Bmatrix))
...
```

The code fragment above (from the coordinate conversion part of the molecular mechanics package) shows an example of the use of the MATMUL intrinsic. Two matrices can be multiplied by indicating their corresponding array names as arguments to the MATMUL intrinsic function and the result can be stored in a third array. One problem with the MATMUL function is the programmer must insure the array dimensions of the arrays being multiplied match for matrix multiplication, and the resulting array must have the required dimensions. For example, if the two arrays to be multiplied are n x m and m x n, they are multiplied in this order and the resulting matrix must have dimensions n x n. The size of an array can be determined by using the SIZE intrinsic function, which takes the array name as input and returns its size.

5.3 Object-Based Programming for Molecular Mechanics Using Fortran 90

Object-based design has many benefits which can be taken advantage of in designing scientific code, however it would be advantageous to be able to create code

using these concepts while still being able to combine the new code with existing Fortran 77 routines. This is possible with the addition of some of the features discussed in the above section to the standard to create Fortran 90. Some of these include the use of modules and derived types to achieve abstraction and encapsulation, and interfaces to achieve polymorphism. In this way, newer scientific code can be written in Fortran 90 which is backward compatible with Fortran 77 so the object-based code can be written effectively without having to rewrite existing numerical routines [35, 38].

In a previous version of Mungauss, written in Fortran 77, a low-level object-based approach was used (OSIPE [39]), which made it easier to convert to Fortran 90. As a result, many of the old features of Fortran 77 have been phased out over the past year.

5.3.1 Achieving Data Abstraction, Encapsulation, and Modularity With Derived Types and Modules

Classes, as discussed in Chapter 3, have a particular structure to their attributes and behaviors. Groups of objects which share in this structure can collectively belong to a class, which describes the objects' attributes and behaviors in a more general sense. The class can then be placed into a single structure in the code. This provides modularity to the program as well as encapsulating the details of the class' implementation [40].

The above can be easily implemented using modules and derived types. The

benefit of the module is that it provides a neat *container* to place the abstractions and behaviors of a group of objects into. The abstractions can then be placed in a new derived type which can be used to create instances of itself, the objects. In doing so, each object has the attributes defined in the derived type and access to the behaviors contained in the module [35, 38, 40]. It also provides some security by only allowing parts of the code who USE the module to access the data and adding the PRIVATE attribute to the more sensitive data of the object prevents any part of the code not belonging exclusively to the object from directly accessing it.

As an example, consider the molecular mechanics energy. It contains seven parts, each of which has a set of components as well as a total number and sum of these components. A class can be defined for the energy pieces and these three abstractions can be placed into a derived type for a generic energy type. These pieces can then be initialized and summed to give a total energy, which is also an energy type. The module for the energy class, with the objects included, is given below:

```
MODULE mm_gradients
* Modules:
USE functions_mm_gradients !first derivative functions
USE constants !global constants


implicit none
```

```
* Class data (each object has these...):
TYPE gradient_contribution
  integer :: NGRADIENTS  !number of gradient contributions
  double precision, pointer :: COMPONENTS(:) !array of gradients
END TYPE gradient_contribution


* Objects:
type(gradient_contribution) :: STRETCH
type(gradient_contribution) :: G_BEND
type(gradient_contribution) :: G_TORSION
type(gradient_contribution) :: G_OOPBEND
type(gradient_contribution) :: G_VANDERWAALS
type(gradient_contribution) :: G_ELECTROSTATIC
...
CONTAINS
SUBROUTINE GRADCLC_STR !initializes stretch gradients
...
SUBROUTINE GRADCLC_BND !initializes bend gradients
...
SUBROUTINE GRADCLC_TOR !initializes torsion gradients
...
SUBROUTINE GRADCLC_OBND !initializes out-of-plane bend gradients
```

```
...
SUBROUTINE GRADCLC_VDW !initializes van der Waals gradients
...
SUBROUTINE GRADCLC_ELEC !initializes electrostatic gradients
....
END MODULE mm_gradients
```

In the above example, the attributes for the number of gradient contributions, and the gradients are given in the derived type gradient_component. The contributions are the individual first derivatives of the energy expression for each interaction.

The gradient objects for the gradient class, seen above, are then defined by declaring each of them as variables of the type gradient_component and initializing them. The initialization is done for each object individually as the gradient expressions for each interaction differ. There is one routine which is called to build the list of them in the cases where they are all needed. This routine is contained in a file for access by all parts of the code.

The class for second derivatives is the same as above, with the addition of the stretch-bends as a separate object (off-diagonal terms for stretches and bends). The attributes are only a list of contributions and the number of gradients in each object.

```
MODULE atom

* Communication...
```

```
USE molecule !molecule information

USE constants !global constants

USE objects_created

USE topology !topology objects

USE mm_interaction_type_params !parameters global to MM


* Class data definition:

TYPE atomtype

* atomic number of the atom

  integer :: atomic_number

* atom's hybridization

  integer :: hybridization

* size of ring atom belongs to

  integer :: ring_size

* atom's "type" defined by the

*   force field

  integer :: type_mm

* Is the atom aromatic?

  logical :: is_aromatic

END TYPE atomtype


* Object definition:
```

```
type(atomtype), dimension(:),
.   allocatable :: ATOMS
CONTAINS
  SUBROUTINE ATOM_TYPE
*****initialize ATOMS here*****
  END SUBROUTINE ATOM_TYPE
END MODULE atom
```

The above example shows an implementation of the atom class. The first line is the beginning of the module and the part following is the definitions of communications this module has with the ones listed. For example, the atom class requests information from the molecule and topology classes, as well as the data stores constants and interaction type parameters (bond types). Objects_created is a set of tools which allow dependency of objects to be defined. The type definition groups the class data into a user-defined type and the methods are separated from this type by the CONTAINS statement. There is only one behavior in this class, which is the initialization, defined by the routine ATOM_TYPE. The objects are defined by allocating them as deferred-shape arrays, specifying the shape (one, two , or higher dimensions), and later allocating them with the correct dimension.

5.3.2 The Implementation of the Functional Model Components with Subroutines and Modules

In the functional model, the processes most often correspond to behaviors of objects or some intermediate transformation process. These are implemented in the molecular mechanics package using SUBROUTINEs. For a further description of the SUBROUTINE, see reference [36]. If the process describes an object's behavior, it is grouped with the class in the module corresponding to that class and its objects. If the process is executed using more than one object of differing classes, this process is placed in a file for use by all parts of the program. Data to be passed into or out of the process appears in the argument list for the SUBROUTINE. Once the data is passed into the process, it is declared locally and transformed.

Actors, the sources and sinks for the data in the program, are usually objects themselves. In the case of input and output, they are actual files. The Cartesian coordinate actor is an object which is included in the coordinate conversion process, but will not be discussed further as it does not correspond to the molecular mechanics problem. It is included because it is an intermediate result needed to compute the nonbonded energies in the molecular mechanics package. It is located in the module *molecule*.

Data stores in the molecular mechanics are an example of another use for a module. In the molecular mechanics package, the data store representing the entire set of parameters is implemented in modules according to the type of parameter data

it stores. The data store is actually a group of modules, each containing a complete parameter set for a particular interaction. For example, the parameter set data store includes the modules for the stretch, bend, torsion, out-of-plane, nonbonded, and stretch-bend parameter sets.

Data flows are implemented as USE statements or function calls. In the case where the information is in a module and one part of the code needs the data from this module, the module is first USEd, then the initialization routine to compute this data is called. Once the data is available for use in the module, any part of the code with a USE statement for the particular module where the data is stored may use this data. For example, consider the following code fragment:

```
SUBROUTINE UPDATE_XYZC (q_new, q_old, Bmatrix)
* Modules:
      ...
USE redundant_coordinates
USE molecule
...
CALL BLD_COORDINATES
...
deltaq = q_new - q_old
...
END SUBROUTINE UPDATE_XYZC
```

In the above example from the redundant internal to Cartesian coordinate conversion, the process UPDATE_XYZC is defined as a subroutine which has the redundant coordinates from the current and previous steps in the optimization (q_new and q_old, respectively) as well as the transformation matrix (Bmatrix) given as inputs. The output is the updated Cartesian coordinate set, which is automatically updated by making it available in the module *molecule*. The parts of q_old are in the module *redundant_coordinates* and need to be updated once the new Cartesians are computed. This is done by first including the module with the parts of q_old already available for modification and modifying them by calling the routine to build the redundant coordinate objects (called BLD_COORDINATES).

5.4 Summary

Fortran 90 is proving to be a viable option for designing scientific code where the use of the Fortran language is still preferred over other popular languages. Fortran 90 has support for several object-based features, for example the module and derived type, which enable programmers to create classes and objects from real-world models. This provides abstraction and encapsulation to a program written in Fortran 90.

Some advantages and disadvantages to the use of Fortran 90 for scientific code are worth noting. First, Fortran 90 provides full support for the use of classes and objects by providing the module and derived type to define classes and data abstractions from which to create objects. This also provides encapsulation. These features are

also available even if the goal of the programmer is only to create modular and not object-based code. Interfaces are also available to provide support for polymorphism and operator overloading, promoting code reuse. However, although the support is available for creating classes and objects, the generic methods to construct and destroy objects are not available in the Fortran 90 language. The programmer must therefore explicitly create these for every class individually! There is also no explicit support for inheritance as in other object-based languages. Because of this last point, Fortran 90 is often referred to as an object-based language because support for the four main parts to the object model (discussed in Chapter 4) are not all available in the standard [35, 36, 37, 38, 40, 41].

Some additional features of Fortran 90, such as backward compatibility with Fortran 77 and explicit support for some matrix and vector operations within the standard combined with the features for use in object-based design make this language a choice for scientific code design.

Chapter 6

Performance of the Molecular Mechanics Package: Numerical Results

6.1 Introduction

The execution of the molecular mechanics package depends on the desired quantity. In some cases an energy of a given structure is needed without modification of the input and in other cases and optimal geometry is sought. These types of computations are called *single-points* and *geometry optimizations*. However, in order to satisfy the quality of controlled printing one must also be able to get the results of the computation of one or more of the objects. This chapter will cover the results of printing the objects, single-point, and geometry optimization computations.

Some comparisons to literature results for differences in energy between two differrent structures of the same molecule (*conformational energy differences*) will also be given.

6.2 Printing the Objects

This section deals with the printing of the objects. The molecular mechanics package is capable, through the menu, of printing the energy, gradient, second derivative, parameter, and coordinate contributions. The objects derived from these classes are all printed, and at this point an individual object cannot be printed. For example, if the user specifies printing of the energy objects, all energy objects are printed as a set of contributions. Printing of data can also be accomplished using local debugging tools which print intermediate results from the routine (or routines) specified in the input file.

```
Molecule
  FreeFormatMatrix
  C1
  H1  C1  HC
  H2  C1  HC  H1  HCH
  H3  C1  HC  H1  HCH  H2  HCHH
  H4  C1  HC  H1  HCH  H2  -HCHH
  end !FreeFormatMatrix
  define
```

```
  HC = 1.0931
  HCH = TETRA
  HCHH = 120.0000
  end !Define
end !molecule
Output
  Object=ENERGY_MM:ENERGY_CONTRIBUTIONS
end
stop
```

The above code segment shows an example of an input file where certain sets of objects are requested. The molecule is input as a Z-matrix (see appendix B). The objects are requested by enclosing the list of objects in an output section using the *Output* menu command and the objects are specified using an *Object=* command. The first part of the object name is the class it belongs to and the second part is the set of objects to be computed and printed. For example, the first set of objects requested are the energy contributions, and they belong to the class ENERGY_MM.

The result of running the above input file is as follows:

```
Welcome to Mungauss - Development version (June 7, 2000)


Free format Z-Matrix for: UNKNOWN
C1
```

```
H1        C1        HC
H2        C1        HC        H1        HCH
H3        C1        HC        H1        HCH       H2        HCHH
H4        C1        HC        H1        HCH       H2       -HCHH

VARIABLES:
HC        = 1.09310000 HCH       =  109.47122 HCHH      =  120.00000
...
BOND_ATOMS
  1     1     2
  2     1     3
  3     1     4
  4     1     5
ANGLE_ATOMS
  1     2     1     3
  2     2     1     4
  3     2     1     5
  4     3     1     4
  5     3     1     5
  6     4     1     5
Pruning graph...
GRAPH_HOMREDUCED> NPRUNE:            0
```

```
Homeomorphically reducing graph...
# Fundamental Rings =            0
total energy:   4.382124831520715E-005
Energy:
Number of stretch contributions =            4
Total stretch energy =   2.183551411268918E-008
Stretch contributions:   5.458878528184422E-009
5.458878528184422E-009   5.458878528160169E-009
5.458878528160169E-009
Number of bend contributions =            6
Total bend energy =   4.344867295603311E-005
Bend contributions:   7.241445492672186E-006
7.241445492672510E-006 7.241445492672510E-006
7.241445492669923E-006  7.241445492669923E-006
7.241445492676065E-006
Number of stretch-bend contributions =            6
Total stretch-bend energy =   3.507398450613559E-007
Stretch-bend contributions:   5.845664084362424E-008
5.845664084356063E-008 5.845664084356063E-008
5.845664084355016E-008  5.845664084355016E-008
5.845664084351007E-008
```

```
         action> end of inputs

...


Job :RUN      ended on :14-Aug-00 at 00:22:43

user: dshaw    on 14-Aug-00 at 00:22:43

Cpu     time:00h00m00s14c on garfield

Elapsed time:00h00m00s00c
```

In the above output file, the input and output Z-matrices are given as well as the printed set of energy objects. Also given, but not shown in the above example, are the cartesian coordinates in bohr and angstroms as well as the connectivity matrix showing how the atoms are connected.

Often the energy at a particular geometry or the geometry at a minimum energy is sought. These two types of computations and the results for a set of molecules will be discussed in the next two sections.

6.3 Single-Point Energies

A single-point energy calculation is the computation of the energy of a molecule at a particular geometry. In the case of molecular mechanics, these energies are often useful to determine if the output of a molecular mechanics package is reasonable, however they have no other practical use. The energy of a molecule computed with different force fields cannot be compared as the parameters and energy expressions

used in each force field are different from those of other force fields. As a result, the single-point energies from this molecular mechanics package are being used to determine if the molecular mechanics package can give results which are not unreasonably large or small.

Table 6.1: Single point energies for a series of hydrocarbon molecules.

Molecule	**Single Point Energy (x10^3 Hartrees)**
methane	0.00438212
staggered ethane	-5.94363
eclipsed ethane	-1.86879
trans-propane	-6.63891
cis-butane	619.999
gauche-butane	0.408991
trans-butane	-5.45631
trans-pentane	-6.68317
trans-hexane	-6.59755
t-butane	26.9702
cyclopropane	55.2188
cyclobutane	16.8973
methylcyclopropane	100.710

continued from previous page

Molecule	**Single Point Energy (x10^3 Hartrees)**
boat cyclohexane	7.14584
chair cyclohexane	1.83973
benzene	26.0829
ethene	13.4039
propene	10.8389
cyclopropene	67.6849
cyclobutene	94.2949
trans-butadiene	13.0006
gauche-butadiene	59.1122
cyclobutadiene	182.218
cyclopentadiene	35.9250

Table 6.1 gives the single-point energies for a series of hydrocarbon molecules covering cyclic and acyclic systems with single, aromatic, and double carbon-carbon bonds. It should be emphasized at this point that these values of the energy are for molecules with particular input structures. Changing any of the input coordinate values will change the energies. These energies correspond to a point on the potential energy surface for the molecule with the given geometry, and are not necessarily optimal structures. However, some trends in the energies can be observed and these

also aid in deciding if the MM energies are reasonable.

Referring again to table 6.1, the energy differences for staggered and eclipsed ethane, chair and boat cyclohexane, and gauche and trans butane are particularly useful in investigating conformational energy differences. In the ethane system, staggered should be lower in energy than eclipsed. In the butane system, trans-butane should be lower in energy than gauche-butane. In the cyclohexane system, the chair form should be lower in energy than the boat form. The energy values for these geometries, shown in the above table for ethane, butane, and cyclohexane, demonstrate these trends, and as a result, it can be concluded that the molecular mechanics package is able to give reasonable single-point energies.

6.4 Geometry Optimizations

The test of a molecular mechanics package is not only to generate single-point energies but also to generate reproducible results. Since direct energy comparisons of systems such as methane and the staggered and eclipsed forms of ethane are not possible, comparisons are often made of conformational differences. The energy values for two different sets of positions of a molecule's atoms in space may not be the same between different force fields, but the difference in these two energies will be comparable. For this reason, comparison of results of the molecular mechanics package to literature values will be done on the basis of existing results for energy differences in a molecule's conformations.

In order to obtain the best geometry and energy for conformational energy comparison, it is useful to run a geometry optimization on the input structure. This involves minimizing the energy of the molecule, thus optimizing its geometry. A geometry optimization is started by including the following in the list of menu commands in the input file: the option for geometry optimization, the method desired, the function to be minimized (in this case, molecular mechanics), the maximum number of iterations, and the desired accuracy. At each iteration, the results are then printed for the coordinates, gradients, total energy, and gradient length.

Table 6.2: Table of geometry optimization results for a series of hydrocarbons.

Molecule Name	**Function Evaluations**	**Energy (x10^{-3} Hartrees)**	**Gradient Hartrees/Bohr)**
methane	4	0.000000	2.366811x10^{-17}
staggered ethane	9	-16.1663	8.952353x10^{-17}
eclipsed ethane	9	-10.7367	4.586337x10^{-19}
trans-propane	11	-23.2853	3.988721x10^{-16}
cis-butane	18	-23.5900	7.999749x10^{-15}
gauche-butane	17	-31.3323	3.651790x10^{-14}
trans-butane	18	-32.5909	3.297194x10^{-13}
t-butane	13	-9.51801	8.573369x10^{-17}

continued from previous page

Molecule Name	Function Evaluations	Energy ($x10^{-3}$ Hartrees)	Gradient Hartrees/Bohr)
trans-pentane	80	-22.1729	$2.979742x10^{-4}$
trans-hexane	80	-19.2031	$4.988806x10^{-4}$
cyclopropane	79	19.9713	$2.911428x10^{-6}$
methylcyclopropane	80	21.9774	$1.799471x10^{-4}$
cyclobutane	26	-2.27935	$1.865883x10^{-16}$
boat cyclohexane	56	-25.0396	$4.121210x10^{-5}$
chair cyclohexane	80	-33.7496	$3.419738x10^{-5}$
benzene	80	-3.27697	$1.516619x10^{-4}$
ethene	8	11.7355	$1.095284x10^{-14}$
propene	10	5.73116	$6.823548x10^{-18}$
cyclopropene	19	6.42416	$2.456490x10^{-17}$
cyclobutene	79	43.9365	$3.189709x10^{-4}$
gauche-butadiene	20	11.9478	$1.616714x10^{-16}$
trans-butadiene	19	10.4657	$2.752949x10^{-14}$
cyclobutadiene	9	16.3081	$2.527553x10^{-15}$
cyclopentadiene	10	-2.91336	$3.873846x10^{-13}$

Table 6.2 gives the geometry optimization results for the molecule set given in

the previous section. The first column of the table gives the name of the molecule, the second column gives the number of function evaluations when either the energy did not change or the maximum number of iterations was reached. For all of the above cases, the maximum number of iterations was set to 40. The third column gives the energies at the given step (from column 2) and the fourth column gives the gradients. The method used in most cases was full Newton-Raphson, but for trans-pentane, trans-hexane, cyclopropane, methylcyclopropane, boat cyclohexane, chair cyclohexane, benzene, and cyclobutene the method used was VA05AD (minimization of the sum of squares method, section 3.3.4). In the cases where Newton-Raphson was used, it seemed to give reasonable results while recalculating the exact second derivative matrix at each step. In the cases where VA05AD was used, problems with convergence were encountered and the results where little improvement in the energy was seen were taken.

Comparison of the energies in tables 6.1 and 6.2 show in nearly all cases the energy decreases and a more optimal geometry is found. Therefore, movement from the initial to a more optimal geometry is possible with the molecular mechanics package using VA05AD.

True comparison of the molecular mechanics package can be done by comparing the results of conformational energy differences using the optimized geometries with those in the literature. Two systems were chosen to demonstrate the molecular mechanics package ability to reproduce trends and energy differences. These two systems are staggered-eclipsed ethane and boat-chair cyclohexane.

Table 6.3: Table of conformational energy differences for the ethane and cyclohexane systems.

Conformer System	**Energy Difference (kCal/mol)**	**Literature Values (kCal/mol)**
eclipsed-staggered ethane	3.28	2.83, 3.02
boat-chair cyclohexane	5.47	5.93
gauche-trans butane	0.79	0.78

Table 6.3 gives the conformational energy differences between eclipsed and staggered ethane, boat and chair cyclohexane, and gauche and trans butane. The literature values are also reported, where the first set of literature values correspond to scaled and not scaled 1-4 nonbonded interactions with the AMBER force field. The results for the AMBER force field were reported as none were found for this particular molecule with MMFF94. As the table shows, the energy difference for gauche and trans butane found in this study is nearly identical to the value reported for the MMFF94 force field. For boat and chair cyclohexane, the energy difference given is not as close to the value given for MMFF94, but it should be noted that the experimental value for this energy difference is 5.5 kCal/mol. This is almost identical to the result obtained for this system in this study. For staggered and eclipsed ethane, the results are not as good, with energy differences for this work different than the

non-scaled energy (from AMBER) by about eight percent.

Overall, the trends in the conformational energy differences are well reported by the molecular mechanics package and the energy differences computed are acceptable[1]. The results need to be improved and two possible ways of accomplishing improvement would be to adjust the guess of the initial second derivative matrix to speed convergence or use a different input which may put the structure and energy closer to another local minimum.

6.5 Summary

The results of this chapter show the ability of the molecular mechanics package to generate single point energies and optimal geometries. The ability of the menu to give the option of printing desired objects was also shown. The results are acceptable, but still show the need for some improvements on the existing molecular mechanics package, especially for optimized geometries.

[1] Acceptable results are those which lie within ten percent of the literature values

Chapter 7

Summary and Conclusions

7.1 Summary

The molecular mechanics method provides an alternative way from the *ab initio* and semiempirical methods to compute molecular geometries and energies. For some systems too large to generate the geometries and energies in a reasonable time with the latter two methods, molecular mechanics is the best alternative.

Molecular mechanics does not only involve the energy computation. In order to obtain the energy as well as an optimized geometry, the positions of the atoms must be determined and the coordinates built. The atomic positions and connectivity can be determined from graph theory concepts and the coordinates can be built from cartesians. The optimization can be done using derivative information as well as the energy and coordinates, and there are several methods available to generate optimized structures.

When designing a molecular mechanics package, consideration into the design method is important. In this work, an object-based design approach was taken with the emphasis on designing a package using logical collections of data with the procedures which initialize and build them. Also emphasized was the way in which these data types interact. The goal of this work was to create a reliable, efficient, dynamically executing, modular, and easily maintained molecular mechanics package. The concepts from object-based analysis and design were used to aid in the visualization of the problem and design of a molecular mechanics package with the above features.

The language of choice for this work was Fortran 90, as it provided many features for the design of an object-based package as well as being backward compatible with Fortran 77. By making use of these two features of the language, the molecular mechanics package could use some existing tools and be integrated into Mungauss, a larger *ab initio* package.

7.2 Conclusions

The resulting molecular mechanics package contains the features of good programs, discussed in the above section. It also has printing capabilities enabling the user to print, in most cases, the desired information without extra data being printed. The integration of the existing Fortran code from Mungauss with the molecular mechanics code was straightforward and the implementation of the molecular mechanics

package using the features of Fortran 90 discussed in chapter 4 resulted in an object-based package.

Comparison of the results of both single-point and geometry optimization calculations shows the ability of the molecular mechanics package to reproduce energies which are reasonable, as well as energy differences between different orientations of the atoms of the same molecule (conformations). However, some improvements can be made on these results in order to obtain better agreement with the literature.

Although most of the goals of this project were reached, some were not fulfilled and are left as future work on the molecular mechanics package. First, the package is not flexible with respect to different groups of molecules, as it is only able to handle systems containing carbon and hydrogen. Since the Merck Molecular Force Field [34] was designed for a wide variety of organic and biomolecules, it contains more than these two atoms, and addition of other atoms is necessary and will be done in the future. Also, integration of force fields to handle inorganic and solvated systems is also a desired feature which will be completed in future work. It is also hoped to combine the molecular mechanics package with the genetic algorithm code as well as modify the existing code to handle transition state systems.

Appendix A

Brief Description of the Merck Molecular Force Field, MMFF94

This appendix gives a brief description of the form of the Merck Molecular Force Field used in this work. It briefly discusses what the force field is designed for, the atom types used, the source of data for parameterization, and the energy expression. Notation within equations is kept consistent with bonds being represented by r, angles by θ, torsions by ϕ, out-of-plane bends by χ, internuclear separations by R, equilibrium values by _0 or _eq and computed values by *. An example of a computed value would be the minimum separation used in the van der Waals interactions, shown as R^*.

A.1 *MMFF94:*

The MMFF94 force field, which was the force field used in our study, was designed to give efficient biomolecular structures mainly for the purpose of drug design. However, it is also designed to handle most organic systems. The force field does not use extended atom types and currently there are 99 parameters available. These cover the organic systems as well as the following metals: iron, lithium, sodium, potassium, zinc, calcium, copper, and magnesium. The parameters were derived by fitting the results of the force field to both *ab initio* and experimental data. The form of the energy expression is as follows [34, 42, 43, 44, 45, 46, 47, 48]:

$$
\begin{aligned}
E_{total} = & \sum_{stretches} 143.9325\left(\frac{K_r}{2}\right)(r-r_0)^2(1+cs(r-r_0)+\frac{7}{12}cs^2(r-r_0)^2) \\
& + \sum_{bends} 0.043844\left(\frac{K_\theta}{2}\right)(\theta-\theta_0)^2(1+cb(\theta-\theta_0)) \\
& + \sum_{tosions} \frac{1}{2}[V_1(1+\cos\phi)+V_2(1-\cos 2\phi)+V_3(1+\cos 3\phi)] \\
& + \sum_{out-of-planebends} 0.043844\left(\frac{K_\chi}{2}\right)\chi^2 + \sum_{electrostatics} 332.0716\frac{q_1q_2}{D(R+\delta)^n} \\
& + \sum_{van\ der\ Waals} \epsilon\left(\frac{1.07R^*}{(R+0.07R^*)}\right)^7\left(\frac{1.12R^*}{(R^7+0.12R^{*7})}-2\right) \\
& + \sum_{stretch-bends} 2.51210(K_{r\theta}(r-r_0)+K_{r'\theta}(r'-r'_0))(\theta-\theta_0)
\end{aligned}
\tag{A.1}
$$

The energy expression above is a sum of contributions due to stretch, bend, torsion, out-of-plane bend, van der Waals, electrostatic, and stretch-bend interactions. The constants included in some of the terms are to insure the energy is in units of kcal/mol [34, 42, 43, 44, 45, 46, 47, 48].

Appendix B

User/Programmer Guide

B.1 Introduction

This guide describes the necessary background into the molecular mechanics package to add to, update, and run the code. It begins with running the program (within the Mungauss suite of programs), gives an outline on how the source code is structured in terms of parts of the molecular mechanics method, and discusses updates and maintenance.

B.2 Performing Molecular Mechanics Calculations

B.2.1 Input

There are several steps involved in obtaining output from a molecular mechanics program. These steps and their order depend on the quantity desired. Before any

computation with a molecular mechanics program is done some decisions should be made first. Often these can depend on the program that is being used as some options desireable to a user may or may not be available. Some decisions to be made before starting a molecular mechanics calculation are as follows [9]:

- What kind of systems are being run?
- What level of accuracy is desired?
- What sort of output is desired, for example structural or spectroscopic data?
- What type of input is needed to run the molecular mechanics program, for example graphical or Z-matrix?

After these decisions are made, the most appropriate molecular mechanics package can be chosen, after which the input must be generated. The Mungauss suite of programs offers a simple menu system to allow the user to select a desired package and choose what printing should be done. This information along with an input geometry for the molecular system is put into an input file to be read in by Mungauss.

This can be done through the use of an input format which will allow the program which builds the molecule to determine what atoms are connected. A popular input format is a Z-matrix [3, 9, 10, 49].

A Z-matrix is an input file which specifies the geometry of a system in terms of internal coordinates (bond lengths, angles, and torsions). The location of an atom with respect to previously specified atoms is given in each row. The first column gives

each atom in the molecule. The third column gives the value of the length of the bond formed between the atoms specified in the first two columns. The fifth column gives the value of the angle formed by the atoms in columns one, two, and four, and the seventh column gives the value of the torsion formed from the atoms specified in columns one, two, four, and six [3, 49].

The first entry in the Z-matrix specifies the first atom, which is placed at the origin. No other information is given on this row. The second row specifies the second atom and a bond to the first atom. The third row specifies the third atom bonded to the second atom with a particular bond length and it makes an angle with the first atom. The fourth row specifies the fourth atom bonded to the third atom, making an angle with the second atom, and a dihedral with the first atom. After this point, specification of each atom needs the bond, angle, and torsion information as outlined in the previous paragraph. The bond, angle, and torsion values can be inserted as numerical results or as a parameter with a label. Labels for the atoms, bonds, angles, and torsions must start with a letter, followed by another letter or a number. In the case of atoms, the first letter or letters must be the elemental symbol of the atom. If a label is used for bonds, angles, or torsions, it must be specified in a parameter list after the Z-matrix [3, 49]. An example of a Z-matrix for methane is given below:

```
INPUT Z_MATRIX FOR METHANE
```

```
C1

H1  C1  CH

H2  C1  CH  H1  HCH

H3  C1  CH  H1  HCH  H2  HCHH

H4  C1  CH  H1  HCH  H2  -HCHH


HC = 1.0931

HCH = 109.4800

HCHH = 120.0000
```

For the molecular mechanics program, there are two options for simulations. First, a single point energy can be computed on a molecule. Selection of this option requires the user to add to the input file the menu option for printing the molecular mechanics energy for the molecule and the energy information will be printed. An example of an input file for the computation of the single point energy for methane is as follows:

```
Molecule

  FreeFormatMatrix

  C1

  H1  C1  HC
```

```
  H2  C1  HC  H1  HCH
  H3  C1  HC  H1  HCH  H2  HCHH
  H4  C1  HC  H1  HCH  H2  -HCHH
  end
  define
  HC = 1.0931
  HCH = TETRA
  HCHH = 120.0000
  end
end
Output
  Object=ENERGY_MM:ENERGY_CONTRIBUTIONS
end
stop
```

In the above example, the free format matrix (Z-matrix) for methane is selected as the input format for the molecule. The next step is to select the method, or in this case, the energy object. If another object is desired, such as the gradients or coordinates, it can be printed in the same manner by replacing the energy with the desired object. The request for an object requires the name of the class to be given followed by the name of the object group for that class, separated by a colon.

If a geometry optimization is required, then the method must be explicitly selected in the case of molecular mechanics, followed by a line selecting a geometry optimization. Using the methane example gives:

```
Molecule
  FreeFormatMatrix
  C1
  H1  C1  HC
  H2  C1  HC  H1  HCH
  H3  C1  HC  H1  HCH  H2  HCHH
  H4  C1  HC  H1  HCH  H2  -HCHH
  end
  define
  HC = 1.0930
  HCH = TETRA
  HCHH = 120.0000
  end
end
!set tralev=50 end
!set debug=BLD_RIC end
!set debug=BLD_RIC_CONTRIBUTIONS end
MM end
```

```
Geom ME=VA ITER=40 ACC=1.0D-06 run end
stop
```

Again, the first step is to give the initial representation of the molecule. The line *MM end* selects the molecular mechanics method. The *Geom* keyword selects the geometry optimization method, where the method in this case is selected as VA, the maximum number of iterations is 40, and the accuracy desired is 10^{-6} (in atomic units). The geometry optimization is then run.

In some molecular mechanics programs additional information such as a connectivity matrix or string to identify the type of simulation may also be needed. This will depend on the package being used [3, 49].

Two debugging tools are seen in the methane geometry optimization example, although in this case they are commented out. However, they were left in the example to show an example of the use of the *debug* and *tralev* debugging tools. These have the advantage of forcing the printing of intermediate results and can be used for this purpose. They are mainly useful to the programmer who can print intermediate results from a specific piece of code only when this keyword is used, giving useful debugging information. The keyword *tralev* causes tracing to be performed and the result is a statement showing the entry and exit from each routine and function as they are called.

B.2.2 Output

The output of the molecular mechanics program is often a geometry as well as an energy. Since most molecular mechanics packages contain a molecule viewer and builder, the output is usually a molecule which can be displayed along with its corresponding energy. For molecular mechanics packages without a molecular viewer or builder, the generation of output is controlled by menu options. The most common menu option for generating output is a printing level. This can be set in the input file and depending on its value, certain computation results will be generated in an output file [10].

B.3 Molecular Mechanics Code Layout

The molecular mechanics code is organized according to classes, utilities, routines that build sets of objects, and global data. There are two main types of files in the molecular mechanics code, *mun* and *mod* files. The *mun* files contain routines which are needed by more than one part of the program, for example utility functions. *mod* files contain either classes or global data. Utility functions which are specific to one type of object but do not necessarily need to be added to a class can also be put in a module. For example, the equilibrium bond lengths, bond force constants, and other global force field data are kept in *mod* files (modules) with names representative of the file contents. Also in modules are the functions for the energy, gradients, and second derivatives. All molecular mechanics classes are kept in modules.

There are three main groups of files in the MM package: classes, utilities, and global data. These files are all contained in a directory called *molecular_mechanics* within the Mungauss main directory. Whenever a file is added, the file can be compiled with the remainder of the molecular mechanics source code by adding two entries in the *Makefile*, one for creating the object file and the other to inform the compiler the object file depends on the source file. This Makefile is also contained within the molecular_mechanics directory.

B.4 Addition of New Parts to the Existing Code

B.4.1 Adding Objects and Classes

A class can be added to the molecular mechanics package in a straightforward manner. The first step once a class is designed is to place the attributes and behaviors into a module file. The name of this file should be representative of the method and class. An example of an existing molecular mechanics class file is the energy class given as *mod_mm_class_energy.f*.

Objects can be placed in a module after the definition of the attributes, before the behaviors. The name of the objects should give a very short description of them and words can be separated by an underscore. Once the objects are known, the names of the class and objects are placed into the get_object function. The purpose of this function is to make sure the proper routine(s) to initialize and compute the

object is called when the object is selected either from the input file or another part of the code. An example of an object name is *SD_STRETCH*, the stretch second derivative object.

Object information can be used by other parts of the program by adding a USE statement where the object information is needed and making a call to get_object with the particular object name in single quotes as the argument, for example:

```
* Modules:
  USE mm_parameters
...
  call get_object ('PARAMETERS:MM_CONTRIBUTIONS')
...
```

In the above example, the calling routine needs the molecular mechanics parameter object, so the USE statement is added at the top and the call to get_object is then done to cause the object to be created. From this point on, the object is available for use anywhere a USE statement is placed.

B.4.2 New Atom Types

Addition of new atom types is also straightforward as the only part of the code which must be greatly modified is the parameter code. First, consider the addition of oxygen. Oxygen contributes a set of its own parameters to the molecular mechanics

package in terms of bonds, angles, torsions, out-of-plane bends, van der Waals, electrostatic, and stretch-bends. Hence, any parts of the code involving the selection of these parameters will need to be modified.

The main difficulty in adding a new atom type will be the computation of *case* values. The computation of case values depends on several features which are used in turn to determine the molecular mechanics atom types for each atom in a particular interaction. These include the atomic number, hybridization, the size of ring the atom belongs to, the bond types it participates in, and whether or not it is part of an aromatic system.

Once the case values are known, the routines to select parameters must be built and added to the file *mun_MM_fconst.f*. In each routine, a set of parameters will be selected from a data module depending on the case number passed in. At this point, the parameter selection routines are available for use and the calls must be added to the appropriate routine in *mod_mm_parameters.f*. This module contains the routines for initializing the parts of the parameter set based on atomic number. So the addition of oxygen to, for example, the bond parameter selection code would involve adding routines for parameter selection for all types of bonds containing oxygen.

Once the parameters are modified to incorporate the new atom and the corresponding types of interactions, the code should be able to function properly with the new atom.

B.4.3 Adding a New Force Field

Addition of a new force field has not been attempted up to this point, but is possible. The force field could be designed in a similar manner to the way the current molecular mechanics package has been designed and an option added to the menu to select a particular force field. For example, if another force field were added, the option for the current force field in the menu could be *MMFF94*. The only parts of the code which would need to be designed specifically for the new force field are those which are specific to that particular force field.

B.5 Summary

Addition and modification of the molecular mechanics package is straightforward and two major additions needed at this point are new atom types and a new force field. A simple menu system allows input files to be easily put together for both geometry optimizations and single point energies and debugging tools allow the user to print intermediate results. Objects can also be selected for printing using the menu system with the name of the class followed by the name of the object, separated by a colon. The output of the final results is then given only for the information requested by the user.

References

[1] Ulrich Burkert and Norman L. Allinger. *Molecular Mechanics.* American Chemical Society, Washington, 1982.

[2] N. L. Allinger and J. P. Bowen. *Reviews in Computational Chemistry*, chapter 2, pages 81–97. VCH, 1991. Molecular Mechanics: The Art and Science of Parameterization.

[3] Ira N. Levine. *Quantum Chemistry: Fifth Edition.* Prentice-Hall, Inc., Upper Saddle River, 2000.

[4] Andrew R. Leach. *Molecular Modeling: Principles and Applications.* Addison Wesley Longman Limited, Essex, 1996.

[5] Anthony K. Rappe and Carla J. Casewit. *Molecular Mechanics Across Chemistry.* University Science Books, Sausalito, 1997.

[6] Jon R. Maple. *Encyclopedia of Computational Chemistry*, chapter 2. John Wiley and Sons Ltd., 1998. Force Fields: A General Discussion.

[7] J. N. Murrell, S. Carter, S. C. Farantos, P. Huxley, and A. J. C. Varandas. *Molecular Potential Energy Functions.* John Wiley and Sons Ltd, West Sussex, 1984.

[8] U. Dinur and A. T. Hagler. *Reviews in Computational Chemistry*, chapter 2, pages 99–164. VCH, 1991. New Approaches to Empirical Force Fields.

[9] Bruce R. Gelin. *Computer Simulation of Biomolecular Systems: Theoretical and Experimental Applications*, chapter 2, pages 127–146. ESCOM, 1993. Testing and Comparison of Empirical Force Fields: Techniques and Problems.

[10] P. Comba and Trevor W. Hambley. *Molecular Modeling of Inorganic Compounds.* VCH, Weinheim, 1995.

[11] N. Trinajstic, S. Nikolic, J. V. Knop, W. R. Muller, and K. Szymanski. *Computational Chemical Graph Theory: Characterization, Enumeration and Generation of Chemical Structures by Computer Methods.* Ellis Horwood Limited, West Sussex, 1991.

[12] Nenad Trinajstic. *Chemical Graph Theory: Second Edition.* CRC Press, Boca Raton, 1992.

[13] Ludek Matyska. Fast Algorithm for Ring Perception. *J. Comp. Chem.*, 9:455, 1988.

[14] E. Bright Wilson Jr., J. C. Decius, and Paul C. Cross. *Molecular Vibrations: The Theory of Infrared and Raman Vibrational Spectra.* McGraw-Hill Book Company, Inc., New York, 1955.

[15] Tamar Schlick. *Reviews in Computational Chemistry*, chapter 3. VCH, 1992. Optimization Methods in Computational Chemistry.

[16] H. Bernhard Schlegel. Optimization of Equilibrium Geometries and Transition Structures. *Adv. Chem. Phys.*, 67:249, 1987.

[17] H. Bernhard Schegel. *Encyclopedia of Computational Chemistry*, chapter 2. John Wiley and Sons Ltd., 1998. Geometry Optimization: 1.

[18] Duane Hanselman and Bruce Littlefield. *The Student Edition of MATLAB: Version 5 Users Guide.* Prentice Hall, Inc., Upper Saddle River, 1997.

[19] Tamar Schlick. *Encyclopedia of Computational Chemistry*, chapter 2. John Wiley and Sons Ltd., 1998. Geometry Optimization: 2.

[20] Stephen G. Nash and Jorge Nocedal. A Numerical Study of the Limited Memory BFGS Method and the Truncated-Newton Method for Large Scale Optimization. *SIAM J. Opt.*, 1:358, 1991.

[21] Pal Csaszar and Peter Pulay. Geometry Optimization by Direct Inversion of the Iterative Subspace. *J. Mol. Str.*, 114:31, 1984.

[22] William C. Davidon. Optimally Conditioned Optimization Algorithms Without Line Searches. *Math. Prog.*, 9:1, 1975.

[23] M. J. D. Powell. *Numerical Methods for Nonlinear Algebraic Equations*, chapter 6. Gordon and Breach Science Publishers, 1970. A Hybrid Method for Nonlinear Equations.

[24] Carl E. Pearson. *Handbook of Applied Mathematics: Selected Results and Methods.* Van Nostrand Reinhold Company, New York, 1974.

[25] P. Pulay and G. Fogarasi. Geometry Optimization in Redundant Internal Coordinates. *J. Chem. Phys.*, 96:2856, 1992.

[26] Chunyang Peng, Philippe Y. Ayala, H. Bernhard Schlegel, and Michael J. Frisch. Using Redundant Internal Coordinates to Optimize Equilibrium Geometries and Transition States. *J. Comp. Chem.*, 17:49, 1996.

[27] Peter Pulay, Geza Fogarasi, Frank Pang, and James E. Boggs. Systematic Ab Initio Gradient Calculation of Molecular Geometries, Force Constants, and Dipole Moment Derivatives. *J. Am. Chem. Soc.*, 101:2550, 1979.

[28] David S. Watkins. *Fundamentals of Matrix Computations.* John Wiley and Sons, Inc., New York, 1991.

[29] B. Meyer. *Object-Oriented Software Construction.* Prentice Hall, Inc., Englewood Cliffs, 1988.

[30] G. Booch. *Object-Oriented Design With Applications.* Benjamin/Cummings Publishing Company, Redwood City, 1991.

[31] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design.* Prentice Hall Inc., Englewood Cliffs, 1991.

[32] P. Coad and E. Yourdon. *Object-Oriented Analysis.* Yourdon Press, Englewood Cliffs, 1991.

[33] G. Booch. *Object Solutions: Managing the Object-Oriented Project.* Addison-Wesley Publishing Company, Menlo Park, 1996.

[34] M. Katherine Holloway, Jenny M. Wai, Thomas A. Halgren, Paula M. D. Fitzgerald, Joseph P. Vacca, Bruce D. Dorsey, Rhonda B. Levin, Wayne J. Thompson, L. Jenny Chen, S. Jane deSolms, Neil Gaffin, Arun K. Ghosh, Elizabeth A. Giuliani, Samuel L. Graham, James P. Guare, Randall W. Hungate, Terry A. Lyle, William M. Sanders, Thomas J. Tucker, Mark Wiggins, Catherine M. Wiscount, Otto W. Woltersdorf, Steven D. Young, Paul D. Larke, and Joan A. Zugay. A Priori Prediction of Activity for HIV-1 Protease Inhibitors Employing Energy Minimization in the Active Site. *J. Med. Chem.*, 38:305, 1995.

[35] Viktor K. Decyk, Charles D. Norton, and Boleslaw K. Szymanski. Introduction to object-oriented concepts using fortran 90. http://www.cs.rpi.edu/ ~szymansk/OOF90/F90_Objects.html.

[36] Jeanne C. Adams, Walter S. Brainerd, Jeanne T. Martin, Brian T. Smith, and Jerrold L. Wagener. *Fortran 90 Handbook: Complete ANSI/ISO Reference.* McGraw-Hill Book Company, New York, 1992.

[37] John L. Volakis and David B. Davidson. Using FORTRAN 90 and Object-Oriented Programming to Accelerate Code Development. *IEEE Antennas and Propagation Magazine*, 41:85, 1999.

[38] Viktor K. Decyk, Charles D. Norton, and Boleslaw K. Szymanski. How to Express C++ Concepts in Fortran 90. *Scientific Programming*, 6:363, 1997.

[39] F. Colonna, L-H Jolly, R. A. Poirier, J. G. Angyan, and G. Jansen. OSIPE - A Tool for Scientific Programming in FORTRAN. *Comp. Phys. Comm.*, 81:293, 1994.

[40] Brian J. Dupee. Object Oriented Methods Using Fortran 90. *ACM Fortran Forum*, 13:21, 1994.

[41] John R. Cary, Svetlana G. Shasharina, Julian C. Cummings, John V. W. Reynders, and Paul J. Hinker. Comparison of C++ and Fortran 90 for Object-Oriented Scientific Programming. *Comp. Phys. Comm.*, 105:20, 1997.

[42] Thomas A. Halgren. Merck Molecular Force Field. I. Basis, Form, Scope, Parameterization, and Performance of MMFF94. *J. Comp. Chem.*, 17:490, 1996.

[43] Thomas A. Halgren. Merck Molecular Force Field. II. MMFF94 van der Waals and Electrostatic Parameters for Intermolecular Interactions. *J. Comp. Chem.*, 17:520, 1996.

[44] Thomas A. Halgren. Merck Molecular Force Field. III. Molecular Geometries and Vibrational Frequencies for MMFF94. *J. Comp. Chem.*, 17:553, 1996.

[45] Thomas A. Halgren and Robert B. Nachbar. Merck Molecular Force Field. IV. Conformational Energies and Geometries for MMFF94. *J. Comp. Chem.*, 17:587, 1996.

[46] Thomas A. Halgren. Merck Molecular Force Field. V. Extension of MMFF94 Using Experimental Data, Additional Computational Data, and Empirical Rules. *J. Comp. Chem.*, 17:616, 1996.

[47] Thomas A. Halgren. Merck Molecular Force Field. VI. MMFF94s Option for Energy Minimization Studies. *J. Comp. Chem.*, 20:720, 1999.

[48] Thomas A. Halgren. Merck Molecular Force Field. VII. Characterization of MMFF94, MMFF94s, and Other Widely Available Force Fields for Conformational Energies and for Intermolecular Interaction Energies and Geometries. *J. Comp. Chem.*, 20:730, 1999.

[49] Warren J. Hehre, Leo Radom, Paul v.R. Schleyer, and John A. Pople. *Ab Initio Molecular Orbital Theory.* John Wiley and Sons Ltd, West Sussex, 1986.