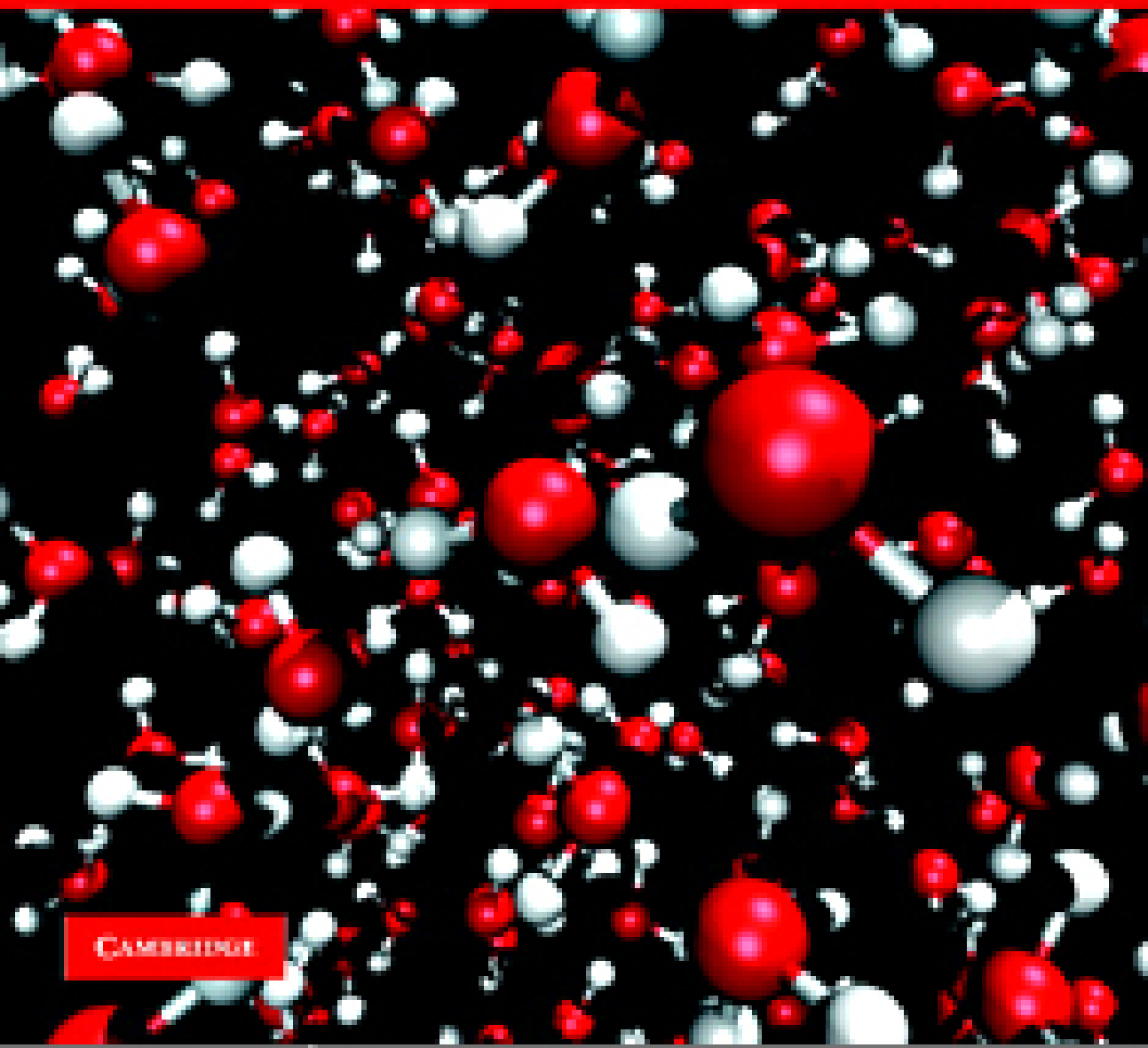
MARTIN J. FIELD

A Practical Introduction to the Simulation of Molecular Systems

SECOND EDITION



CAMBRIDGE

CAMBRIDGE www.cambridge.org/9780521852524

This page intentionally left blank

A PRACTICAL INTRODUCTION TO THE SIMULATION OF MOLECULAR SYSTEMS

Second Edition

Molecular simulation is a powerful tool in materials science, physics, biophysics, chemistry, drug design and many other areas of research. This updated edition provides a pragmatic introduction to a wide range of techniques for the simulation of molecular systems at the atomic level. The first part of the book concentrates on methods for calculating the potential energy of a molecular system, with new chapters on quantum chemical, molecular mechanical and hybrid potential techniques. The second part covers ways of investigating the conformational, dynamical and thermodynamical properties of systems, discussing such techniques as geometry-optimization, normal-mode analysis, and molecular dynamics and Monte Carlo simulation.

Now employing Python, the second edition includes a wealth of examples and program modules for each simulation technique. This allows readers to carry out the calculations and to appreciate the inherent difficulties involved in each. This is a valuable resource for researchers and graduate students wanting to know how to perform atomic-scale molecular simulations.

Additional resources for this title, including the program library, technical information and instructor-solutions, are available online at www.cambridge.org/9780521852524.

MARTIN J. FIELD is Group Leader of the Laboratoire de Dynamique Moléculaire at the Institut de Biologie Structurale – Jean-Pierre Ebel, Grenoble. He was awarded his Ph.D. in Quantum Chemistry from the University of Manchester, UK, in 1985. His areas of research include using molecular modeling and simulation techniques to study biological problems. More specifically, his current interests are in the development and application of hybrid potential techniques to study enzymatic reaction mechanisms and other condensed phase processes.

Reviews of the first edition:

'... a valuable teaching aid for those presenting this topic. It should be of interest not only to the physical chemist, but also to those involved in computational biophysics, biochemistry or molecular physics.' *Scientific Computing World*

'... this book is a valuable addition to my shelf and one that I must make sure doesn't disappear because my research group has taken off with it!' *Nell L. Allan. Chemistry and Industry*

A PRACTICAL INTRODUCTION TO THE SIMULATION OF MOLECULAR SYSTEMS

Second Edition

MARTIN J. FIELD

Institut de Biologie Structurale – Jean-Pierre Ebel, Grenoble, France

CAMBRIDGE UNIVERSITY PRESS

CAMBRIDGE UNIVERSITY PRESS
Cambridge, New York, Melbourne, Madrid, Cape Town, Singapore, São Paulo

Cambridge University Press
The Edinburgh Building, Cambridge CB2 8RU, UK
Published in the United States of America by Cambridge University Press, New York
www.cambridge.org
Information on this title: www.cambridge.org/9780521852524

First published in print format 2007

ISBN-13 978-0-511-34918-8 eBook (EBL)
ISBN-10 0-511-34918-1 eBook (EBL)

ISBN-13 978-0-521-85252-4 hardback
ISBN-10 0-521-85252-8 hardback

Contents

Preface to the first edition

The reason that I have written this book is simple. It is the book that I would have liked to have had when I was learning how to carry out simulations of complex molecular systems. There was certainly no lack of information about the theory behind the simulations but this was widely dispersed in the literature and I often discovered it only long after I needed it. Equally frustrating, the programs to which I had access were often poorly documented, sometimes not at all, and so they were difficult to use unless the people who had written them were available and preferably in the office next door! The situation has improved somewhat since then (the 1980s) with the publication of some excellent monographs but these are primarily directed at simple systems, such as liquids or Lennard-Jones fluids, and do not address many of the problems that are specific to larger molecules.

My goal has been to provide a practical introduction to the simulation of molecules using molecular mechanical potentials. After reading the book, readers should have a reasonably complete understanding of how such simulations are performed, how the programs that perform them work and, most importantly, how the example programs presented in the text can be tailored to perform other types of calculation. The book is an *introduction* aimed at advanced undergraduates, graduate students and confirmed researchers who are newcomers to the field. It does not purport to cover comprehensively the entire range of molecular simulation techniques, a task that would be difficult in 300 or so pages. Instead, I have tried to highlight some of the basic tasks that can be done with molecular simulations and to indicate some of the many exciting developments which are occurring in this rapidly evolving field. I have chosen the references which I have put in carefully as I did not want to burden the text with too much information. Inevitably such a choice is subjective and I apologise in advance to those workers whose work or part of whose work I did not explicitly acknowledge.

There are many people who directly or indirectly have helped to make this book possible and whom I would like to thank. They are: my early teachers in the

field of computational chemistry, Nicholas Handy at Cambridge and Ian Hillier at Manchester; Martin Karplus and all the members of his group at Harvard (too numerous to mention!) during the period 1985–9 who introduced me to molecular dynamics simulations and molecular mechanics calculations; Bernie Brooks and Rich Pastor, at the NIH and FDA, respectively, whose lively discussion and help greatly improved my understanding of the simulations I was doing; and all the members of my laboratory at the IBS, past and present, Patricia Amara, Dominique Bicout, Celine Bret, Laurent David, Lars Hemmingsen, Konrad Hinsen, David Jourand, Flavien Proust, Olivier Roche and Aline Thomas. Finally, special thanks go to Patricia Amara and to Dick Wade at the IBS for comments on the manuscript, to Simon Capelin and the staff of Cambridge University Press for their guidance with the production of the book, to the Commissariat à l'Energie Atomique and the Centre National de la Recherche Scientifique for financial support and to my wife, Laurence, and to my sons, Mathieu and Jeremy, for their patience.

Martin J. Field
Grenoble, 1998

Preface to the second edition

This edition of *A Practical Introduction* has two major differences from the previous one. The first is a discussion of quantum chemical and hybrid potential methods for calculating the potential energies of molecular systems. Quantum chemical approaches are more costly than molecular mechanical techniques but are, in principle, more 'exact' and greatly extend the types of phenomena that can be studied with the other algorithms described in the book. The second difference is the replacement of FORTRAN 90 by Python as the language in which the DYNAMO module library and the book's computer programs are written. This change was aimed to make the library more accessible and easier to use. As well as these major changes, there have been many minor modifications, some of which I wanted to make myself but many that were inspired by the suggestions of readers of the first edition.

Once again, I would like to acknowledge my collaborators at the Institut de Biologie Structurale in Grenoble and elsewhere for their comments and feedback. Special thanks go to all members, past and present, of the Laboratoire de Dynamique Moléculaire at the IBS, to Konrad Hinsen at the Centre de Biophysique Moléculaire in Orléans and to Troy Wymore at the Pittsburgh Supercomputing Center. I would also like to thank Michelle Carey, Anna Littlewood and the staff of Cambridge University Press for their help during the preparation of this edition, Johny Sebastian from TechBooks for answers to my many LaTeX questions, and, of course, my family for their support.

Martin J. Field
Grenoble, 2006

1

Preliminaries

1.1 Introduction

The aim of this book is to give a practical introduction to performing simulations of molecular systems. This is accomplished by summarizing the theory underlying the various types of simulation method and providing a programming library, called pDynamo, which can be used to perform the calculations that are described. The style of the book is pragmatic. Each chapter, in general, contains some theory about related simulation topics together with descriptions of example programs that illustrate their use. Suggestions for further work (or exercises) are listed at the end.

By the end of the book, readers should have a good idea of how to simulate molecular systems as well as some of the difficulties that are involved. The pDynamo library should also be a reasonably convenient starting point for those wanting to write programs to study the systems they are interested in. The fact that users have to write their own programs to do their simulations has advantages and disadvantages. The major advantage is flexibility. Many molecular modeling programs come with interfaces that supply only a limited range of options. In contrast, the simulation algorithms in pDynamo can be combined arbitrarily and much of the data generated by the program is available for analysis. The drawback is that the programs have to be written – a task that many readers may not be familiar with or have little inclination to do themselves. However, those who fall into the latter category are urged to read on. pDynamo has been designed to be easy to use and should be accessible to everyone even if they have only a minimum amount of computing experience.

This chapter explains some essential background information about the programming style in which pDynamo and the example programs are written. Details of how to obtain the library for implementation on specific machines are left to the appendices.

1.2 Python

All the example programs in this book and much of the programming library are written in the programming language Python. The rest of the library, which most readers will never need to look at, consists of code for which computational efficiency is paramount and is written in C. The reasons for the choice of Python were threefold. First, it is a powerful and modern programming language that is fun to use! Unlike languages such as C and FORTRAN, it is an interpreted language, which means that programs can be run immediately without going through separate compilation and linking steps. Second, Python is open-source software that is free and runs under a wide variety of operating systems and, third, there is a very active development community that is continually enhancing the language and adding to its capabilities.

Most computer languages are easiest to learn by example and Python is no exception. The following, simple program illustrates several basic features of the language:

```
1  """Example 0."""
2
3  import math
4
5  # . Define a squaring function.
6  def Square ( x ):
7      return x**2
8
9  # . Create a list of integers.
10 values = range ( 10 )
11
12 # . Loop over the integers.
13 for i in values:
14     x = float ( i )
15     print "%5d%10.5f%10.5f%10.5f" \
               % ( i, x, math.sqrt ( x ), Square ( x ) )
```

Line 1 is the program's *documentation string* which, in principle, should give a concise description of what the program is supposed to do. All the examples in this book, however, have documentation strings of the type `"""Example n."""` to save space and to avoid duplicating the explanations that occur in the text.

Lines 2, 4, 8 and 11 are blank and are ignored.

Line 3 makes the standard Python *module* `math` accessible to the program. Python itself and programs written using Python – including pDynamo – consist of modules which must be explicitly *imported* if their contents are to be used.

The `import` statement has a number of different forms and the one shown is the simplest. With this form, module items are accessed by prefixing the item's name with the module name, followed by a dot (`.`) character. Thus, the function `sqrt` from the module `math`, which is used on *line 15* of the program, is accessed as `math.sqrt`. An alternative form, which is sometimes preferable, is `from math import sqrt`. This makes it possible to refer to the function `sqrt` by its name only without the `math.` prefix.

Lines 5, 9 and 12 are *comments* which are included to make the program easier to understand. Python ignores all characters from the hash character (`#`) until the end of the line.

Lines 6–7 define a very simple Python *function*. Functions are named collections of instructions that can be *called* or *invoked* at different points in a program. They behave similarly in Python to functions in other languages, such as C and FORTRAN.

Line 6 is the function definition line. It starts with the word `def` which tells Python that a function definition is coming and terminates with a colon (`:`). The second word on the line, `Square`, is the name that we are giving to the function and this is followed by the function's *arguments* which appear in parentheses. Arguments are variables that the function needs in order to work. Here there is only one, `x`, but there can be many more.

The function definition line is followed by the *body* of the function. This would normally consist of several lines but here there is only one, *line 7*. Python is unusual among programming languages in that the lines in the function body are determined by line indentation. In other languages, such blocks of code are delimited by specific characters, such as the matching braces {...} of C or the `FUNCTION` ... `END FUNCTION` keywords of FORTRAN 90. The number of spaces to indent by is arbitrary – in this book it is always four – but all lines must be indented by the same amount and instructions after the end of the function must return to the original indentation level.

Line 7 is very simple. The second part of the line contains the expression `x**2` which computes the square of the function's argument `x`. The `**` symbol denotes the power operator and so the expression tells Python to raise `x` to the power of 2. The first part of the line is the keyword

`return` which says that the result of the squaring calculation is to be *returned* to the place from which the function was called.

Line 10 is the first executable line of the example and illustrates several more features of the Python language – *built-in functions*, *sequence types* and *variable assignment*. `range` is one of Python's built-in functions and is always available whenever the Python interpreter is invoked. It produces a sequence of integers, in this case ten of them, starting with the value `0` and finishing with the value `9`. Python, like C, but unlike FORTRAN, starts counting from zero and not from one. The integers are returned as a *list* which is one of Python's built-in sequence data types and is one of the things that makes Python so attractive to use. Finally the list of integers is assigned to a variable with the name `values`. Python differs from many languages in that variables do not need to be declared as being of a particular type. In C, for example, an integer variable would have to be declared with a statement such as "`int i ;`" before it could be used. These declarative statements do not exist in Python and so `values` can be assigned arbitrarily to refer to any data type.

Line 13 shows one of the forms of *iteration* in Python. The statement takes the list referred to by `values` and assigns each of its elements to the variable `i` in turn. The iteration stops when the end of the list is reached. The lines over which iteration is to occur are determined by line identation in exactly the same way as those in the body of a function.

Line 14 is the first line of the loop specified by the `for` construct in *line 13*. It takes the integer referred to by the variable `i`, converts it to a *floating-point number* using the built-in function `float` and then assigns it to the variable `x`.

Line 15 is printed as two lines in the text, due to the restricted page width, but it is logically a single statement. The presence of the backslash character (\) at the end of the line indicates to Python that the subsequent line is to be treated as a continuation of the current one.

The statement prints the values of `i` and of `x`, the square root of `x`, which is calculated by invoking the function `sqrt` from the module `math`, and the square of `x`, calculated using the previously defined function `Square`. These items are grouped together at the end of the line in a *tuple*. Tuples, like lists, are one of Python's built-in sequence data types and are constructed by enclosing the items that are to be in the tuple in parentheses. Tuples differ from lists, though, in that they are *immutable*, which means that their contents cannot be changed once they have been created.

The style in which the quantities are printed is determined by the *formatting string*, which is enclosed in double quotes `""`. This is placed after the `print` keyword and is separated from the tuple of items to be printed by the `%` character. Python employs a syntax for formatting operations that is very similar to that of the C language. Output fields start with a `%` character and so, in this example, there are four output fields in the string, one for each of the items to be printed. The first output field is `%5d`, which says that an integer, coded for by the letter `d`, is to be printed in a field `5` characters wide. The remaining fields are identical and have the form `%10.5f`. They are for the output of floating-point numbers (`f`) in fields `10` characters wide but with `5` of these characters occurring after the decimal point.

It is not, of course, possible to master a language from a single, short program but readers should gain in expertise and come to appreciate more fully the capabilities of the language as they work through the examples and exercises in the book. One of the great advantages of Python for learning is that it can be used interactively and so it is quick and easy to write simple programs to test whether one really understands what the language is doing.

1.3 Object-oriented programming

Python admits various programming styles but all the modules in pDynamo are written using an *object-oriented* approach in which the basic unit of programming is the *class*. A class encapsulates the notion of an *object*, such as a file or a molecule, and groups together the data or *attributes* needed to describe the object and the functions or *methods* that are required to manipulate it. Classes are used by *instantiating* them so that, for example, a program for modeling the molecules methanol and water would create two instances of the class molecule, one to represent methanol and one water.

There are other important aspects of object-oriented programming that pDynamo employs but which will only be alluded to briefly, if at all, later on. Two of these are *inheritance* and *polymorphism*. Inheritance is the mechanism by which a new class is defined in terms of an existing one. For example, a class for manipulating organic molecules could be derived from a more general class for molecules. The new class would inherit all the attributes and methods defined by its parent class but would also have attributes and methods specific for organic molecules. Polymorphism is related to inheritance and is the ability to redefine methods for derived classes. Thus, the general molecule class could have a method for chemical reactions but this would

be *overridden* in the organic molecule class because the rules for implementing reactions for organic molecules are different from those of the general case.

It is time to consider a simple, hypothetical example. Suppose there were a *class hierachy* designed for writing to text files. The *base class* would be a general class, `TextFileWriter`, and there would be a *subclass*, `DataFileWriter`, designed for writing either a specific type of data or data in a specific format. The base class has the following (partial) specification:

```
1 class TextFileWriter ( object ):
2     """The base class for objects that write to text files."""
3
4     def __init__ ( self, filename ):
5         """Instance initializer from |filename|."""
6         self.name = filename
7         ... other initialization here ...
8
9     def Close ( self ):
10        """Close the file."""
11        ... contents here ...
```

Line 1 says that a class of name `TextFileWriter` is being defined and that it is subclassed from the class `object` which is the base class for all Python objects.

Line 2 is the documentation string for the class.

Lines 4 to 7 define a special method, `__init__`, that is called when an instance of a class is created.

The method has two arguments. The first, `self`, denotes the instance of the class that is calling the method (hence the name `self`). Python requires that this argument appears in the specification of all instance methods in a class but that it should not be present when a method is actually invoked. We shall see examples of this later in the section. The second argument, `filename`, gives the name of the file to which data are to be written.

The body of the method, *line 6* onwards, would be used to perform various 'start-up' operations on the newly created instance. This could include the initialization of various attributes that the instance may need and the setting up of the necessary data structures for writing to a file with the given name. The only operation that we show explicitly is on *line 6* because it illustrates how an attribute of an instance can be defined – in this case, the attribute, `name`, of the instance `self` that points to the

name of the file. The dot-notation identifies the attribute as belonging to the instance and is employed when accessing an attribute as well as for its definition.

Lines 9 to 11 define a method, `Close`, that would be called when writing to the file has terminated.

The subclass is specified as follows:

```
1 class DataFileWriter ( TextFileWriter ):
2     """A class for writing data to a text file."""
3
4     def WriteData ( self, data ):
5         """Write |data| to the file."""
6         ... contents here ...
```

Line 1 defines `DataFileWriter` as a subclass of `TextFileWriter`.

Lines 4 to 6 define a method, `WriteData`, that would be called to write the argument `data` to the file.

The methods `__init__` and `Close` are absent from the specification of the subclass, which means that they are inherited from the parent class, `TextFileWriter`, and behave in an identical fashion.

Once the classes and its methods have been defined, data could be written to a file of the class `DataFileWriter` with the following series of commands:

```
1 datafile = DataFileWriter ( "myfile.dat" )
2 datafile.WriteData ( data )
3 datafile.Close ( )
4 print "Data written to the file", datafile.name
```

Line 1 creates an instance of the class `DataFileWriter` that is called `datafile`. The instance is produced using the name of the class followed by parentheses that contain the arguments, excluding `self`, that are to be passed to the class's `__init__` method. In this case there is a single argument, `"myfile.dat"`, which is a character string that contains the name of the file to be written. In the rest of this book, we shall refer to statements in which instances of a class are generated using the class name as *constructors*.

Line 2 calls the `WriteData` method of the instance with the data to be written. Methods of an instance are most usually invoked using the dot-notation. This is similar to the way in which the attributes of an instance are

accessed but with the difference that the method's arguments appear in parentheses after the method name. As discussed above, the `self` argument that occurs in the specification of the method is absent.

Line 3 closes the file as writing to the file has terminated. The `Close` method takes no arguments but parentheses are still required.

Line 4 prints a short informational message. This version of the print statement is simpler than that used previously in that no formatting information is present. Instead, Python chooses suitable defaults for the way in which the string `"Data written to the file"` and the `name` attribute of `datafile` are written.

Although the class notation is very elegant, it can be a little cumbersome to use. This is why, for many classes, pDynamo supplies 'helper' functions that provide a shorthand way of using the class without having to explicitly instantiate it. An appropriate helper function for the class `DataFileWriter` would simply be one that 'wraps' the four-line program given previously. It would have the form:

```
def DataFile_Write ( filename, data ):
    """Write |data| to the data file with name |filename|."""
    datafile = DataFileWriter ( filename )
    datafile.WriteData ( data )
    datafile.Close ( )
    print "Data written to the file", datafile.name
```

and would be used as follows

```
DataFile_Write ( "myfile.dat", data )
```

1.4 The pDynamo library

Like many large Python libraries, pDynamo is hierarchically organized into *packages* and modules. A package is a named collection of modules that are put together because they perform logically related tasks, whereas a module is a collection of Python classes, functions and other instructions that are grouped in a single Python file. As we saw in the example program of Section 1.2, modules from a library can be used in Python programs by importing them with the `import` keyword. The same syntax is possible with packages, so the

statement `import mypackage` would make the contents of the package called `mypackage` accessible.

pDynamo consists of three principal packages. The first and most fundamental package is `pCore`. It contains modules implementing various basic data structures and algorithms that are independent of molecular applications. The second package is `pDynamo`, which has modules for representing and manipulating molecular systems and for performing molecular simulation. The third package is `pBabel`, which has modules that read and write information for chemical systems in various formats. The packages are arranged hierarchically because `pBabel` depends upon both `pDynamo` and `pCore`, `pDynamo` upon `pCore`, but not `pBabel`, and `pCore` upon neither.

The purpose of this book is not to provide a detailed description of each of the pDynamo packages. Instead, only a subset of pDynamo's classes and functions will be introduced as needed. Some, whose behaviour and construction are deemed important for the arguments being pursued in the text, will be described in detail, whereas others will be mentioned only in passing. A summary of all the items from the pDynamo library appearing in the book is given in Appendix 1, whereas full documentation will be found online with the library's source code and the book's example programs.

1.5 Notation and units

To finish, a few general points about the notation and units used in this book and the program library will be made. In the text, all program listings and the definitions of classes, methods, functions and variables have been represented by using characters in typewriter style, e.g. `molecule`. For other symbols, normal typed letters are used for scalar quantities whereas bold face italic letters are employed for vectors and bold face roman for matrices. Lower case letters have generally been taken to represent the properties of individual atoms whereas upper case letters represent the properties of a group of atoms or, more usually, the entire system. Lower case roman subscripts normally refer to atoms, upper case roman subscripts to entire structures and Greek subscripts to other quantities, such as the Cartesian components of a vector or quantum chemical basis functions. The more common symbols are listed in Tables 1.1, 1.2 and 1.3.

The units of most of the quantities either employed or calculated by pDynamo are specified in Table 1.4. All the quantum chemical algorithms use *atomic units* internally although little input or output is done in them. Nevertheless, for completeness, Table 1.5 lists some quantities in atomic units and their pDynamo equivalents.

Table 1.1 *Symbols that denote quantities for atoms or for the entire system*

Symbol	Description
Atomic quantities	
α_i	Isotropic dipole polarizability for atom i
$\boldsymbol{\zeta}_i$	Mass-weighted first derivatives of potential energy
$\boldsymbol{a}_i$ ($\equiv \ddot{\boldsymbol{r}}_i$)	Acceleration of atom i
$\boldsymbol{f}_i$	Force on atom i
$\boldsymbol{g}_i$	First derivatives of potential energy with respect to coordinates of atom i
$\mathbf{h}_{ij}$	Second derivatives of potential energy with respect to coordinates of atoms i and j
m_i	Mass of atom i
$\boldsymbol{p}_i$	Momentum vector for atom i
$\boldsymbol{q}_i$	Vector of mass-weighted Cartesian coordinates for atom i
q_i	Partial charge for atom i
$\boldsymbol{r}_i$	Vector of Cartesian coordinates, (x_i, y_i, z_i), for atom i
r_{ij}	Distance between two atoms i and j
$\boldsymbol{s}_i$	Vector of Cartesian fractional coordinates for atom i
$\boldsymbol{v}_i$ ($\equiv \dot{\boldsymbol{r}}_i$)	Velocity vector for atom i
w_i	Weighting factor for atom i
x_i	x Cartesian coordinate of atom i
y_i	y Cartesian coordinate of atom i
z_i	z Cartesian coordinate of atom i
System quantities	
$\boldsymbol{\mu}$	Dipole-moment vector
σ_{IJ}	Root mean square coordinate deviation between structures I and J
$\boldsymbol{A}$	$3N$-dimensional vector of atom accelerations
$\boldsymbol{D}$	$3N$-dimensional coordinate displacement vector
$\boldsymbol{F}$	$3N$-dimensional vector of atom forces
$\boldsymbol{G}$	$3N$-dimensional vector of first derivatives
G_{RMS}	Root mean square (RMS) gradient
$\mathbf{H}$	$(3N \times 3N)$-dimensional matrix of second derivatives of system
$\boldsymbol{\mathcal{I}}$	Inertia matrix
$\mathbf{M}$	$3N \times 3N$ diagonal atomic mass matrix
O	System observable or property
$\boldsymbol{P}$	$3N$-dimensional vector of atom momenta
Q	Charge
$\boldsymbol{R}$	$3N$-dimensional vector of atom coordinates
$\boldsymbol{R}_{\mathrm{c}}$	Centre of charge, geometry or mass
$\boldsymbol{S}$	$3N$-dimensional vector of atom fractional coordinates
$\boldsymbol{V}$	$3N$-dimensional vector of atom velocities

Table 1.2 *Quantum chemical symbols (Chapters 4 and 6)*

Symbol	Description
General symbols	
α, β	Electron spin-up and spin-down functions
ϵ_a	Energy of orbital a
ϵ_{ab}	Lagrange multiplier for orbitals a and b
ζ	Basis function exponent
η	Basis function
Θ	Angular momentum function
ρ	Charge or electron density
σ	Electron spin function (either α or β)
ϕ	One-electron space orbital
Φ	Slater determinant
Ψ	Wavefunction
A_I	Configuration interaction coefficient for state I
$c_{\mu a}$	Molecular orbital coefficient for basis function μ and orbital a
$\mathbf{C}$	Matrix of molecular orbital coefficients
E	Energy of a stationary state
$\mathbf{F}$, $F_{\mu\nu}$	Fock matrix and one of its elements
$\mathbf{H}$, $H_{\mu\nu}$	One-electron matrix and one of its elements
$\hat{\mathcal{H}}$	Quantum mechanical Hamiltonian
$\hat{\mathcal{K}}$	Quantum mechanical kinetic energy operator
M	Number of electrons
n_a	Occupancy of orbital a
N_b	Number of basis functions
$\mathbf{P}$, $P_{\mu\nu}$	Density matrix and one of its elements
Q_{lm}	Multipole moment
$\mathbf{S}$, $S_{\mu\nu}$	Overlap matrix and one of its elements
S_{lm}	Real spherical harmonic function
$\hat{\mathcal{V}}$	Quantum mechanical potential energy operator
$\mathbf{W}$, $W_{\mu\nu}$	Energy-weighted density matrix and one of its elements
Z	Nuclear charge
Subscripts and superscripts	
λ, μ, ν, σ	Basis functions
a, b, c	Orbitals
e, el	Electronic
eff	Effective
i, j	Quantum chemical atoms
m	Molecular mechanical atom
MM	Molecular mechanical
n	Nuclear
nn	Nuclear–nuclear
QC	Quantum chemical
s, t	Electrons
t	Total (electronic plus nuclear)

Table 1.3 *Miscellaneous symbols*

Symbol	Description
Matrix and vector operations	
$\dot{\boldsymbol{a}}$	First time derivative of a vector
$\ddot{\boldsymbol{a}}$	Second time derivative of a vector
$\hat{\boldsymbol{a}}$	Normalized vector, $\boldsymbol{a}/a$, where $a = \|\boldsymbol{a}\|$
$\boldsymbol{a}^{\mathrm{T}}$	Transpose of vector $\boldsymbol{a}$
$\|\mathbf{A}\|$	Determinant of matrix $\mathbf{A}$
$\boldsymbol{a}^{\mathrm{T}}\boldsymbol{b}$	Dot or scalar product of two vectors
$\boldsymbol{a}\boldsymbol{b}^{\mathrm{T}}$	Outer product of two vectors
$\boldsymbol{a} \wedge \boldsymbol{b}$	Cross or vector product
Other symbols	
A	Helmholtz free energy
ϵ	Dielectric constant for system
ϵ_0	Permittivity of vacuum
Λ	Constraint condition
θ, ϕ	Spherical polar angles
E	Total energy of system
G	Gibbs free energy
H	Enthalpy
$\mathcal{H}$	Classical Hamiltonian for system
$\mathcal{K}$	Kinetic energy for a system
k_{B}	Boltzmann's constant
L	Length of side of cubic box
N	Number of atoms in system
N_{df}	Number of degrees of freedom in system
N_{o}	Number of observables
p	Parameter
$\boldsymbol{p}$	Parameter vector
P	Pressure
$\mathcal{P}$	Instantaneous pressure
R	Molar gas constant
S	Entropy
t	Time
T	Temperature
$\mathcal{T}$	Instantaneous temperature
U	Internal energy
$\mathbf{U}$	Proper or improper rotation matrix
$\mathcal{V}$	Potential energy of system
V	Volume of a system
Z	Partition function

Table 1.4 *The units employed by the pDynamo library*

Quantity	Name	Symbol	SI equivalent
Angle (input and output)	Degrees	°	$\equiv \pi/180$ rad
Angle (internally)	Radians	rad	
Charge	Elementary charge	e	$\simeq 1.602 \times 10^{-19}$ C
Dipole	Debyes	D	$\simeq 3.336 \times 10^{-30}$ C m
Energy	Kilojoules per mole	kJ mol^{-1}	$\simeq 1.066 \times 10^{-21}$ J
Frequency	Wavenumbers	cm^{-1}	$\simeq 2.998 \times 10^{10}$ Hz
	Terahertz	THz	10^{12} Hz
Length	Ångströms	Å	$\equiv 10^{-10}$ m
Mass	Atomic mass units	a.m.u.	$\simeq 1.661 \times 10^{-27}$ kg
Pressure	Atmospheres	atm	$\equiv 1.013250 \times 10^{5}$ Pa
Temperature	Kelvins	K	
Time	Picoseconds	ps	$\equiv 10^{-12}$ s
Velocity	Ångströms per picosecond	Å ps^{-1}	$\equiv 10^{2}$ m s^{-1}
Volume	Ångströms cubed	Å^3	$\equiv 10^{-30}$ m^3

Table 1.5 *Atomic units and their pDynamo equivalents*

Quantity	Name	Symbol	pDynamo equivalent
Angular momentum	Dirac's constant	$\hbar$	$\simeq 6.351 \times 10^{-2}$ kJ mol^{-1} ps
Charge	Elementary charge	e	$\equiv 1\ e$
Dipole		ea_0	$\simeq 2.541$ D
Energy	Hartrees	E_h	$\simeq 2625.5$ kJ mol^{-1}
Length	Bohrs	a_0	$\simeq 0.529$ Å
Mass	Electron rest mass	m_e	$\simeq 5.486 \times 10^{-4}$ a.m.u.
Time		$\hbar/E_\mathrm{h}$	$\simeq 2.419 \times 10^{-5}$ ps
Velocity		$a_0 E_\mathrm{h}/\hbar$	$\simeq 2.188 \times 10^{4}$ Å ps^{-1}

2

Chemical models and representations

2.1 Introduction

Models and representations are crucial in all areas of science. They are employed whenever one thinks about an object or a phenomenon and whenever one wants to interpret or to predict how it is going to behave. Models need not be unique. In fact, it is normal to have multiple representations of varying complexity and to choose the one that is most appropriate given the circumstances.

The same is true when thinking about chemical and molecular systems. Representations encompass the traditional chemical ones that employ atoms and bonds, and also span the range from the very fundamental, in which a molecule is considered as a collection of nuclei and electrons, through to the highly abstract, in which a molecule is treated as a *chemical graph*. Several of these are illustrated in Figure 2.1.

The purpose of this chapter is twofold. First, we introduce the way in which pDynamo represents and manipulates molecular systems. Of course, we can only start to discuss these topics here as pDynamo has a diversity of approaches that will take the rest of the book to explain. Second, we describe several common molecular representations that are used in modeling and simulation studies and show how pDynamo can transform between them.

2.2 The System class

`System` is the central class of the pDynamo program and it will be used directly or indirectly in most of the examples in this book. Its purpose is to gather together the data that are required to represent, to model and to simulate a molecular system. The fundamental data in a system are the sequence of atoms that it contains. For various reasons, systems were designed to be immutable with respect to their atom composition. This means that once a system exists the atoms it contains and their order cannot be modified. These changes can be effected but only by creating a new system, independent of the old one, with the desired composition.

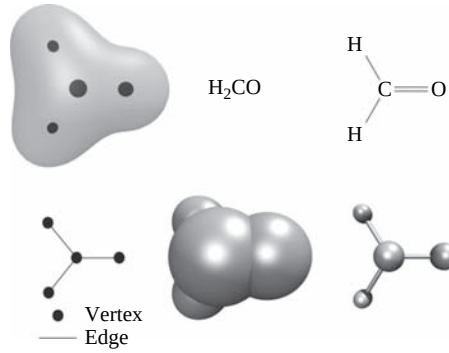


Fig. 2.1. Different representations of the molecule formaldehyde. Clockwise from top left: electron cloud and (oversized) nuclei; a chemical formula; a two-dimensional model; a three-dimensional ball and stick model; a three-dimensional space filling model; a chemical graph.

The `System` class is an extensive one and so only a few of its methods and attributes will be described here. More will be introduced later on in the book as they are needed. The definition of the class that will suffice for the moment is as follows:

Class `System`

A class to represent a molecular system.

Constructor

Construct an instance of the `System` class given some atom data.

Usage: `new = System ( atoms )`

`atoms` is a sequence that defines the atoms in the system. In the only case that we shall consider, `atoms` is a Python list containing the atomic numbers of the system's atoms.

`new` is the new instance of `System`.

Remarks: It will rarely be necessary to use the constructor directly as new instances of `System` will most often be obtained by alternative routes.

Method `Summary`

Output a summary of the system.

Usage: `system.Summary ( )`

`system` is an instance of `System`.

Attributes

`atoms`	is an instance of the class `AtomContainer` and contains the sequence of atoms in the system. Each atom in the container is represented by an instance of the class `Atom`.
`coordinates3`	contains the Cartesian x, y and z coordinates for each atom in the system. The coordinates and their manipulation will be described in more detail in the next chapter.
`label`	is a string containing the name or a short description of the system.

The operation of summarizing is a common one that it is often useful to do on other pDynamo objects. As a result, many pDynamo classes have a method called `Summary`, all of which have similar characteristics to that described above. There are a number of other operations, similar to summarizing, that work on a range of pDynamo classes and which we shall require later in this book. Three of these are cloning, merging and pruning which, unlike summarizing, are implemented via helper functions:

Function `Clone`

Create a new instance of an object by duplicating an existing one.

Usage: `new = Clone ( old )`

`old`	is the old instance.
`new`	is the new instance.
Remarks:	Cloning creates a new copy of the old instance so that removal of either instance will not affect the other. The existing instance is unchanged. This function works with all pDynamo classes.

Function `MergeByAtom`

Create a new instance of a class by merging existing instances of the class.

Usage: `new = MergeByAtom ( old1, old2, ... )`

`old1`, etc.	are the old instances.
`new`	is the new, larger instance.
Remarks:	Merging by atom is like cloning in that an instance is produced that is independent of the instances from which it is created. It is less general, however, in that merging only makes sense for classes that have some concept of an atom, such as `System`. For classes for

which this is not the case, no merging will be done. The order of merging is significant as the old instances are merged in the order in which they are passed to the function.

Function `PruneByAtom`

Create a new instance of a class by pruning atoms from an existing instance of the class.

Usage:	`new = PruneByAtom ( old, selection )`
`old`	is the old instance.
`selection`	specifies the atoms that are to be kept in the pruning process. Most usually this argument is an instance of the class `Selection` that contains the indices of the atoms to be retained.
`new`	is the new, smaller instance.
Remarks:	Pruning by atom is similar to merging in that only classes with the notion of an atom can be pruned.

2.3 Example 1

The first example is designed to illustrate some of the basic capabilities of the class `System`. It is:

```
1 """Example 1."""
2
3 from Definitions import *
4
5 # . Create a water molecule.
6 water       = System ( [ 8, 1, 1 ] )
7 water.label = "Water"
8 water.Summary ( )
9
10 # . Create a water dimer.
11 waterdimer       = MergeByAtom ( water, water )
12 waterdimer.label = "Water Dimer"
13 waterdimer.Summary ( )
14
15 # . Create a hydroxyl.
16 oh = Selection ( [ 0, 1 ] )
17 hydroxyl       = PruneByAtom ( water, oh )
18 hydroxyl.label = "Hydroxyl"
19 hydroxyl.Summary ( )
```

Line 3 The program starts by importing all quantities from the module `Definitions`. This module was designed especially for the examples in this book and serves two purposes. First, it defines various variables that are needed by some of the example programs. Mostly these are strings that give the location of data files. Second, it imports from the pDynamo packages, `pCore`, `pDynamo` and `pBabel`, all the classes, functions and variables that the examples use. The use of a module in this fashion is a convenience as it means that definitions that otherwise would straddle several lines can be replaced by a single statement.

In this example, the definitions that are required are the ones that import the classes `Selection` and `System` and the functions `MergeByAtom` and `PruneByAtom`.

Line 6 creates an instance of the class `System` and assigns it to the variable `water`. The argument to `System` is a list that contains the atomic numbers of the atoms in the molecule – one oxygen and two hydrogens. Note the use of square brackets which is the syntax that Python uses to construct a `list` object.

Line 11 creates a second instance of the class `System` by merging `water` with itself. The instance pointed to by `water` remains unchanged as the function `MergeByAtom` duplicates all the instance data.

Line 16 creates an instance of the class `Selection` whose purpose is to define the atoms that will be kept in the pruning operation of the following line. The argument to the `Selection` constructor is a list containing the indices of the oxygen atom and the first hydrogen atom in `water`. These are `0` and `1`, respectively, remembering that Python starts counting from zero.

Line 17 produces a hydroxyl group from `water` by using the `PruneByAtom` function and the `Selection` instance defined on the previous line.

Lines 7, 12 and 18 give labels to the systems that have been created.

Lines 8, 13 and 19 print summaries of the systems. The information printed is minimal as only the systems' atoms and labels have been defined.

2.4 Common molecular representations

The aim of this section is to explore a selection of the more common formats used for representing molecular species. The choice was made to show the variety of representations that exist but it is by no means exhaustive because of the many

different formats currently employed. All the formats described here are text-based so that they are human-readable. Most are also file-based in that they store their data in text files with either one or several molecular representations per file. Formats readable only by machine – *binary* formats – are also available and these will be more efficient if many or large molecular data sets are to be processed.

2.4.1 XYZ format

The XYZ format is one of the simplest and one of the most widely used. An example for water is:

```
1 3
2 Water - XYZ format.
3 O  0.00  0.00  0.0
4 H  0.58  0.75  0.0
5 H  0.58 -0.75  0.0
```

Line 1 lists the number of atoms in the molecule.
Line 2 contains a title. If there is no title the line is left blank.
Lines 3–5 are the lines that contain the atom data. The number of lines should equal the number given at the top of the file. Each line lists the elemental symbol or the atomic number of the atom and three floating-point numbers corresponding to its x, y and z coordinates.

An important point about XYZ format is that it is *free format*, not *fixed format*. In fixed format, *tokens*, either words or numbers, have to be placed in particular columns. In addition, floating-point numbers often have to be written with a certain number of figures after the decimal point. In contrast, there is no restriction on how tokens are written in free format as long as they can be distinguished from each other – usually by separating them with spaces. Most modern formats are free format.

2.4.2 MOL format

The MOL format is a fixed format and was introduced and is maintained by the company MDL Information Systems. An example for water is:

```
1 Water - MOL format.
2
3
4   3  2  0  0  0  0  0  0  0  0  0
```

```
5      0.0000    0.0000    0.0000 O   0  0  0  0  0  0
6      0.5800    0.7500    0.0000 H   0  0  0  0  0  0
7      0.5800   -0.7500    0.0000 H   0  0  0  0  0  0
8   1  2  1  0  0  0
9   1  3  1  0  0  0
10 M  END
```

The MOL format consists of two parts. *Lines 1–3* constitute the *header block* whereas *lines 4–10* define the *connection table* for the molecule.

Line 1 contains a title.

Lines 2–3 have been left blank although in the complete specification of the MOL format these lines may contain other, optional, information that we do not need here.

Line 4 defines various integer counters and options necessary for specification of the connection table. In all there are 11 fields each of which is three characters wide. The integers within each field are right-justified and the maximum integer that a field can hold is `999`. In Python syntax the line format would be written `11 * "%3d"`. The first two counters are the only ones that are needed in our example and they give the number of atom lines and the number of bond lines in the connection table, respectively. The remaining counters can be safely ignored although they should be present.

Lines 5–7 are the atom lines, each of which has a Python format of `3 * "%10.4f" + " %-3s%2d" + 5 * "%3d"`. The first three fields hold the x, y and z coordinates for the atom. Each coordinate field is 10 characters wide and contains a right-justified, floating-point number with four figures after the decimal point. The fourth field is the atom's elemental symbol and it is placed, left-justified, in columns 32–34 of the line. The remaining fields are integers whose values we do not need here.

Lines 8–9 are the bond lines with Python formats of `6 * "%3d"`. The first two fields give the integer indices of the atoms that are involved in the bond. Unlike in Python, counting starts at one, so that the first atom in the molecule has an index of `1`. The third field specifies the type of bond. A value of `1` corresponds to a single bond whereas values of `2`, `3` and `4` indicate double, triple and aromatic bonds, respectively. Thus, *line 8* says that there is a single bond between the first and second atoms in the molecule.

Line 10 terminates the file.

The MOL format specification allows the storage of multiple molecule definitions in the same file. This is done by concatenating the individual definitions

and separating them with a line containing the characters `$$$$`. MOL files of this type are called structure-data or SD files.

2.4.3 PDB format

Protein Data Bank or PDB format was designed for storing the structures of protein and nucleic acid polymers determined by experimental X-ray crystallographic and nuclear magnetic resonance (NMR) techniques. These structures are normally very complicated and involve thousands of atoms. Consider, as an example, the protein myoglobin which is illustrated schematically in Figure 2.2. The protein has a single chain comprising about 150 amino acids and a haem group. Both in vivo and in the crystal from which the structure is determined, the protein is bathed in a solution of water and counter-ions. The great majority of these molecules will not be in the experimentally determined structure but, nevertheless, a few solvent molecules are often present.

To cope with molecular systems of this complexity, the PDB format has a hierarchical classification scheme that consists of *chains*, *residues* and *atoms*. A chain is a single contiguous peptide, protein or nucleic acid strand and each chain in the system is given a unique, single-character name. A chain is itself subdivided into residues so that peptide and protein chains consist of sequences of amino acid residues, whereas nucleic acid chains comprise nucleotide residues. Each residue is identified by a three-character name and also by a code, most usually an integer, that gives its position in the sequence. Finally, residues are divided into atoms with each atom in the residue having a unique name of, at most, four characters.

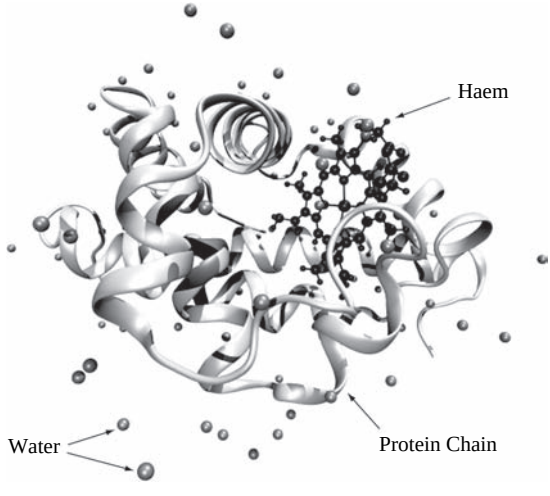


Fig. 2.2. A schematic of a protein, myoglobin.

The classification is actually slightly more complicated than this because residues can be of two types. First of all there are *standard* residues which comprise the twenty or so amino acids that occur in proteins and the five or six nucleotides that occur in ribo- and deoxyribo-nucleic acids. All other residues are *non-standard* or *hetero-atom* residues. Standard residues consist of standard atoms, or just atoms, whereas hetero-atom residues consist of hetero-atoms. Most residues in chains will be standard residues but there can be hetero-atom residues as well. In the case of myoglobin, for example, there will be a hetero-atom residue, representing a haem, in the protein chain. Most hetero-atom residues are, in fact, not classified into chains at all, but are lumped together into what can be considered as an unnamed 'hetero-atom chain'. These hetero-atom declarations are usually the last part of a system's specification and, most commonly, comprise solvent and counter-ion molecules.

The PDB format, like the MOL format, is a fixed format. Each line is a separate *record* and contains data of a particular type. The type of record is identified by the first few (usually six) characters of the line. The full PDB specification is a comprehensive one as it defines many records devoted to details of the experimental techniques used for determining the structure of the system in the file. These are unnecessary in many modeling studies and so are normally omitted.

The following illustrates a PDB file for water using the residue and atom names for the molecule that are defined in the PDB specification:

```
1 TITLE     WATER -  PDB FORMAT.
2 AUTHOR    GENERATED BY PDYNAMO 1.3
3 HETATM    1  O   HOH     1       0.000   0.000   0.000  0.00  0.00           O
4 HETATM    2 1H   HOH     1       0.580   0.750   0.000  0.00  0.00           H
5 HETATM    3 2H   HOH     1       0.580  -0.750   0.000  0.00  0.00           H
6 CONECT    1    2    3
7 MASTER        0    0    0    0    0    0    0    0    3    0    1    0
8 END
```

Line 1 contains a title. The first five characters of the line, `TITLE`, identify the type of line. The title itself, which is an arbitrary sequence of characters, starts in column 11 and can continue up to column 70.

Line 2 is of a similar format to *line 1* except that it specifies the author of the file. In this case, it is version 1.3 of the pDynamo program.

Lines 3–5 are the lines that contain the atom coordinate data. These records are hetero-atom records because water is not one of the standard amino acid or

nucleic acid residues that qualify for atom status. Nevertheless, `HETATM` and `ATOM` records have the same format except for the initial record identifier. In the example here, the following fields are present on each line:

Columns 7–11 contain the *serial number* of the atom – i.e. the order of the atom in the molecular sequence.

Columns 13–16 hold the atom name. In PDB format, each atom in a residue has a unique name which can consist of, at most, four characters. Atom names start in column 13 if the name is four characters long or if the atom is a hydrogen whose name starts with an integer. Otherwise, the name starts in column 14.

Columns 18–20 identify the name of the residue. `HOH` is the standard PDB name for water.

Columns 23–26 contain the *residue sequence number* which is the number that indicates the order of the residue in the system defined in the file. In this case, there is only a single residue in the file and so this number is `1`.

Columns 31–54 hold the x, y and z coordinates for the atom in ångströms. The format of the numbers in the field is `3 * "%8.3f"`.

Columns 55–66 define two floating-point numbers in `2 * "%6.2f"` format. They can be ignored here as they pertain to experiment-specific information.

Columns 77–78 contain the element symbol which is right-justified.

Line 6 is a record that identifies the *connections* in the system. The integer in columns 7–11 denotes the serial number of the first atom in the bond and the integers in columns 12–16 and 17–21 the serial numbers of the atoms to which it is bound. Thus, in this case, there are two bonds, between the oxygen and each hydrogen. Note that in contrast to MOL format there is no way of specifying the type of covalent bond that is present.

As well as the fields shown explicitly in *line 6*, there are extra fields in the record which can be used to specify additional covalent bonds and also hydrogen bonds and salt-bridge interactions.

Line 7 is the `MASTER` record. It is used for book-keeping and contains the numbers of particular types of record in the file. There are 12 counters in all. They start in column 11 and each occupies five spaces. In the case of water, all counters are zero except for the one in columns 51–55 which indicates the number of `ATOM` and `HETATM` records and the one in columns 61–65 which gives the number of `CONECT` records.

Line 8 denotes the end of the file.

2.4.4 CML format

Chemical markup language (CML) is a format based on *extensible markup language* (XML). XML is a language that is similar in many respects to *hypertext markup language* (HTML) and with which many readers will be familiar as being the language employed for writing web pages. The goals of XML and HTML, however, are different. HTML was designed for formatting text documents, much like a word processor, whereas XML was designed to provide a framework for organizing and representing arbitrary data. XML is not used directly. Instead, markup languages (MLs) are defined in terms of the XML specification to represent data coming from a particular domain.

CML is an ML that was developed for chemical applications by P. Murray-Rust, H. Rzepa and their collaborators. It comes in several flavours and is capable of representing a wide range of chemical data. The following example illustrates a CML file for water. It has been simplified somewhat by removing lines not directly concerned with the molecule and its atom and bond definitions.

```
1  <molecule title="Water - CML format.">
2    <atomArray>
3      <atom id="a1" elementType="O" x3="0"    y3="0"     z3="0"/>
4      <atom id="a2" elementType="H" x3="0.58" y3="0.75"  z3="0"/>
5      <atom id="a3" elementType="H" x3="0.58" y3="-0.75" z3="0"/>
6    </atomArray>
7    <bondArray>
8      <bond atomRefs2="a1 a2" order="1"/>
9      <bond atomRefs2="a1 a3" order="1"/>
10   </bondArray>
11 </molecule>
```

Lines 1 and 11 mark the beginning and the end of the molecule specification. In XML all data are expressed in terms of *elements*, so `molecule` is the CML element that represents a molecule. The beginning and the end of the element are given by *start* and *end tags* – lines 1 and 11, respectively – and the elements in between are its *child elements*. A starting tag is enclosed in angular brackets <...> whereas an end tag is enclosed by </...>. An element can be *empty*, in which case it is written as <.../>. In contrast to the MOL and PDB formats, XML-based formats are free format because the starting and stopping points of data blocks are explicitly indicated by the tag syntax.

Start tags and empty tags can contain *attributes* which are named properties of the element. Thus, `title` is an attribute of the `molecule` element and has the value `"Water - CML format."`.

Lines 2–6 define the atoms in the molecule. The atoms are grouped together in an `atomArray` element and are specified individually by `atom` elements. The `atom` elements are empty but they have attributes that give the atom's identifier, `id`, its elemental type, `elementType`, and its coordinates in three dimensions, `x3`, `y3` and `z3`.

Lines 7–10 define the bonds in the molecule using a syntax that is very similar to that of the atoms. The attributes of the `bond` elements are `atomRefs2` which lists the identifiers of the two atoms in the bond and `order` which gives the bond order. A bond order of `1` indicates a single bond.

CML is not the only markup language that pDynamo supports. Two others are pDynamoML and HTML. pDynamoML is pDynamo's own ML and it is used for reading and writing pDynamo objects to and from external files. These, by convention, are called XPK files. All pDynamo classes can be written to and from XPK files. In contrast, HTML is used when writing the output of pDynamo programs, thereby allowing the results of a calculation or a simulation to be viewed as a web page. Output in HTML format is achieved by using instances of an HTML-subclass of the `LogFileWriter` class that we shall meet in Section 3.4.

2.4.5 SMILES format

The SMILES (Simplified Molecular Input Line Entry System) format is an elegant and powerful one but it is different from the formats discussed previously because it aims to provide a way of compactly representing the composition of a molecular species as a string of characters. The architect of the format was D. Weininger and it is now maintained by the company Daylight Chemical Information Systems.

The full SMILES specification allows the representation of any molecular species and can describe such things as isotopic composition, isomerism and chirality. However, for many chemical species, and for the purposes of the examples in this book, the following subset of SMILES rules is sufficient:

- All atoms are represented by their elemental symbols enclosed in square brackets. Thus, `[C]` represents a carbon atom and `[Fe]` an iron atom.
- Hydrogens that are bound to an atom and the atom's nominal charge are specified inside the atom's square brackets. So, a hydroxide ion is written as `[OH-]` and a ferrous ion as `[Fe+2]`.
- The square brackets of atoms within the *organic subset* – B, C, N, O, F, P, S, Cl, Br and I – can be omitted if these atoms are uncharged and have standard valence – 1 for F, Cl, Br and I, 2 for O and S, 3 for B, N and P and 4 for C. Any empty valences that these atoms may have are assumed to be filled by bonding to hydrogen. This means

that the one-character SMILES `C`, `N` and `O` stand for methane, CH_4, ammonia, NH_3, and water, H_2O, respectively.

- Atoms adjacent to each other in a SMILES are bonded together. In the absence of a specific character, a single bond is assumed. Otherwise, the type of bond can be explicitly given using the symbols `-`, `=` and `#` for single, double and triple bonds, respectively. Therefore, `CC` and `C-C` both represent ethane (C_2H_6), `C=O` and `O=C` formaldehyde (H_2CO) and `C#C` ethyne (C_2H_2).

  Another useful symbol of this type, which we shall require later on, is the dot (`.`). It indicates that there is no bond or, in SMILES terminology, a *disconnection*. This means, for example, that `O.O` represents two water molecules.
- *Branching* in a structure is indicated by parentheses, so the SMILES `CC(=O)O` and `NC(C)C(=O)O` are those for ethanoic acid and for the amino acid alanine, respectively.
- *Ring closures* are indicated by pairs of matching digits. A SMILES for cyclohexane is `C1CCCCC1` as the two 'ones' indicate that the first and sixth carbons are bonded together. In contrast, the SMILES `C1CCCC1C` stands for methylcyclopentane as the first and fifth carbons are bonded together and the sixth carbon is outside the ring.
- The atoms of some elements, notably C, N and O, can be specified as being *aromatic* if their symbols are written in lower case. Aromatic atoms are always part of rings and they have less free valences for bonding to hydrogen than non-aromatic atoms because they donate electrons to the π-systems of which they are a part. Aromaticity is determined using *Hückel's rule* so that a ring, or a combination of fused rings, is taken to be aromatic if its constituent atoms donate $4n + 2$ electrons to the π-system, where n is a positive integer. Of course, an aromatic set of rings must consist uniquely of aromatic atoms.

  The definition of aromaticity is one of the more complicated aspects of SMILES, but for carbon it is relatively straightforward as each carbon can donate one electron to an aromatic system and has one less valence available for bonding to hydrogen. Valid aromatic SMILES are `c1ccccc1` for benzene, `c1cc2ccccc2cc1` for naphthalene and `c1c[cH-]cc1` for the cyclopentadienyl anion. An invalid aromatic SMILES is `c1ccc1` because the ring contains only four donated electrons, not the minimal six that would be needed.

It will quickly become apparent as readers try out SMILES for themselves that many different representations of the same chemical species are possible. In many applications, though, it is advantageous or even necessary to have a single representation of a species. Consider, for example, how much easier it is to search databases of molecules if each molecule has a unique identifier. An extension to SMILES, called unique SMILES, permits just this.

A project with aims that are, in many respects, similar to those of unique SMILES is the IUPAC (International Union of Pure and Applied Chemistry) chemical identifier or InChI project. This work was underway at the time of writing this book and so no details will be given here. It is probable, however, that InChI will be widely adopted in the future.

2.5 Example 2

pDynamo is capable of interpreting all the molecular representations outlined in the previous section and also a number of others. For the CML, MOL, PDB and XYZ formats, pDynamo defines separate classes for reading and writing data to external files. These classes are constructed in very much the same way as the example given in Section 1.3. Use of the classes directly allows fine control over how data are read from or written to a file but, for normal usage, it is simpler to employ the helper functions that pDynamo provides. There are two basic functions for each format, one for reading and one for writing. For the XYZ case they are:

Function `XYZFile_ToSystem`

Read a system from a file in XYZ format.

Usage: `system = XYZFile_ToSystem ( filename )`

`filename` is the name of the file to be read.

`system` is a new instance of the class `System` created using the data on the file.

Function `XYZFile_FromSystem`

Write a system to a file in XYZ format.

Usage: `XYZFile_FromSystem ( filename, system )`

`filename` is the name of the file to be written.

`system` is the system to be written.

The functions for the other formats are identical except that their names are changed accordingly.

SMILES stores its data in a string, unlike the other representations which use a file. Nevertheless, the basic way in which pDynamo handles the formats is similar. There are classes for reading and writing SMILES to and from strings and two helper functions that circumvent direct use of the SMILES classes. The helper function definitions are:

Function `SMILES_ToSystem`

Convert a SMILES into a system.

Usage: `system = SMILES_ToSystem ( smiles )`

`smiles` is a string containing the SMILES.

`system` is a new instance of the class `System` decoded from the SMILES string.

Function `SMILES_FromSystem`

Generate a SMILES representation of a system.

Usage: `smiles = SMILES_FromSystem ( system )`
`system` is the system whose SMILES is to be generated.
`smiles` is a string containing the SMILES.

It is convenient at this stage to introduce two functions that read and write XPK files, as files in XPK format are employed quite extensively in this book. The function definitions are:

Function `XMLPickle`

Write Python objects in XPK format to a file.

Usage: `XMLPickle ( filename, objects )`
`filename` is the name of the file to which data are to be written.
`objects` are the objects to be stored. They will be written to the file in the same fashion as they are passed to the function.

Function `XMLUnpickle`

Read Python objects from a file in XPK format.

Usage: `objects = XMLUnpickle ( filename )`
`filename` is the name of the file to be read.
`objects` are the Python objects stored on the file. They are reconstructed in the same way as they were written to the file.

We illustrate the use of some of these functions with a small molecule, *N*-methyl-alanyl-acetamide, which is shown in Figure 2.3. It is also referred to as blocked alanine (bALA) or as the alanine dipeptide. bALA is often used in modeling studies because it is relatively small but still sufficiently complex that it displays interesting behaviour. It has two peptide-like bonds, which are the bonds that link the consecutive amino acids in a protein, and so it is often employed as a model of protein systems. bALA has a number of distinct conformations and we shall meet several of them in the course of the book. In this example, though, we require only the C7 equatorial (C7eq) conformation.

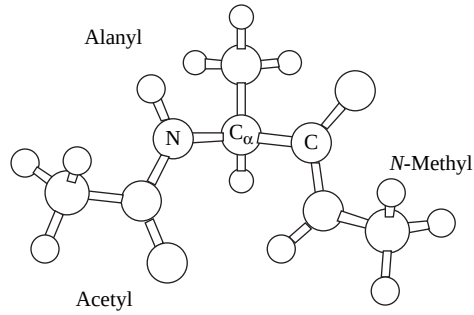


Fig. 2.3. *N*-Methyl-alanyl-acetamide.

The aim of the example in this section is to make use of the functions defined above. The program creates different representations of the bALA molecule by reading files in various formats and by converting a SMILES string. Summaries of each of the representations are then printed. The program is:

```
1  """Example 2."""
2
3  from Definitions import *
4
5  # . Initialize a list to contain the molecules.
6  molecules = []
7
8  # . Read all molecules.
9  molecules.append ( \
       MOLFile_ToSystem ( os.path.join ( molpath, "bala_c7eq.mol" ) ) )
10 molecules.append ( \
       PDBFile_ToSystem ( os.path.join ( pdbpath, "bala_c7eq.pdb" ) ) )
11 molecules.append ( \
       XYZFile_ToSystem ( os.path.join ( xyzpath, "bala_c7eq.xyz" ) ) )
12
13 # . Generate a molecule from a SMILES string.
14 molecules.append ( SMILES_ToSystem ( "CC(=O)NC(C)C(=O)NC" ) )
15
16 # . Print summaries of the molecules.
17 for molecule in molecules:
18     molecule.Summary ( )
```

Line 3 imports items from the module `Definitions`. In this case, definitions of the four `ToSystem` functions are required as well as those for the variables `molpath`, `pdbpath` and `xyzpath` which indicate where the data files of different format are located.

Line 6 defines an empty Python list called `molecules` which will be used to hold the different molecular representations.

Lines 9–11 read three different files each of which contains a representation of the bALA molecule. The files are read using the helper functions discussed earlier in the section and each file returns a new instance of the class `System`. These instances are added directly to the list `molecules` using the `append` method of the Python list class.

The files to be read are stored in directories whose names are given by the variables `molpath`, `pdbpath` and `xyzpath` in the module `Definitions`. The names of the individual data files are joined to the names of the directories using the function `join` from the standard Python module `os.path`. The reason for this is that different computer operating systems use different conventions for file names but `join` will use the convention that is correct for the machine upon which the program is running.

Line 14 generates a fourth representation of the bALA molecule by converting an appropriate SMILES string.

Lines 17–18 iterate over the molecules in the list `molecules` and print a summary of each one.

Exercises

2.1 Taking Example 2 as reference, read and write molecular species in different formats. What happens when one starts and finishes with a particular format but in between passes through formats of other kinds? Are the beginning and ending representations the same? Is information conserved?

2.2 Experiment with the SMILES language. Try, for example, writing aromatic and non-aromatic SMILES for various molecules (e.g. toluene). What happens when systems created from these representations are transformed into PDB or MOL format?

3

Coordinates and coordinate manipulations

3.1 Introduction

In the last chapter we explored various ways of specifying the composition of a molecular system. Many of these representations contained not only information about the number and type of atoms in the system but also the atoms' coordinates, which are an essential element for most molecular simulation studies. Given the nature of the system's atoms and their coordinates the *molecular structure* of the system is known and it is possible to deduce information about the system's physical properties and its chemistry. The generation of sets of coordinates for particular systems is the major goal of a number of important experimental techniques, including X-ray crystallography and NMR spectroscopy, and there are data banks, such as the PDB and the Cambridge Structural Database (CSD), that act as repositories for the coordinate sets of molecular systems obtained by such methods.

There are several alternative coordinate systems that can be used to define the atom positions. For the most part in this book, *Cartesian coordinates* are employed. These give the absolute position of an atom in three-dimensional space in terms of its x, y and z coordinates. Other schemes include *crystallographic coordinates* in which the atom positions are given in a coordinate system that is based upon the crystallographic symmetry of the system and *internal coordinates* that define the position of an atom by its position relative to a number of other atoms (usually three).

The aim of this chapter is to describe the various ways in which coordinates can be analysed and manipulated. Because numerous analyses can be performed on a set of coordinates, only a sampling of some of the more common ones will be covered here.

3.2 Connectivity

One of the most important properties that a chemical system possesses is the number and type of *bonds* between its atoms. A rigorous determination of the bonding pattern would require an analysis of the electron density of a system

or of its geometry, in combination with a database of chemical information that has the bonding behaviours of each element in different molecular environments. Once the bonding arrangement is known, it is straightforward to determine other aspects of the *connectivity* of a system, such as its *bond angles* and its *torsion* or *dihedral angles*.

In this section the bonding pattern of a system is not determined using a database or the electron density, although this will be done in later chapters. Instead a less rigorous approach is adopted that is based upon a simple search of the distances between atoms and produces a list of probable bonds.

If $\boldsymbol{r}_i$ is the three-dimensional vector containing the x, y and z Cartesian coordinates of atom i, then the distance, r_{ij}, between two atoms, i and j, can be written as

$$\begin{aligned} r_{ij} &= \left|\boldsymbol{r}_i - \boldsymbol{r}_j\right| \\ &= \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2} \end{aligned} \tag{3.1}$$

The algorithm for finding bonds works by calculating the distances between two atoms in a system and then checking to see whether they are less than a certain maximum *bonding distance* apart. The bonding distance is determined as the sum of radii that are typical 'bonding' radii for each element and a *safety factor*, which is an empirical parameter that can be set by the user. The bonding radii are also empirical parameters that have been derived from the covalent, van der Waals and ionic radii for the elements and have been shown to give reasonable results in this application.

The simplest implementation of the algorithm would calculate the distances between all possible pairs of atoms and perform the distance comparison for each pair. This is sufficient for systems with small numbers of particles but becomes prohibitively expensive as the number increases. To see this, assume that the system contains N atoms. The number of possible pairs of atoms in the system can then be calculated with the following procedure. The first atom can pair with all the atoms from atom 2 to atom N giving $N-1$ pairs. The second atom can pair with atom 1 and with the atoms from atom 3 to atom N. However, the pair with atom 1 has already been counted so only the pairs with atoms 3 to N are new ones. The procedure can be continued for all the atoms up until atom N, which contributes no new pairs. The total number of pairs, N_{pair}, is thus

$$\begin{aligned} N_{\text{pair}} &= (N-1) + (N-2) + \cdots + 1 + 0 \\ &= \sum_{i=1}^{N} (N-i) \\ &= \frac{1}{2} N(N-1) \end{aligned} \tag{3.2}$$

This says that the number of pairs is approximately equal to the square of the number of particles. To denote this it is common to use the notation $N_{\text{pair}} \simeq O(N^2)$, which means that N_{pair} is *of the order of* the square of N. The reason for the expense of the search for large N should now be apparent because the number of distances to be calculated increases roughly as the square of N. So, for a small system of 100 atoms the number of pairs is $O(10^4)$, whereas for a system 100 times larger (10 000 atoms) the number of pairs is $O(10^8)$ or 10^4 times greater.

The problem of how the effort needed for an algorithm scales with increasing size is ubiquitous in all areas of computational science, not only molecular modeling, and it is one that we shall meet repeatedly. The aim when designing any algorithm is to develop one with as low a scaling behaviour as possible and ideally with *linear scaling*, $O(N)$, or less.

There are several possible approaches that can be used to achieve linear scaling for the bond-finding algorithm. The one that we use is as follows.

(i) Determine a *bounding box* for the system which is the smallest rectangular box that will enclose every atom. As most molecular systems have relatively homogeneous distributions of atoms, the volume of a bounding box will scale linearly with the number of atoms in the system.

This step takes $O(N)$ operations as it entails finding the minimum and maximum values of the Cartesian coordinates of the atoms in each of the x, y and z directions.

(ii) Create a grid that fills the bounding box. This is done by choosing an origin for the grid, such as the lower left-hand corner or the centre of the bounding box, and a grid spacing in each direction. If $\boldsymbol{r}_{\text{o}}$ is the position of the grid origin and δ_x, δ_y and δ_z are the grid spacings, the position of a grid point in the box is given by $\boldsymbol{r}_{\text{o}} + (i\delta_x, j\delta_y, k\delta_z)$ where i, j and k are integers that label the grid point.

This step scales as $O(N)$ because the number of grid points is proportional to the volume of the bounding box and, hence, to the system size. In practice the cost is much less because the coordinates of the grid points need never be explicitly calculated and can be determined, if required, directly from the previous formula.

(iii) Assign atoms to grid points by finding the grid point that falls nearest to the atom. This step scales as $O(N)$ because finding the nearest grid point takes only a small, constant number of arithmetic operations per atom.

(iv) Construct the bond lists for the atoms using a two-step procedure. First, a list of potential bonding partners is obtained for each atom by searching over neighbouring grid points that are in bonding range and, second, the list of potential partners is refined by calculating the distances between the atoms on the list and accepting only those that are close enough.

This step is the most expensive in the algorithm but it too is linear scaling. This is because the number of neighbouring points over which it is necessary to search is approximately constant for each atom – it depends only on the bonding range, a quantity that is independent of system size. It should now be clear why the grid was

constructed. Its purpose is to provide a way in which atoms within a particular volume of space can be rapidly identified.

Although this algorithm is a linear scaling one, it will be less efficient than the straightforward $O(N^2)$ distance search for small systems (such as bALA) because there is an *overhead* cost associated with the initial grid-construction steps of the algorithm. This illustrates another important principle of algorithm design – that for differently sized systems different algorithms will be optimal. This is shown schematically in Figure 3.1. Although a particular algorithm, algorithm 1, may exhibit worse scaling properties with system size than does another algorithm, algorithm 2, the extra overhead costs associated with algorithm 2 make it less efficient for smaller systems. It is only when a certain *critical system size* is reached that algorithm 2 will, in fact, become faster.

Once the bonds in a system are known, the generation of other connectivity-related information is possible. In particular, it is straightforward to obtain lists of the bond angles and dihedral angles using algorithms whose costs scale linearly with system size. The `System` class contains methods that generate the connectivity and also attributes that hold the bond, angle and dihedral information. Their definitions are:

Class `System`

Connectivity methods and attributes.

Method `BondsFromCoordinates`

Construct the bonds for a system using a distance-search algorithm.

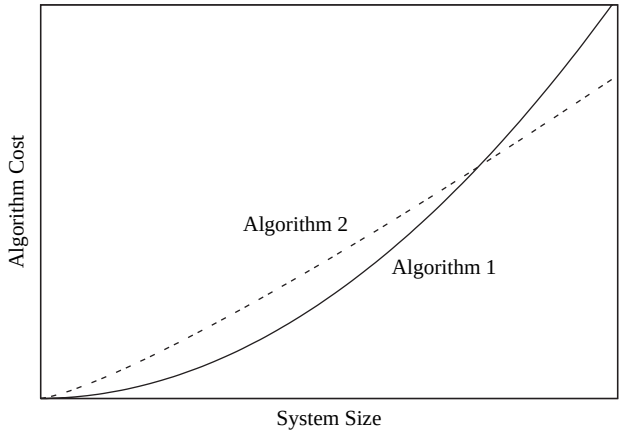


Fig. 3.1. The scaling properties of two algorithms, 1 and 2, versus the system's size.

Usage: `system.BondsFromCoordinates ( safety = 0.45 )`

`safety` defines a distance that is added to the sum of the atomic bonding radii when determining whether two atoms are bound. This argument is a *keyword argument* as opposed to the *positional arguments* that we have met up to now. Keyword arguments have three distinguishing features: (i) they occur after any positional argument in an argument list; (ii) they are passed into a function or method using the `keyword = value` syntax; and (iii) they are optional and, if absent, will be assigned a default value that is specified in the function or method definition. In this case, the default value for the argument `safety` is `0.45`.

`system` is the instance of `System` for which the bonds are to be generated.

Remarks: The method employs standard, elemental radii to determine bonding distances between atoms and uses coordinates from the `coordinates3` attribute of `system` to perform the search. The bonds generated by the method are stored in the `bonds` attribute of the instance. The method also automatically generates other connectivity information, including angles and dihedrals, once the bonds are known.

Attributes

`bonds` are the bonds in the system.

`angles` are the angles in the system.

`dihedrals` are the dihedrals in the system.

Remarks: These attributes contain lists of instances of the `Bond`, `Angle` and `Dihedral` classes that will be defined in the next section.

3.3 Internal coordinates

The lists of bonds, angles and dihedrals are a start in the analysis of a system's chemical structure but they define only its overall connectivity, which will be the same for all systems in the same chemical state. To investigate differences between the structures of systems with identical connectivities it is important to calculate the values of their internal coordinates. The three principal internal coordinates are bond lengths, bond angles and dihedral angles, which are illustrated in Figure 3.2.

The distance between two atoms has already been defined in Equation (3.1). The angle, θ_{ijk}, subtended by three atoms i, j and k is calculated from

$$\theta_{ijk} = \arccos\left(\hat{\boldsymbol{r}}_{ij}^{\mathrm{T}}\hat{\boldsymbol{r}}_{kj}\right) \tag{3.3}$$

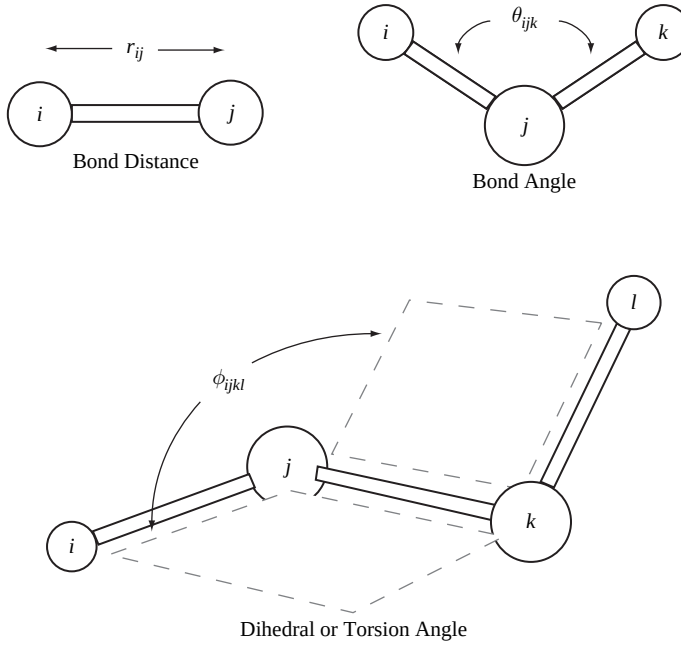


Fig. 3.2. The three principal types of internal coordinate.

where the difference vectors are defined as

$$\boldsymbol{r}_{ij} = \boldsymbol{r}_i - \boldsymbol{r}_j \tag{3.4}$$

and the hat over a vector indicates that it is *normalized*, i.e. $\hat{\boldsymbol{r}} = \boldsymbol{r}/r$.

The definition of a dihedral angle, ϕ_{ijkl}, is slightly more ambiguous because several exist. The one used here is

$$\phi_{ijkl} = \pm \arccos\left(\hat{\boldsymbol{a}}^{\mathrm{T}}\hat{\boldsymbol{b}}\right) \tag{3.5}$$

where the vectors $\boldsymbol{a}$ and $\boldsymbol{b}$ are defined as

$$\boldsymbol{a} = \boldsymbol{r}_{ij} - (\boldsymbol{r}_{ij}^{\mathrm{T}}\hat{\boldsymbol{r}}_{kj})\boldsymbol{r}_{kj} \tag{3.6}$$

$$\boldsymbol{b} = \boldsymbol{r}_{lk} - (\boldsymbol{r}_{lk}^{\mathrm{T}}\hat{\boldsymbol{r}}_{kj})\boldsymbol{r}_{kj} \tag{3.7}$$

The sign of the dihedral (i.e. whether to use $+$ or $-$ in Equation (3.5)) is the same as the sign of the scalar quantity, $-\boldsymbol{r}_{ij}^{\mathrm{T}}(\boldsymbol{r}_{kj} \wedge \boldsymbol{r}_{lk})$. A dihedral angle of 0° indicates that all the atoms are in the same plane in a *cis* conformation, whereas angles of 180° or −180° mean that the atoms are coplanar in a *trans* conformation.

pDynamo stores information about bonds, angles and dihedrals in specific classes. A minimal definition that is sufficient for the purposes of this book is as follows:

Class `Bond`

A class to represent a chemical bond.

Attributes

`i` is the index of the first atom in the bond.
`j` is the index of the second atom in the bond.

Class `Angle`

A class to represent a bond angle.

Attributes

`i` is the index of the first atom in the angle.
`j` is the index of the second atom in the angle.
`k` is the index of the third atom in the angle.

Class `Dihedral`

A class to represent a dihedral or torsion angle.

Attributes

`i` is the index of the first atom in the dihedral.
`j` is the index of the second atom in the dihedral.
`k` is the index of the third atom in the dihedral.
`l` is the index of the fourth atom in the dihedral.

The three-dimensional Cartesian coordinates of a molecular system are stored in instances of the `Coordinates3` class. An initial definition which includes methods for determining distances, angles and dihedral angles is:

Class `Coordinates3`

A class for holding sets of Cartesian coordinates in three dimensions.

Method `Angle`

Calculate an angle given the indices of three points.

Usage:	`angle = coordinates3.Angle ( i, j, k )`
`i, j, k`	are the indices of the points in the angle.
`coordinates3`	is the instance of `Coordinates3` for which the angle is to be calculated.
`angle`	is the value of the calculated angle in degrees.

Method `Dihedral`

Calculate a dihedral given the indices of four points.

Usage:	`dihedral = coordinates3.Dihedral ( i, j, k, l )`
`i, j, k, l`	are the indices of the points in the dihedral.
`coordinates3`	is the instance of `Coordinates3` for which the dihedral is to be calculated.
`dihedral`	is the value of the calculated dihedral in degrees.

Method `Distance`

Calculate a distance given the indices of two points.

Usage:	`distance = coordinates3.Distance ( i, j )`
`i, j`	are the indices of the points in the distance.
`coordinates3`	is the instance of `Coordinates3` for which the distance is to be calculated.
`distance`	is the value of the calculated distance in the same units as those of the coordinates (usually ångströms).

3.4 Example 3

The example program in this section uses the classes and methods described in Sections 3.2 and 3.3 to analyse the structure of blocked alanine. The program is:

```
1 """Example 3."""
2
3 from Definitions import *
4
5 # . Read in a system.
6 molecule = XYZFile_ToSystem ( \
            os.path.join ( xyzpath, "bala_c7eq.xyz" ) )
```

```
7 molecule.Summary ( )
8
9 # . Define a table for the results.
10 table = logfile.GetTable ( columns = [ 15, 15 ] )
11 table.Start ( )
12 table.Title ( "Bond Analysis" )
13 table.Heading ( "Safety Factor" )
14 table.Heading ( "Bonds Found"   )
15
16 # . Loop over the buffer sizes.
17 for i in range ( 21 ):
18     safety = 0.1 * float ( i )
19     molecule.BondsFromCoordinates ( safety = safety )
20     table.Entry ( "%4.1f" % ( safety, ) )
21     table.Entry ( `len ( molecule.bonds )`, )
22
23 # . Finish up the table.
24 table.Stop ( )
25
26 # . Generate the bonds with the default safety factor.
27 molecule.BondsFromCoordinates ( safety = 0.5 )
28 molecule.Summary ( )
29
30 # . Print the bonds.
31 table = logfile.GetTable ( columns = 4 * [ 5, 5, 10 ] )
32 table.Start ( )
33 table.Title ( "Bond Lengths (Angstroms)" )
34 for bond in molecule.bonds:
35     table.Entry ( `bond.i` )
36     table.Entry ( `bond.j` )
37     table.Entry ( "%6.3f" % ( molecule.coordinates3.Distance \
                                ( bond.i, bond.j ), ) )
38 table.Stop ( )
39
40 # . Print the angles.
41 table = logfile.GetTable ( columns = 3 * [ 5, 5, 5, 10 ] )
42 table.Start ( )
43 table.Title ( "Angles (Degrees)" )
44 for angle in molecule.angles:
45     table.Entry ( `angle.i` )
46     table.Entry ( `angle.j` )
47     table.Entry ( `angle.k` )
48     table.Entry ( "%6.1f" % ( molecule.coordinates3.Angle \
                                ( angle.i, angle.j, angle.k ), ) )
```

```
49 table.Stop ( )
50
51 # . Print the dihedrals.
52 table = logfile.GetTable ( columns = 4 * [ 5, 5, 5, 5, 10 ] )
53 table.Start ( )
54 table.Title ( "Dihedrals (Degrees)" )
55 for dihedral in molecule.dihedrals:
56     table.Entry ( ‘dihedral.i‘ )
57     table.Entry ( ‘dihedral.j‘ )
58     table.Entry ( ‘dihedral.k‘ )
59     table.Entry ( ‘dihedral.l‘ )
60     table.Entry ( "%6.1f" % ( molecule.coordinates3.Dihedral \
                                ( dihedral.i, dihedral.j, \
                                  dihedral.k, dihedral.l ), ) )
61 table.Stop ( )
```

There are two parts to this program. In the first part (*lines 1–24*), bonds are generated using the algorithm of Section 3.2 with different values of the safety factor. In the second part (*lines 26–61*), the connectivity for the molecule is generated with a ‘reasonable’ value of the safety factor and the values of the internal coordinates printed out.

Lines 6–7 create `molecule` whose connectivity is to be analysed.

Line 10 defines a table to which the results of the first part of the program will be output. In the pDynamo library, the Python `print` keyword is never used directly. Instead all output is handled through instances of subclasses of the class `LogFileWriter`. One of the principal reasons why this is done is that it allows output to occur in different formats depending upon the subclass of `LogFileWriter` that is being used. Thus, for example, output can occur in regular text format by using the class `TextLogFileWriter` or in HTML format using the class `XHTMLLogFileWriter`. In this book, all output is to the object `logfile` which is an instance of the `TextLogFileWriter` class. `logfile` is predefined by the pDynamo library and is the default destination for output if no other instance of `LogFileWriter` is specified.

Line 10 uses the `GetTable` method of `logfile` to create `table` which is an instance of the `TextTable` class. The argument to the method defines the number of columns and their widths – in this case, two columns each 15 characters wide.

Lines 11–14 activate writing to the table and output a title and the headings for each column.

Lines 17–21 generate bonds for `molecule` using the `BondsFrom Coordinates` method and values of the safety factor that range in 0.1 Å increments from 0 to 2 Å. From the output file (which is not given here) it will be seen that the number of bonds is constant (at 21) with safety factors from 0.2 to 0.8 Å. With smaller values the number of bonds is less and with larger values it is bigger. This is typical behaviour for systems (with well-defined coordinate sets!) that contain primarily non-metal atoms – a safety factor of about 0.5 Å is usually a good compromise value.

Output of the results is performed using the `Entry` method of `table` which takes a single, string argument. *Line 20* outputs the value of the safety factor and *line 21* the number of bonds. The latter is obtained by using the Python built-in function `len` which returns the length of the `bonds` attribute of `molecule` as an integer. Python then permits a string representation of the integer (or indeed of any object) to be obtained by enclosing the whole expression in backward quotes ‘...‘.

Line 24 comes after the loop and terminates writing to the table.

Lines 27–28 regenerate the connectivity for `molecule` with a value of 0.5 Å for the safety factor.

Lines 31–38 print the bond distances for `molecule` to a table. The way in which output is done is very similar to that in *lines 10–21* except that the table has 12 columns and no column headings.

Line 34 loops over the bonds in `molecule.bonds` by extracting them one by one as instances of the `Bond` class. The indices of the atoms in the bond are then output to the table in *lines 35–36* and the bond distance in *line 37*. The bond distance is calculated using the `Distance` method of the `coordinates3` attribute of `molecule`.

Lines 41–49 print a table of bond angles for `molecule`.

Lines 52–61 print a table of dihedral angles for `molecule`.

3.5 Miscellaneous transformations

The previous sections presented analyses that involve the positions of atoms relative to each other. Often, however, it is useful to be able to manipulate the Cartesian coordinates themselves, either for part or the whole of a system. This section presents a number of simple transformations of this type and provides a necessary preliminary to the more complex operations of the next section.

The most fundamental and probably the most useful transformations are those that rotate and translate the atom coordinates. A translation of a coordinate is

effected by adding a vector that specifies the translation, $\boldsymbol{t}$, to the coordinate vectors of the atoms, i.e.

$$\boldsymbol{r}_i' = \boldsymbol{r}_i + \boldsymbol{t} \tag{3.8}$$

where the prime denotes a modified coordinate. A rotation involves the multiplication of the coordinate vector of each atom by a 3×3 matrix, $\mathbf{U}$, which specifies the rotation:

$$\boldsymbol{r}_i' = \mathbf{U}\boldsymbol{r}_i \tag{3.9}$$

Rotation matrices are *orthogonal matrices*, which means that the inverse of the matrix is also its transpose, i.e. $\mathbf{U}^{\mathrm{T}}\mathbf{U} = \mathbf{I}$, where $\mathbf{I}$ is the identity matrix. If the matrix has a determinant of 1 ($\|\mathbf{U}\| = 1$), the rotation is said to be a *proper rotation* and the handedness of the coordinate system is preserved. If the matrix has a determinant of -1 ($\|\mathbf{U}\| = -1$), the rotation is an *improper rotation* and involves a mirror reflection or an inversion and can lead to changes in the stereochemistry of the system.

In many cases a coordinate set for a system will be centred and oriented arbitrarily. It is often useful to be able to remove these effects and to orient the system in a systematic fashion. A standard way to centre a system is to translate it so that it is centred at the origin. The position vector of the centre of a system, $\boldsymbol{R}_{\mathrm{c}}$, is defined by

$$\boldsymbol{R}_{\mathrm{c}} = \frac{\sum_{i=1}^{N} w_i \boldsymbol{r}_i}{\sum_{i=1}^{N} w_i} \tag{3.10}$$

where w_i are *weights* associated with each particle. For the *centre of mass*, these are the atomic masses, but it is possible to employ other values. For example, if the weights are all equal to 1 then the *centre of geometry* is calculated, whereas if the weights are the elemental nuclear charges, it is the *centre of charge* that is obtained. To centre a system at the origin it is necessary to calculate $\boldsymbol{R}_{\mathrm{c}}$ and translate the coordinates of all atoms by $-\boldsymbol{R}_{\mathrm{c}}$ (using Equation (3.8)).

To orient a system in a systematic fashion it is common to use a *principal axis transformation* that rotates it so that the off-diagonal elements of its *inertia matrix* are zero. The inertia matrix, $\boldsymbol{\mathcal{I}}$, is a 3×3 matrix that is defined as

$$\boldsymbol{\mathcal{I}} = \sum_{i=1}^{N} w_i \left(r_i^2 \mathbf{I} - \boldsymbol{r}_i \boldsymbol{r}_i^{\mathrm{T}} \right) \tag{3.11}$$

where $\mathbf{I}$ is the 3×3 identity matrix. The inertia matrix is *symmetric*, which means that it is equal to its own transpose, i.e. $\boldsymbol{\mathcal{I}} = \boldsymbol{\mathcal{I}}^T$. This can be seen explicitly by writing out the individual components of the matrix, such as

$$\begin{aligned}\mathcal{I}_{xx} &= \sum_{i=1}^{N} w_i \left(r_i^2 - x_i^2\right) \\ \mathcal{I}_{xy} &= -\sum_{i=1}^{N} w_i x_i y_i \\ &= \mathcal{I}_{yx}\end{aligned} \tag{3.12}$$

As before, the weights for the system can be the masses for each atom or other values depending upon the property that is being studied.

The *moments of inertia* and the *principal axes* of the system are the *eigenvalues*, $\mathcal{I}_\alpha$, and *eigenvectors*, $\boldsymbol{e}_\alpha$, of the inertia matrix. They satisfy the following equation

$$\boldsymbol{\mathcal{I}}\boldsymbol{e}_\alpha = \mathcal{I}_\alpha \boldsymbol{e}_\alpha \qquad \alpha = 1,\ 2 \text{ or } 3 \tag{3.13}$$

and are obtained by *diagonalizing* the inertia matrix. More details concerning eigenvalues and eigenvectors can be found in Section A2.1. The inertia matrix and the moments of inertia are important properties because they help characterize the behaviour of a system under rotational motion.

The principal axis transformation is the one that makes the inertia matrix diagonal and converts all the diagonal elements of the matrix to the moments of inertia. This is achieved by rotating the coordinates of the atoms of the system with a rotation matrix, **U**, which is equal to the transpose of the matrix of eigenvectors:

$$\mathbf{U} = (\boldsymbol{e}_1, \boldsymbol{e}_2, \boldsymbol{e}_3)^{\mathrm{T}} \tag{3.14}$$

All the transformations discussed above are implemented as methods of the `Coordinates3` class. Their definitions are:

Class `Coordinates3`

Methods for transforming coordinates.

Method `Center`

Determine a centre for the coordinates.

Usage: `center = coordinates3.Center ( )`

`coordinates3` is the instance of `Coordinates3` for which the centre is to be calculated.

`center` is an instance of the class `Vector3` that contains the coordinates of the centre.

Method `InertiaMatrix`

Determine the inertia matrix for the coordinates.

Usage: `matrix = coordinates3.InertiaMatrix ( )`
`coordinates3` is an instance of `Coordinates3`.
`matrix` is an instance of the class `Matrix33` that contains the matrix of inertia.

Method `Rotate`

Rotate the coordinates given a rotation matrix.

Usage: `coordinates3.Rotate ( rotation )`
`coordinates3` is an instance of `Coordinates3`.
`rotation` is an instance of the class `Matrix33` that contains a rotation matrix.

Method `ToPrincipalAxes`

Perform a principal axis transformation on a set of coordinates.

Usage: `coordinates3.ToPrincipalAxes ( )`
`coordinates3` is an instance of `Coordinates3`.

Method `Translate`

Translate the coordinates given a translation.

Usage: `coordinates3.Translate ( translation )`
`coordinates3` is an instance of `Coordinates3`.
`translation` is an instance of the class `Vector3` that contains a translation.

Method `TranslateToCenter`

Translate a set of coordinates to its centre.

Usage: `coordinates3.TranslateToCenter ( )`
`coordinates3` is an instance of `Coordinates3`.

All the methods listed above can take a keyword argument `selection`. If the argument is present, it must be an instance of the `Selection` class and

it specifies the subset of coordinates upon which the particular transformation is to be effected. If the argument is absent, all coordinates will be transformed. In addition, all methods, except `Rotate` and `Translate`, have a keyword argument `weights` for defining the coordinate weights, w_i. If absent, values of 1 are assumed for each weight, whereas, if present, the argument must be a vector of floating-point numbers with the same length as the coordinate array. In pDynamo, vectors of this type are implemented with the class `Vector`. Weights arrays can be constructed explicitly using the methods of the `Vector` class but more usually they will be obtained using the methods of other classes. One of the more useful of these is given in the example in Section 3.7.

3.6 Superimposing structures

The transformations introduced in the previous section modified a single set of coordinates. In many cases, though, it is necessary to compare the structures defined by two or more sets of coordinates. It is possible, of course, to compare two structures by comparing the values of their internal coordinates. This gives useful information but can be cumbersome when the system and, hence, the number of internal coordinates are large. A simpler and widely used measure of the difference between two structures, I and J, is the RMS coordinate deviation, σ_{IJ}. It provides a quicker, albeit cruder, measure insofar as it is a single number. It is defined as

$$\sigma_{\mathrm{IJ}} = \sqrt{\frac{\sum_{i=1}^{N} w_i \left(\boldsymbol{r}_i^{\mathrm{I}} - \boldsymbol{r}_i^{\mathrm{J}}\right)^2}{\sum_{i=1}^{N} w_i}} \tag{3.15}$$

where the superscripts I and J refer to the coordinates of the first and second structures, respectively.

A moment's consideration shows that the RMS coordinate deviation will not be a useful measure for the comparison of structures unless the coordinate sets are somehow oriented with respect to each other. It is evident, for example, that if one set has undergone a translation with respect to the other then the RMS coordinate deviation can take any value. A possible solution is to orient the two sets of coordinates separately using a principal axis transformation and then calculate the RMS coordinate deviation between them. This can provide a satisfactory measure of comparison in some circumstances. However, a better method is to choose one set of coordinates as a reference structure and then find the transformation that superimposes the other coordinate set upon it.

There are various methods for superimposing structures. One of the original ones was developed by W. Kabsch but a more recent one that uses *quaternions* has been proposed by G. Kneller. The algorithm works by initially translating

the coordinate set of the structure to be moved so that it has the same centre as the reference set. The first structure is then rotated so that its RMS coordinate deviation from the reference structure is as small as possible. If **U** is the rotation matrix that operates upon the first structure, the RMS coordinate deviation between it and the reference structure is a function of **U** and is written as

$$\sigma_{\mathrm{IJ}}^2(\mathbf{U}) \propto \sum_{i=1}^{N} w_i \left(\boldsymbol{r}_i^{\mathrm{I}} - \mathbf{U}\boldsymbol{r}_i^{\mathrm{J}}\right)^2 \tag{3.16}$$

The **U** that produces the smallest deviation is found by *minimizing* the value of Equation (3.16) with respect to various parameters that define the rotation matrix. In the quaternion algorithm, the rotation matrix is expressed in terms of four parameters q_0, q_1, q_2 and q_3 as follows:

$$\mathbf{U} = \begin{pmatrix} q_0^2+q_1^2-q_2^2-q_3^2 & 2(-q_0q_3+q_1q_2) & 2(q_0q_2+q_1q_3) \\ 2(q_0q_3+q_1q_2) & q_0^2-q_1^2+q_2^2-q_3^2 & 2(-q_0q_1+q_2q_3) \\ 2(-q_0q_2+q_1q_3) & 2(q_0q_1+q_2q_3) & q_0^2-q_1^2-q_2^2+q_3^2 \end{pmatrix} \tag{3.17}$$

The matrix defined in Equation (3.17) automatically defines an orthogonal matrix as long as the quaternion parameters satisfy the expression $q_0^2 + q_1^2 + q_2^2 + q_3^2 = 1$. The minimization of Equation (3.16) is done using standard minimization approaches and the normalization condition is imposed using the method of *Lagrange multipliers*. Further examples of minimization problems can be found in Sections 4.5 and 7.3, whereas Lagrange multipliers are discussed more fully in Section A2.2.

The calculation of the RMS coordinate deviation and the superposition of two coordinate sets are handled by two methods of the class `Coordinates3`. Their definitions are:

Class `Coordinates3`

Methods for determining the RMS deviation between coordinates sets and for their superposition.

Method `RMSDeviation`

Determine the RMS deviation between two sets of coordinates.

Usage: `rmsd = coordinates3.RMSDeviation( reference )`

`coordinates3` is an instance of `Coordinates3`.

`reference` is a reference set of coordinates with respect to which the RMS deviation is to be calculated. It must be of the same size as the instance calling the method.

`rmsd` is a floating-point number containing the value of the RMS deviation.
Remarks: Both sets of coordinates are unchanged after the operation.

Method `Superimpose`

Superimpose a set of coordinates onto a reference set.

Usage: `coordinates3.Superimpose ( reference )`
`coordinates3` is an instance of `Coordinates3`.
`reference` is a reference set of coordinates with respect to which the instance that calls the method will be reoriented.
Remarks: The quaternion algorithm is used by this method.

Both of these methods have keyword arguments `selection` and `weights` that behave identically to those of the methods described in Section 3.5.

3.7 Example 4

The program in this section illustrates the use of the transformation and superimposition methods. It is:

```
"""Example 4."""

from Definitions import *

# . Define the list of structures.
xyzfiles = [ "bala_alpha.xyz", "bala_c5.xyz", \
             "bala_c7ax.xyz", "bala_c7eq.xyz" ]

# . Define a molecule.
xyzfile  = xyzfiles.pop ( )
molecule = XYZFile_ToSystem ( os.path.join ( xyzpath, xyzfile ) )
molecule.Summary ( )

# . Translate the system to its center of mass.
masses = molecule.atoms.GetItemAttributes ( "mass" )
molecule.coordinates3.TranslateToCenter ( weights = masses )

# . Calculate and print the inertia matrix before reorientation.
inertia = molecule.coordinates3.InertiaMatrix ( weights = masses )
inertia.Print ( title = "Inertia Matrix Before Reorientation" )

```

```
21 # . Transform to principal axes.
22 molecule.coordinates3.ToPrincipalAxes ( weights = masses )
23
24 # . Calculate and print the inertia matrix after reorientation.
25 inertia = molecule.coordinates3.InertiaMatrix ( weights = masses )
26 inertia.Print ( title = "Inertia Matrix After Reorientation" )
27
28 # . Define a table for the results.
29 table = logfile.GetTable ( columns = [ 20, 10, 10 ] )
30 table.Start ( )
31 table.Title ( "RMS Coordinate Deviations" )
32 table.Heading ( "Structure" )
33 table.Heading ( "Before"    )
34 table.Heading ( "After"     )
35
36 # . Loop over the remaining structures.
37 for xyzfile in xyzfiles:
38     crd3 = XYZFile_ToCoordinates3 ( \
                                os.path.join ( xyzpath, xyzfile ) )
39     rms0 = crd3.RMSDeviation ( molecule.coordinates3, \
                                                 weights = masses )
40     crd3.Superimpose ( molecule.coordinates3, weights = masses )
41     rms1 = crd3.RMSDeviation ( molecule.coordinates3, \
                                                 weights = masses )
42     table.Entry ( xyzfile[0:-4], alignment = "l" )
43     table.Entry ( "%.2f" % ( rms0, ) )
44     table.Entry ( "%.2f" % ( rms1, ) )
45
46 # . Finish up the table.
47 table.Stop ( )
```

The example consists of two halves. In the first part, *lines 1–26*, a single set of coordinates is manipulated using the transformation methods, whereas in the second part, *line 28* onwards, additional coordinate sets are read and superimposed onto a reference set.

Line 6 defines a list with the names of files in XYZ format that contain the structures to be analysed.

Lines 9–11 create an instance of `System` from the XYZ file `"bALA_C7eq.xyz"`.

Line 9 uses the Python list method `pop` to define the name of the XYZ file to be read. The method removes the last element of the list and assigns it to the variable `xyzfile`. As the element is removed, the length of the list is reduced by one.

Line 14 generates a vector of weights that contains the masses of the atoms in the system. The method that is used is `GetItemAttributes` from the `AtomContainer` class. It takes a single argument which is the name of the attribute to extract for each atom. In this case it is `"mass"` but other names are possible, including `"atomicnumber"`.

Line 15 moves the system to its centre of mass.

Lines 18–26 determine and print the system's inertia matrix before and after a principal axis transformation. The matrix will, in general, have non-zero values for all its components before the transformation but afterwards it should be diagonal. The printing is done with the `Print` method of `inertia` which is an instance of the `Matrix33` class.

Lines 29–34 set up the table that is to be used to output the results of the subsequent analysis.

Line 37 loops over the XYZ files that remain in the list `xyzfiles`.

Line 38 employs the helper function `XYZFile_ToCoordinates3` of the `XYZFileReader` class to read the coordinates from the XYZ file that is currently being iterated over. This function behaves similarly to the function `XYZFile_ToSystem` except that only the coordinates in the file are returned and not a full system. Other file-reader classes, such as `MOLFileReader` and `PDBFileReader`, have equivalent functions.

Lines 39–41 calculate the RMS deviations between the coordinates in `crd3` and the reference set in `molecule`. The calculation is done before and after superimposing the two sets.

Lines 42–44 output the results. The `xyzfile[0:-4]` syntax in *line 42* means that the last four characters of the string `xyzfile` are not output to the table. These are not needed as they correspond to the string `".xyz"`. Likewise, the keyword argument `alignment` in the call to the method `Entry` means that the text is to be output left-justified in the column. The default is to right-justify it.

Line 47 terminates table writing.

Exercises

3.1 A property of a molecule closely related to its moments of inertia is its *radius of gyration*, R_{gyr}, which is defined as

$$R_{\mathrm{gyr}} = \sqrt{\frac{\sum_{i=1}^{N} w_i \left(\boldsymbol{r}_i - \boldsymbol{R}_{\mathrm{c}}\right)^2}{\sum_{i=1}^{N} w_i}} \tag{E3.1}$$

where $\boldsymbol{R}_c$ is the centre of the molecule calculated with the same weights as those used for the calculation of R_{gyr}. Write a function that calculates this quantity and use it to analyse the molecules examined in Example 4.

3.2 Stereochemistry is an important branch of chemistry. Part of the way in which the stereochemistry of a molecule is described is based upon the Cahn–Ingold–Prelog notation. In this system, knowledge of the connectivity of the atoms in a molecule together with their coordinates, atomic numbers and masses is sufficient to label a molecule's *stereocentres*. Write a method to identify the number and types of a molecule's stereocentres and apply it to the structures of the blocked alanine molecules appearing in the text. Are the alanyl groups of these structures in their R or S forms?

4

Quantum chemical models

4.1 Introduction

In the last chapter we dealt with how to manipulate a set of coordinates and how to compare the structures defined by two sets of coordinates. This is useful for distinguishing between two different structures but it gives little indication regarding which structure is the more probable; i.e. which structure is most likely to be found experimentally. To do this, it is necessary to be able to evaluate the intrinsic stability of a structure, which is determined by its *potential energy*. The differences between the energies of different structures, their *relative energies*, will then determine which structure is the more stable and, hence, which structure is most likely to be observed.

This chapter starts off by giving an overview of the various strategies that are available for calculating the potential energy of molecular systems and then goes on to describe a specific class of techniques based upon the application of the theory of *quantum mechanics* to chemical systems.

4.2 The Born–Oppenheimer approximation

Quantum mechanics was developed during the first decades of the twentieth century as a result of shortcomings in the existing *classical mechanics* and, as far as is known, it is adequate to explain all atomic and molecular phenomena. In an oft-quoted, but nevertheless pertinent, remark, P. A. M. Dirac, one of the founders of quantum mechanics, said in 1929:

> The underlying physical laws necessary for the mathematical theory of a large part of physics and the whole of chemistry are thus completely known, and the difficulty is only that the exact application of these laws leads to equations much too complicated to be soluble. It therefore becomes desirable that approximate practical methods of applying quantum mechanics should be developed, which can lead to an explanation of the main features of complex atomic systems without too much computation.

One of the equations that Dirac was talking about and which, in principle, determines the complete behaviour of a (non-relativistic) molecular system is the *time-dependent Schrödinger equation*. It has the form

$$\hat{\mathcal{H}}\Psi = \mathrm{i}\hbar\frac{\partial\Psi}{\partial t} \tag{4.1}$$

In this equation, $\hat{\mathcal{H}}$ is what is known as the *Hamiltonian operator* which operates on the *wavefunction* for the system denoted by Ψ. The other symbols are the imaginary number, i, Dirac's constant, $\hbar$, which is Planck's constant, h, divided by 2π, and the time, t.

A little further qualitative explanation of some of these terms is necessary. In general, the wavefunction, Ψ, is a function of the position coordinates of all the particles in the system, the time and some specifically quantum mechanical variables that determine each particle's *spin*. It is the wavefunction that gives complete information about the system and is the goal when solving Equation (4.1). The wavefunction is important because its square is the probability density distribution for the particles in the system. To make this clearer, consider a system consisting of one particle that is constrained to move in one dimension. Then the wavefunction is a function of two variables, the position coordinate, x, and the time, t (for convenience spin is ignored) – this dependence is written as $\Psi(x, t)$. The probability that the particle will be found in the range x to $x + \delta x$, where δx is a small number, at a time t is $|\Psi(x, t)|^2\,\delta x$. In quantum mechanics this is the best that can be done – it is possible to know only the probability that a particle will have certain values for its variables – unlike classical mechanics, in which, in principle, values for a particle's variables can be defined precisely.

The wavefunction is important but its behaviour is determined by the Hamiltonian operator of the system. This operator will consist of a sum of two other operators – the kinetic energy operator, $\hat{\mathcal{K}}$, and the potential energy operator, $\hat{\mathcal{V}}$. The former determines the *kinetic energy* of the system, which is the energy due to 'movement' of the particles, and the latter the energy due to interactions between the particles and with their environment.

In certain cases the system can exist in what is called a *stationary state* in which its wavefunction and, thus, its particles' probability distribution do not change with time. For stationary states the time-dependent equation (Equation (4.1)) reduces to a simpler form, the *time-independent Schrödinger equation*:

$$\hat{\mathcal{H}}\Psi = E\Psi \tag{4.2}$$

where E is the energy of the stationary state which is a constant.

For atomic and molecular systems, there are essentially two types of particles – electrons and the atomic nuclei. The latter will differ in their mass and their charge depending upon the element and its isotope. These classes of particles have

greatly disparate masses. As an example, the lightest nucleus – that for hydrogen which consists of a single proton – has a mass about 1836 times greater than that of an electron. These very different masses will cause the electrons and the nuclei to have very different motions and this means that, to a good approximation, their dynamics can be treated separately. This is the basis of the *Born–Oppenheimer approximation*. It leads to the following procedure. The first step is to tackle the electronic problem by solving the *electronic Schrödinger equation* for a specific set of nuclear variables. That is, the nuclear coordinates are regarded as fixed and the wavefunction that is determined gives the electronic distribution only. The energy for which this equation is solved is no longer a constant but is a function of the nuclear positions. The second stage is to treat the nuclear dynamics by using the energy obtained from the solution of the electronic problem as an *effective potential* for the interaction of the nuclei.

The electronic equation is

$$\hat{\mathcal{H}}_{\text{el}}\Psi_{\text{el}} = E_{\text{el}}(\boldsymbol{r}_1, \boldsymbol{r}_2, \ldots, \boldsymbol{r}_N)\Psi_{\text{el}} \tag{4.3}$$

where the Hamiltonian is the same as that in Equation (4.2) except that the kinetic energy operator for the nuclei has been omitted and the wavefunction, Ψ_{el}, gives the distribution of electrons only. Because the nuclei have been fixed the wavefunction and the energy, E_{el}, both depend parametrically upon the nuclear coordinates, $\{\boldsymbol{r}_i\}$, and so for each different configuration or structure there will be a different electronic distribution and a different electronic energy.

The electronic energy, E_{el}, is the energy that will be of primary interest to us. Because it is a function of the positions of all the nuclei in the system it is a multidimensional function that will often have a very complex form. This function defines the *potential energy surface* for the system and it is this that determines the effective interactions between the nuclei and, hence, the system's structure and dynamics.

4.3 Strategies for obtaining energies on a potential energy surface

One of the most important problems when performing molecular simulations is how to obtain accurate values for the electronic energy of a system as a function of its nuclear coordinates. Various strategies are possible.

The most fundamental approach is to attempt to calculate the energies from first principles by solving the electronic Schrödinger equation (Equation (4.3)) for the electronic energy at each nuclear configuration. Many methods for doing this are available but most are based upon one of the following theories – *density functional theory*, *molecular orbital theory*, *quantum Monte Carlo theory* or *valence bond theory*. All these theories are first principles or *ab initio*, in the sense that they

attempt to solve Equation (4.3) with as few assumptions as possible. Although *ab initio* methods can give very accurate results in many circumstances they are computationally expensive and so cheaper alternatives have been developed.

One way of making progress is to drop the restriction of performing first-principles calculations and seek ways of simplifying the *ab initio* methods outlined above. These so-called *semi-empirical* methods have the same overall formalism as that of the *ab initio* methods but they approximate time-consuming parts of the calculation. As approximations have been introduced, these methods must be calibrated to ensure that the results they produce are meaningful. This often means that the values of the various *empirical parameters* in the methods have to be chosen so that the results of the calculations agree with experimental data or with the results of accurate *ab initio* calculations. Semi-empirical versions of many *ab initio* methods exist. Together the *ab initio* and semi-empirical methods constitute the class of *quantum chemical* (QC) approaches.

A second and even cheaper alternative is to employ an entirely *empirical potential energy function*. This consists in choosing an analytic form for the function that is to represent the potential energy surface for the system and then, like the semi-empirical QC methods, *parametrizing* the function so that the energies it produces agree with experimental data or with the results of accurate quantum mechanical calculations. Very many different types of empirical energy function have been devised. Some are designed for the description of a single system. For example, studies of simple reactions such as $H_2 + H \rightarrow H + H_2$ often use surfaces of special forms. For studies on larger molecules more general functions have been developed and it is these that we shall discuss in Chapter 5. Methods that combine elements of both quantum chemical and empirical potential energy function calculations will be the subject of Chapter 6.

4.4 Molecular orbital methods

Molecular orbital (MO) and density functional theory (DFT) based methods are the most popular *ab initio* approaches for performing quantum chemical calculations. For a long time, MO methods dominated quantum chemistry but they have slowly been giving way to DFT approaches since the beginning of the 1990s. This is because DFT methods have approximately the same computational expense as the cheapest MO methods but can give results that are more reliable for many phenomena. Despite this, we describe only MO methods here because semi-empirical methods, based upon MO theory, are the only ones that we shall employ in the examples in this book. Nevertheless, DFT methods share similarities with MO methods, both in their formulation and in the way that they are implemented

for calculating the potential energy of a molecular system, and so much of what is said will be applicable to the DFT case.

MO methods express the wavefunction of a particular *electronic state* of a system as

$$\Psi_{\mathrm{el}} = \sum_I A_I \Phi_I \tag{4.4}$$

Here A_I are expansion coefficients, normally called *configuration interaction* (CI) coefficients, and the Φ_I are *Slater determinants* comprising products of M *one-electron, orthonormal spin-orbitals* where M is the number of electrons in the system. The Φ_I are written as matrix determinants because electrons are *fermions* whose wavefunctions must be *antisymmetric* with respect to particle exchange and determinants automatically have this property. A Slater determinant for M electrons is

$$\Phi_I = \frac{1}{\sqrt{M!}} \begin{vmatrix} (\phi\sigma)_a(1) & (\phi\sigma)_b(1) & \dots & (\phi\sigma)_m(1) \\ (\phi\sigma)_a(2) & (\phi\sigma)_b(2) & \dots & (\phi\sigma)_m(2) \\ \vdots & \vdots & \vdots & \vdots \\ (\phi\sigma)_a(M) & (\phi\sigma)_b(M) & \dots & (\phi\sigma)_m(M) \end{vmatrix} \tag{4.5}$$

where $(\phi\sigma)$ is a spin-orbital and $(\phi\sigma)_a(1)$ indicates that the ath spin-orbital is occupied by electron 1. Expansion of the Slater determinant leads to a sum of $M!$ terms each of which consists of a product of M spin-orbitals. To see this, consider the case of two electrons for which a Φ_I is

$$\begin{aligned} \Phi_I(1,2) &= \frac{1}{\sqrt{2}} \begin{vmatrix} (\phi\sigma)_a(1) & (\phi\sigma)_b(1) \\ (\phi\sigma)_a(2) & (\phi\sigma)_b(2) \end{vmatrix} \\ &= \frac{1}{\sqrt{2}} \{(\phi\sigma)_a(1)(\phi\sigma)_b(2) - (\phi\sigma)_a(2)(\phi\sigma)_b(1)\} \end{aligned} \tag{4.6}$$

The spin-orbitals, themselves, are the product of a spin function, σ, and a space-orbital, ϕ. Electrons have a spin of one-half and the spin function can take one of two values, α or β, corresponding to spins of $+1/2$ (spin-up) or of $-1/2$ (spin-down), respectively. In contrast, space-orbitals are usually expanded in terms of *basis functions* taken from a *basis set*. If we suppose that there are N_{b} functions in the set and that these functions are denoted η_μ (with μ in the range 1 to N_{b}), a space-orbital expansion can be written as

$$\phi_a(\boldsymbol{r}) = \sum_{\mu=1}^{N_{\mathrm{b}}} c_{\mu a} \eta_\mu(\boldsymbol{r}) \tag{4.7}$$

In this equation, the $c_{\mu a}$ are the *molecular orbital coefficients*. In general, both they and the CI coefficients will be complex but in this book, for simplicity, we shall take them to be real.

In MO methods, the spin-orbitals are orthonormal, which means that the *overlap* is zero between different spin-orbitals and one between identical orbitals. The spin functions α and β are orthonormal by construction whereas the space-orbitals must satisfy an *orthonormality constraint*. Denoting the constraint between space-orbitals a and b as Λ_{ab} gives

$$\Lambda_{ab} = \int_{\boldsymbol{r}} \phi_a(\boldsymbol{r})\,\phi_b(\boldsymbol{r})\,\mathrm{d}\boldsymbol{r} - \delta_{ab} \tag{4.8}$$

$$= \sum_{\mu} c_{\mu a} \sum_{\nu} c_{\nu b} \int_{\boldsymbol{r}} \eta_\mu(\boldsymbol{r})\,\eta_\nu(\boldsymbol{r})\,\mathrm{d}\boldsymbol{r} - \delta_{ab} \tag{4.9}$$

$$= \sum_{\mu\nu} c_{\mu a} c_{\nu b} S_{\mu\nu} - \delta_{ab} \tag{4.10}$$

Here δ_{ab} is the *Kronecker delta*, which takes a value of one if $a = b$ and zero otherwise, $S_{\mu\nu}$ is the element of the *overlap matrix* between the basis functions η_μ and η_ν and the integrations in Equations (4.8) and (4.9) are performed over all space. The condition of orthonormality on the space-orbitals limits the number of independent orbitals that can be formed from a given basis set. Thus, if there are N_b functions in the set, the maximum number of independent orbitals derivable from the set will also be N_b.

To perform an MO calculation, it is necessary to choose a basis set and the form that the expansion in Equation (4.4) is to take. Once this has been done, the CI and MO coefficients of Equations (4.4) and (4.7), respectively, are the quantities that remain to be determined. In *Hartree–Fock* (HF) theory, which is the simplest MO method, the expansion of Equation (4.4) consists of only a small number of determinants – normally one, in fact – and the values of the CI coefficients are fixed. This leaves the MO coefficients, which are found using a *variational procedure* that will be detailed below. More elaborate approaches include the CI methods in which only the CI coefficients are allowed to vary and the values of the MO coefficients are fixed, having been taken, usually, from a previous HF calculation. The most general MO methods are *multiconfigurational* methods in which the CI and MO coefficients are varied simultaneously. It is also possible to determine wavefunctions *non-variationally*. One example is methods based upon *Møller–Plesset perturbation theory* which, like CI methods, are employed most often to improve upon a previously determined HF wavefunction.

4.5 The Hartree–Fock approximation

The simplest HF wavefunctions consist of a single determinant and there are two principal types. There are *spin-restricted* HF (RHF) wavefunctions, in which a single set of orthonormal space-orbitals is used for electrons of both α and

β spin, and *spin-unrestricted* HF (UHF) wavefunctions in which there are two distinct sets of space-orbitals, one for the electrons of α spin and another for those of β spin. The orbitals within each set are orthonormal but there are no such conditions between the orbitals of different sets. The commonest single-determinant RHF wavefunctions are *closed-shell* wavefunctions in which each space-orbital that is occupied has two electrons, one of α spin and one of β spin. These wavefunctions, by construction, have no unpaired electrons and so always correspond to *singlet* states or, equivalently, to states with a *spin multiplicity* of one. By contrast, single-determinant RHF wavefunctions, with unpaired α and β electrons, and UHF wavefunctions are suitable for describing *open-shell* states which have multiplicities of two (*doublet*), three (*triplet*), four (*quartet*) and so on.

In this book, we consider only closed-shell RHF and UHF approaches as these will be sufficient for our needs. The wavefunctions of these methods are determined with very similar procedures but we shall concentrate upon the closed-shell RHF case because the formulae are less cumbersome. To find a HF wavefunction, use is made of the *variational principle* which states that the energy of an approximate wavefunction, obtained by solving Equation (4.3), is always an upper bound to the energy of the exact wavefunction. This implies that the optimum HF wavefunction can be found by determining the values of the MO coefficients that give the lowest electronic energy. Practically this is done by minimizing the expression for the electronic energy of the HF wavefunction with respect to values of the MO coefficients, but making sure that the coefficients obey the orthonormality constraints of Equation (4.10).

An expression for the electronic energy of a system can be obtained by rearranging Equation (4.3) and is

$$\mathcal{V} = \frac{\int_{\boldsymbol{r}_1} \mathrm{d}\boldsymbol{r}_1 \int_{\boldsymbol{r}_2} \mathrm{d}\boldsymbol{r}_2 \ldots \int_{\boldsymbol{r}_M} \mathrm{d}\boldsymbol{r}_M \Psi(\boldsymbol{r}_1, \boldsymbol{r}_2, \ldots, \boldsymbol{r}_M) \hat{\mathcal{H}}_{\mathrm{el}} \Psi(\boldsymbol{r}_1, \boldsymbol{r}_2, \ldots, \boldsymbol{r}_M)}{\int_{\boldsymbol{r}_1} \mathrm{d}\boldsymbol{r}_1 \int_{\boldsymbol{r}_2} \mathrm{d}\boldsymbol{r}_2 \ldots \int_{\boldsymbol{r}_M} \mathrm{d}\boldsymbol{r}_M |\Psi(\boldsymbol{r}_1, \boldsymbol{r}_2, \ldots, \boldsymbol{r}_M)|^2} \quad (4.11)$$

where Ψ is the HF wavefunction and we have replaced E_{el} by $\mathcal{V}$ as the latter is the symbol that we shall use, henceforth, to denote the potential energy of a system, no matter how it is obtained.

In atomic units, the electronic Hamiltonian, in its most basic version, has the form

$$\hat{\mathcal{H}}_{\mathrm{el}} = -\frac{1}{2} \sum_s \nabla_s^2 - \sum_{si} \frac{Z_i}{r_{si}} + \sum_{st} \frac{1}{r_{st}} + \sum_{ij} \frac{Z_i Z_j}{r_{ij}} \quad (4.12)$$

Here the subscripts s and t and i and j refer to electrons and nuclei, respectively, Z is a nuclear charge and r_{xy} is the distance between particles x and y. The first term on the right-hand side of the equation is the operator describing the kinetic

energy of the electrons, whereas the remaining terms account for the *electrostatic interactions* between the charges of the various subgroups of particles. Thus, the second term is the *electron–nuclear attraction*, the third term the *electron–electron repulsion* and the fourth term the *nuclear–nuclear repulsion*. This latter term, which we shall denote by the symbol $\mathcal{V}_{\mathrm{nn}}$, is a constant for a given arrangement of the nuclei (or atoms) because it is independent of the electronic coordinates.

Substitution of the expressions for a single-determinant HF wavefunction, Equation (4.5), and the electronic Hamiltonian, Equation (4.12), into Equation (4.11) enables the electronic energy to be written as a function of the MO coefficients that define the wavefunction. The derivation is straightforward, although long, and particular care must be taken in properly accounting for each of the $M!$ terms in the determinantal expansion. The final expression, for a closed-shell RHF wavefunction, is compactly written as

$$\mathcal{V} = \frac{1}{2}\sum_{\mu\nu} P_{\mu\nu}\left\{H_{\mu\nu} + F_{\mu\nu}\right\} + \mathcal{V}_{\mathrm{nn}} \tag{4.13}$$

where the sum is a double sum over all the basis functions in the basis set and $P_{\mu\nu}$, $H_{\mu\nu}$ and $F_{\mu\nu}$ are elements of the *density matrix*, the *one-electron matrix* and the *Fock matrix*, respectively. The density matrix elements are quadratic functions of the MO coefficients and are

$$P_{\mu\nu} = \sum_{a} n_a c_{\mu a} c_{\nu a} \tag{4.14}$$

$$= 2 \sum_{a \text{ occupied}} c_{\mu a} c_{\nu a} \tag{4.15}$$

In Equation (4.14) the sum is over all MOs in the system and n_a are the orbital occupancies. For a closed-shell RHF wavefunction, orbitals are either occupied with an occupancy of two or unoccupied with an occupancy of zero and so the sum reduces to one over occupied orbitals only, as in Equation (4.15).

The elements of the one-electron matrix consist of integrals of the basis functions over the terms in the electronic Hamiltonian that depend upon the coordinates of one electron only. These are the electron kinetic energy and the electron–nuclear attraction terms. The full expression for $H_{\mu\nu}$ is

$$H_{\mu\nu} = \int_{\boldsymbol{r}} \eta_\mu(\boldsymbol{r}) \left\{ -\frac{1}{2}\nabla^2 - \sum_i \frac{Z_i}{|\boldsymbol{r} - \boldsymbol{r}_i|} \right\} \eta_\nu(\boldsymbol{r}) \mathrm{d}\boldsymbol{r} \tag{4.16}$$

The expression for the Fock matrix is more complicated and is

$$F_{\mu\nu} = H_{\mu\nu} + \sum_{\lambda\sigma} P_{\lambda\sigma} \left[(\mu\nu|\lambda\sigma) - \frac{1}{2}(\mu\sigma|\lambda\nu) \right] \tag{4.17}$$

The first term on the right-hand side is the one-electron part of the Fock matrix whereas the remaining term is a sum of products of density matrix elements and *two-electron integrals*. The latter arise from the electron–electron repulsion terms of the electronic Hamiltonian. They are six-dimensional because they depend upon the coordinates of two electrons and take the form

$$(\mu\nu|\lambda\sigma) = \int_{\boldsymbol{r}_1} \mathrm{d}\boldsymbol{r}_1 \eta_\mu(\boldsymbol{r}_1)\eta_\nu(\boldsymbol{r}_1) \int_{\boldsymbol{r}_2} \mathrm{d}\boldsymbol{r}_2 \frac{\eta_\lambda(\boldsymbol{r}_2)\eta_\sigma(\boldsymbol{r}_2)}{|\boldsymbol{r}_1 - \boldsymbol{r}_2|} \tag{4.18}$$

The two-electron integrals enter into the expression for the Fock matrix in two distinct ways. The first two-electron term on the right-hand side of Equation (4.17) is the *Coulomb* portion of the matrix and corresponds to the electrostatic interaction between different parts of the electron distribution. It takes the standard form that would be expected from the classical theory of electromagnetism. By contrast, the second two-electron term is purely quantum mechanical in origin and is called the *exchange term*. It arises from the antisymmetrization requirement on the electronic wavefunction.

The minimization of the electronic energy, subject to the orthogonalization constraints on the orbitals, can be performed in a similar way to that of Section 3.6, using the method of Lagrange multipliers. The procedure is a little more complicated in this case because there are multiple constraints, one for each pair of orbitals in the system, and so there are multiple multipliers, one for each constraint. Denoting the Lagrange multiplier for the constraint between orbitals a and b as ϵ_{ab}, the equations to be satisfied by the optimum MO coefficients are

$$\frac{\partial}{\partial c_{\mu a}} \left[\mathcal{V} - \sum_{bc} \epsilon_{bc} \Lambda_{bc} \right] = 0 \quad \forall \quad \mu, a \tag{4.19}$$

$$\Lambda_{bc} = 0 \quad \forall \quad b, c \tag{4.20}$$

In these equations, the subscripts a, b and c are orbital indices and the sum in Equation (4.19) is a double one over all pairs of orbitals.

The derivatives in Equation (4.19) can be obtained by differentiating Equations (4.13) and (4.10), respectively. This is straightforward to do, although a little tedious, as long as it is noted that the matrices $\mathbf{F}$ and $\mathbf{S}$ and the matrix defined by the Lagrange multipliers, ϵ_{bc}, are all symmetric. The final results are

$$\frac{\partial \mathcal{V}}{\partial c_{\mu a}} = 4 \sum_\nu F_{\mu\nu} c_{\nu a} \tag{4.21}$$

$$\frac{\partial}{\partial c_{\mu a}} \sum_{bc} \epsilon_{bc} \Lambda_{bc} = 2 \sum_b \epsilon_{ab} \sum_\nu S_{\mu\nu} c_{\nu b} \tag{4.22}$$

The derivative expressions from Equations (4.21) and (4.22) can now be substituted into Equation (4.19). In matrix-vector form the result is

$$\mathbf{F}\boldsymbol{c}_a = \sum_b \left(\frac{1}{2}\,\epsilon_{ab}\right)\mathbf{S}\boldsymbol{c}_b \quad \forall \quad a \tag{4.23}$$

where $\boldsymbol{c}_a$ and $\boldsymbol{c}_b$ are the N_b-dimensional vectors of MO coefficients for orbitals a and b, respectively.

Equation (4.23) is reminiscent of the eigenvalue equation that we met in Section 3.5, although it is more complicated due to the presence of the sum over orbitals and of the overlap matrix **S** on its right-hand side. The equation can, however, be simplified by considering transformations that rotate the set of occupied MOs amongst themselves. Suppose that there are N_occ occupied orbitals and that we write them as a matrix, **C**, in which each column corresponds to an orbital, as follows:

$$\mathbf{C} = \left(\boldsymbol{c}_1\boldsymbol{c}_2\cdots\boldsymbol{c}_{N_\mathrm{occ}-1}\boldsymbol{c}_{N_\mathrm{occ}}\right) \tag{4.24}$$

A set of transformed orbitals, $\mathbf{C}'$, can be produced by multiplying the original matrix by an orthogonal matrix, **U**, that describes the rotation:

$$\mathbf{C}' = \mathbf{C}\mathbf{U} \tag{4.25}$$

The rotation matrix will have dimensions, $N_\mathrm{occ} \times N_\mathrm{occ}$, and, because it is orthogonal, the orthonormality of the original orbitals will be preserved.

The interest of transformations of this type is that many quantities, such as the density matrix, the Fock matrix and the electronic energy, are invariant to them. As an example, let us take the RHF density matrix, expressed in terms of the transformed orbitals, and show that it is identical to that expressed in terms of the original set. In matrix-vector notation, the argument runs as follows:

$$\begin{aligned}\mathbf{P}' &= 2\mathbf{C}'(\mathbf{C}')^\mathrm{T}\\ &= 2(\mathbf{C}\mathbf{U})(\mathbf{C}\mathbf{U})^\mathrm{T}\\ &= 2\mathbf{C}\mathbf{U}\mathbf{U}^\mathrm{T}\mathbf{C}^\mathrm{T}\\ &= 2\mathbf{C}\mathbf{C}^\mathrm{T}\\ &= \mathbf{P}\end{aligned} \tag{4.26}$$

It should be emphasized that this result only holds if the transformed orbitals have the same occupancy. Thus, it applies to the occupied orbitals of closed-shell RHF wavefunctions and, separately, to the occupied orbitals of α and β spin in the UHF case.

This property of invariance of quantities to rotations amongst the occupied orbitals means that we have significant freedom in choosing which set of orbitals

to find as long as it spans the occupied space. In particular, we can choose the set that simplifies Equation (4.23) by rendering the matrix of Lagrange multipliers for the occupied orbitals diagonal. Doing this, and at the same time absorbing the factor of a half into the definition of the new multipliers, ϵ_a, gives the final expression that the optimized orbitals must obey:

$$\mathbf{F}\boldsymbol{c}_a = \epsilon_a \mathbf{S}\boldsymbol{c}_a \quad \forall \quad a \tag{4.27}$$

This equation is called the *Roothaan–Hall* equation, named after C. C. J. Roothaan and G. G. Hall who proposed it in the 1950s.

4.5.1 Solving the Roothaan–Hall equation

The Roothaan–Hall equation, Equation (4.27), is a *generalized eigenvalue equation* and can be diagonalized to obtain the orbital vectors, $\boldsymbol{c}_a$, and the Lagrange multipliers or *orbital energies*, ϵ_a. The procedure for doing this is described in Section A2.1. Diagonalization, however, is not sufficient to solve for the optimum orbitals as the Fock matrix (Equation (4.17)) is itself a function of the density matrix and, hence, the orbital coefficients, $\mathbf{c}_a$.

A range of methods have been developed for solving this equation. The approach that we employ in this book is called the *self-consistent field* (SCF) procedure and is the one that has traditionally been used in the quantum chemistry community. Nevertheless, other methods may be more appropriate or efficient for certain applications, such as when treating systems with many atoms or crystalline solids.

The SCF procedure is an iterative one and, in the spin-restricted case, would typically consist of the following steps:

(i) Define the quantities that are needed for the HF calculation and, in particular:
   (a) The elemental composition of the system that is to be studied.
   (b) A set of atomic coordinates at which the calculation is to be performed.
   (c) The charge and spin multiplicity for the system. These will determine how many α and β spin electrons there are and whether a closed-shell RHF or a UHF description is appropriate.
   (d) A basis set.

(ii) Calculate and store items that will be required in the subsequent procedure. These include the overlap and one-electron matrices, **S** and **H**, and the two-electron integrals, $(\mu\nu|\lambda\sigma)$. For large systems, it often proves impossible to store the two-electron integrals, simply because there are too many of them. In such cases, they are recalculated each time they are required.

(iii) Guess a starting density matrix, **P**. A crude approach is to take **P** to be zero, which means that the Fock matrix built in the first SCF iteration will be equal to the one-electron matrix. A more sophisticated estimate is obtained by building **P** from the densities of the isolated atoms.

(iv) Construct the Fock matrix using the current value of the density matrix and the appropriate integrals and other quantities. Those integrals that were not precalculated and stored in Step (ii) have to be recalculated.

(v) Diagonalize the new Fock matrix to obtain orbital energies and vectors. This step will yield a set of up to N_b vectors.

(vi) Build a new density matrix from the orbital vectors obtained in Step (v) using Equation (4.15). It is normal, although not obligatory, to choose the orbitals of lowest energy as those that are occupied.

(vii) Decide whether the SCF procedure has *converged*. This can be done in a number of ways but a common one is to check how different the new density matrix is from the one of the previous cycle. If this difference is too big the iterations are continued by returning to Step (iv). Otherwise the iterations are stopped and the SCF procedure terminates.

A similar procedure to this one is followed in a spin-unrestricted calculation. The major difference is that there are two sets of orbitals to determine, one for each spin, and each set of orbitals has its own associated quantities. This means, for example, that there will be two density matrices, $\mathbf{P}^{\alpha}$ and $\mathbf{P}^{\beta}$, and two Fock matrices, $\mathbf{F}^{\alpha}$ and $\mathbf{F}^{\beta}$.

The SCF method described above will often fail to converge. This can either be because convergence is occurring but it is so slow that it will not take a reasonable number of steps or because the procedure is oscillating and will never converge no matter how many steps are taken. In practice, therefore, it is normal to augment the procedure with methods that enhance or stabilize convergence. A number of such methods exist but most use, in one way or another, information from previous steps to help guide the construction of the density and/or Fock matrices at the current step. Even so, none of the methods in use is foolproof and convergence can often be a problem, particularly for systems, such as those that contain transition metals, for which the differences in energies between the *highest occupied orbitals* and the *lowest unoccupied orbitals* are very small.

4.5.2 Basis sets

The choice of basis set is crucial in determining the precision of the results of a quantum chemical calculation. The three most common types of function

used in basis sets are *Slater* or *exponential* functions, *Gaussian* functions and *plane-waves*. They have the following forms

$$\eta_\mu^{\text{Slater}}(\boldsymbol{r}) \propto \Theta(\boldsymbol{r}) \exp[-\zeta r] \tag{4.28}$$

$$\eta_\mu^{\text{Gaussian}}(\boldsymbol{r}) \propto \Theta(\boldsymbol{r}) \exp[-\zeta r^2] \tag{4.29}$$

$$\eta_\mu^{\text{plane-wave}}(\boldsymbol{r}) \propto \exp[-\mathrm{i}\boldsymbol{k}^{\mathrm{T}}\boldsymbol{r}] \tag{4.30}$$

Slaters and Gaussians are localized functions in that they are centred about a particular point in space. Normally they are placed upon the atomic nuclei so that the $\boldsymbol{r}$ in Equations (4.28) and (4.29) would be replaced by $\boldsymbol{r} - \boldsymbol{r}_i$, where $\boldsymbol{r}_i$ are the atomic coordinates of the ith atom. The ζ in these equations are the basis functions' *exponents* whereas the $\Theta(\boldsymbol{r} - \boldsymbol{r}_i)$ are functions that determine the *angular behaviour* of the basis functions, i.e. whether they behave as s or p or d orbitals and so on. In contrast to the atom-centred Slater and Gaussian functions, plane-waves are delocalized, periodic functions that spread over all space. They are complex by definition and are characterized by the value of their *wavevector*, $\boldsymbol{k}$.

These functions have different advantages and disadvantages. The orbitals of hydrogen, and related one-electron cations, are Slater functions and so these functions probably provide the most compact description of the wavefunctions of atomic and molecular systems in terms of one-electron functions. Unfortunately, many of the integrals involving Slater functions are difficult to evaluate and this has limited their use in quantum chemical applications. By contrast, Gaussians are much more amenable to manipulation although they provide a less accurate description of the wavefunction near to and far from the nucleus. Plane-waves, like Gaussians, are also very easy to manipulate but, as they are delocalized, they are very poor at representing the wavefunction near the atomic nuclei where the electron density is at its highest. In addition, they are almost never used for HF calculations due to difficulties associated with evaluating the two-electron exchange term. Plane-wave basis sets are especially well adapted for treating periodic systems, such as crystals, with DFT methods that do not have exchange-like terms.

Gaussians were first introduced into quantum chemistry by S. F. Boys in 1950. Since then, a huge variety of Gaussian basis sets have been developed and calibrated for molecular calculations. Many of the most widely used were proposed by one of the pioneers of computational quantum chemistry, J. A. Pople, and his collaborators. The trade-off when devising a basis set is between the potential improvements in accuracy when additional functions are added and the extra computational expense that a larger basis set incurs. This is why a range of basis sets are commonly used in any particular study. One starts with a small

basis set to explore the system and to obtain an idea of how it behaves and then one uses basis sets of increasing size to get results of higher precision.

The nomenclature used to describe Gaussian basis sets is as diverse as the basis sets themselves but a basic categorization can be made using the ζ ('zeta') notation. A *single-*ζ or *minimal* basis set has one basis function for each *core* and *valence* atomic orbital. Thus a hydrogen atom would have one function in its basis set corresponding to its 1s orbital whereas a carbon would have five functions corresponding to its 1s, 2s and three 2p orbitals. Minimal basis set descriptions are not very precise and so most serious studies would start with *double-*ζ basis sets in which there are two basis functions per atomic orbital. More accurate calculations require *triple-*ζ and *quadrupule-*ζ basis sets or higher.

The basic basis sets are often supplemented with additional functions. Two examples are *diffuse functions* and *polarization functions*. Diffuse functions are functions with small exponents that are found necessary to describe the more extended charge distributions in anionic systems. They are usually of s or p type. In contrast, polarization functions have exponents similar to those of the other, valence functions in the basis set but they are of higher angular momentum. Thus, for example, hydrogen would have polarization functions of p-type whereas the second and third row elements would have functions of d-type. Polarization functions are essential for the correct description of hypervalent bonding, such as occurs for phosphorus and sulphur, but they benefit calculations on many other systems as well.

Two clarifications need to be made concerning the form of Gaussian basis sets. The first is that, in practice, many functions in a Gaussian basis set are not single functions, as in Equation (4.29), but rather fixed linear combinations, or *contractions*, of several functions

$$\eta_{\mu}^{\text{Gaussian}}(\boldsymbol{r}) \propto \Theta(\boldsymbol{r}) \sum_{u} d_u \exp[-\zeta_u r^2] \tag{4.31}$$

In this equation, the individual Gaussian functions are denoted *primitives* and the d_u, *contraction coefficients*. Contraction is most common for core atomic orbitals and is done so that a function based upon Gaussians can more closely approximate the 'correct' exponential-like behaviour that a basis function should have near the atomic nucleus.

The second clarification concerns the form of the angular function, Θ, in Equation (4.31). Two alternative forms are in routine use and these can be summarized as

$$\Theta_l^{\text{Spherical}}(r, \theta, \phi) = r^l S_{lm}(\theta, \phi) \quad m = -l, \ldots, 0, \ldots, l \tag{4.32}$$

$$\Theta_l^{\text{Cartesian}}(x, y, z) = x^{l_x} y^{l_y} z^{l_z} \tag{4.33}$$

In these equations l is a positive or zero integer that denotes the *angular-momentum character* of the function. Values of 0, 1, 2, 3 and 4 indicate s, p, d, f and g functions, respectively. Equation (4.32) is a representation based upon the *real spherical harmonics*, S_{lm}. These are functions of the spherical polar angles θ and ϕ and for each value of l, there are $2l+1$ different functions corresponding to the allowed values of m. In contrast, Equation (4.33) is a Cartesian representation in which l_x, l_y and l_z are all possible combinations of positive or zero integers such that $l_x + l_y + l_z = l$. The two representations are equivalent for s and p functions but diverge thereafter. For example, the Cartesian representation of a d orbital will have six different components in contrast to the spherical harmonic's five, whereas the f orbital representations will have ten and seven components, respectively. The difference is due to contamination in the Cartesian representation with functions of lower angular momentum. Thus, a Cartesian d function is equivalent to a spherical harmonic s and d, and a Cartesian f function to a spherical harmonic p and f.

4.5.3 Scaling

There are two principal computational bottlenecks in a HF calculation. The first is the evaluation of the two-electron integrals and their use in the construction of the Fock matrix required at each SCF iteration. From Equation (4.18) it can be seen that, formally, the number of two-electron integrals scales as $O(N_b^4)$. As the number of basis functions required to describe a system scales roughly with the number of electrons and, hence, the number of atoms, the number of integrals scales approximately as $O(N^4)$. This scaling is very unfavourable as it implies, for example, that a calculation on a system twice as big will take 16 times as long, whereas one on a system ten times as big will take 10 000 times as long!

It is not quite as bad as this. With localized basis functions, such as Gaussians, which are almost invariably used for HF calculations, it can be shown that the scaling should ultimately become $\sim O(N^2)$ as the size of the system increases. This is because integrals involving functions that are centred on three or four widely separated atoms will have negligible values and so need not be evaluated. In practice, it requires quite large systems to attain this behaviour and so scalings of $\sim O(N^3)$, or slightly less, are more commonly observed.

The second bottleneck concerns the manipulation of the matrices, such as **F**, **P** and **S**, in terms of which HF theory is formulated. The most expensive matrix operations that are required are matrix multiplication and diagonalization, both of which scale as the cube of the matrix dimension. As the largest HF matrices have dimension $\sim N_b$, these operations scale as $O(N_b^3) \sim O(N^3)$. In spite of this scaling,

very efficient computational implementations of most matrix operations exist and so it is normally the time for Fock matrix construction that predominates.

Although we do not make use of any of them in this book, methods with improved scaling properties are available for both of the bottlenecks mentioned above. Thus, $\sim O(N)$ algorithms have been devised for the construction of both the Coulomb and the exchange portions of the Fock matrix. We shall discuss some of the principles behind equivalent fast Coulomb methods in Section 10.8. Likewise, linear-scaling methods are known that enable the optimum MO coefficients or density matrix to be found and which serve as alternatives to the SCF procedure of Section 4.5.1. Several strategies are employed by these latter algorithms to achieve their speed-up, including: (i) the reformulation of the QC procedure in terms of density matrices, instead of orbitals, which permits the costly diagonalization of the Fock matrix to be avoided; and (ii) the exploitation of the fact that, for large systems, the density and other matrices required in a QC calculation become *sparse*. This means that a significant fraction of the matrix elements have very small or zero values and so operations, such as matrix multiplication, can be performed more rapidly.

4.5.4 Semi-empirical methods

Semi-empirical methods are an important complement to *ab initio* techniques. Although they can be considered to be more 'approximate' than their *ab initio* counterparts, they are normally much faster and so can be applied to systems or processes that it would not otherwise be possible to investigate with QC methods. It would be difficult to give a comprehensive overview of semi-empirical methods because a huge diversity of schemes, derived from different *ab initio* theories, have been developed. As a result, we shall restrict our discussion to the methods that we employ in this book and which are probably the most popular semi-empirical methods currently in use by the quantum chemistry community.

These methods are called *modified neglect of diatomic overlap* (MNDO) methods and were first introduced by M. J. S. Dewar and W. Thiel in the 1970s. Since then, newer versions of the MNDO methods have been developed, both by the original authors and by other workers, most notably J. J. P. Stewart. The MNDO approaches are MO-based and calculations are usually performed within the HF framework. The methods use minimal basis sets consisting of Slater functions and only the valence electrons are treated explicitly, the core electrons being merged into the nuclei of their respective atoms. This means, for example, that both carbon and silicon atoms have four electrons and 'nuclear' charges of +4 whereas oxygen and sulphur have six electrons and 'nuclear' charges of +6.

The principal approximation made by the MNDO methods in the electronic part of the calculation is the neglect of diatomic differential overlap (NDDO), which was first employed by Pople and co-workers in the 1960s. It states that integrals involving overlap between basis functions on different atoms are zero and has the consequence of greatly decreasing the number of integrals that have to be evaluated. Thus, the overlap matrix **S** reduces to the identity matrix **I** and the number of two-electron integrals becomes $O(N^2)$ instead of the formal $O(N^4)$ of *ab initio* methods.

Further simplifications are obtained by evaluating the remaining integrals using empirical expressions and not with the formulae in Equations (4.16) and (4.18) that would be appropriate for Slater basis functions. The 'nuclear'–'nuclear' or *core–core* repulsion terms are likewise calculated with empirical formulae and not using the analytic expression given in Equation (4.12). It would be inappropriate to list here the formulae that are used for the evaluation of the integrals and other terms. It is important to note, though, that they require parameters which depend upon the elemental types of the atoms involved in the interaction and that, in all, approximately 10–20 parameters per element are needed to perform a calculation with any of the MNDO methods. Discussion of the types of procedure necessary to obtain these parameters will be left until Section 5.5.

The use of the NDDO approximation, and the other simplifications, makes calculations with MNDO methods much less expensive than *ab initio* ones. In particular, Fock matrix construction is no longer the most costly part of the computation due to the reduced number of two-electron integrals. Instead, the $O(N^3)$ matrix operations, especially diagonalization of the Fock matrix, are rate-limiting unless some of the specialized techniques mentioned at the end of the last section are employed.

4.6 Analysis of the charge density

Many properties of a system can be obtained from the QC procedures described in this chapter, not just the potential energy. One of the more useful of these is the total charge density whose analysis often constitutes a routine part of a QC study. The aim of this section is to summarize some of the simpler analyses that can be performed on this density although readers should bear in mind that a whole range of analyses of varying sophistication exist.

The total charge density of the system is defined as the sum of the charge densities due to the electrons and to the nuclei. It can be written as

$$\rho_t(\boldsymbol{r}) = \rho_e(\boldsymbol{r}) + \rho_n(\boldsymbol{r}) \tag{4.34}$$

where ρ denotes a charge density and the subscripts t, e and n stand for total, electron and nuclear, respectively.

An expression for the electron charge density is obtained by integrating the square of the wavefunction over all electronic coordinates, bar one. Remembering that electrons have a charge of minus one, the appropriate equations are

$$\begin{aligned}\rho_{\mathrm{e}}(\boldsymbol{r}) &= -\int_{\boldsymbol{r}_2} \mathrm{d}\boldsymbol{r}_2 \int_{\boldsymbol{r}_3} \mathrm{d}\boldsymbol{r}_3 \ldots \int_{\boldsymbol{r}_M} \mathrm{d}\boldsymbol{r}_M \, |\Psi(\boldsymbol{r}, \boldsymbol{r}_2, \ldots \boldsymbol{r}_M)|^2 \\ &= -\sum_{\mu\nu} P_{\mu\nu}\eta_\mu(\boldsymbol{r})\eta_\nu(\boldsymbol{r}) \end{aligned} \tag{4.35}$$

Equation (4.35) says that ρ_{e} can be written as a sum of products of basis functions multiplied by density matrix elements. This derivation is valid for any wavefunction as long as the density matrix is appropriately defined. For closed-shell RHF wavefunctions the density matrix elements are given by Equation (4.15), whereas for UHF wavefunctions they would be given by the sum of the elements coming from the density matrices of α and β spin.

The electron charge density, ρ_{e}, is a smoothly varying function that spreads over a wide region of space. By contrast, the nuclear density, ρ_{n}, is zero everywhere except at the atomic nuclei. This type of discrete behaviour can be expressed using the *Dirac delta function*, δ, which is a function that is non-zero only when its argument is zero and whose integral over all space is one. With this notation the nuclear density is

$$\rho_{\mathrm{n}}(\boldsymbol{r}) = \sum_i Z_i \delta(\boldsymbol{r} - \boldsymbol{r}_i) \tag{4.36}$$

The total charge density, ρ_{t}, is a complicated function of the space coordinate, $\boldsymbol{r}$, and analyses seek to determine quantities that somehow make its interpretation easier. One such set of quantities is the *multipole moments* of a charge distribution. They provide a convenient, hierarchical representation of how a distribution will interact with another, non-overlapping charge distribution or an external electric field and are important because some of them are accessible experimentally and so can be compared to calculated values.

The definition of a multipole moment requires the specification of a point in space, $\boldsymbol{R}_{\mathrm{c}}$, about which the multipole is calculated and it makes use of the spherical harmonic functions that we met in Section 4.5.2. The equation is

$$Q_{lm} = \int \mathrm{d}\boldsymbol{r} \rho_{\mathrm{t}}(\boldsymbol{r}) S_{lm}(\theta, \phi) \, |\boldsymbol{r} - \boldsymbol{R}_{\mathrm{c}}|^l \tag{4.37}$$

where the integration is over all space and θ and ϕ are the spherical polar angles about the point $\boldsymbol{R}_{\mathrm{c}}$. The first few terms in the series are given names so that the $l = 0$ term is the *monopole* and the subsequent terms are the *dipole* ($l = 1$), the *quadrupole* ($l = 2$), the *octupole* ($l = 3$) and the *hexadecapole* ($l = 4$), respectively.

The first two multipoles of the total charge density, ρ_t, are straightforwardly evaluated. The monopole expression reduces to an integral over the charge distribution because S_{00} is a constant

$$Q_{00} = \int \mathrm{d}\boldsymbol{r}\rho_t(\boldsymbol{r}) \tag{4.38}$$

This integral is simply the total charge of the system. The dipole moment consists of three terms and can be written as a vector denoted $\boldsymbol{\mu}$

$$\boldsymbol{\mu} = \int \mathrm{d}\boldsymbol{r}\rho_t(\boldsymbol{r})(\boldsymbol{r} - \boldsymbol{R}_c) \tag{4.39}$$

The total charge is always independent of the multipole origin, $\boldsymbol{R}_c$, whereas the dipole moment will only be independent if the system's total charge is zero.

A second type of analysis is the evaluation of *atomic charge populations*. Its aim is the estimation of *effective charges* for the atoms in a system by partitioning the charge density between them. Two aspects of charge population analyses should be emphasized. First, a unique way of effecting this analysis does not exist. Clearly the nuclear charge belongs to its atom but a choice has to be made of how to divide the electron density in an arbitrary region of space between atoms. Second, atomic charges, unlike multipole moments, cannot be measured experimentally and so they are mostly useful for interpretative purposes.

Many recipes for calculating atomic charges exist, but probably the most widespread is one due to R. S. Mulliken, although it only works for basis sets consisting of atom-centred functions, such as Gaussians. Consider the integral of the expression for the electron charge density which, using Equation (4.35), is

$$\int \mathrm{d}\boldsymbol{r}\rho_e(\boldsymbol{r}) = -\sum_\mu \sum_\nu P_{\mu\nu}S_{\mu\nu} \tag{4.40}$$

$$= Q_e \tag{4.41}$$

This expression relates the total electronic charge, Q_e, to a sum of 'charge-like' terms of the form $P_{\mu\nu}S_{\mu\nu}$, each of which only involves a pair of basis functions. As the basis functions are atom-centred, the Mulliken analysis supposes that each term can be split between the atoms upon which the functions are centred and that, in the absence of any other information, the most unbiased splitting occurs when half of the term is given to each atom.

It takes a little juggling, but the expression for the Mulliken charge, q_i, on atom i, including the nuclear charge, is

$$q_i = Z_i - \sum_{\mu \in i} \sum_\nu P_{\mu\nu}S_{\mu\nu} \tag{4.42}$$

where the first sum is over basis functions, μ, centred on atom i and the second sum is over all basis functions no matter to which atom they belong. Mulliken

charges are simple to calculate but they are notoriously fickle because charges for identical atoms can change substantially if calculations are repeated with different basis sets.

For spin-unrestricted methods it is often useful to be able to identify the atoms upon which there are differences in the populations of electrons with α and β spin. This is easily done within the Mulliken scheme by replacing the total charge density by the *spin density*, which is just the difference in the densities of the α and β electrons, i.e. $\rho_{\mathrm{e}}^{\alpha} - \rho_{\mathrm{e}}^{\beta}$.

4.7 Example 5

In this book, all QC calculations will be done with semi-empirical methods of the MNDO-type. Within pDynamo, these are specified using the class `QCModelMNDO` whose definition is:

Class `QCModelMNDO`

A class to represent an MNDO-type semi-empirical QC model.

Constructor

Construct an instance of `QCModelMNDO`.

Usage: `new = QCModelMNDO ( method )`

`method` is a string that indicates the method to use. Possible values are `"am1"`, `"mndo"` and `"pm3"`. This argument can be left out, in which case the AM1 method is chosen.

`new` is the new instance of `QCModelMNDO`.

Remarks: MNDO methods have not been parametrized for some elements, including many of the transition metals. An error will occur if an attempt is made to perform calculations on systems with these atoms.

Attributes

`label` is a string containing the name or a short description of the model.

`QSPINRESTRICTED` is a *Boolean* that indicates whether the model is spin-restricted or spin-unrestricted. Booleans are Python variables that can take only two values, either `True` or `False`.

To specify the charge and the spin multiplicity of the system another class, `ElectronicState`, is required. Its specification is:

Class `ElectronicState`

A class to represent the electronic state of a system.

Constructor

Generate an instance of `ElectronicState` with a given charge and spin multiplicity.

Usage:
```
new = ElectronicState ( charge = 0,
                        multiplicity = 1 )
```

`charge` is a keyword integer argument that gives the total charge of the system.

`multiplicity` is a keyword argument that defines the spin multiplicity of the system. It can either be a positive integer or be one of the strings `"singlet"`, `"doublet"`, `"triplet"`, `"quartet"`, `"quintet"`, `"sextet"` or `"septet"`. These correspond to integer multiplicities of 1–7, respectively.

`new` is the new instance of `ElectronicState`.

A QC model and an electronic state must be assigned to an instance of the `System` class if a QC energy is to be calculated. These and other related tasks are performed using the following extensions to the `System` class:

Class `System`

QC-related methods and attributes.

Method `AtomicCharges`

Calculate the atomic charges for a system using a Mulliken population analysis.

Usage:
```
charges = system.AtomicCharges ( )
```

`system` is the instance of `System` for which the charges are to be determined.

`charges` are the calculated atomic charges returned as an instance of `Vector`.

Method `DefineQCModel`

Assign a QC model to a system.

Usage: `system.DefineQCModel ( qcmodel )`

`qcmodel` is the QC model which is to be used for calculating the QC energy for the system.

`system` is the instance of `System` for which the QC model is being defined.

Method `DipoleMoment`

Calculate the dipole moment for a system.

Usage: `dipole = system.DipoleMoment ( )`

`system` is the instance of `System` for which the dipole is to be determined.

`dipole` is the dipole moment returned as an instance of the class `Vector3`.

Remarks: The dipole is calculated at the centre of geometry of the system.

Method `Energy`

Calculate the potential energy for a system.

Usage: `energy = system.Energy ( log = logfile )`

`log` is an instance of `LogFileWriter` to which output about the energy calculation is to occur. By default this will be to `logfile` which is predefined by the pDynamo library. Output can be suppressed entirely by setting this argument to `None`.

`system` is the instance of `System` for which the energy is to be determined.

`energy` is the calculated potential energy.

Remarks: The energy is calculated for the structure defined in the system's `coordinates3` attribute.

Attributes

`electronicstate` is the electronic state for the system. It is usual to define it by direct assignment with an instance of the class `ElectronicState`. This attribute is not used directly in any of the examples in this book because all the systems that we study are singlet states of zero charge. This is the default state for a system if none is explicitly given.

Example 5 employs some of the QC capabilities of pDynamo by calculating the potential energy and various electronic properties for a water molecule with three different semi-empirical methods. The program is:

```
1  """Example 5."""
2
3  from Definitions import *
4
5  # . Define the energy models.
6  energymodels = [ QCModelMNDO ( "am1"  ), \
                    QCModelMNDO ( "mndo" ), \
                    QCModelMNDO ( "pm3"  ) ]
7
8  # . Get the filename.
9  filename = os.path.join ( xyzpath, "water.xyz" )
10
11 # . Loop over the energy models.
12 results = []
13 for model in energymodels:
14     molecule = XYZFile_ToSystem ( filename )
15     molecule.DefineQCModel ( model )
16     molecule.Summary ( )
17     energy  = molecule.Energy ( )
18     charges = molecule.AtomicCharges ( )
19     dipole  = molecule.DipoleMoment  ( )
20     results.append ( ( model.label, energy, charges, \
                          dipole.Norm2 ( ) ) )
21
22 # . Output the results.
23 table = logfile.GetTable ( columns = [ 10, 20, 20, 20, 20, 20 ] )
24 table.Start  ( )
25 table.Title  ( "Energy Model Results for Water" )
26 table.Heading ( "Model"  )
27 table.Heading ( "Energy" )
28 table.Heading ( "Charges", columnspan = 3 )
29 table.Heading ( "Dipole" )
30 for ( label, energy, charges, dipole ) in results:
31     table.Entry ( label )
32     table.Entry ( "%.1f" % ( energy, ) )
33     for charge in charges: table.Entry ( "%.3f" % ( charge, ) )
34     table.Entry ( "%.3f" % ( dipole, ) )
35 table.Stop ( )
```

Line 6 creates a list, `energymodels`, with the three different QC models that are to be tested.

Line 12 defines an empty list, `results`, that will be used to store quantities that are calculated with the QC models.

Lines 13–20 loop over the QC models in the list `energymodels`. At each iteration, an instance of a water molecule is created (*line 14*), its QC model defined (*line 15*) and some properties calculated (*lines 17–19*). Note that the energy must be calculated before the other analyses because it is during the energy calculation that the wavefunction and, hence, the electron density for the system are determined.

The results of the calculations for each model are appended, as a tuple, to the list `results` on *line 20*. Only the magnitude of the dipole is saved and not its individual components. This is done with the method `Norm2` of the class `Vector3`.

Lines 23–35 output the results as a table. The syntax is similar to what we have met before except for *Line 30*. This shows how the tuple of quantities stored for each QC model can be unpacked and the elements assigned directly to the variables `label`, `energy`, `charges` and `dipole`.

4.8 Derivatives of the potential energy

Up to now the potential energy of a system and how it is calculated has been the centre of attention but, in many applications, it is equally or even more important to know the values of derivatives of the energy with respect to certain parameters. In such cases, it is crucial that the derivatives of the energy be calculable in as efficient a manner as the energy itself. A useful property of all the ways of determining the potential energy that are discussed in this book is that most of the derivatives we require can be calculated *analytically*, which means that it is possible to derive explicit formulae for the derivatives by direct differentiation.

The alternative to analytically calculated derivatives is derivatives that are calculated *numerically*. One particularly common and reasonably effective way of calculating numerical derivatives is to use a *central-step finite-difference method*. If $\mathcal{V}$ is the potential energy of a system and p is the parameter with respect to which the derivative is required, the first derivative is approximated as

$$\frac{\partial \mathcal{V}(p)}{\partial p} \sim \frac{\mathcal{V}(p+\delta p) - \mathcal{V}(p-\delta p)}{2\ \delta p} \tag{4.43}$$

where δp is a small change in the parameter value. When calculating the derivative with respect to p, only the value of p is changed and the values of all the other parameters in the function are kept constant.

Analytically calculated derivatives have a number of advantages over those that are numerically calculated. First, they are almost always more accurate. Analytic derivatives will be accurate to machine precision whereas the accuracy

of numerical ones will depend to a large extent on the values of the finite steps (δp) taken in the numerical algorithm. Second, analytic derivatives are usually much less expensive, especially when the number of derivatives to be calculated is large.

Several different types of derivative of the energy are useful, but by far the most common are the derivatives of the energy with respect to the positions of the atoms. The first derivatives or *gradients* are employed most extensively throughout this book, although the second derivatives are also necessary in certain applications. Because the potential energy is dependent upon all the coordinates of the atoms in a system, there will be $3N$ first derivatives of the energy with respect to the coordinates, i.e.

$$\boldsymbol{g}_i = \frac{\partial \mathcal{V}}{\partial \boldsymbol{r}_i} = \begin{pmatrix} \frac{\partial \mathcal{V}}{\partial x_i} \\ \frac{\partial \mathcal{V}}{\partial y_i} \\ \frac{\partial \mathcal{V}}{\partial z_i} \end{pmatrix} \quad \forall \quad i = 1, \ldots, N \tag{4.44}$$

The second derivatives of the energy with respect to the coordinates of two atoms, i and j, are

$$\mathbf{h}_{ij} = \frac{\partial^2 \mathcal{V}}{\partial \boldsymbol{r}_i \, \partial \boldsymbol{r}_j} = \begin{pmatrix} \frac{\partial^2 \mathcal{V}}{\partial x_i \partial x_j} & \frac{\partial^2 \mathcal{V}}{\partial x_i \partial y_j} & \frac{\partial^2 \mathcal{V}}{\partial x_i \partial z_j} \\ \frac{\partial^2 \mathcal{V}}{\partial y_i \partial x_j} & \frac{\partial^2 \mathcal{V}}{\partial y_i \partial y_j} & \frac{\partial^2 \mathcal{V}}{\partial y_i \partial z_j} \\ \frac{\partial^2 \mathcal{V}}{\partial z_i \partial x_j} & \frac{\partial^2 \mathcal{V}}{\partial z_i \partial y_j} & \frac{\partial^2 \mathcal{V}}{\partial z_i \partial z_j} \end{pmatrix} \tag{4.45}$$

Whereas the first derivatives can be considered to form a vector of dimension $3N$, the second derivatives of the energy with respect to the coordinates form a $3N \times 3N$ matrix. Owing to the properties of differentiation, the second-derivative matrix is symmetric and only $3N \times (3N+1)/2$ components of the matrix will be different. This means that the second derivatives involving the coordinates of the i and j particles will be the same, no matter in which order the differentiation is performed; i.e. $\partial^2 \mathcal{V} / \partial x_i \, \partial y_j = \partial^2 \mathcal{V} / \partial y_j \, \partial x_i$. If the particles are the same ($i = j$) then, of course, the symmetric nature of the second derivatives is immediately apparent. The second-derivative matrix is often called the *Hessian*.

4.8.1 Quantum chemical derivatives

To make the discussion above concrete, let us consider how to calculate derivatives for QC algorithms. We shall focus on the HF method but the reasoning is analogous for other approaches. The potential energy of a system described within

the closed-shell RHF approximation is given by Equation (4.13). Differentiating with respect to an atomic coordinate, x_i, gives

$$\frac{\partial \mathcal{V}}{\partial x_i} = \frac{1}{2}\frac{\partial}{\partial x_i}\left[\sum_{\mu\nu} P_{\mu\nu}\left\{H_{\mu\nu} + F_{\mu\nu}\right\} + \mathcal{V}_{\mathrm{nn}}\right] \tag{4.46}$$

$$= \sum_{\mu\nu} P_{\mu\nu}\left\{\frac{\partial H_{\mu\nu}}{\partial x_i} + \frac{1}{2}\sum_{\lambda\sigma} P_{\lambda\sigma}\frac{\partial}{\partial x_i}\left[(\mu\nu|\lambda\sigma) - \frac{1}{2}(\mu\lambda|\lambda\nu)\right]\right\} + \sum_{\mu\nu}\frac{\partial P_{\mu\nu}}{\partial x_i}F_{\mu\nu} + \frac{\partial \mathcal{V}_{\mathrm{nn}}}{\partial x_i} \tag{4.47}$$

Getting to Equation (4.47) from (4.46) requires a little manipulation. In particular, the elements of the Fock matrix, $F_{\mu\nu}$, need to be expanded, using Equation (4.17), and differentiated individually. Some of them can subsequently be recombined, after a permutation of basis-function indices, to give the second term on the right-hand side of Equation (4.47). Although the details of the differentiation have been omitted, the important point is that the final expression has three terms. The first involves derivatives of elements of the one-electron matrix, $H_{\mu\nu}$, and of two-electron integrals, $(\mu\nu|\lambda\sigma)$, multiplied by elements of the density matrix, $P_{\mu\nu}$, whereas the third is the derivative of the nuclear repulsion energy. These terms can be calculated analytically because the one- and two-electron integrals and the nuclear repulsion energy are known functions of the atomic positions and so their derivatives can be determined explicitly.

The second term, by contrast, is more complicated as it involves the derivatives of density matrix elements multiplied by elements of the Fock matrix. Although the density matrix is not an explicit function of the atomic positions, it depends implicitly upon them because it and the electronic wavefunction change, and must be redetermined, when the nuclei move. This term can be simplified by expressing the density matrix in terms of molecular orbitals, Equation (4.15), and then employing the Roothaan–Hall equation, Equation (4.27), as follows:

$$\sum_{\mu\nu}\frac{\partial P_{\mu\nu}}{\partial x_i}F_{\mu\nu} = \sum_{\mu\nu}\frac{\partial \sum_a 2c_{\mu a}c_{\nu a}}{\partial x_i}F_{\mu\nu} \tag{4.48}$$

$$= 4\sum_a\sum_{\mu\nu}\frac{\partial c_{\mu a}}{\partial x_i}F_{\mu\nu}c_{\nu a} \tag{4.49}$$

$$= 4\sum_a \epsilon_a \sum_{\mu\nu}\frac{\partial c_{\mu a}}{\partial x_i}S_{\mu\nu}c_{\nu a} \tag{4.50}$$

Equation (4.49) is obtained from Equation (4.48) by interchanging the indices μ and ν and using the fact that the Fock matrix is symmetric.

Equation (4.50) still involves derivatives of the molecular orbital coefficients but it can be reduced further by using the orthonormality condition for the molecular orbitals, Equation (4.10). Differentiation of this condition for the orbital a gives

$$\frac{\partial}{\partial x_i}\sum_{\mu\nu} c_{\mu a} S_{\mu\nu} c_{\nu a} = 0 \tag{4.51}$$

$$= 2\sum_{\mu\nu} \frac{\partial c_{\mu a}}{\partial x_i} S_{\mu\nu} c_{\nu a} + \sum_{\mu\nu} c_{\mu a} \frac{\partial S_{\mu\nu}}{\partial x_i} c_{\nu a} \tag{4.52}$$

Rearranging this expression and substituting into Equation (4.50) gives

$$4\sum_a \epsilon_a \sum_{\mu\nu} \frac{\partial c_{\mu a}}{\partial x_i} S_{\mu\nu} c_{\nu a} = -2\sum_a \epsilon_a \sum_{\mu\nu} c_{\mu a} \frac{\partial S_{\mu\nu}}{\partial x_i} c_{\nu a} \tag{4.53}$$

$$= \sum_{\mu\nu} \frac{\partial S_{\mu\nu}}{\partial x_i} W_{\mu\nu} \tag{4.54}$$

where the *energy-weighted density matrix* has been defined as

$$W_{\mu\nu} = -2\sum_a \epsilon_a c_{\mu a} c_{\nu a} \tag{4.55}$$

Expression (4.54), like the first and third terms on the right-hand side of Equation (4.47), can be evaluated analytically as it involves derivatives of the overlap integrals, which are explicit functions of the atomic positions.

The term in Equation (4.54) is often named the *Pulay term* after P. Pulay who was one of the first workers to derive derivative expressions within the HF approximation. Terms of this type are of general occurrence and are not limited to quantum chemical methods. They arise whenever one has an expression that is a function both of a set of independent variables (the atomic positions for the QC case) and of a set of variables (the density matrix elements or MO coefficients) that depends implicitly upon the independent set, either as a result of an optimization process or in some other way. Note that the Pulay term will be zero if the overlap matrix is set to the identity matrix, either because the basis functions are orthogonal (as occurs for plane-waves) or because it is assumed to be so (as in the MNDO semi-empirical methods).

Expressions for the second, and higher, derivatives may be obtained by further differentiation but the resulting formulae are much more complicated and so will not be given here. In addition, starting with the second derivatives, it is not possible to avoid the calculation of the derivatives of the density matrix elements (or molecular orbital coefficients) and so these must be determined explicitly. For HF methods this is done using the *coupled perturbed* HF (CPHF)

algorithm. The calculation of many other molecular properties, not only the second derivatives of the energy with respect to the atomic positions, also requires the derivatives of the density or wavefunction and so makes use of coupled perturbed methods.

4.9 Example 6

The majority of the simulation algorithms to be described in this book modify the structure or the geometry of a system using information that is obtained by the calculation of the potential energy and, usually, its derivatives with respect to a set of geometrical variables. Many of the algorithms, however, are general in that they can act upon any function that depends upon a set of variables. To take advantage of this generality, simulation algorithms in pDynamo are written so that they work upon instances of a class called `ObjectFunction`. This class is then subclassed so as to create object functions of the appropriate type.

`SystemGeometryObjectFunction` is the subclass of `ObjectFunction` that provides the interface between the simulation algorithms and instances of the class `System`. It defines the potential energy of a system as the function to be manipulated and the variables as those that determine the system's structure – usually its Cartesian coordinates. `SystemGeometryObjectFunction` need rarely be used directly because pDynamo's simulation algorithms provide helper functions that should suffice for most routine purposes. Nevertheless, a very basic outline of the class is introduced here because it is required in Example 6.

Class `SystemGeometryObjectFunction`

A class used to create an object function of a system's potential energy. The variables are those that define the geometry of the system – in this case the Cartesian coordinates.

Constructor

Construct an instance of `SystemGeometryObjectFunction` given a system.

Usage: `new = SystemGeometryObjectFunction ( system )`

`system` is the instance of `System` for which the object function is to be defined. The system must have a valid set of coordinates and a defined energy model.

`new` is the new instance of `SystemGeometryObjectFunction`.

Method `TestGradients`

Calculate gradients for the function analytically and numerically, using a central-step finite-difference algorithm. After calculation, the differences between the two sets of gradients are compared and printed.

Usage: `of.TestGradients ( delta = 1.0e-4 )`

`delta` is a keyword argument that gives the step length for the numerical derivative calculation. If the argument is absent, a value of 10^{-4} Å is assumed. This is usually a good compromise value, although, to be certain, it is often necessary to try several values for the step size to see how the values of the derivatives change.

`of` is the instance of `SystemGeometryObjectFunction` for which the gradients are to be calculated.

Remarks: This method is usually used for testing purposes so as to ensure that the analytical derivatives of a function have been correctly implemented.

Example 6 is straightforward. It employs the class and its methods described above and has as its goal the comparison of the analytically and numerically calculated first derivatives of the potential energy with respect to the system's Cartesian coordinates.

```
1  """Example 6."""
2
3  from Definitions import *
4
5  # . Generate the molecule.
6  molecule = XYZFile_ToSystem ( \
               os.path.join ( xyzpath, "bala_c7eq.xyz" ) )
7  molecule.DefineQCModel ( QCModelMNDO ( ) )
8  molecule.Summary ( )
9
10 # . Create an object function for the molecule.
11 of = SystemGeometryObjectFunction ( molecule )
12
13 # . Test the gradients.
14 of.TestGradients ( )
```

Lines 6–8 define the system that is to be studied and its energy model. They resemble closely the equivalent lines of Example 5.

Line 11 creates an object function for the molecule.

Line 14 does the comparison of the analytical and numerical gradients and prints out the results. Typically the biggest differences between components of the two sets of derivatives are $\sim 10^{-4}$ kJ mol^{-1} Å^{-1}.

Exercises

4.1 Taking the program of Example 5 as a model, calculate the energy and multipoles of different molecules with different QC methods. How do the results differ? Can one compare the results with values obtained from experiment and, if so, how?

4.2 Calculate the energies and properties of a molecule using the same QC approximation but at different molecular geometries. A good example is to consider a simple molecule, such as molecular hydrogen, H_2, or ethane, C_2H_6, and determine the potential energy curves for dissociation of the molecule into two fragments. How does the energy change as a function of the distance between the two fragments? Do the dissociated fragments have a radical character or are they cationic or anionic? How do the results change on going from a spin-restricted to a spin-unrestricted model?

5

Molecular mechanics

5.1 Introduction

In the last chapter we discussed quantum chemical methods for calculating the potential energy of a system whereas in this chapter we present an alternative class of approaches, those that use empirical energy functions. To start, though, a point of notation will be clarified. Several different terms are employed to denote empirical energy functions in the literature and, no doubt, inadvertently, in this book. Common terms, which all refer to the same thing, include *empirical energy function*, *potential energy function*, *empirical potential* and *force field*. The use of empirical potentials to study molecular conformations is often termed *molecular mechanics* (MM).

Some of the earliest empirical potentials were derived by vibrational spectroscopists interested in interpreting their spectra (this was, in fact, the origin of the term 'force field'), but the type of empirical potential that is described here was developed at the end of the 1960s and the beginning of the 1970s. Two prominent proponents of this approach were S. Lifson and N. Allinger. These types of force field are usually designed for studying conformations of molecules close to their equilibrium positions and so would be inappropriate for studying processes, such as chemical reactions, in which this is not the case.

5.2 Typical empirical energy functions

This section presents the general form of the empirical energy functions that are used in molecular simulations. A diversity exists, because the form of an empirical potential function is, to some extent, arbitrary, but most functions have two categories of terms that deal with the *bonding* and the *non-bonding interactions* between atoms, respectively. These will be discussed separately.

5.2.1 Bonding terms

The bonding energy terms are those that help define the bonding or covalent structure of the molecule, i.e. its local shape. In a typical, simple force field, the bonding or covalent energy, $\mathcal{V}_{\text{cov}}$, will consist of a sum of terms for the *bond*, *angle*, *dihedral* (or *torsion*) and *out-of-plane distortion* (or *improper dihedral*) energies:

$$\mathcal{V}_{\text{cov}} = \mathcal{V}_{\text{bond}} + \mathcal{V}_{\text{angle}} + \mathcal{V}_{\text{dihedral}} + \mathcal{V}_{\text{improper}} \tag{5.1}$$

The bond energy is often taken to have a *harmonic* form:

$$\mathcal{V}_{\text{bond}} = \sum_{\text{bonds}} \frac{1}{2} k_b (b - b_0)^2 \tag{5.2}$$

where k_b is the *force constant* for the bond, b is the actual bond length in the structure between the two atoms defining the bond and b_0 is the *equilibrium distance* for the bond. The sum runs over all the bonds that have been defined in the system. Because the energy is harmonic in form (see Figure 5.1) it means that the energy of the bond will increase steadily without limit as it is distorted from its equilibrium value, b_0.

Harmonic terms are sufficient for many studies, but sometimes it is important to have a form for the bond energy that permits dissociation. An example would be if a reaction were being studied. One form that does this is the *Morse potential* which is shown in Figure 5.2. The Morse energy, $\mathcal{V}_{\text{Morse}}$, is given by

$$\mathcal{V}_{\text{Morse}} = \sum_{\text{bonds}} D\{\exp[-a(b - b_0)] - 1\}^2 - D \tag{5.3}$$

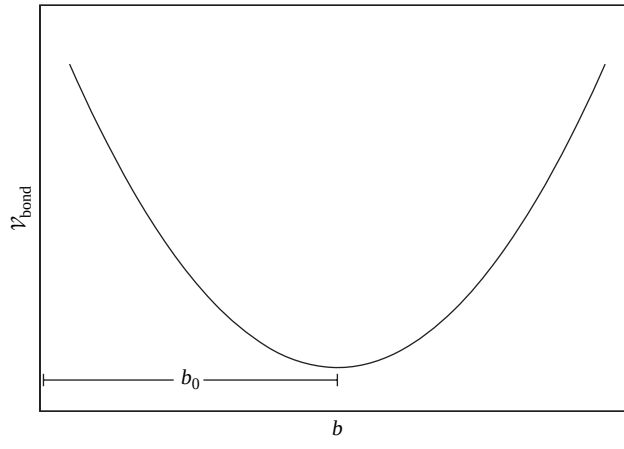


Fig. 5.1. The harmonic bond energy term.

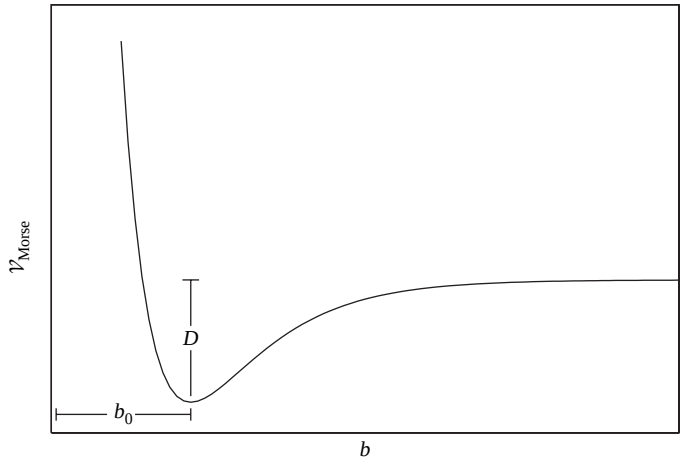


Fig. 5.2. The Morse function bond energy term.

where the two new parameters are D, which is the dissociation energy of the bond, and a, which determines the width of the potential well.

The angle energy term is designed to imitate how the energy of a bond angle changes when it is distorted away from its equilibrium position. Like the bond energy term it too is often taken to be harmonic:

$$\mathcal{V}_{\text{angle}} = \sum_{\text{angles}} \frac{1}{2} k_\theta (\theta - \theta_0)^2 \tag{5.4}$$

The extra parameters are similar to those of the bond energy – k_θ is the force constant for the angle and θ_0 is its equilibrium value. The sum runs over all the angles in the system and each angle is defined in the same way as described in Section 3.3.

The third type of bonding term is the term that describes how the energy of a molecule changes as it undergoes a rotation about one of its bonds, i.e. the dihedral or torsion energy for the system. In contrast to the bond and angle terms a harmonic form for the dihedral energy is not usually appropriate. This is because, for many dihedral angles in molecules, the whole range of angles from 0° to 360° can be accessible with not too large differences in energy. Such effects can be reproduced with a periodic function that is continuous throughout the complete range of possible angles (see Figure 5.3). The dihedral energy can then be written as

$$\mathcal{V}_{\text{dihedral}} = \sum_{\text{dihedrals}} \frac{1}{2} V_n [1 + \cos(n\phi - \delta)] \tag{5.5}$$

Once again the sum is over all the dihedrals that are defined in the system and the form of the dihedral angle is the same as that given in Equation (3.5). In the

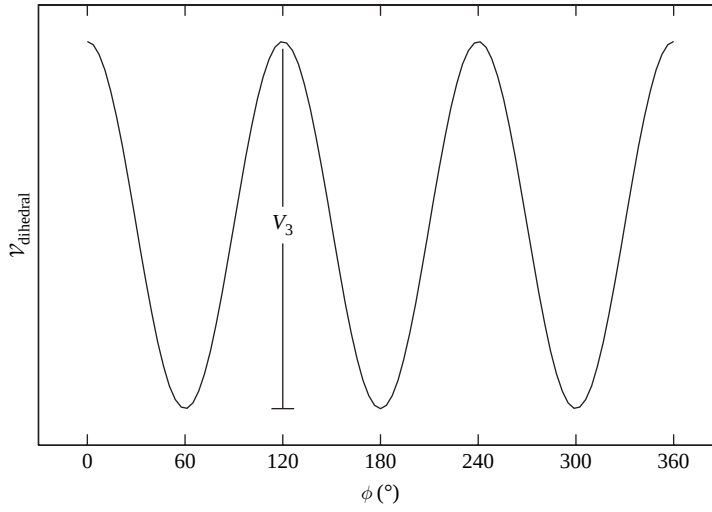


Fig. 5.3. A dihedral angle energy term with a periodicity of 3.

formula, n is the *periodicity* of the angle (which determines how many peaks and wells there are in the potential), δ is the *phase* of the angle and V_n is the force constant. Often δ is restricted to taking the values 0° or 180°, in which case it is only the sign of the cosine term in the expression that will change. It is to be noted that the periodicity of each term in the sum can change depending upon the type of dihedral and that values of n from 1 to 6 are most commonly used. It is also worth remarking that, in many force fields, multiple terms with different periodicities are used for some dihedral angles. Thus, it may be that a single term with a periodicity of 3, for example, is unable to reproduce accurately the change in energy during a torsional motion and terms with other periodicities will be added.

The fourth term in the sum in Equation (5.1) is a more complicated one that describes the energy of out-of-plane motions. It is often necessary for planar groups, such as sp^2 hybridized carbons in carbonyl groups and in aromatic systems, because it is found that use of dihedral terms alone is not sufficient to maintain the planarity of these groups during calculations. A common way to avoid this problem is to define an *improper dihedral angle*, which differs from the *proper dihedral angle* in that the atoms which define the dihedral angle, *i–j–k–l*, are not directly bonded to each other. The calculation of the angle, however, remains exactly the same. An example is shown in Figure 5.4. With this definition of an improper dihedral angle, which we denote by ω, some force fields use the same

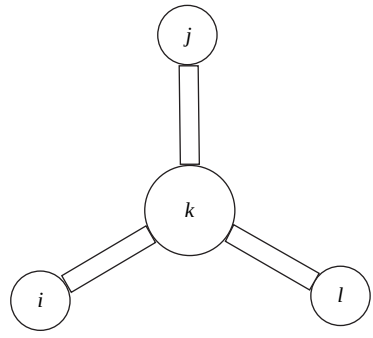


Fig. 5.4. The arrangement of atoms in an improper dihedral angle.

form for the energy as that in Equation (5.5), i.e.

$$\mathcal{V}_{\text{improper}} = \sum_{\text{impropers}} \frac{1}{2} V_n [1 + \cos(n\omega - \delta)] \tag{5.6}$$

while others employ a harmonic form:

$$\mathcal{V}_{\text{improper}} = \sum_{\text{impropers}} \frac{1}{2} k_\omega (\omega - \omega_0)^2 \tag{5.7}$$

where k_ω and ω_0 are the force constant for the energy term and the equilibrium value of the improper dihedral angle, respectively. The improper dihedral angle, ω, is not the only variable that is used to define the distortion due to out-of-plane motions. Another common one is the angle between one of the bonds to the central atom and the plane defined by the central atom and the other two atoms (e.g. the bond *i*–*k* and the plane *jkl* in Figure 5.4).

The four terms mentioned above are the only bonding terms that we shall consider, although other types can be encountered in some force fields. In general, the extra terms are added to obtain better agreement with experimental data (especially vibrational spectra) but they increase the complexity of the force field and the number of parameters that need to be obtained. Extra terms that are sometimes added include bond and angle terms of the same form as Equations (5.2) and (5.4) except that the terms are no longer harmonic – linear, cubic and quartic terms are possible. Other types sometimes seen are *cross-terms* that couple distortions in different internal coordinates. For example, a bond/angle cross-term could be proportional to $(b - b_0)(\theta - \theta_0)$.

5.2.2 Non-bonding terms

The bonding energy terms help to define the covalent energy of a molecule. The non-bonding terms describe the interactions between the atoms of different

molecules or between atoms that are not directly bonded together in the same molecule. These interactions help to determine the overall conformation of a molecular system.

The non-bonding interactions arise from the interactions between the electronic distributions surrounding different atoms. The theory of intermolecular interactions is well developed and leads to the identification of a number of important types of interaction. At short range the interactions are primarily repulsive due to the interactions between the electron clouds and to the purely quantum mechanical effect of *exchange repulsion*, which arises when the two clouds are pushed together. At long ranges there are several important classes of interaction. The first are the electrostatic interactions that arise from the interaction of the charge distributions (including the nuclei) about each molecule or portion of a molecule. Second are the *dispersion* interactions that are produced by correlated fluctuations in the charge distributions of the two groups. Finally, there are *induced* or *polarization* interactions that are caused by the distortion of the charge distribution of a molecule as it interacts with neighbouring groups.

The non-bonding terms in an empirical force field attempt to reproduce all these types of interaction. Here we shall consider a non-bonding energy consisting of the sum of three terms:

$$\mathcal{V}_{\text{nb}} = \mathcal{V}_{\text{elect}} + \mathcal{V}_{\text{LJ}} + \mathcal{V}_{\text{polar}} \tag{5.8}$$

The *electrostatic energy*, $\mathcal{V}_{\text{elect}}$, mimics the energy arising from the electrostatic interactions between two charge distributions. As we saw in the last chapter, charge distributions, and the electrostatic energies arising from them, can be quite easily evaluated using quantum chemical methods. The aim with force fields though is different. We seek models for the charge distribution that are simple enough to allow fast calculation of the electrostatic energy but sufficiently accurate that the major effects due to the interaction are reproduced. The simplest representation of a charge distribution and the one that is most widely used is one in which a fractional charge is assigned to each atom. This is the total nett charge of the atom obtained as the sum of the nuclear charge and the charge in the part of the electron cloud that surrounds it. The electrostatic energy is calculated as

$$\mathcal{V}_{\text{elect}} = \frac{1}{4\pi\epsilon_0\epsilon} \sum_{ij \text{ pairs}} \frac{q_i q_j}{r_{ij}} \tag{5.9}$$

where q_i and q_j are the fractional charges on atoms i and j and r_{ij} is the distance between the two particles. The terms in the prefactor are $1/(4\pi\epsilon_0)$, which is the standard term when calculating electrostatic interactions in the MKSA (metre, kilogram, second, ampere) system of units, and ϵ, which is the dielectric constant that will have the value 1 when the system is in vacuum. The sum in Equation

(5.9) runs over all pairs of atoms for which an electrostatic interaction is to be calculated. Note that the fractional charges on the atoms are constants and do not change during a calculation.

It is possible to define other representations of the charge distribution. For example, instead of fractional charges at the atoms' centres (i.e. on the nuclei), charges could be assigned off-centre along bonds or higher moments, such as dipoles, could be used. Such representations are not usually favoured because they are more complex and more expensive than the simple point-charge model and they have not been shown to give proportionately better results.

The second term in Equation (5.8) is the *Lennard-Jones energy* which mimics the long-range dispersion interactions and the short-range repulsive interactions. It has the form

$$\mathcal{V}_{\mathrm{LJ}} = \sum_{ij \text{ pairs}} \frac{A_{ij}}{r_{ij}^{12}} - \frac{B_{ij}}{r_{ij}^{6}} \tag{5.10}$$

where A_{ij} and B_{ij} are positive constants whose values depend upon the types of the atoms, i and j, and the sum is over all pairs of atoms for which the interaction is to be calculated.

The shape of the Lennard-Jones potential is plotted in Figure 5.5. The repulsive part of the curve is produced by the $1/r_{ij}^{12}$ term and the attractive part by $1/r_{ij}^{6}$. The inverse sixth power form of the attraction arises naturally from the theory of dispersion interactions. The choice of an inverse twelfth power for the repulsion is less well founded and other forms for the repulsion have been used, including other inverse powers, such as eight and ten, and an exponential form that leads to the so-called *Buckingham potential*. Most simple force fields seem to use the Lennard-Jones form.

To complete the specification of the Lennard-Jones energy it is necessary to have a recipe for determining the parameters A_{ij} and B_{ij}. These are usually defined in terms of the depth of the Lennard-Jones well, ε_{ij}, and either the distance at which the energy of the interaction is zero, s_{ij}, or the position of the bottom of the well, σ_{ij} (see Figure 5.5). With these parameters the Lennard-Jones interaction between atoms i and j, $\mathcal{V}_{\mathrm{LJ}}^{ij}$, takes the form

$$\mathcal{V}_{\mathrm{LJ}}^{ij} = 4\varepsilon_{ij}\left[\left(\frac{s_{ij}}{r_{ij}}\right)^{12} - \left(\frac{s_{ij}}{r_{ij}}\right)^{6}\right] \tag{5.11}$$

$$= \varepsilon_{ij}\left[\left(\frac{\sigma_{ij}}{r_{ij}}\right)^{12} - 2\left(\frac{\sigma_{ij}}{r_{ij}}\right)^{6}\right] \tag{5.12}$$

where $\sigma_{ij}^{6} = 2s_{ij}^{6}$.

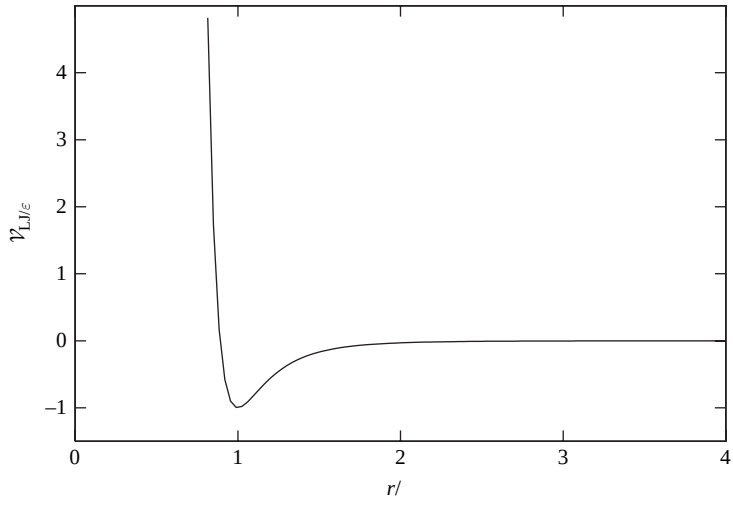


Fig. 5.5. The Lennard-Jones energy for a pair of atoms.

Although each of the parameters, ε_{ij}, s_{ij} and σ_{ij}, depends formally on two atoms, it is normal to specify a set of *combination rules* so that they can be defined from the parameters for single atoms. It is usual to use the geometrical mean as a combination rule for the well depths:

$$\varepsilon_{ij} = \sqrt{\varepsilon_{ii}\varepsilon_{jj}} \tag{5.13}$$

For the distance parameters both arithmetical and geometrical mean combination rules are common. For the σ_{ij} parameter, for example, we have either

$$\sigma_{ij} = \sqrt{\sigma_{ii}\sigma_{jj}} \tag{5.14}$$

or

$$\sigma_{ij} = \frac{1}{2}\left(\sigma_{ii} + \sigma_{jj}\right) \tag{5.15}$$

The third type of non-bonding energy term considered in this section is the *polarization energy*. In contrast to the previous two terms, the electrostatic and Lennard-Jones energies, this energy term is not a standard term in many force fields and it will not be used in any of the calculations in this book. However, it is instructive to introduce it here for a number of reasons. First, polarization interactions are important in many systems. Second, the nature of the polarization energy term is very different from that of the other force field terms that have been discussed and resembles, in many respects, the quantum chemical terms that were discussed in Chapter 4. Third, it is a term that will be increasingly used in future molecular modeling studies.

As mentioned above, the polarization energy arises from the fact that the charge distribution of a group or molecule is distorted by interactions with its neighbours. In the point-charge model of electrostatic interactions (Equation (5.9)) the charges assigned to atoms are constants and so the charge distribution of the molecule is constant for a given nuclear configuration irrespective of its environment. To model changes in the charge distribution of a molecule a number of techniques have been developed, but one of the more common ones is to let each atom be polarizable by giving it an *isotropic dipole polarizability*. This means that, in the presence of an electric field produced by the charge distribution of the environment, a dipole moment is induced on the atom that is proportional in size and parallel to the field at the atom. If the polarizability for an atom i is denoted by α_i and the field at the atom by $\boldsymbol{E}_i$ (it is a vector quantity) then the dipole induced at the atom, $\boldsymbol{\mu}_i$, is

$$\boldsymbol{\mu}_i = \alpha_i \boldsymbol{E}_i \tag{5.16}$$

The polarizability model we use here is isotropic because α_i is a scalar quantity. For an *anisotropic* model it would be a 3×3 matrix, which means that the atom could be more polarizable in some directions (such as along a bond) than in others. The polarizability is called a dipole polarizability because the field induces a dipole at the atom. In more complicated versions of the theory the electric field (or its derivatives) can induce different effects in the charge distribution.

The field at each atom is produced by the sum of the fields due to the charges, $\boldsymbol{E}_i^q$, and the induced dipoles, $\boldsymbol{E}_i^\mu$, on the other atoms. These fields have the form

$$\boldsymbol{E}_i^q = \frac{1}{4\pi\epsilon_0\epsilon} \sum_{j\neq i=1}^{N} \frac{q_j \boldsymbol{r}_{ij}}{r_{ij}^3} \tag{5.17}$$

$$\boldsymbol{E}_i^\mu = \frac{1}{4\pi\epsilon_0\epsilon} \sum_{j\neq i=1}^{N} \mathbf{T}_{ij}\boldsymbol{\mu}_j \tag{5.18}$$

where $\mathbf{T}_{ij}$ is a 3×3 matrix that is given by

$$\begin{aligned} \mathbf{T}_{ij} &= -\nabla_i \frac{\boldsymbol{r}_{ij}}{r_{ij}^3} \\ &= \frac{1}{r_{ij}^5} \begin{pmatrix} 3x_{ij}^2 - r_{ij}^2 & 3x_{ij}y_{ij} & 3x_{ij}z_{ij} \\ 3y_{ij}x_{ij} & 3y_{ij}^2 - r_{ij}^2 & 3y_{ij}z_{ij} \\ 3z_{ij}x_{ij} & 3z_{ij}y_{ij} & 3z_{ij}^2 - r_{ij}^2 \end{pmatrix} \end{aligned} \tag{5.19}$$

It is now possible to substitute the equations for the fields, Equations (5.17) and (5.18), into the equation for the dipole, Equation (5.16). From the form for the field due to the induced dipoles, it is evident that the dipole on atom i depends

upon the induced dipoles of all the other atoms. To make the dependence more explicit it is possible to combine the N equations for the dipoles on each atom into a single equation leading, after a little manipulation, to

$$\mathbf{A}\boldsymbol{\mu} = \boldsymbol{B} \tag{5.20}$$

$\boldsymbol{\mu}$ and $\boldsymbol{B}$ are both vectors of $3N$ components with the forms

$$\boldsymbol{\mu} = \begin{pmatrix} \boldsymbol{\mu}_1 \\ \boldsymbol{\mu}_2 \\ \vdots \\ \boldsymbol{\mu}_N \end{pmatrix} \qquad \boldsymbol{B} = \begin{pmatrix} \alpha_1 \boldsymbol{E}_1^q \\ \alpha_2 \boldsymbol{E}_2^q \\ \vdots \\ \alpha_N \boldsymbol{E}_N^q \end{pmatrix} \tag{5.21}$$

$\mathbf{A}$ is a $3N \times 3N$ matrix that can be taken to consist of N^2 3×3 submatrices, $\mathbf{a}_{ij}$:

$$\mathbf{A} = \begin{pmatrix} \mathbf{a}_{11} & \mathbf{a}_{12} & \dots & \mathbf{a}_{1N} \\ \mathbf{a}_{21} & \mathbf{a}_{22} & \dots & \mathbf{a}_{2N} \\ \vdots & \vdots & \vdots & \vdots \\ \mathbf{a}_{N1} & \mathbf{a}_{N2} & \dots & \mathbf{a}_{NN} \end{pmatrix} \tag{5.22}$$

The diagonal matrices and the off-diagonal matrices have different forms:

$$\mathbf{a}_{ii} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \qquad \mathbf{a}_{ij} = -\frac{1}{4\pi\epsilon_0\epsilon}\alpha_i \mathbf{T}_{ij} \tag{5.23}$$

Equation (5.20) defines a set of $3N$ linear equations that can be solved to obtain the induced dipoles, $\boldsymbol{\mu}_i$, on each atom. Once these are known the polarization energy can be computed. It is

$$\mathcal{V}_{\text{polar}} = -\frac{1}{2}\sum_{i=1}^{N} \boldsymbol{\mu}_i^{\mathrm{T}} \boldsymbol{E}_i^q \tag{5.24}$$

This energy arises from the sum of three contributions. There is the energy due to the interaction of the induced dipoles in the system, $\mathcal{V}_{\mu\mu}$, the energy due to the interaction of the induced dipoles with the permanent charges, $\mathcal{V}_{\mu q}$, and an energy (which is positive) that arises because it costs a certain amount to produce the induced dipoles, $\mathcal{V}_{\text{induced}}$. Their expressions are

$$\mathcal{V}_{\mu\mu} = -\frac{1}{2}\sum_{i=1}^{N} \boldsymbol{\mu}_i^{\mathrm{T}} \boldsymbol{E}_i^{\mu} \tag{5.25}$$

$$\mathcal{V}_{\mu q} = -\sum_{i=1}^{N} \boldsymbol{\mu}_i^{\mathrm{T}} \boldsymbol{E}_i^q \tag{5.26}$$

$$\mathcal{V}_{\text{induced}} = \frac{1}{2}\sum_{i=1}^{N} \boldsymbol{\mu}_i^{\mathrm{T}} \boldsymbol{E}_i \tag{5.27}$$

The fact that there is a set of linear equations to solve (Equation (5.20)) marks the major difference between the calculation of the previous force field energy terms that have been mentioned and the calculation of the polarization energy. The electrostatic and Lennard-Jones energies are what are known as *pairwise additive*. Each interaction can be calculated separately and is independent of the others. For the polarization energy this is not the case, in that the magnitude of the induced dipole on each atom depends upon the induced dipoles of all the other atoms and so the dipoles for all the atoms must be calculated before the energy. The polarization energy is a type of *many-body* term.

The other distinguishing feature of the calculation of the polarization energy is its expense. The calculations of the electrostatic and Lennard-Jones energies as written above both involve $O(N^2)$ operations. In contrast, the calculation of $\mathcal{V}_{\text{polar}}$ is more expensive because the solution of the $3N$ linear equations, which is the most time-consuming part of the calculation, formally scales as $O(N^3)$.

For the bonding energy terms, the bond energies are pairwise additive while the remaining terms are, strictly speaking, many-body terms because they depend on either three or four atoms (for the angles and the proper and improper dihedral energies, respectively). The number of each of these four types of terms is, however, roughly proportional to the number of atoms, so the expense of calculating them is only about $O(N)$.

It is, therefore, the calculation of the non-bonding energies – the electrostatic, Lennard-Jones and polarization terms – that is the most expensive part of an MM energy calculation. In the next few chapters the simple, $O(N^2)$ method for the calculation of the electrostatic and Lennard-Jones energies will be employed, but, as alluded to in Sections 3.2 and 4.5.3, there are ways of reducing the cost of an $O(N^2)$ calculation to more manageable proportions. This is a very important topic whose discussion will be left until a later chapter.

There is one additional point that needs elaborating for the calculation of the electrostatic, Lennard-Jones and polarization energies. This concerns which interactions between particles are to be included in the sums for the electrostatic and Lennard-Jones energies (Equations (5.9) and (5.10), respectively) or for the calculation of the fields in the case of the polarization energy (Equations (5.17) and (5.18)). For particles that are far from each other there is little problem and the interactions can be calculated as described above. For particles that are bonded together or are separated by only a few bonds there are two problems. First, the non-bonding interactions between them are large because their interparticle

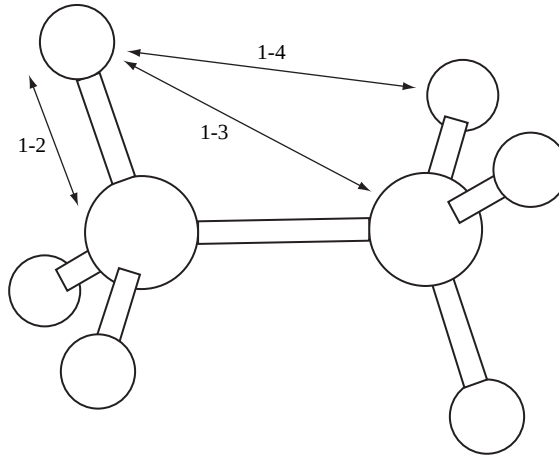


Fig. 5.6. Examples of 1–2, 1–3 and 1–4 non-bonding exclusions for a hydrogen atom in an ethane molecule.

separations are small and, second, there will also be bonding terms (bonds, angles, dihedrals, etc.) between such atoms.

This dilemma is resolved by introducing the concept of *non-bonding exclusions* (see Figure 5.6). Non-bonding interactions are calculated only for particles that are not involved in direct bonding interactions. For particles that are bonded or separated by only a few bonds, the non-bonding interactions between them are not calculated and it is the bonding terms that determine their energy of interaction. This avoids the problems of having very large interaction energies and of the overcounting that would result if both types of interaction were included. If both bonding and non-bonding terms between atoms close to each other were to be calculated then the analytic forms of the interactions described above would probably need to be significantly modified.

The number and type of non-bonding exclusions used depend on the force field. It is typical to exclude interactions between atoms that are directly bonded together (the so-called *1–2 interactions*) and those that are separated by two bonds (*1–3 interactions*). The treatment of interactions between atoms separated by three bonds (*1–4 interactions*) is the most variable. In some force fields they are excluded, in others they are included and in yet other cases they will be included but either the interactions will be scaled by some factor or special 1–4 sets of charges or Lennard-Jones parameters will be used. The reason for the different treatment of 1–4 interactions is that it is the combination of the dihedral angle bonding terms and the 1–4 electrostatic and Lennard-Jones terms that determines the barriers to rotation about bonds.

To close this section it is worth mentioning that, as was the case for the bonding energy terms, other types of non-bonding term are employed in some force fields.

These include alternatives to the explicit forms of the electrostatic and Lennard-Jones interactions mentioned above and also other terms. An important example included in some force fields is an energy introduced to mimic *hydrogen bonding*. Various forms for this energy have been used but this term is often omitted because the combination of electrostatic and dispersion/repulsion interactions is sufficient to reproduce the hydrogen-bonding interaction to the required precision.

5.3 Calculating a molecular mechanics energy

In the last section we introduced in a general way some of the concepts that underlie the design of many molecular mechanics force fields. In this section we define the force field that will be used in this book and show how molecular mechanics energies can be calculated for a given system with the pDynamo library.

5.3.1 The OPLS all-atom force field

Many force fields exist for performing simulations of molecular systems, a partial list of which is given in the references. Some are fairly intricate and designed for highly accurate calculations of smaller molecules (such as the MM2, MM3 and MM4 force fields developed by N. Allinger and co-workers) whereas others, such as those devised for the simulation of biomacromolecules, are simpler and resemble the 'typical' force field that was discussed in the last section. For a number of reasons, primarily because it is widely used and a large variety of parameters for it have been published in the literature, the programs in this book employ the all-atom version of the *optimized potentials for liquid simulations* force field (OPLS-AA) that has been developed by W. Jorgensen and his collaborators. This is a force field of the simpler type which has been applied to a wide range of systems, including biomacromolecules, such as proteins and nucleic acids, organic liquids and solutes in solution. It should be emphasized that, although the specific energies or properties for a system that are calculated with this force field may differ from those obtained using other force fields, the general principles of the calculations that we perform are completely independent of the specific choice of the force field that has been made.

The exact analytic form of the force field is repeated here for convenience. The total potential energy of the system, $\mathcal{V}$, is the sum of bond, angle, torsion, improper torsion and non-bonding terms. The expressions for the bond and angle

energies are the same as those in Equations (5.2) and (5.4) except that the factor of one-half has been omitted, i.e.

$$\mathcal{V}_{\text{bond}} = \sum_{\text{bonds}} k_b(b - b_0)^2 \tag{5.28}$$

$$\mathcal{V}_{\text{angle}} = \sum_{\text{angles}} k_\theta(\theta - \theta_0)^2 \tag{5.29}$$

The dihedral energy is similar to that in (5.5). For each dihedral, three terms in the Fourier expansion are included with periodicities of 1, 2 and 3. The phase of each of the terms is 0° for periodicities 1 and 3 and 180° for periodicity 2. This gives the following expression in the most general case in which none of the Fourier coefficients, V_n, is zero:

$$\mathcal{V}_{\text{dihedral}} = \sum_{\text{dihedrals}} \frac{V_1}{2}(1 + \cos\phi) + \frac{V_2}{2}(1 - \cos 2\phi) + \frac{V_3}{2}(1 + \cos 3\phi) \tag{5.30}$$

The improper dihedral energy, $\mathcal{V}_{\text{improper}}$, has the same form as the dihedral energy except that the dihedral angles, ϕ, are replaced by improper ones, ω.

The non-bonding energy, $\mathcal{V}_{\text{nb}}$, consists of the sum of electrostatic and Lennard-Jones terms with the form

$$\mathcal{V}_{\text{nb}} = \sum_{ij \text{ pairs}} \left\{ \frac{q_i q_j}{4\pi\epsilon_0 \epsilon r_{ij}} + 4\varepsilon_{ij} \left[\left(\frac{s_{ij}}{r_{ij}} \right)^{12} - \left(\frac{s_{ij}}{r_{ij}} \right)^{6} \right] \right\} f_{ij} \tag{5.31}$$

The variable f_{ij} is a weighting factor for the interactions. For 1–2 and 1–3 interactions it has the value 0 and so these interactions are excluded from the sum. For the 1–4 interactions the value is $\frac{1}{2}$ and for all other interactions its value is 1. The OPLS-AA force field uses geometrical combination rules for the Lennard-Jones parameters ε_{ij} and s_{ij}.

A final point about the origin of some of the parameters in the OPLS-AA force field should be made. In their work Jorgensen and co-workers concentrated on the optimization of the non-bonding and torsional parameters. The bond, bond angle and improper dihedral angle parameters were taken mostly from the AMBER force field developed by P. Kollman and his group and, to a lesser extent, from the CHARMM force field developed by M. Karplus and his collaborators.

5.3.2 Example force field representations

A quick glance at the equations describing the molecular mechanics energy function and its constituent terms shows that there are many quantities that need to be defined before the energy of a system can be calculated. These include the number of terms of each type, the atoms involved in each energy term and the values of

Table 5.1 *The TIP3P water model*

2 atom types Frequency	Name	q_i (e)	s_{ii} (Å)	ε_{ii} (kJ mol^{-1})
1	OW	−0.834	3.15	0.64
2	HW	0.417	0.0	0.0
3 bond terms Frequency	Types		k_b (kJ mol^{-1} Å^{-2})	b_0 (Å)
2	OW–HW		2215.85	0.9572
1	HW–HW		160.04	1.5139
1 angle term Frequency	Types		k_θ (kJ mol^{-1} rad^{-2})	θ_0 (°)
1	HW–OW–HW		142.47	104.52

their parameters. As examples of the way in which systems are represented with the OPLS-AA force field, we shall consider in detail two small molecules, water and bALA.

Several different models of water have been developed within the OPLS-AA framework but the one that we shall employ is called TIP3P, details of which are given in Table 5.1. The first part of the table lists the *atom types* in the molecule. Some explanation is needed here. In principle, if the energy function being used were completely accurate, the only property needed in order to identify an atom for the calculation of an energy would be its element type. In practice, because the empirical energy functions in use are not sufficiently flexible in this regard, it is necessary to define different types of atom that correspond to the same element. Thus, for example, a hydrogen bound to an oxygen atom will be defined to be of a type different from a hydrogen bound to an aliphatic carbon and the two types will have different parameters associated with them. Similarly, aliphatic carbons will have different types from aromatic carbons and carbonyl carbons. In the water case, atom typing is straightforward and there is one type for oxygen, OW, and one for hydrogen, HW. The non-bonding parameters for the types are also shown in the table using the same symbols as occur in Equation (5.31). It is to be noted that the charges are such that the molecule is neutral and that only the oxygen atom will have Lennard-Jones interactions because the s_{ii} and ε_{ii} parameters for HW are zero.

The next sections in the table list the bonding terms in the model along with their parameters. There are no dihedral terms for water and so only bond and angle terms occur. In total there are four bonding terms, three of which – the two OW–HW bonds and the HW–OW–HW angle – correspond to terms that arise from the covalent structure of the molecule. The remaining term is a bond that has been defined between the two hydrogens. Such 'unchemical' bonds (and other terms) are not uncommon in force field models of molecules and are included so as to obtain better calculated properties.

The second example is more complicated and concerns the molecule bALA. Table 5.2 provides a summary of some aspects of the OPLS-AA model. As before, the table starts by listing the atom types in the system, of which there are six. In addition to the single types for nitrogen and oxygen, OPLS uses two types for carbon (C for a carbonyl sp^2 carbon and CT for an sp^3 carbon) and two types for hydrogen (H for a hydrogen attached to a nitrogen and HC for a hydrogen attached to an sp^3 carbon). The identification of which atoms have which types should be straightforward by referring to Figure 2.3. The Lennard-Jones parameters for the types are not given in the table but these will, in general, be different for each type. The charges are, however, listed. For bALA, these are constant for each type, with the exception of atoms of CT type. For these atoms, the charges are such that the overall charges of the chemical groups in which the atoms occur are neutral. Thus, the side-chain and C-terminal methyl carbon atoms have charges of -0.18, the C_α atom has a charge of 0.14 and the N-terminal methyl carbon has a charge of 0.02.

The bonding terms are listed after the atom types and, in this case, there are terms for each type that appears in Equation (5.1). The choice of which bonding terms to include is based upon the covalent structure of the molecule and, so unlike the TIP3P model, there are no 'unchemical' entries in the table. In the bALA molecule there are 21 covalent bonds and so there are 21 bonding terms, one per covalent bond. Likewise, from these 21 bonds it is possible to generate 36 bond angles and so there are 36 angle terms, one per bond angle. There are 7 different types of bonding term and 13 types of angle term which are defined with respect to the atom types that take part in the interaction. In contrast to the bond and angle terms, dihedral terms are present for only 21 of the 41 possible dihedral angles that occur in the molecule. This is because the force constants, V_1, V_2 and V_3 in Equation (5.30), for the dihedral terms applicable to the 20 remaining dihedrals are all zero and so they are excluded from the model. There are 4 improper terms, one of which is present for each of the four sp^2-hybridized atoms in the molecule. In all, there are 11 different types of dihedral term and 2 types of improper term.

Table 5.2 *The OPLS-AA representation of the bALA molecule*

6 atom types			
Frequency	Name	Description	Charge (e)
2	H	Amide hydrogen	0.30
10	HC	Aliphatic hydrogen	0.06
2	C	Amide carbon	0.50
4	CT	Aliphatic carbon	—
2	N	Amide nitrogen	−0.50
2	O	Amide oxygen	−0.50

21 bond terms	
Frequency	Types
2	C – CT
2	C – O
2	C – N
10	CT – HC
1	CT – CT
2	CT – N
2	H – N

36 angle terms	
Frequency	Types
2	CT – C – O
2	CT – C – N
2	N – C – O
9	HC – CT – HC
4	HC – CT – C
4	HC – CT – N
4	HC – CT – CT
1	C – CT – CT
1	C – CT – N
1	CT – CT – N
2	C – N – CT
2	C – N – H
2	CT – N – H

21 dihedral terms	
Frequency	Types
1	N – C – CT – CT
1	N – C – CT – N
2	CT – C – N – CT
2	CT – C – N – H
2	O – C – N – CT
2	O – C – N – H
3	C – CT – CT – HC
3	HC – CT – CT – HC
3	HC – CT – CT – N

Table 5.2 (cont.)

21 dihedral terms (cont.) Frequency	Types
1	C – CT – N – C
1	CT – CT – N – C

4 improper terms Frequency	Type of Central Atom
2	C
2	N

5.3.3 Generating the MM representation

It will be apparent from the examples in the last section that the MM representations of even relatively simple molecules can be quite complicated. Indeed, setting up the force field terms for a system can often be one of the most frustrating and time-consuming parts of a simulation study! Because of this, efforts have been made in pDynamo to make this process as automatic and painless as possible. The procedure that is adopted is as follows.

(i) Define the system for which the MM representation is being created. The definition should comprise the elemental type of the system's atoms and a full connectivity that includes the type (i.e. whether it is single, double, etc.) as well as the number of its covalent bonds. The easiest way of doing this is to generate the system from a file or representation that has this information. There are three representations that we met in Section 2.4 that satisfy this criterion. They are the CML, MOL and SMILES formats, although the latter has the disadvantage that it does not contain atomic coordinates and so these have to be input in some other fashion.

It is also possible to employ files in PDB format. This format has a section in which bonds between atoms can be listed (although it is often missing in practice) but nowhere can the bond types be given. However, the PDB organization keeps a library of all standard and non-standard residues that occur in files deposited in the PDB and the definitions in this library contain this information. pDynamo has its own version of the residue library which can be accessed when PDB files are read and from which the full connectivity for a system can be generated. This is done with a Boolean keyword argument `QUSERESIDUELIBRARY` to the function `PDBFile_ToSystem`. If the argument is `True`, data for the residues occurring in the PDB file are extracted from the library, assuming that the appropriate entries exist. If they do not an error is raised. If the argument has the value `False` or is absent (the default) no cross-referencing against the PDB residue library is done.

(ii) Classify the atoms in the system according to their MM type. This is done via a hierarchical sequence of rules which identify the types of atoms or groups of atoms according to their chemical environments. The rules are applied in turn to each of the atoms in the system until matches for all atoms have been found.

The rules themselves are of varying complexity, with those that are more specific occurring before those that are more general in the hierarchy. The simplest rules may only identify the types of single atoms. Using the notation of the last section, an example is: if the atom is an oxygen and possesses two single bonds to hydrogens, the atom's type is OW. The obvious corollary is a subsequent rule that says: if the atom is a hydrogen with a single bond to an atom of type OW, the atom's type is HW. More complicated rules identify the types of several atoms at once. Thus, a rule to identify peptide bonds could be: if there is a tetravalent carbon with single bonds to another carbon and to nitrogen and a double bond to oxygen, and the attached nitrogen is trivalent with additional single bonds to hydrogen and to a third carbon, the types of the first carbon, its attached oxygen, its attached nitrogen and the hydrogen attached to the nitrogen are C, O, N and H, respectively.

(iii) Assign bonding and non-bonding parameters according to the types of the atoms. The choice of which bonding terms to include in the representation is, for the most part, based upon the connectivity of the system although, in certain cases, such as for the TIP3P model of water, extra bonding terms will be added.

Clearly, the feasibility of the above procedure depends upon the existence of a set of rules for atom typing and of lists of accompanying bonding and non-bonding parameters. In pDynamo, all the definitions for particular classes of systems are collected together in single files which can be accessed as appropriate. These files, and all others that contain parameter data or definitions, are stored in a subdirectory, called `parameters`, that is part of the `pDynamo` package. It is possible for users to generate their own MM definitions if the pre-existing definitions are not sufficient but the details of how to do so will not be given here as the process is rather lengthy.

MM force fields are represented in pDynamo with instances of subclasses of the class `MMModel`. The appropriate subclass for the OPLS-AA force field is `MMModelOPLS` which has the following definition:

Class `MMModelOPLS`

A class to represent an OPLS-AA MM model.

Constructor

Construct an instance of `MMModelOPLS`.

Usage: `new = MMModelOPLS ( datafile )`

`datafile` is a string with the name of the file that contains the appropriate OPLS-AA atom type and parameter definitions. The name refers to one of the files in the `opls` section of pDynamo's `parameters` directory. In this book, the following parameter sets are employed: `"booksmallexamples"` which is sufficient for all small-molecule examples; `"lennardjones"` which is used for the example in Section 9.7; and `"protein"` which is valid for simple proteins.

`new` is the new instance of `MMModelOPLS`.

The calculation of the non-bonding part of the MM energy requires instances of subclasses of the class `NBModel`. A distinct non-bonding (NB) class is necessary because the non-bonding energy can be evaluated in a number of different ways, even for a single MM model. We shall meet several of these alternatives later, but, for the moment, we employ an algorithm in which the full interaction is determined using straightforward $O(N^2)$ summation. The appropriate class is `NBModelFull`:

Class `NBModelFull`

A class to calculate non-bonding interactions with a simple $O(N^2)$ summation algorithm.

Constructor

Construct an instance of the `NBModelFull` class.

Usage: `new = NBModelFull ( )`

`new` is the new instance of `NBModelFull`.

MM and NB models are assigned to a system in a similar way as QC models (see Section 4.7). The appropriate methods from the `System` class are

Class `System`

MM and NB model-related methods.

Method `DefineMMModel`

Assign an MM model to a system.

Usage: `system.DefineMMModel ( mmmodel )`

`mmmodel` is the MM model that is to be used for calculating the MM energy of the system.

`system` is the instance of `System` for which the MM model is being defined.

Remarks: This method constructs the MM representation for the system using the definitions it contains. It both types the atoms in the system and assigns parameters to the bonding and non-bonding terms. For the method to work, `system` must have a fully defined connectivity, including bonds, angles and dihedrals. An error will occur if this is not the case or if `system` contains atom types or parameters that are not covered by the definitions in `mmmodel`.

Method `DefineNBModel`

Assign an NB model to a system.

Usage: `system.DefineNBModel ( nbmodel )`

`nbmodel` is the NB model that is to be used for calculating the non-bonding energy for the system.

`system` is the instance of `System` for which the NB model is being defined.

Remarks: This method should only be called for a system after its MM model has been set up via a call to `DefineMMModel`. Only the bonding portion of the MM energy will be calculated if a system has no NB model.

5.4 Example 7

The example in this section makes use of some of the MM and NB model features described in the preceding section. Much of the syntax is similar to Examples 4 and 5 (Sections 3.7 and 4.7, respectively) because the program treats the same set of conformations of the bALA molecule as Example 4 and calculates similar quantities as Example 5 for each conformation.

The program is:

```
1 """Example 7."""
2
3 from Definitions import *
4
5 # . Define the list of structures.
6 xyzfiles = [ "bala_alpha.xyz", "bala_c5.xyz", \
              "bala_c7ax.xyz", "bala_c7eq.xyz" ]
```

```
7
8  # . Define the MM and NB models.
9  mmmodel = MMModelOPLS ( "booksmallexamples" )
10 nbmodel = NBModelFull ( )
11
12 # . Generate the molecule.
13 molecule = MOLFile_ToSystem ( \
                    os.path.join ( molpath, "bala_c7eq.mol" ) )
14 molecule.DefineMMModel ( mmmodel )
15 molecule.DefineNBModel ( nbmodel )
16 molecule.Summary ( )
17
18 # . Loop over the structures in the xyz files.
19 results = []
20 for xyzfile in xyzfiles:
21     molecule.coordinates3 = XYZFile_ToCoordinates3 ( \
                             os.path.join ( xyzpath, xyzfile ) )
22     energy = molecule.Energy ( )
23     dipole = molecule.DipoleMoment  ( )
24     results.append ( ( xyzfile[5:-4], energy, dipole.Norm2 ( ) ) )
25
26 # . Output the results.
27 table = logfile.GetTable ( columns = [ 20, 20, 20 ] )
28 table.Start  ( )
29 table.Title  ( "Energy Model Results for bALA" )
30 table.Heading ( "Conformation" )
31 table.Heading ( "Energy" )
32 table.Heading ( "Dipole" )
33 for ( label, energy, dipole ) in results:
34     table.Entry ( label )
35     table.Entry ( "%.1f" % ( energy, ) )
36     table.Entry ( "%.3f" % ( dipole, ) )
37 table.Stop ( )
```

The major differences with the previous examples concern the definition of the MM and NB models. These are:

Lines 9–10 create instances of the classes `MMModelOPLS` and `NBModelFull`. The MM model is defined so that it employs data from the parameter set `"booksmallexamples"` when constructing the MM potential energy function for the system.

Line 14 defines the MM model for the bALA system to be that in `mmmodel`. This method sets up the OPLS potential energy function for the molecule using the parameters from the set `"booksmallexamples"`.

Line 15 assigns an NB model to the system. Note that it is called after the definition of the system's MM model.

5.5 Parametrizing potential energy functions

By the end of this book we will have spent much time presenting a wide range of algorithms for performing simulations of molecular systems and how to apply them correctly. No matter how good our simulation techniques are though, the usefulness of the results produced will be heavily dependent upon how accurately the potential energy function reproduces the potential energy surface of the system being studied. In other words, the quality of parametrization of the energy function is crucial to the results that we obtain.

Parametrization is an essential part of the development of all the methods that we discuss in this book for the calculation of the potential energy. This applies to the Gaussian basis sets used in *ab initio* QC calculations, the parameters employed for the evaluation of integrals and core–core terms in MNDO-type semi-empirical QC methods, and the parameters of the various terms in MM force fields. The way in which all these energy functions are parametrized is similar and, in broad outline, as follows.

(i) Gather a set of *reference data* that correspond to the physical and chemical properties of systems of the type for which the energy function is to be parametrized. The data values can come from either experiment or accurate *ab initio* QC calculations and must pertain to properties that can be computed with the energy function and an appropriate simulation methodology. It is essential that the number and variety of reference data be sufficient to provide an adequate test of the parametrized potential energy function. Examples of the types of data that can be used are: structures determined experimentally using techniques such as X-ray crystallography, microwave spectroscopy and electron diffraction; vibrational frequencies from infrared and Raman spectroscopy; ionization potentials; electrostatic properties such as dipole moments; physical properties such as densities; and thermodynamic quantities such as enthalpies of formation, enthalpies of vaporization and heat capacities. We shall meet how to calculate many of these properties in later chapters.

(ii) Separate the data into a *training set* and a *test* or *validation set*.

(iii) Guess a set of starting values for the parameters that are to be found.

(iv) Refine the values of the parameters using the following procedure.

  (a) Calculate the properties of the systems in the training set using the energy function and the current parameter values.

  (b) Determine the agreement between the calculated and reference data and stop the parameter refinement if this is judged to be adequate.

  (c) Modify the values of the parameters and go back to step (iv) (a). This step will involve checking the new values of the parameters to ensure that they remain chemically and physically reasonable. Thus, for example, the radius of an atom

is of the order of 1 Å and so values for Lennard-Jones radius parameters should be of roughly the same magnitude. Likewise, the maximum stabilization to be expected from a van der Waals interaction is a few kJ mol^{-1}, which will limit, in turn, the size of the Lennard-Jones well-depth parameters.

There are various ways in which parameter refinement can be accomplished but a popular one is to choose a *non-linear least squares* algorithm, in which a function, $\mathcal{F}$, of the following form is minimized with respect to the parameter values:

$$\mathcal{F}(\boldsymbol{p}) = \sum_{I=1}^{N_o} w_I (O_I^{calc}(\boldsymbol{p}) - O_I^{ref})^2 \tag{5.32}$$

In this equation, $\boldsymbol{p}$ is the vector of parameter values, N_o is the number of *observables* (i.e. the number of reference data), w_I are observable weights and O_I^{calc} and O_I^{ref} are the calculated and reference values of the observables, respectively.

(v) Calculate the properties of the systems in the validation set with the fitted parameter values and compare the results with the reference values. This step is a crucial one and serves to determine how *transferable* the parameters are to properties and systems against which they were not fitted.

The parametrization of all but the simplest energy functions is a complex process that can be a long and time-consuming business. Normally it is impractical to try to optimize all the parameters against all the available experimental data simultaneously so the problem must be broken down into smaller pieces. How this is done varies greatly but a number of strategies are possible. One is to focus upon a subset of 'core' systems and parametrize the energy function for these first. Once this has been achieved, the parametrization can be extended to other species assuming that the parameters for the core species remain fixed. When developing semi-empirical QC methods, for example, it is common to start with molecules that contain the elements C, H, N and O before continuing on to compounds of other elements. Similarly, the parametrization of MM force fields often proceeds in a stepwise fashion. Hydrocarbons are considered first, followed by the parametrization of groups of increasing complexity such as alcohols, amines, carbonyls and so on.

A second strategy that can be adopted is to try to avoid the parametrization of certain parameters altogether. One approach of this type is the calculation of the partial charges on the atoms in an MM force field using a QC method. This is done in a number of force fields, including AMBER. Due to the vagaries of the Mulliken population analysis, other schemes are preferred, most notably one in which the charges are obtained by fitting them so that they reproduce the *electrostatic potential* (ESP) calculated at various points around the molecule. The advantage of calculating the charges in this way is that it decouples their determination from that of the remainder of the force field parameters, but its

disadvantage is that the charge parameters are no longer transferable and must be recalculated using ESP data for each different type of molecule. By contrast, in other force fields, such as OPLS-AA, charges are parametrized but some degree of transferability is assumed and so they are derived, insofar as it is possible, such that the same charges can be used for the same chemical group in different environments.

A last point needs to be emphasized before ending, which is that the final result of any parametrization procedure will inevitably be a compromise. This is both because of the difficulties of the parametrization process itself and because the approximations inherent in the formulation of the energy function will limit the precision that can be achieved and the type of observational data that can be reproduced.

5.6 Soft constraints

To conclude this chapter we make a brief aside and introduce a subject, together with related pDynamo classes and methods, that will not be needed immediately but will prove essential later. In previous chapters, we have already met instances in which it was necessary to impose constraints upon the values of some of the variables that were intrinsic to the problem under discussion. One example was the normalization condition for the quaternion parameters in Section 3.6 and another the orthonormality constraints placed upon the molecular orbital coefficients in Section 4.4. Both of these are examples of *hard-* or *rigid-constraint* methods in which the constraint equations are satisfied 'exactly' (to within a specified numerical precision). A second class of constraint methods also exists that employ *soft constraints*. These methods are cruder and only attempt to approximately satisfy the constraint conditions.

In molecular simulations it is often valuable to be able to constrain the values of particular geometrical variables. A few types of geometrical constraint are straightforward to implement with hard-constraint methods but, in the general case, the imposition of hard constraints can be difficult, requiring special methods or significant reformulations of the simulation algorithms that are being employed. We shall come across some of these techniques in Section 11.8. By contrast, soft constraints, although less precise, are very simple to implement and to apply.

Soft constraints are conveniently implemented by adding extra MM-like terms to the potential energy function. These give low energies when the geometrical variables are close to the desired values and progressively higher energies as the values deviate further. The form of the soft-constraint energy is arbitrary but the

ones that we employ in this book are all of *piecewise harmonic form* as follows:

$$\mathcal{V}_{\mathrm{sc}} = \begin{cases} k_{\mathrm{sc}}\,(v - v_{\mathrm{low}})^2 & v < v_{\mathrm{low}} \\ 0 & v_{\mathrm{low}} \leq v \leq v_{\mathrm{high}} \\ k_{\mathrm{sc}}\,(v - v_{\mathrm{high}})^2 & v > v_{\mathrm{high}} \end{cases} \tag{5.33}$$

In this equation, v is the value of the geometrical variable being constrained and $\mathcal{V}_{\mathrm{sc}}$ and k_{sc} are the constraint's potential energy and force constant, respectively. The parameters v_{low} and v_{high} determine the range over which the constraint energy is zero. If $v_{\mathrm{low}} = v_{\mathrm{high}} (= v_0)$, Equation (5.33) becomes a regular harmonic function of the type illustrated in Figure 5.1.

Due to their diversity, use of soft constraints in pDynamo requires a number of different classes. The first group specify the form of the soft-constraint energy function and have the definitions:

Class `SoftConstraintEnergyModelHarmonicRange`

A class to represent an energy function of the form given by Equation (5.33).

Constructor

Construct an instance of `SoftConstraintEnergyModelHarmonicRange`.

Usage: `new = SoftConstraintEnergyModelHarmonicRange ( vlow, vhigh, fc )`

`vlow` is the value of v_{low}.
`vhigh` is the value of v_{high}.
`fc` is the force constant, k_{sc}.
`new` is the new instance of `SoftConstraintEnergyModelHarmonicRange`.

Class `SoftConstraintEnergyModelHarmonic`

A class to represent a harmonic energy function. This is a special case of Equation (5.33) in which $v_{\mathrm{low}} = v_{\mathrm{high}}$.

Constructor

Construct an instance of `SoftConstraintEnergyModelHarmonic`.

Usage: `new = SoftConstraintEnergyModelHarmonic ( v0, fc )`
`v0` is the equilibrium value of the variable v.

`fc` is the force constant, k_{sc}.
`new` is the new instance of `SoftConstraintEnergyModelHarmonic`.

The second group of classes define the geometrical variable that is to be constrained along with the constraint's energy model. These classes are all subclasses of the class `SoftConstraint` with the simplest being one that constrains the distance between two atoms. Its definition is:

Class `SoftConstraintDistance`

A class for constraining the distance between two atoms.

Constructor

Construct an instance of `SoftConstraintDistance`.

Usage:
```
new = SoftConstraintDistance ( atom1, atom2,
                               energymodel )
```
`atom1` is the index of the first atom in the constraint.
`atom2` is the index of the second atom in the constraint.
`energymodel` is the energy model to use for the constraint. It should be an instance of one of the `SoftConstraintEnergyModel` classes.
`new` is the new instance of `SoftConstraintDistance`.

Similar classes exist for constraining the angle between three atoms or the dihedral angle between four atoms. These have the same specification as `SoftConstraintDistance` except that their names are changed appropriately – `Angle` or `Dihedral` instead of `Distance` – and that their constructors require three (for an angle) or four (for a dihedral) atom arguments.

The classes above apply constraints that are functions of atom coordinates only. It is also possible to have constraints that involve other geometrical objects, such as points, lines or planes. The only one of this type that we shall require is a *tether* constraint that limits the position of a particle to a particular region of space. It is implemented as follows:

Class `SoftConstraintTether`

A class for constraining the position of an atom about an absolute position in space.

Constructor

Construct an instance of `SoftConstraintTether`.

Usage: `new = SoftConstraintTether (atom, point, energymodel)`

`atom` is the index of the atom to be tethered.

`point` is an instance of the class `Vector3` that holds the Cartesian coordinates of the point about which the atom is to be tethered.

`energymodel` is the energy model to use for the constraint. It should be an instance of one of the `SoftConstraintEnergyModel` classes.

`new` is the new instance of `SoftConstraintTether`.

Remarks: The variable that is being constrained is $v = |\boldsymbol{r}_i - \boldsymbol{r}_0|$, where $\boldsymbol{r}_i$ is the position of atom i and $\boldsymbol{r}_0$ is the position of the reference point.

To be used and assigned to a system, soft constraints must be gathered together in an instance of the class `SoftConstraintContainer`. This class behaves very like a *dictionary* which is another of Python's built-in sequence types. Dictionaries are unlike lists and tuples in that access to the items they contain is done by an item's unique *key* and not by the integer index that gives its order in the sequence. In this case, the container keys are strings that give the names of the constraints and the items are instances of the class `SoftConstraint`. We shall see how to add and remove constraints to and from a container with a dictionary-like syntax in Sections 9.7 and 11.7, but for the moment a sufficient definition of the class is:

Class `SoftConstraintContainer`

A container class that holds soft constraints. This class behaves like a Python dictionary whose keys are strings and whose items are instances of `SoftConstraint`.

Constructor

Construct an instance of `SoftConstraintContainer`.

Usage: `new = SoftConstraintContainer ( )`

`new` is the new instance of `SoftConstraintContainer`.

Remarks: The container is empty when it is created.

Finally, soft constraints are assigned to a system with a method similar to those for assigning QC, MM and NB energy models. The extension to the `System` class is:

Class `System`

Soft constraint-related methods.

Method `DefineSoftConstraints`

Assign soft constraints to a system.

Usage:	`system.DefineSoftConstraints (constraints)`
`constraints`	is an instance of `SoftConstraintContainer` that contains the constraints to be added to the system. This argument can also take the value `None`, in which case all existing soft constraints on the system are removed.
`system`	is the instance of `System` for which the soft constraints are being defined.

Exercises

5.1 Derive an expression for the derivatives of the polarization energy, Equation (5.24), with respect to the atomic coordinates. Relate the result to the discussion of Section 4.8.1.

5.2 Repeat the calculation of the energy using the example in Section 5.4 but for other systems of interest. The pDynamo library includes MOL and PDB coordinate files for miscellaneous molecules so these can be used if necessary. In addition, the library provides parameter sets that are appropriate for certain classes of systems. Are these sets sufficient? If not, which parameters are missing and for which groups?

5.3 The parameter set `"booksmallexamples"` contains definitions for water and also sodium and chloride ions. Using this set, determine the energies of some water–water and water–ion complexes. In each case the geometry of the water molecules can be kept fixed, but the relative orientation of the molecules or molecule and ion can be altered. Devise a search procedure to investigate automatically and systematically a range of configurations for the complexes. What is the shape of the potential energy surface for the system? What are the most stable configurations for the interaction of a water molecule with the sodium cation, the chloride anion and another water molecule? The energy of interaction between the molecules is, of course, due to non-bonding terms only. What are the relative contributions of the electrostatic and Lennard-Jones terms for each of the three complexes?

6

Hybrid potentials

6.1 Introduction

The last two chapters considered two distinct classes of methods for calculating the potential energy of a system. Chapter 4 discussed QC techniques. These are, in principle, the most 'exact' methods but they are expensive and so are limited to studying systems with relatively small numbers of atoms. MM approaches were introduced in Chapter 5. These represent the interactions between particles in a simpler way than do QC methods and so are more rapid and, hence, applicable to much larger systems. They have the disadvantage, though, of being unsuitable for treating some processes, notably chemical reactions. *Hybrid potentials*, which are described in this chapter, seek to overcome some of the limitations of QC and MM methods by putting them together.

6.2 Combining QC and MM potentials

Although a hybrid potential is, in principle, any method that employs different potentials to treat a system, either spatially or temporally, the potentials we focus upon in this chapter use a combination of QC and MM techniques. These methods are also known as *QC/MM* or *QM/MM potentials*. The first potential of this type was developed in the 1970s by A. Warshel and M. Levitt who were studying the mechanism of the chemical reaction catalyzed by the enyzme lysozyme. Enzymes are proteins that can greatly accelerate the rate of certain chemical reactions. How they achieve this is still a matter of active research but the reaction itself occurs when the substrate species are bound close together in a specific part of the enzyme called the *active site*. The full system of enzyme, solvent and substrates was clearly too large to study by QC methods (especially in the 1970s!) so what Warshel and Levitt did was to treat a small number of atoms, comprising the substrates and active site region of the protein, with a semi-empirical QC potential and the, much larger, remainder of the system with a force field method. After

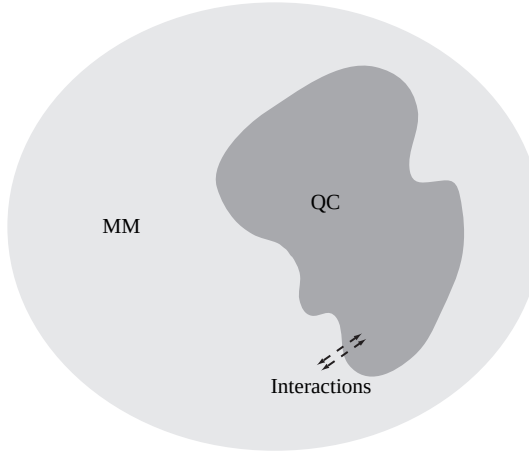


Fig. 6.1. Partitioning a system between QC and MM regions.

this initial work there was a lull in the application of QC/MM methods until the beginning of the 1990s, since when their use has mushroomed. This renewed interest was spurred, in large part, by the increase in computer power, and also by a number of technical developments in the potentials themselves. Notable amongst these were contributions by U. C. Singh and P. A. Kollman and by M. J. Field, P. A. Bash and M. Karplus.

There are many variants of QC/MM potentials, some of which we shall mention later in the section, but the schemes that we outline here divide a system into two regions, one QC and one MM. QC/MM methods are especially useful for studying chemical reactions in large systems, such as in enzymes or in solvents or on surfaces, and so, in these cases, the QC region will contain the reactive atoms. In studies where reactions are not being investigated, it may be that it is desired to have a QC description of a portion of the system, either because certain atoms should be treated at a higher level of precision or because they are not adequately represented by the MM potentials that are being used. A schematic of partitioning is shown in Figure 6.1.

How a hybrid potential is formulated depends upon the QC and MM methods that it employs. For the MO QC methods and pairwise-additive force fields that we use in this book, a formulation in terms of an *effective Hamiltonian*, $\hat{\mathcal{H}}_{\text{eff}}$, is convenient. When there are two regions, $\hat{\mathcal{H}}_{\text{eff}}$ consists of a sum of three terms, one for the QC region, one for the MM region and one for the interactions between the two:

$$\hat{\mathcal{H}}_{\text{eff}} = \hat{\mathcal{H}}_{\text{QC}} + \hat{\mathcal{H}}_{\text{MM}} + \hat{\mathcal{H}}_{\text{QC/MM}} \tag{6.1}$$

In this equation, the Hamiltonian for the QC region, $\hat{\mathcal{H}}_{\mathrm{QC}}$, takes the form that is appropriate for the QC method. Likewise, the Hamiltonian for the MM region, $\hat{\mathcal{H}}_{\mathrm{MM}}$, will be identical to that of the MM method but, as the ones that we employ do not contain any operators, this term reduces to the normal potential energy for the MM atoms, $\mathcal{V}_{\mathrm{MM}}$.

The crucial part of a QC/MM method, therefore, lies in how the interaction Hamiltonian, $\hat{\mathcal{H}}_{\mathrm{QC/MM}}$, is expressed. If we restrict ourselves, for the moment, to cases in which there are only non-bonding interactions between the atoms of the QC and MM regions, a representation, compatible with the force fields we are using, consists of a sum of electrostatic and Lennard-Jones terms. The equation is

$$\hat{\mathcal{H}}_{\mathrm{QC/MM}} = -\sum_{sm} \frac{q_m}{r_{sm}} + \sum_{im} \frac{Z_i q_m}{r_{im}} + \sum_{im} \left\{ \frac{A_{im}}{r_{im}^{12}} - \frac{B_{im}}{r_{im}^{6}} \right\} \tag{6.2}$$

where the subscripts s, i and m refer to electrons, QC nuclei and MM atoms, respectively. The first and second terms on the right-hand side of this equation are the electrostatic interactions between the partial charges, q_m, of the MM atoms and the electrons and nuclei of the QC region, respectively, whereas the last term is the Lennard-Jones interaction between the MM and QC atoms. Only the first term is an operator as it contains the coordinates of the electrons. The remaining terms are constants, for a given set of atomic coordinates, like the MM energy.

Once the effective Hamiltonian for the system has been defined, the appropriate time-independent Schrödinger equation can be solved:

$$\hat{\mathcal{H}}_{\mathrm{eff}} \Psi = \mathcal{V} \Psi \tag{6.3}$$

Here Ψ is the wavefunction for the electrons on the QC atoms and $\mathcal{V}$ is the potential energy of the full QC/MM system.

For a HF calculation, solution of Equation (6.3) gives rise to the Roothaan–Hall equation (Equation (4.27)) as normal, with the exception that there are extra terms in the one-electron matrix that arise due to the electrostatic interaction between the electrons and MM atoms. These terms are added directly to the one-electron matrix elements of Equation (4.16) and take the form

$$H_{\mu\nu}^{\mathrm{QC/MM}} = \int_{\boldsymbol{r}} \eta_\mu(\boldsymbol{r}) \left\{ \sum_m \frac{-q_m}{|\boldsymbol{r} - \boldsymbol{r}_m|} \right\} \eta_\nu(\boldsymbol{r}) \, \mathrm{d}\boldsymbol{r} \tag{6.4}$$

The fact that the one-electron matrix has been altered means that the wavefunction and, hence, the electron density will be adapted to the electrostatic environment in which the QC atoms are embedded.

After solution, the potential energy of the full system, like the effective Hamiltonian, can be expressed as the sum of three terms

$$\mathcal{V} = \mathcal{V}_{\mathrm{QC}} + \mathcal{V}_{\mathrm{MM}} + \mathcal{V}_{\mathrm{QC/MM}} \tag{6.5}$$

The first term is the QC energy, and is the same as that given in Equation (4.13), the second term is the MM energy and the third is the QC/MM interaction energy. The latter has the same form as does Equation (6.2) but with the first term on the right-hand side replaced by the expression $\sum_{\mu\nu} P_{\mu\nu} H_{\mu\nu}^{\mathrm{QC/MM}}$. The derivatives of $\mathcal{V}$ with respect to the atomic coordinates are straightforward to determine because all terms are fully differentiable except for those terms involving density matrix derivatives and these can be evaluated using the procedure described in Section 4.8.1.

The scheme outlined above is the one that we shall use in this book but it is far from unique. Many alternative methods follow the same general framework but differ in how they treat the QC/MM coupling or how many regions they partition the system into. Thus, for example, some schemes have extra terms, such as MM polarization, in Equation (6.2), whereas others divide the system into three or more regions consisting of various mixes of *ab initio* and semi-empirical QC methods and MM potentials.

There are also schemes that are quite different in their formulation. Extreme examples are potentials that treat the σ-electron, single-bond structure of a system with an empirical energy function and the delocalized π-electron structure with a QC technique. Other approaches, which have gained some popularity, are the 'layered' ONIOM methods first developed by F. Maseras and K. Morokuma. As an example, suppose, as before, that we have a system which we wish to treat with QC and MM potentials. ONIOM works by considering two systems, the first is the full (or *real*) system and the second is a *model* system that comprises the atoms in the QC region but excludes all others. The energy of the real system is then approximated as

$$\mathcal{V} = \mathcal{V}_{\mathrm{QC}}^{\mathrm{model}} - \mathcal{V}_{\mathrm{MM}}^{\mathrm{model}} + \mathcal{V}_{\mathrm{MM}}^{\mathrm{real}} \tag{6.6}$$

where $\mathcal{V}_{\mathrm{QC}}^{\mathrm{model}}$ and $\mathcal{V}_{\mathrm{MM}}^{\mathrm{model}}$ are the QC and MM energies of the model system, respectively, and $\mathcal{V}_{\mathrm{MM}}^{\mathrm{real}}$ is the MM energy of the real system.

There are clearly major differences between this approach and the one that we shall use. One is that the energy of the atoms in the QC region is calculated three times, twice as part of the model system and once as part of the real system, and that, to avoid overcounting, the MM energy of the model system must be subtracted in the final expression. Another difference is that, at least for the simpler ONIOM methods, the interactions between the atoms in the QC region and those exterior to it are only calculated with the MM potential because the model system excludes the surrounding atoms.

Hybrid potentials, no matter what scheme is used, resolve some of the problems associated with pure QC and pure MM potentials. They are not without shortcomings of their own, however, many of which relate to how the atoms in a system are divided between the different regions. One limitation is that the identities of the atoms in each region are fixed during a calculation. This is apparent, for example, when studying chemical reactions in which potentially reactive species diffuse or otherwise move out of the QC region and are replaced by unreactive MM species. Some schemes have been developed that can change the partitioning during the course of a simulation but none is, as yet, very general. Another limitation arises from the relatively small number of atoms that it is possible to put in the QC region. This requires that the process being studied with the QC potential be localized. Some, such as many chemical reactions, are, but others, such as electron transfer which can occur over large distances, are not.

6.3 Example 8

Hybrid potential models in pDynamo are specified using a combination of the MM and QC model classes that we have already met. The only additional element that is needed is an extra keyword argument, called `qcselection`, to the method `DefineQCModel` of the class `System`. This argument should be an instance of the class `Selection` and indicates the indices of the atoms that are to be treated quantum chemically. As we saw in Example 5 of Section 4.7, absence of the argument implies that all atoms will be in the QC region.

The following example shows how to combine the various models.

```
"""Example 8."""

from Definitions import *

# . Define the MM, NB and QC models.
mmmodel = MMModelOPLS ( "booksmallexamples" )
nbmodel = NBModelFull ( )
qcmodel = QCModelMNDO ( )

# . Define the molecule.
molecule = MOLFile_ToSystem ( \
               os.path.join ( molpath, "waterdimer_cs.mol" ) )

# . Define the selection for the first molecule.
firstwater = Selection ( [ 0, 1, 2 ] )

```

```
16 # . Define the energy model.
17 molecule.DefineMMModel ( mmmodel )
18 molecule.DefineQCModel ( qcmodel, qcselection = firstwater )
19 molecule.DefineNBModel ( nbmodel )
20 molecule.Summary ( )
21
22 # . Calculate an energy.
23 molecule.Energy ( )
```

Lines 6–8 create instances of the MM, NB and QC models that will be used for the hybrid potential calculation.

Line 11 defines an instance `molecule` of `System` that represents a water dimer. The structure on the file is illustrated in Figure 6.2. It has C_s symmetry and corresponds (approximately) to the dimer's most stable configuration.

Line 14 creates a selection that contains the indices of the atoms of the first water in the dimer.

Lines 17–19 define the hybrid potential energy model for the system in the order MM, QC and then NB. The atoms of the first water in the dimer are specified as being in the QC region by passing the selection `firstwater` as the keyword argument `qcselection` on *line 18*. It is important to note that pDynamo requires that an MM representation of the full system exist before a hybrid potential for the system can be specified. This is why the call to `DefineMMModel` precedes that to `DefineQCModel`.

Line 23 calculates the potential energy of the system using the combined QC/MM hybrid potential.

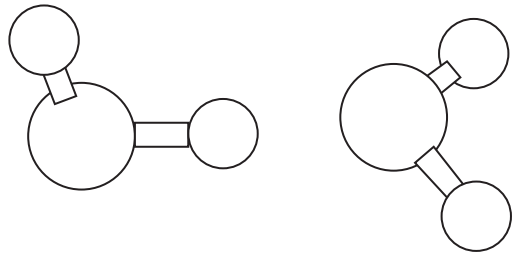


Fig. 6.2. The water dimer studied in Example 8. One molecule is treated with a QC potential and the other with an MM potential.

6.4 Covalent bonds between QC and MM atoms

So far, in this chapter, we have only considered examples in which there are non-bonding interactions between the atoms of the QC and MM regions. But what happens if there are covalent bonds as well? This is a common occurrence when studying systems composed of large molecules because, in these cases, it is often necessary to split a molecule between the different regions.

The treatment of covalent bonds between QC and MM atoms requires special methods. To make the discussion concrete, consider a molecule that contains, somewhere, a single bond between two sp^3-hybridized carbon atoms. The atoms each have four valence electrons and four single bonds to neighbouring atoms. To a good approximation, each atom can be viewed as having four *hybrid orbitals*, one of which participates in each single bond and to each of which an atom donates one electron. This is illustrated in Figure 6.3. Now suppose that the molecule is partitioned at this bond, with one atom being in the QC region and the other in the MM region. As MM atoms do not have electrons or orbitals, this leaves the hybrid orbital of the QC atom that points towards the MM atom with a single, unpaired electron. The presence of this 'dangling' or 'unsatisfied' bond makes the system radical in character and radicals have very different electronic structures from those in which there are no unpaired electrons.

The electronic problem of dangling bonds is perhaps the most crucial one that methods for handling bonds across the boundary of the QC and MM regions

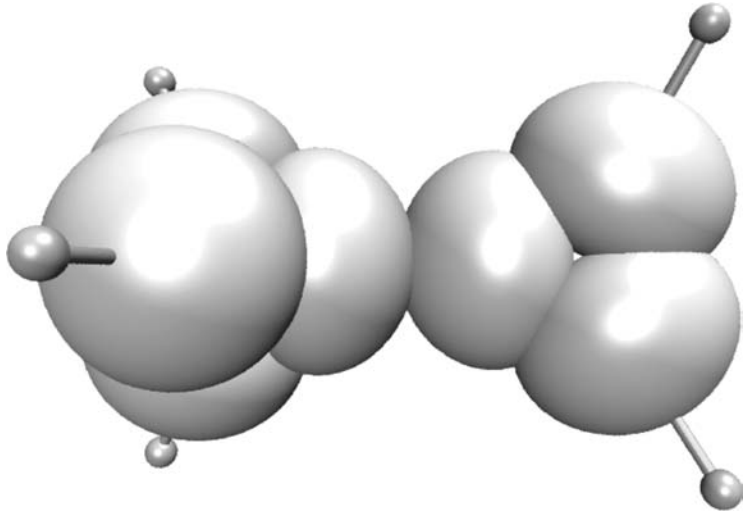


Fig. 6.3. The sp^3 hybrid orbitals of the carbon atoms in the ethane molecule. The molecule is in a staggered conformation which obscures one of the hydrogens and one of the hybrid orbitals for the carbon on the right. The image shows the electronic densities due to the orbitals, not the orbitals themselves.

must be able to tackle, but it is not the only one as they must also ensure that the partitioned system behaves structurally and energetically as much as possible like the unpartitioned one. A variety of methods have been developed to treat QC–MM bonds. Most fall into three classes depending upon how they solve the dangling-bond problem:

(i) Methods that employ hybrid orbitals, or other more general types of *localized* orbital. In one variant of these schemes, hybrid orbitals are generated for the QC atom of the broken QC–MM bond, one of which is forced to point towards the MM atom. All the hybrid orbitals of the QC atom enter into the QC calculation but the form of the orbital pointing towards the MM atom is fixed and it is constrained to be occupied by one electron. The presence of this fixed orbital satisfies the extra valence of the QC atom but the QC–MM bond itself must be represented by an MM bond term.

In other variants of hybrid-orbital approaches, hybrid orbitals are placed on the MM atom of the broken QC–MM bond and enter into the QC calculation. This means, in principle, that there is no need for an MM term to represent the broken bond as its properties are reproduced by the QC part of the hybrid potential. Some methods build a single orbital on the MM atom that points towards the QC atom and it is this orbital whose form and occupancy are fixed during the QC calculation. Others build the full complement of hybrid orbitals for the MM atom and constrain the form and occupancy of all of them during the QC calculation, except the one that points to the QC atom.

A hybrid-orbital method was first used by Warshel and Levitt in their hybrid potential work but other contributions, in the context of more modern hybrid potentials, have been made, among others, by the groups of R. Friesner, of J. Gao and of J.-L. Rivail. The methods in this class are, in many ways, the most elegant of those discussed here but they suffer from the disadvantage that they can entail significant modifications of the algorithms required for performing a QC calculation. This is due to the fact that hybrid and localized orbitals are normally not orthogonal, unlike the orbitals around which most standard QC methods are constructed.

(ii) Methods that use a univalent *capping atom* of fictitious elemental type to replace the MM atom of the broken QC–MM bond in the QC calculation. The position of the capping atom is taken to be the same as that of the MM atom and no MM bond term is normally necessary as the properties of the bond arise from the QC calculation. Examples of fictitious elements include those that are hydrogen-like, with one valence electron, and those that are halogen-like, with seven valence electrons. These methods have the advantage that they require no modification of standard QC algorithms but the disadvantage that fictitious elements have to be conceived for each different type of broken QC–MM bond so that the properties of the respective bonds are correctly reproduced.

(iii) Methods that introduce an extra, univalent atom into the system for each covalent bond between QC and MM atoms. These *dummy* or *link atoms* serve to replace the MM atom in the QC calculation and ensure that the QC atoms of the broken bonds

have no unsatisfied valencies. Link atoms are usually hydrogens and, unlike capping atoms, are not assumed to have the same position as the MM atoms that they represent. Instead they are placed along the bond between the QC and MM atoms at a suitable distance away from the QC atom. The bond itself must be treated with an MM bond term. Although addition of extra atoms into a system is rather inelegant, link-atom methods are probably the easiest to develop and implement and, as such, have been the most widely used in hybrid potential simulation studies.

It should be emphasized that all the methods described above require a careful balancing of the QC and MM contributions to the full potential so that the properties of the partitioned system are correctly described. Thus, each method will have a recipe detailing which MM bonding and non-bonding terms in the region around the QC–MM bonds should be omitted and which included, and whether or not the parameters, and even the analytic forms, of these MM terms remain the same as in the unpartitioned case or need to be modified in some way. These recipes will have been derived by performing calculations on a set of test systems with known properties and then refining the ingredients of the recipe until the desired agreement between known and calculated properties is reached.

pDynamo employs a link-atom method, a schematic of which is shown in Figure 6.4. The method has been aimed to be as simple as possible thereby minimizing the changes needed to the individual QC and MM potentials and the number of new terms and parameters that have to be introduced. Link atoms are added for every covalent bond that occurs between atoms of the QC and MM

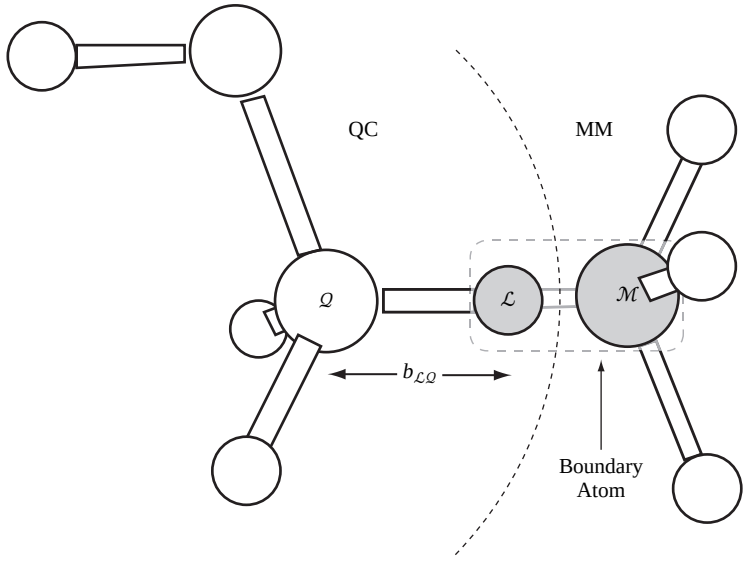


Fig. 6.4. The link-atom scheme illustrated by its application to ethanol.

regions. They are not treated as independent atoms but are considered to be part of the MM atoms of the QC–MM bonds. These combined link and MM atoms are called *boundary atoms*. Link atoms are hydrogens with one electron whose coordinates are constructed automatically whenever they are needed, most notably when the potential energy of the system is evaluated. If $\mathcal{L}$ denotes the link atom and $\mathcal{Q}$ and $\mathcal{M}$ the QC and MM atoms of the QC–MM bond, respectively, the position of a link atom is determined as

$$\boldsymbol{r}_{\mathcal{L}} = \boldsymbol{r}_{\mathcal{Q}} + b_{\mathcal{L}\mathcal{Q}} \frac{\boldsymbol{r}_{\mathcal{M}} - \boldsymbol{r}_{\mathcal{Q}}}{|\boldsymbol{r}_{\mathcal{M}} - \boldsymbol{r}_{\mathcal{Q}}|} \tag{6.7}$$

In this equation, $b_{\mathcal{L}\mathcal{Q}}$ is the distance between the link atom, $\mathcal{L}$, and its QC partner, $\mathcal{Q}$. Distance $b_{\mathcal{L}\mathcal{Q}}$ can be a constant whose value depends upon the identity of the atom $\mathcal{Q}$, typically ~ 1Å, or it can be a more complicated function of various quantities pertaining to the atoms $\mathcal{Q}$ and $\mathcal{M}$.

The link atoms, $\{\mathcal{L}\}$, replace the atoms $\{\mathcal{M}\}$ in the QC calculation which is otherwise carried out as normal. As far as the MM portion of the hybrid potential calculation is concerned, link atoms do not enter in at all because they have no associated MM parameters, even those for Lennard-Jones terms. As for the other interactions, the MM bond term between the atoms $\mathcal{Q}$ and $\mathcal{M}$ is calculated, as are any MM bond, angle or dihedral energy terms that include at least one other MM atom in addition to the atom $\mathcal{M}$. Lennard-Jones interactions between QC and MM atoms are calculated in the way that is normal for the force field that is being used. Thus, in the OPLS-AA case, 1–2 and 1–3 interactions are excluded and 1–4 interactions are scaled.

Of all the QC/MM interactions, the link-atom method is most sensitive to the electrostatic interactions. A general recipe to handle these interactions for all types of QC and MM potentials is still an active area of research and so only a summary of some possible approaches will be given here. Readers should refer to the online pDynamo documentation for full details of the current implementation. In some schemes all interactions are included fully but this often induces instabilities that arise principally from the very short-range and, hence, strong interaction within the boundary atom itself, i.e. between $\mathcal{L}$ and $\mathcal{M}$. One solution to this is simply to omit this interaction, although this can lead to unwanted distortions of the electron density around the link atom due to the fact that the electrons are unduly attracted or repelled by the charges of the MM atoms that are covalently bound to $\mathcal{M}$. More balanced approaches modify the partial charge on $\mathcal{M}$, and sometimes those of its MM neighbours, for the calculation of all the electrostatic interactions, both MM/MM and QC/MM, that involve these atoms. Most drastically this is done by zeroing the charges of these atoms. A gentler method is to replace the point δ-function charge distribution on each of these atoms by *smeared* charge distributions, often Gaussians, which are spread over a finite, but small

region of space. This spreading has the effect of reducing the strength of the electrostatic interactions involving these distributions at short range whilst leaving them unchanged at long range.

We conclude this section with a few general remarks about the use of link atoms in pDynamo. First, the link-atom method is designed to work only for cases in which partitioning of a molecule results in single bonds being broken, so do not expect reliable results if link atoms are put in the middle of aromatic or conjugated groups! Link atoms can be placed along any single bond but those between sp^3-hybridized atoms, ideally carbons, are best because this helps to minimize the perturbation to the electronic structure of the QC region. Second, care must be taken when partitioning a molecule that the MM portion has the desired, usually zero or integer, total charge. This can be troublesome when using some MM potentials but it is straightforwardly achieved with the OPLS-AA force field as most chemical groups, such as aliphatic CH_2 and CH_3 groups, are parametrized so that they have zero or integral charge. Partitioning should therefore occur at the boundaries of these groups. Finally, it is desirable when performing a hybrid-potential study to test different partitionings of a molecule to see how sensitive the results are. If this is not possible, or if in doubt, a general rule of thumb is to use the biggest QC region that is feasible.

6.5 Example 9

The example in this section shows how to split a molecule between QC and MM regions. No extra syntax is needed to perform the calculation because pDynamo detects the presence of covalent bonds between atoms of the QC and MM regions automatically. Whenever the method `DefineQCModel` is called, a check is done to see whether there are bonds across the boundary between the two regions and, if so, the appropriate number of boundary atoms are flagged.

The program is:

```
1 """Example 9."""
2
3 from Definitions import *
4
5 # . Define the MM, NB and QC models.
6 mmmodel = MMModelOPLS ( "booksmallexamples" )
7 nbmodel = NBModelFull ( )
8 qcmodel = QCModelMNDO ( )
9
10 # . Define the molecule.
```

```
molecule = MOLFile_ToSystem ( \
                    os.path.join ( molpath, "bala_c7eq.mol" ) )

# . Define the selection for the first molecule.
methylgroup = Selection ( [ 10, 11, 12, 13 ] )

# . Define the energy model.
molecule.DefineMMModel ( mmmodel )
molecule.DefineQCModel ( qcmodel, qcselection = methylgroup )
molecule.DefineNBModel ( nbmodel )
molecule.Summary ( )

# . Calculate an energy.
molecule.Energy ( )
```

The program is nearly identical to that of Section 6.3. The system that is treated is bALA, instead of the water dimer, and the atoms in the QC region are those of the methyl group sidechain of the alanyl moiety. The indices of these atoms are 10–13 and are contained in the selection `methylgroup` which is passed to the method `DefineQCModel` on *line 18*. This method detects the presence of a single covalent bond between the QC and MM regions and identifies the C_α atom of the molecule as being a boundary atom.

Exercises

6.1 Using Example 8 as a reference, calculate the binding energy of the water dimer using QC, MM and QC/MM models. In the latter case, try partitionings in which first one and then the other water molecule is in the QC region. How different are the results? Note that the binding energy is defined as the energy of the dimer minus the sum of the energies of the isolated monomers.

6.2 Choose a molecule or series of molecules that can be partitioned between QC and MM regions in different ways. Suitable examples are long-chain alcohols and amines. Investigate how changing the partitioning changes the values of particular properties and, if possible, relate these values to the pure QC and pure MM values. Properties that can be tested include atomic charges and dipole moments. If the molecules can be protonated or deprotonated, check how the energy required for removal or addition of a proton changes with the energy model.

7

Finding stationary points and reaction paths on potential energy surfaces

7.1 Introduction

In the last three chapters we have discussed how to calculate the potential energy, and some of its derivatives, for a single geometry of the atoms in a system. Although the calculation of an energy for one or a small number of configurations may sometimes be necessary, it can give only limited information about a system's properties. To investigate the latter more thoroughly it is necessary to identify interesting or important regions on the system's potential energy surface and develop ways in which they can be explored. Methods to do this will be investigated in this chapter.

7.2 Exploring potential energy surfaces

The function that represents a system's potential energy surface is a multidimensional function of the positions of all the system's atoms. It is this surface that determines, in large part, the behaviour and the properties of the system. A little reflection shows that the number of configurations or geometries available to a system with more than a few atoms is enormous. A simple example should make this clear. Take a diatomic molecule or, more generally, any system comprising two atoms in vacuum. The geometry of such a molecule is completely determined by specifying the distance between the two atoms and so the potential energy surface is a function of only one geometrical variable. It is easy to search the entire potential energy surface for this system. Start with a small interatomic distance, calculate the energy, increase the distance by a certain amount and then repeat the procedure. In this way we can obtain a picture similar to those in Figures 5.1, 5.2 and 5.5.

Consider next a three-atom system, such as a molecule of water, for which there are three independent geometrical parameters. These can be specified, for example, as the three interatomic distances or by two distances and an angle. If the

atoms are collinear then there will be only two independent variables. In either case, the potential energy surface can again be explored, although substantially more calculations will be required. To see this, suppose that n values of each independent parameter are required in the search. It will be necessary to calculate n^2 energies for the linear system and n^3 energies in the general case. If n takes the value 10 – a reasonable number – then 100 and 1000 energy calculations will be needed, respectively. The extension of this argument to larger systems shows that to search a potential energy surface in this simplistic fashion for a system with m geometrical parameters requires n^m energy calculations – a prohibitively large number except for small values of m and n.

One problem with the simplistic search scheme outlined above is that it is very wasteful, in that most of the configurations for which the energy will be calculated will not be chemically reasonable. Their energies will be too high, either because some atoms will be much too close and overlap or because some atoms that should be bonded together will be too far apart. Important configurations will be those that have a low energy and so it is for these that we would like to look in any search procedure that we adopt. To make this criterion more precise, we normally search for points on the surface that are *minima*, i.e. points that have the lowest energy on a particular part of a potential energy surface. A minimum is characterized by the property that any small changes in the geometry of a system that is at a minimum will lead to an increase in energy. Minima and the regions of the potential energy surface around them correspond, roughly speaking, to the *stable states* in which a system will normally be found.

Minima are an example of *stationary points* on a potential energy surface. Stationary points are defined as points for which the first derivatives of the energy with respect to the geometrical parameters are zero. For the special case in which the Cartesian coordinates of the atoms are the geometrical parameters, we have the condition

$$\boldsymbol{G} = \frac{\mathrm{d}\mathcal{V}}{\mathrm{d}\boldsymbol{R}} = \boldsymbol{0} \tag{7.1}$$

To distinguish between different types of stationary points, a second condition, which uses the second derivatives of the energy with respect to the geometrical parameters, is needed. The condition states that the stationary points of a potential energy surface can be classified into different types depending upon the number of negative eigenvalues that its second derivative matrix possesses. If there are no negative eigenvalues then the point is a minimum, if there is one negative eigenvalue then the point is a *first-order saddle point* (more usually shortened to just *saddle point*) and if there are n negative eigenvalues the point is an *nth-order saddle point*.

As we have seen before, the eigenvalues of a matrix are determined by diagonalizing it. If **H** denotes the second-derivative matrix, the eigenvalue equation it obeys is

$$\mathbf{H}\boldsymbol{e}_i = \lambda_i \boldsymbol{e}_i \tag{7.2}$$

where λ_i is the ith eigenvalue and $\boldsymbol{e}_i$ its associated eigenvector. If Cartesian coordinates are used, the matrix **H** will have a dimension $3N \times 3N$, where N is the number of atoms, and so there will be $3N$ eigenvalues and eigenvectors. As the matrix **H** is symmetric, the eigenvalues are guaranteed to be real (either negative or positive, but not complex) and the eigenvectors will form an orthonormal set.

Physically, if a stationary point has a negative eigenvalue it means that a small (or, to be exact, an infinitesimal) displacement of the geometry of the system along the direction defined by the eigenvector corresponding to the negative eigenvalue will lead to a reduction in the system's energy. Equally, a small displacement along the eigenvector of a positive eigenvalue will lead to an increase in energy. Thus, a minimum has no negative eigenvalues and so all displacements increase the energy, whereas for an nth-order saddle point there will be n displacements that decrease the energy and $3N - n$ displacements that increase it. If an eigenvalue has a value of zero then a small displacement along its eigenvector results in no change in energy.

Minima are not the only stationary points on the surface that are of interest, although they are probably the most important. We are also often interested in searching for first-order saddle points. The reason for this is illustrated in Figure 7.1, which shows a model potential energy surface. This surface has three minima and two (first-order) saddle points. As stated above, each minimum corresponds to a stable state of the system but, if there is a reaction or the system undergoes a conformational change, the system will move from one stable state and, hence, minimum to another. Looking at Figure 7.1 it can be seen that there are many possible *reaction paths* that join different minima. Remembering that a system will spend most of its time in low-energy regions of a surface, it is easily seen that the paths with the lowest energy pass through the saddle points. In other words, the first-order saddle point is the point of highest energy along the reaction path of lowest energy that joins two minima. Thus, the location of saddle points is important when studying reactions and transitions between different geometrical configurations in molecular systems.

Now that we have a specific mathematical criterion for the identification of points on the surface in which we are interested it is possible to formulate algorithms to search for such points. In this chapter, we shall discuss a few of these that all have the common property that they are *local search algorithms*. That is they start at a given geometrical configuration of the system and then

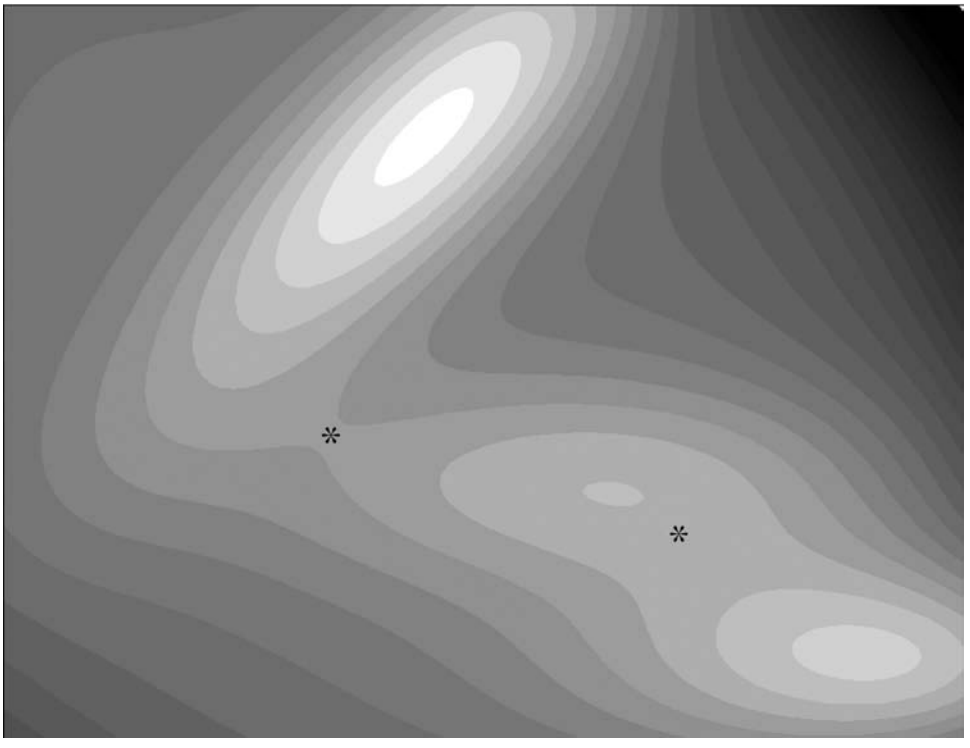


Fig. 7.1. A contour plot of the Müller–Brown two-parameter model potential energy surface. The darker the shading the larger the function's value. The minima are located in the three regions of lightest shading and the approximate positions of the two saddle points are indicated by asterisks.

look for a stationary point that is in the vicinity of this starting geometry. This is a useful approach but it does have drawbacks because we may be interested in finding the minimum, for example, which has the lowest energy on the entire potential energy surface. The algorithms given here will not usually be useful in this case and so-called *global search algorithms* will be required. We shall return to this subject later.

To see that local search algorithms will not always be useful when searching for the *global minimum* (or the near-global minimum) we can look at particular classes of systems for which the number of minima either is known or can be estimated. One such class that is widely used as a test case for optimization algorithms is the *Lennard-Jones clusters*. These are groups of a given number of identical atoms that interact solely via Lennard-Jones terms (Equation (5.10)) and that act as models of clusters of rare-gas atoms, such as argon. For small numbers of atoms it is possible to enumerate or find all the minima. For larger numbers of atoms it is necessary to estimate the number. The numbers of minima

Table 7.1 *Numbers of minima and the lowest energies of some Lennard-Jones clusters*

Number of atoms	Number of minima	Lowest energy (reduced units)
8	8	−19.822
9	18	−24.113
10	57	−28.420
11	145	−32.765
12	366	−37.967
13	988	−44.327
14	2 617	−47.845
15	6 923	−52.323
16	18 316	−56.816
17	48 458	−61.318
18	1.28×10^5	−66.531
19	3.39×10^5	−72.660

and the energies of the global minima for the clusters consisting of 8–19 atoms are given in Table 7.1. The energies are reported in *reduced units*, which means that they have been calculated assuming that the well depths and the atom radii in the Lennard-Jones energy term both take the value of 1. It can be seen that the number of minima for the system increases very rapidly with the number of atoms. In fact the number of minima increases, to a reasonable approximation, as the exponential of the number of atoms in the cluster, i.e. as $\exp(N)$. This should give some idea of the difficulty of finding the global minimum for a system such as a protein, which consists of 1000 atoms or more!

7.3 Locating minima

The location of a minimum on a potential energy surface is an example of a well-studied mathematical problem – that of the minimization of a multidimensional function. There is a large number of standard methods available for performing such a task, but the choice of the best method for a particular problem is determined by such factors as the nature of the function to be minimized, whether and which derivatives are available, and the number of variables.

A description of local optimization methods will not be undertaken here as the details are rather lengthy and good expositions are available in texts on numerical techniques. Instead, we classify optimization algorithms by the type of derivatives that they use. We mention four types.

(i) *No derivatives.* These algorithms make use of function (or, in our case, energy) values only. An example of these algorithms would be the *simplex* method and the crude search algorithm discussed in the last section.
(ii) *First derivatives.* These algorithms use the values of the energy and its first derivatives. Examples include *conjugate gradient* and *steepest descent* methods.
(iii) *First and exact second derivatives.* These algorithms use energy values and the first and second derivatives. Examples are the *exact Newton* and *Newton–Raphson* methods.
(iv) *First and approximate second derivatives.* These algorithms use energy values and the exact first derivatives of the energy. They also use approximations to the second-derivative matrix, either to the full matrix or to the derivatives in a subspace of the geometrical parameters. Methods of this type include the *quasi-Newton*, *reduced basis-set Newton* and *truncated Newton* algorithms.

As a rule, the algorithms that use derivative information are more efficient at finding minima. For this reason and because the derivatives of the energy are readily calculated, methods that use derivatives are more commonly employed for molecular problems. In most cases, it is likely that the exact Newton algorithms would require the least number of energy and derivative evaluations, but there are three factors that can limit the use of second-derivative methods. These are:

(i) The second-derivative matrix uses a lot of storage space. For N atoms there are $3N(3N+1)/2 \simeq O(N^2)$ elements, which is often too much space when there are more than a few hundred atoms in the system. First-derivative methods require storage that scales as $O(N)$.
(ii) Most algorithms that use second derivatives require that the eigenvalues and eigenvectors of the matrix be found and, because the diagonalization of a matrix is an operation that scales as $O(N^3)$, this becomes difficult or impossible when there is a large number of atoms. The computational cost associated with handling the first-derivative vector scales as only $O(N)$.
(iii) The second derivatives can be significantly more expensive to calculate than the energy and first derivatives. This is true for QC methods and for MM methods that contain non-pairwise additive terms, such as those employing polarization terms of the type discussed in Section 5.2.2. By contrast, the cost of calculating the second derivatives when using the simple, pairwise-additive MM potentials described in Chapter 5 is comparable to and has the same scaling behaviour as that required for the calculation of the energy and first derivatives.

To summarize, for systems with small numbers of atoms, exact or quasi-Newton methods can be used and are likely to be the most efficient. The only methods practicable for systems with large numbers of atoms are those that use the first derivatives, such as conjugate gradient algorithms, or a reduced set of approximate second derivatives.

In this section we present a conjugate gradient approach for minimizing the energy of the system. It is a good general-purpose algorithm that is appropriate even for very large systems. The function implementing the algorithm is `ConjugateGradientMinimize_SystemGeometry` and its definition is:

Function `ConjugateGradientMinimize_SystemGeometry`

Minimize the geometry of a system using a first-derivative conjugate gradient algorithm.

Usage:

```
ConjugateGradientMinimize_SystemGeometry (
                    system,
                    logfrequency          = 1,
                    maximumiterations     = 50,
                    rmsgradienttolerance = 0.001 )
```

`system` is the system whose geometry is to be minimized.

`logfrequency` specifies the frequency at which information about the minimization procedure is printed to the log file. A negative or zero value means that no printing is done.

`maximumiterations` is the number of iterations of the conjugate gradient algorithm to perform. All optimization algorithms are iterative, in that they start with a given set of variables and then refine those variables in successive steps. This argument defines the maximum number of iterations or steps that are to be performed during the minimization. If a minimum is found in fewer steps then the minimization will exit, otherwise it will stop when `maximumiterations` have been performed.

`rmsgradienttolerance` is the parameter that determines the *convergence criterion* for termination of the minimization process. A large variety of convergence criteria are in use but this one refers to a condition on the value of the RMS gradient of the energy, G_{RMS}, which is defined as follows:

$$G_{\mathrm{RMS}} = \sqrt{\frac{1}{3N}\sum_{i=1}^{N} \boldsymbol{g}_i^{\mathrm{T}}\boldsymbol{g}_i} \tag{7.3}$$

At a stationary point, of course, the gradient vector is zero. In actual calculations the gradient vector will rarely, if ever, be able to attain this value at the end of an optimization and so some finite value for the RMS gradient is employed to indicate that a stationary point has been reached. If the value of the RMS gradient falls below

the value given by `rmsgradienttolerance` at any step then the optimization is assumed to have converged and the algorithm stops.

Remarks: The variables that are minimized in this function are the system's Cartesian coordinates in its `coordinates3` attribute. When the function exits, either because convergence has been achieved or the number of iterations has been exhausted, this attribute will contain the coordinates of the point with the lowest energy that was found during the minimization procedure.

7.4 Example 10

To illustrate the use of the minimization function we present a program for the optimization of the blocked alanine molecule, bALA:

```
"""Example 10."""

from Definitions import *

# . Define the molecule and its QC model.
molecule = XYZFile_ToSystem ( \
                    os.path.join ( xyzpath, "bala_c7eq.xyz" ) )
molecule.DefineQCModel ( QCModelMNDO ( "am1" ) )
molecule.Summary ( )

# . Save a copy of the starting coordinates.
coordinates3 = Clone ( molecule.coordinates3 )

# . Determine the starting energy.
estart = molecule.Energy ( )

# . Optimization.
ConjugateGradientMinimize_SystemGeometry ( molecule, \
                         logfrequency         =  100, \
                         maximumiterations    = 2000, \
                         rmsgradienttolerance = 0.001 )

# . Determine the final energy.
estop = molecule.Energy ( )

# . Determine the RMS coordinate deviation between the structures.
masses = molecule.atoms.GetItemAttributes ( "mass" )
coordinates3.Superimpose ( molecule.coordinates3, \
```

```
                                                    weights = masses )
25 rms = coordinates3.RMSDeviation ( molecule.coordinates3, \
                                                    weights = masses )
26
27 # . Print the results.
28 table = logfile.GetTable ( columns = [ 30, 30 ] )
29 table.Start ( )
30 table.Title ( "Minimization Results" )
31 table.Entry ( "Energy Change",            alignment = "l" )
32 table.Entry ( "%20.4f" % ( estop - estart, ) )
33 table.Entry ( "RMS Coordinate Deviation", alignment = "l" )
34 table.Entry ( "%20.4f" % ( rms, ) )
35 table.Stop ( )
```

Lines 6–8 define the molecule and its energy model.

Line 11 creates a copy of the starting coordinates for later use by cloning the `coordinates3` attribute of `molecule`.

Line 14 calculates the energy of the molecule at the initial configuration.

Line 17 minimizes the geometry of the molecule using the conjugate gradient method. A maximum of 2000 iterations is requested with details about the minimization to be printed at 100-iteration intervals.

Line 20 determines the molecule's energy at the final geometry.

Lines 23–25 calculate the mass-weighted RMS coordinate deviation between the unoptimized and optimized structures. This provides a crude measure of how much the structure has altered as a result of the minimization process.

Lines 28–35 print out the results of the calculation. In this case, the minimization performs about 500 iterations before the convergence criterion on the RMS gradient (10^{-3} kJ mol^{-1} Å^{-1}) is satisfied, resulting in a total reduction in energy between the two configurations of $\sim$40 kJ mol^{-1}. The RMS coordinate deviation between the two structures is about 0.1 Å.

7.5 Locating saddle points

Whereas algorithms for the location of minima have been the subject of intense research by mathematicians, algorithms for the location of saddle points have been studied much less widely. As a result, many of these algorithms have been developed by chemists and other researchers directly interested in looking at potential energy surfaces. The location of saddle points is usually a much more demanding task than the location of minima. The main reason for this is intuitive. When searching for a minimum one always wants to reduce the value of the energy, so one can choose to go in any direction that gives this

result. When looking for saddle points, however, one is trying to find a point that is a *maximum* in one direction but a minimum in all the others and so the algorithm has to perform a delicate balancing act between the two conflicting types of search.

A wide range of saddle-point-location algorithms has been proposed. In this section we shall discuss a *mode-following* algorithm developed by J. Baker, which is widely used and appears to be one of the most efficient methods available. It uses both first- and second-derivative information and so its application is restricted to relatively small systems, although this is a limitation of most saddle-point-location subroutines. In principle, whenever a stationary point has been obtained by an algorithm for the location of minima or saddle points, the only way to verify that the point is of the type required is to determine how many negative eigenvalues there are in the second-derivative matrix. This is especially desirable for saddle points due to the difficulties with saddle-point searches but less so for minima because the algorithms for the location of minima are more robust. When treating large systems for which the second-derivative matrix cannot be diagonalized due to computational constraints it is generally assumed that the stationary point obtained is a minimum. Saddle-point searches in these cases are very difficult.

To understand the theory behind Baker's algorithm we consider a point on the potential energy surface of the system with coordinates $\boldsymbol{R}_0$. We then approximate the energy, $\mathcal{V}$, of neighbouring points on the surface using a Taylor series. If the coordinates of the displaced point on the surface are $\boldsymbol{R}$, where $\boldsymbol{R} = \boldsymbol{R}_0 + \boldsymbol{D}$ and $\boldsymbol{D}$ is a displacement vector, then the Taylor series to second order in the displacements is

$$\mathcal{V}(\boldsymbol{R}) = \mathcal{V}(\boldsymbol{R}_0) + \boldsymbol{G}^{\mathrm{T}}\boldsymbol{D} + \frac{1}{2}\boldsymbol{D}^{\mathrm{T}}\mathbf{H}\boldsymbol{D} + \cdots \tag{7.4}$$

where $\boldsymbol{G}$ is the vector of first derivatives and $\mathbf{H}$ is the matrix of second derivatives, both of which are evaluated at the point $\boldsymbol{R}_0$. To find the displacement vector, $\boldsymbol{D}$, which minimizes the energy of this expression, we differentiate Equation (7.4) with respect to $\boldsymbol{D}$ to obtain $\mathrm{d}\mathcal{V}/\mathrm{d}\boldsymbol{D}$, set the result to zero and solve to get

$$\boldsymbol{D} = -\mathbf{H}^{-1}\boldsymbol{G} \tag{7.5}$$

This is the *Newton–Raphson (NR) step*. The inverse of $\mathbf{H}$ can be written in terms of the eigenvalues and eigenvectors of $\mathbf{H}$, in which case Equation (7.5) becomes

$$\boldsymbol{D} = -\sum_i \frac{\boldsymbol{e}_i^{\mathrm{T}}\boldsymbol{G}}{\lambda_i}\boldsymbol{e}_i \tag{7.6}$$

The NR step is structured so that it will minimize along the eigenvectors or *modes* with positive eigenvalues and maximize along modes with negative eigenvalues. Thus, it will tend to optimize to structures that have the same number of negative eigenvalues or *curvature* as the original structure. If the starting structure is of the correct curvature, the NR step is a good one to take, but, in the general case, the NR step must be modified so that structures with the desired curvature can be obtained.

The required modification, developed by C. Cerjan and W. Miller and by J. Simons and his co-workers, is a simple one and involves altering the denominator of Equation (7.6) by adding parameters, γ_i, which shift the values of the eigenvalues:

$$\boldsymbol{D} = -\sum_i \frac{\boldsymbol{e}_i^{\mathrm{T}} \boldsymbol{G}}{\lambda_i - \gamma_i} \boldsymbol{e}_i \tag{7.7}$$

Normally, only two different shift parameters are used – one for the modes along which a minimization is to be done and another along which there is to be a maximization. In the special case of searching for a minimum, only the first of these will be required. There are several prescriptions for choosing the values of the shift parameters, which need not concern us here. The important point is that, with an appropriate choice, it is possible to force a search along a particular mode opposite to that in which it would normally go. For example, when searching for a minimum the shift parameter would be negative and have a value less than that of the smallest eigenvalue. This would ensure that the denominators in Equation (7.7) are always positive and so a minimization occurs along all the modes. When the optimization reaches a region of the correct curvature the value of the shift parameter can be reduced. For a saddle-point search there will be one mode along which there is a maximization and for which the shift parameter will be such that the denominator for that mode in Equation (7.7) will be negative. This means that it is possible to start at a minimum on the potential energy surface and 'walk' up one of the modes until a saddle point is reached.

The definition of the function that implements Baker's algorithm in pDynamo is as follows.

Function `BakerOptimize_SystemGeometry`

Optimize the geometry of a system using Baker's algorithm.

Usage:

```
BakerOptimize_SystemGeometry (
                system,
                followmode           = 1,
                logfrequency         = 1,
                maximumiterations    = 50,
                QMINIMIZE            = False,
                rmsgradienttolerance = 0.001 )
```

`system` is the system whose geometry is to be optimized.

`followmode` is an integer that specifies, for a saddle-point search, the mode along which the maximization is to be performed. If no value is given then the mode chosen is automatically the one with the lowest eigenvalue. The mode-following option is most useful when starting out from a minimum. In that case all the eigenvalues are positive. Often it turns out that walking up the 'softest' mode (the lowest eigenvalue mode) will lead to a saddle point. This is not always true and so it is sometimes useful to search in other directions.

`QMINIMIZE` is a Boolean argument that specifies whether a minimum or a saddle point is to be searched for. The default is a saddle-point search.

Remarks: The remaining arguments have the same behaviour as those of `ConjugateGradientMinimize_SystemGeometry` in Section 7.3.

There are two technical points that can be made about the implementation of the algorithm. First of all, the full second-derivative matrix is required at each iteration. In principle, this could be calculated afresh each time it is needed but it is wasteful for QC energy models for which the evaluation of the second derivatives is expensive. Instead, the matrix is calculated once at the beginning of the calculation and then *updated* using the values of the gradient vector calculated at the current point. Although the updated matrix is only an approximation to the true second-derivative matrix, it is usually sufficiently accurate to allow a saddle point to be found and results in a much faster algorithm. In the case of molecular mechanics energy functions, it is the time required for the diagonalization of the second-derivative matrix that limits the application of the method, not the time required for the calculation of the second derivatives, and so evaluation of the second-derivative matrix at each step is feasible.

The second technical point concerns the use of Cartesian coordinates. From the arguments at the beginning of the chapter it will be remembered that one parameter

completely defines the geometry for a two-atom system, three parameters for a three-atom system and so on but, for a two-atom system, there are six Cartesian coordinates and for a three-atom system there are nine. This means that the set of Cartesian coordinates contains too many variables and is redundant. The redundancies, in fact, are related to the overall translational and rotational motions which the entire system can undergo. For first-derivative algorithms these motions are unimportant. For algorithms that use second derivatives it is necessary to modify the second-derivative matrix so that they are removed. This is done by pDynamo's Baker algorithm but a more detailed discussion of this point will be left to the next chapter.

7.6 Example 11

To illustrate the Baker algorithm we choose to study the molecule cyclohexane, which undergoes a well-known transition between different conformational forms. In this example we start with the chair form of the molecule and look for a saddle point that leads to another conformer.

The program is very similar to that of Example 10 in Section 7.4 and is

```
1  """Example 11."""
2
3  from Definitions import *
4
5  # . Define the molecule and its QC model.
6  molecule = XYZFile_ToSystem ( \
                  os.path.join ( xyzpath, "cyclohexane_chair.xyz" ) )
7  molecule.DefineQCModel ( QCModelMNDO ( "am1" ) )
8  molecule.Summary ( )
9
10 # . Determine the starting energy.
11 estart = molecule.Energy ( )
12
13 # . Optimization.
14 BakerSaddleOptimize_SystemGeometry ( molecule, \
                     logfrequency         =  100, \
                     maximumiterations    = 2000, \
                     rmsgradienttolerance = 0.001 )
15
16 # . Determine the final energy.
17 estop = molecule.Energy ( )
18
19 # . Print the energy change.
```

```
20 logfile.Paragraph ( "Energy change after search = %20.4f\n" % \
                                                ( estop - estart, ) )
21
22 # . Save the coordinates.
23 molecule.label = "Cyclohexane saddle conformation."
24 XYZFile_FromSystem ( os.path.join ( scratchpath, \
                                "cyclohexane_saddle.xyz" ), molecule )
```

Lines 6–8 define the molecule and its energy model. The coordinates in the XYZ file correspond to the chair conformation of cyclohexane.

Line 11 calculates the energy of the molecule at the initial configuration.

Line 14 searches for a saddle point using the Baker algorithm by following the mode with the lowest eigenvalue.

Line 17 determines the energy of the molecule at the final configuration.

Line 20 prints out the energy difference between the chair and saddle point conformations. The algorithm finds a saddle point and requires about 40 steps to reach convergence. The structures of the chair conformer and the saddle point found in the search are shown in Figure 7.2 together with those of the boat and twist-boat conformers of cyclohexane.

Lines 23–24 save the optimized coordinates of the saddle point in XYZ format. The name of the directory used for storing the file is in `scratchpath` which is where all files written by the examples in this book are put.

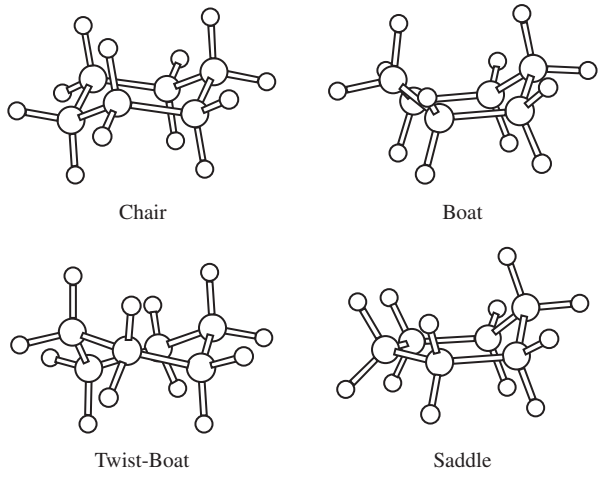


Fig. 7.2. The structures of the chair, boat and twist-boat conformers of cyclohexane and the saddle-point structure found in Example 11.

7.7 Following reaction paths

A reaction, which can involve either the breaking and forming of bonds or a change in the conformation of a molecule, may be defined in terms of the minima on the potential energy surface and the saddle points that lie between them. The minima comprise the reactant and product configurations and any stable intermediate states. In principle, knowledge of the minima and saddle points is sufficient to give a detailed picture of the *mechanism* of the transition. In practice, though, it is often useful to be able to generate points intermediate between the minima and saddle points so that the geometrical and other changes occurring during the transition can be analysed in greater detail.

Although the stationary points on the potential energy surface have a unique definition, there is no single definition of a reaction path. It is not even necessary, as we shall see later, to define it so that it goes through the minima and saddle points that correspond to the transition being studied. The definition that we shall adopt in this section is that of the *steepest descent reaction path* (SDRP), which is probably the one in most common use, at least for the study of reactions in systems with small numbers of atoms. K. Fukui was one of the earliest proponents of this definition, although he called it the *intrinsic reaction coordinate* (IRC).

The SDRP is usually defined in terms of *mass-weighted* Cartesian coordinates, although unweighted coordinates can also be used. If $\boldsymbol{r}_i$ is the coordinate vector of atom i and m_i its mass, the mass-weighted coordinate, $\boldsymbol{q}_i$, is

$$\boldsymbol{q}_i = \sqrt{m_i}\boldsymbol{r}_i \tag{7.8}$$

The mass-weighted gradient of the energy for an atom, $\boldsymbol{\zeta}_i$, is defined in a similar way in terms of the unweighted gradient, $\boldsymbol{g}_i$:

$$\boldsymbol{\zeta}_i = \frac{\boldsymbol{g}_i}{\sqrt{m_i}} \tag{7.9}$$

It is normal to introduce a parameter, s, that denotes the distance along the reaction path. If we do this the coordinates of the atoms in the structures along the path are also functions of s and the SDRP can be formally described by the differential equation

$$\frac{\mathrm{d}\boldsymbol{q}_i(s)}{\mathrm{d}s} = \pm\frac{\boldsymbol{\zeta}_i}{\sqrt{\sum_{i=1}^{N}\boldsymbol{\zeta}_i^2}} \tag{7.10}$$

where the plus sign refers to a path going up from a minimum and the minus sign to one descending from a saddle point. Mathematically what this equation means is that the reaction path at any point follows the direction of the mass-weighted gradient, which is equivalent to the direction of steepest descent.

Practically, the easiest way to generate structures on a reaction path and the one that will be used here, is to start at a saddle point and to take a series of finite steps in the direction of descent. If the current structure that has just been generated is denoted by the index α, the coordinates of the next structure are calculated from

$$\boldsymbol{q}_i^{\alpha+1} = \boldsymbol{q}_i^{\alpha} - \left(\frac{\boldsymbol{\zeta}_i^{\alpha}}{\sqrt{\sum_{i=1}^{N} (\boldsymbol{\zeta}_i^{\alpha})^2}} \right) \delta s \tag{7.11}$$

where δs is a small step to be taken down the path. This algorithm works reasonably well if the step size is small. Even with small step sizes, though, the path generated will have a tendency to oscillate about the true path and it will not end up exactly at the minimum, although it should terminate in its vicinity. Other more sophisticated algorithms have been developed to generate smoother reaction paths. These often start off with a step similar to that given in Equation (7.11) but then try to refine the point using *line-search* or *constrained minimization* techniques. We do not present an implementation of one of these techniques here, although an alternative method for calculating smooth reaction paths is described at the end of the chapter.

The only problem with the definition of the path in Equation (7.10) is that it does not hold at a stationary point. At these points the gradient is zero and the denominator on the right-hand side of the equation is undefined. Therefore, when starting at a saddle point, a different type of starting step must be chosen. This is done by calculating the eigenvector of the (mass-weighted) second-derivative matrix at the saddle point that corresponds to the mode with the negative frequency. This mode produces a reduction in energy of the system and so points downhill when the system is displaced along it away from the stationary point. Thus, the step taken from a saddle point is the one that goes along this mode (in either the plus or the minus direction because there are two downhill directions).

The algorithm described above has been implemented as the function `SteepestDescentPath_SystemGeometry`. It has the following definition:

Function `SteepestDescentPath_SystemGeometry`

Generate a reaction path for a system.

```
SteepestDescentPath_SystemGeometry (
                system,
                functionstep  =  2.0,
                logfrequency  =  1,
```

Usage:
```
maximumiterations  =  50,
pathstep           =  0.025,
QMASSWEIGHTING     =  False,
savefrequency      =  0,
trajectory         =  trajectory)
```

`system` is the system for whose geometry a reaction path is to be calculated.

`functionstep` is the parameter which determines the size of the initial step away from the saddle point. It specifies the approximate reduction in energy required as a result of the step and is calculated using the formula $\sqrt{-2\Delta E/\lambda_-}$, where ΔE is the value of `functionstep` and λ_- is the value of the negative eigenvalue.

`pathstep` is the value of δs to be taken for the steepest-descent steps. The units are $\sqrt{\text{a.m.u.}}$ Å if mass-weighting is used and Å otherwise.

`QMASSWEIGHTING` is a Boolean argument indicating whether mass-weighted coordinates are to be used.

`savefrequency` specifies the frequency with which to write out calculated structures to an external file for subsequent analysis. If the frequency is less than unity or greater than `maximumiterations` no structures are written out. When structures are saved the initial structure for the steepest descent procedure is also saved.

`trajectory` is a *trajectory* object to which the structures are to be saved. We shall encounter trajectories many times in this book but fuller discussions will be postponed to Section 7.9 and to Chapter 9 when we consider how to analyse trajectory data.

Remarks: The remaining arguments have the same behaviour as those in `ConjugateGradientMinimize_SystemGeometry`.

The function generates a sequence of structures using Equation (7.11). It automatically checks whether the input configuration is a saddle point and, if so, the path is generated in both directions. If not, a single-sided path is generated by following the gradient vector down from the starting point.

A final point can be made about the use of this algorithm. Having found a saddle-point structure (using the algorithm of the previous section, for example) it is not uncommon not to know to which minima it leads (either one or both)! In this case, the reaction-path-following algorithm can be used to identify them.

7.8 Example 12

In this section we illustrate the use of the reaction-path-following algorithm by tracing out the path from the saddle point for cyclohexane calculated in the previous example. The program is:

```
1  """Example 12."""
2
3  from Definitions import *
4
5  # . Define the molecule and its QC model.
6  molecule = XYZFile_ToSystem ( \
             os.path.join ( scratchpath, "cyclohexane_saddle.xyz" ) )
7  molecule.DefineQCModel ( QCModelMNDO ( "am1" ) )
8  molecule.Summary ( )
9
10 # . Calculate an energy.
11 molecule.Energy ( )
12
13 # . Create an output trajectory.
14 trajectory = SystemGeometryTrajectory ( \
           os.path.join ( scratchpath, "cyclohexane_sdpath.trj" ), \
                                              molecule, mode = "w" )
15
16 # . Optimization.
17 SteepestDescentPath_SystemGeometry ( molecule,                  \
                                  functionstep      =       2.0, \
                                  logfrequency      =        10, \
                                  maximumiterations =       400, \
                                  pathstep          =     0.025, \
                                  QMASSWEIGHTING    =      True, \
                                  savefrequency     =        10, \
                                  trajectory        = trajectory )
18
```

Lines 6–11 define the molecule and its energy model and calculate its energy at the saddle point.

Line 14 defines an instance of the `SystemGeometryTrajectory` class which will be used to store the reaction path structures. We shall introduce this class in the next section.

Line 17 generates the reaction path. It is calculated with 400 structures at separations of 0.025 $\sqrt{\text{a.m.u.}}$ Å and the initial step is such that there will be a drop in energy of approximately 2 kJ mol^{-1} on displacement from

the saddle point. The details about the path points are printed every ten steps and structures are written to the trajectory every tenth step, giving 41 structures to be written out in all.

The path generated by this example leads to the chair form of cyclohexane in one direction and to the twist-boat form in the other direction. Four hundred steps suffice to bring the path down into regions of the potential energy surface that correspond to the two different conformers, although the structures generated are relatively far from the exact ones and the path tends to meander around them. The parameters employed here give reasonable results for cyclohexane but for other systems it is advisable to try other values to see which produce the best paths.

7.9 Determining complete reaction paths

In Section 7.7 we saw how to calculate reaction paths when a saddle-point structure was available. However, as indicated in Section 7.5, the calculation of saddle-point structures, using methods that require second derivatives, is much more difficult for large systems and alternative approaches for determining reaction paths are needed. The development of algorithms for such calculations is an area of active research and a range of competing methods have been proposed. We shall discuss one of the earlier, and one of the more elegant, methods of this type which was developed by R. Elber and co-workers in the early 1990s. It may no longer be the most efficient algorithm that exists but it illustrates well the principles involved in this general class of methods.

A different approach from that of the algorithms discussed in the previous sections of this chapter is taken. Instead of locating saddle points, the method attempts to generate a discretized version of the reaction path that consists of a sequence of intermediate structures lying between reactants and products. It does this by first generating a *chain* of structures to represent the path and then refining the chain by minimizing an object function, $\mathcal{F}$, with respect to the coordinates of *all* the chain structures simultaneously. Clearly the form of the object function is the crux of the algorithm because it must be designed so that the structures that result from its minimization lie on the required reaction path. Elber and co-workers chose a function consisting of two terms, one a sum of the potential energies of the chain structures, and the other a set of *constraint conditions* that limit how the structures are arranged with respect to each other. The form is

$$\mathcal{F} = \sum_{I=1}^{M} \mathcal{V}_I + \gamma \sum_{I=0}^{M} (d_{I,I+1} - \langle d \rangle)^2 + \frac{\rho}{\kappa} \sum_{I=0}^{M-1} \exp\left[-\kappa \left(\frac{d_{I,I+2}}{\langle d \rangle} \right)^2 \right] \qquad (7.12)$$

where $\mathcal{V}_I$ is the energy of the Ith structure, $d_{I,J}$ is the distance between two structures I and J, $\langle d \rangle$ is the average distance between neighbouring structures

and γ, κ and ρ are constants. The number of structures in the chain is $M+2$. The reactant's structure is labelled 0, the product's structure $M+1$ and the intermediate structures by values ranging from 1 to M. The distances between structures are defined as

$$d_{I,J} = |\boldsymbol{R}_I - \boldsymbol{R}_J| \tag{7.13}$$

where $\boldsymbol{R}_I$ is the $3N$-dimensional vector containing the coordinates of the atoms of structure I. The average distance between neighbouring structures is

$$\langle d \rangle = \frac{1}{M+1} \sum_{I=0}^{M} d_{I,I+1} \tag{7.14}$$

If the system contains N atoms, the object function, $\mathcal{F}$, is a function of $3NM$ variables (note that the coordinates of the reactant and product structures are not optimized). The first constraint term in the function (the γ term) is a term that keeps adjacent structures roughly the same distance apart. The second term (the ρ term) keeps structures separated by one other structure. This avoids the problem of having the chain collapse or fold back on itself during the optimization procedure. Once the object function has been defined, the algorithm works by minimizing the function with respect to all the atomic coordinates, using a standard technique, until the convergence criteria have been satisfied.

There is one extra complication, which arises due to the fact that we are using Cartesian coordinates to define the atom positions in each structure. We discussed the same problem at the end of Section 7.5 and it concerns the translational and rotational degrees of freedom of the structures. Once the structures in the chain to be minimized have been defined it is normal to reorient each of them with respect to a reference structure so that the distances, $d_{I,J}$, between the structures are minimized. The quaternion technique discussed in Section 3.6 is appropriate for the reorientation because minimization of the distance between structures is equivalent to minimization of their RMS coordinate deviation. During the subsequent optimization, the structures must not be allowed to rotate or to translate with respect to each other so that the calculation of the distances between structures using Equation (7.13) remains valid. A full discussion of this point and the way in which it is treated will be left until the next chapter.

The above algorithm was called the *self-avoiding walk* (SAW) method by Elber and co-workers and so it has been implemented as a function `SAWOptimize_SystemGeometry` with the following definition:

Function `SAWOptimize_SystemGeometry`

Generate a reaction path for a system using a self-avoiding walk optimization.

```
SAWOptimize_SystemGeometry (
  system,
  trajectory,
  gamma                 = 100.0,
  kappa                 = 2.0,
  rho                   = 5000.0,
  logfrequency          = 1,
  maximumiterations     = 50,
  rmsgradienttolerance = 0.001 )
```

Usage:

`system` is the system for which a reaction path is to be calculated.

`trajectory` is a trajectory object that contains the structures along the path that are to be optimized. The structures in the trajectory will change as the optimization proceeds.

`gamma`, `kappa`, `rho` are the parameters that enter into the expression for the SAW object function (Equation (7.12)).

Remarks: The remaining arguments have the same behaviour as those in `ConjugateGradientMinimize_SystemGeometry`. In fact, the implementation of the chain optimization uses exactly the same conjugate-gradient-minimization procedure.

Trajectories are required by many algorithms in this book, including the SAW algorithm and the reaction path methods of Section 7.7. Various types of trajectory exist, depending upon the data that are to be stored on them, but we shall mostly use the one that stores a system's coordinates and other geometrical information. This is implemented by the `SystemGeometryTrajectory` class which has the following definition:

Class `SystemGeometryTrajectory`

A class to handle trajectories that contain the geometrical data for a system.

Constructor

Construct an instance of `SystemGeometryTrajectory` from a filename.

```
new = SystemGeometryTrajectory (
                              filename,
                              system,
                              mode = None )
```

Usage:

`filename` is the name of the file that will contain the trajectory.

`system` is the instance of `System` that is to be associated with the trajectory.

`mode` specifies how the trajectory or, more specifically, the file containing the trajectory is to be accessed. This argument can take several possible values but we shall meet two, `"r"` and `"w"`. The former applies to trajectories that already exist as it means 'read'. It implies that a trajectory can be read, but should not be written to. By contrast, the latter means 'write' and implies that the trajectory can be written to, whether it already exists or not. If the argument `mode` is absent, the default is to choose the value `"r"` for trajectories that already exist and the value `"w"` for those that do not.

`new` is the new instance of `SystemGeometryTrajectory`. It will be empty.

Constructor `LinearlyInterpolate`

Construct an instance of `SystemGeometryTrajectory` by linearly interpolating between two end-point structures.

Usage:
```
new = SystemGeometryTrajectory.LinearlyInterpolate (
                    filename, system, nframes,
                    startframe, stopframe )
```

`filename` is the name of the file that will contain the trajectory.

`system` is the system for which the trajectory is to be written.

`nframes` is the number of *frames* or structures to be put on the trajectory. This number must be at least three as the starting and stopping structures are automatically stored on the trajectory, leaving one structure to be determined by linear interpolation.

`startframe` is the first frame to be put on the trajectory. It must be an instance of the class `Coordinates3`.

`stopframe` is the last frame to be put on the trajectory. It must be an instance of the class `Coordinates3`.

`new` is the new instance of `SystemGeometryTrajectory` which will contain `nframes` structures.

Remarks: The determination of the structures to be put on the trajectory is accomplished by first orienting the `stopframe` structure onto the `startframe` structure and then using linear interpolation to obtain the intermediate structures. Employing the notation of the beginning of this section, the coordinates of the Ith structure would be constructed as

$$\boldsymbol{R}_I = \boldsymbol{R}_0 + \frac{I}{M+1}\left(\boldsymbol{R}_{M+1} - \boldsymbol{R}_0\right) \tag{7.15}$$

After construction, all the structures, including `startframe` and `stopframe`, are written to the trajectory file.

This method is an example of a *class method* in Python because it is invoked using the name of the class instead of the name of the instance of the class. The method is classified as a constructor because it returns a new instance of `SystemGeometryTrajectory`.

Method `LinearlyExpand`

Expand a trajectory by inserting linearly interpolated structures between existing structures.

Usage: `trajectory.LinearlyExpand ( ninsert )`

`ninsert` is the number of structures to insert between each existing structure in the trajectory. If there are $M+2$ structures in the existing chain and `ninsert` takes the value n, there will be $M+2+n(M+1)$ structures after expansion. The coordinates of the inserted structures are calculated by linearly interpolating between the coordinates of adjacent pairs of structures using a formula similar to that of Equation (7.15).

`trajectory` is the instance of `SystemGeometryTrajectory` that is to be expanded.

7.10 Example 13

We test the self-avoiding walk algorithm by applying it to the calculation of the reaction path for the transition between the chair and the twist-boat conformations of cyclohexane. Although the algorithm was designed for use on large systems it is equally applicable, as we shall see, to small systems.

The program is:

```
1 """Example 13."""
2
3 from Definitions import *
4
5 # . Define the molecule and its QC model.
6 molecule = XYZFile_ToSystem ( \
          os.path.join ( xyzpath, "cyclohexane_chair.xyz" ) )
7 molecule.DefineQCModel ( QCModelMNDO ( "am1" ) )
```

```
8 molecule.Summary ( )
9
10 # . Assign the reactant and product coordinates.
11 reactants = Clone ( molecule.coordinates3 )
12 products  = XYZFile_ToCoordinates3 ( \
          os.path.join ( xyzpath, "cyclohexane_twistboat.xyz" ) )
13
14 # . Create a starting trajectory.
15 trajectory = SystemGeometryTrajectory.LinearlyInterpolate ( \
          os.path.join ( scratchpath, "cyclohexane_sawpath.trj" ), \
                                molecule, 11, reactants, products )
16
17 # . Optimization.
18 SAWOptimize_SystemGeometry ( molecule,                   \
                               trajectory,                 \
                               gamma             = 1000.0, \
                               maximumiterations = 200     )
```

Lines 6–8 define the molecule and its energy model.

Line 11 creates a clone of the `coordinates3` attribute of `molecule`. This contains the chair conformation of cyclohexane and will be taken as the starting structure of the SAW path.

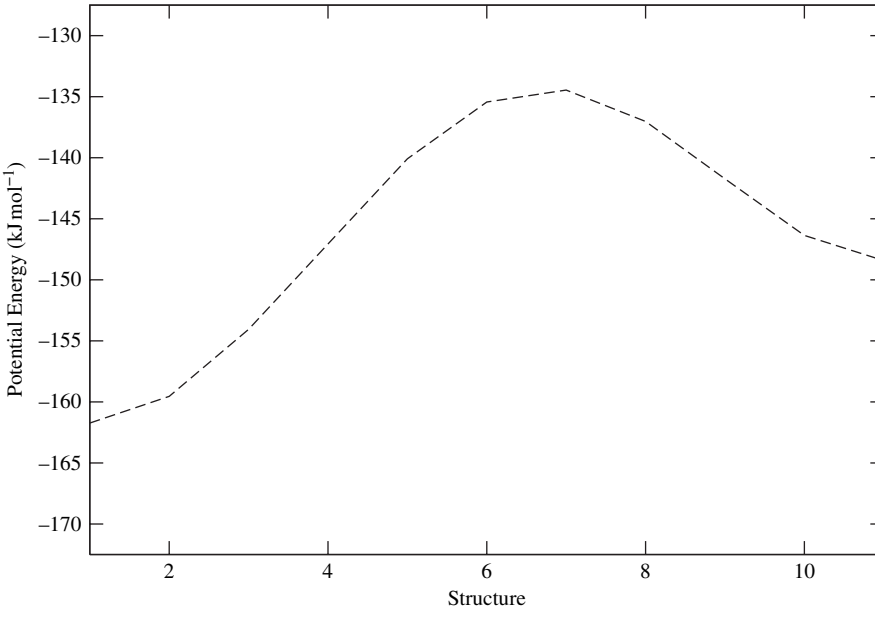


Fig. 7.3. A plot of the energies of the structures along the transition pathway between the chair and twist-boat forms of the cyclohexane molecule calculated using the SAW algorithm.

Line 12 reads in the coordinates of the twist-boat form of cyclohexane which will be used as the final structure of the SAW path.

Line 15 creates a trajectory containing 11 structures that are linearly interpolated between cyclohexane's chair and twist-boat conformations. The structures on the trajectory will serve as the starting point for the SAW optimization.

Line 18 generates a SAW path. The results are plotted in Figure 7.3, which shows the energies of the structures along the path. The seventh structure in the chain is within about 0.1 kJ mol^{-1} of the saddle-point structure optimized in Example 11. The convergence criteria for the optimization were satisfied after about 90 cycles, i.e. after 9×90 energy and derivative calculations, which is approximately twice as many as the number required by the path-tracing algorithm of Section 7.8.

Exercises

7.1 The blocked alanine molecule has a number of different minimum energy structures. Try to find some of these minima using the starting coordinate files that are provided. Use various minimization algorithms and, in all cases, a reasonably stringent criterion on the value of the RMS gradient tolerance (say a value of, at least, 10^{-3} kJ mol^{-1} $Å^{-1}$). Once some minima have been located, find the pathways that connect them, either by searching for saddle points and then using the reaction-path-tracing algorithm or by using the SAW method directly between two minima. Finally, construct a schematic potential energy surface for the molecule, illustrating the various minima, the saddle points and the reaction paths connecting them. How does the representation of the molecule's potential energy surface change with the energy model?

7.2 Another way of investigating the structure of bALA's potential energy surface is to optimize the molecule's structure with its ϕ and ψ dihedral angles fixed at particular values. Devise a program that does this by constraining ϕ and ψ during an optimization with a set of soft-constraint energy terms. Terms of the appropriate type were described in Section 5.6. Ensure that enough optimizations are performed so that the full range of both angles (-180 to 180°) is explored. Plot the resulting optimized energies as a function of the angles using a contour plot similar to that of Figure 7.1. How does this surface compare with the one that was obtained in the previous example?

7.3 Using the Baker algorithm, search for a saddle-point structure starting from the twist-boat form of cyclohexane. Is this saddle point the same as that obtained when starting from the chair form of cyclohexane? If not, to which reaction path does it correspond? Try to follow different modes, other than

the lowest one, to see the different structures that are produced. Also try to characterize more fully the boat and twist-boat conformers of the molecule and the reaction paths that lead between them. Are the boat and twist-boat forms both minima on the potential energy surface?

8

Normal mode analysis

8.1 Introduction

A characterization of stationary points on a system's potential energy surface provides structural and energetic information about its stable states and about possible pathways for transitions between them. But can we get more? In particular, can we use our knowledge about these local regions to obtain dynamical and thermodynamic information about the system? The most accurate way to do this is to use methods such as *molecular dynamics* and *Monte Carlo simulations*, which will be covered in later chapters. However, there is a useful intermediate technique, *normal mode analysis*, which we shall now discuss, which can give an idea about the dynamics of a system in the neighbourhood of a stationary point. We shall also see how this information, together with other data, can be used to estimate various thermodynamic quantities.

8.2 Calculation of the normal modes

We have already met some of the concepts that underlie normal mode analysis. The method relies on being able to write an expansion of the potential energy of a system about any configuration in terms of a Taylor series. If $\boldsymbol{R}_0$ is the coordinate vector of a reference structure and $\boldsymbol{R} = \boldsymbol{R}_0 + \boldsymbol{D}$ is the coordinate vector of a structure displaced by a small amount, $\boldsymbol{D}$, the Taylor series, up to terms of second order, is

$$\mathcal{V}(\boldsymbol{R}) = \mathcal{V}(\boldsymbol{R}_0) + \boldsymbol{G}^{\mathrm{T}}\boldsymbol{D} + \frac{1}{2}\boldsymbol{D}^{\mathrm{T}}\mathbf{H}\boldsymbol{D} + \cdots \tag{8.1}$$

where the first-derivative vector of the energy, $\boldsymbol{G}$, and the second-derivative matrix, $\mathbf{H}$, are determined at the reference structure, $\boldsymbol{R}_0$.

By definition, the gradient vector is zero at a stationary point. If the reference structure, $\boldsymbol{R}_0$, is taken to be a stationary point, Equation (8.1) can be simplified. Neglecting terms after second order gives

$$\Delta\mathcal{V}(\boldsymbol{D}) = \mathcal{V}(\boldsymbol{R}) - \mathcal{V}(\boldsymbol{R}_0) = \frac{1}{2}\boldsymbol{D}^{\mathrm{T}}\mathbf{H}\boldsymbol{D} \tag{8.2}$$

This equation says that the change in energy on displacement from a stationary point is a quadratic function of the displacement. This is called the *harmonic approximation* and is valid when the displacements involved are small and the terms of higher order in $\boldsymbol{D}$ can be ignored. If these terms are not small then *anharmonic* theories that include them are needed.

The important point about the expression for the change in energy in Equation (8.2) is that we can solve analytically for the dynamics of a system subject to such a potential. Both classical and quantum mechanical solutions are possible, although we shall restrict ourselves to a classical description. In this case, we can use Newton's laws to describe the motion of the atoms in the system. For each atom we have an equation of the form

$$\boldsymbol{f}_i = m_i\boldsymbol{a}_i \tag{8.3}$$

where $\boldsymbol{f}_i$ is the force on the atom, m_i is its mass and $\boldsymbol{a}_i$ is its acceleration. The force on an atom is defined as the negative of the first derivative of the potential energy with respect to the position vector of the atom:

$$\boldsymbol{f}_i = -\frac{\partial\mathcal{V}}{\partial\boldsymbol{r}_i} = -\boldsymbol{g}_i \tag{8.4}$$

whereas the acceleration is the second time derivative of the atom's position vector:

$$\boldsymbol{a}_i = \frac{\mathrm{d}^2\boldsymbol{r}_i}{\mathrm{d}t^2} = \ddot{\boldsymbol{r}}_i \tag{8.5}$$

Equation (8.3) can be rewritten for the full system as

$$\boldsymbol{F} = \mathbf{M}\boldsymbol{A} \tag{8.6}$$

where $\boldsymbol{F}$ and $\boldsymbol{A}$ are the $3N$-dimensional vectors of forces and accelerations for the atoms in the system and $\mathbf{M}$ is a $3N \times 3N$ diagonal matrix that contains the masses of the atoms and is of the form

$$\mathbf{M} = \begin{pmatrix} m_1 & 0 & 0 & 0 & \cdots & 0 \\ 0 & m_1 & 0 & 0 & \cdots & 0 \\ 0 & 0 & m_1 & 0 & \cdots & 0 \\ 0 & 0 & 0 & m_2 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & m_N \end{pmatrix} \tag{8.7}$$

Let us apply these equations to the case in which the potential energy of the system is given by Equation (8.2). First we note that the vector representing the configuration of the system, $\boldsymbol{R}$, can be replaced by the displacement vector, $\boldsymbol{D}$, because $\boldsymbol{R}_0$ is a reference structure and so is constant. The forces on the atoms in the system are thus obtained by differentiating the energy expression of Equation (8.2) with respect to the displacement for that atom, giving

$$\mathbf{M}\frac{\mathrm{d}^2\boldsymbol{D}}{\mathrm{d}t^2} = -\mathbf{H}\boldsymbol{D} \tag{8.8}$$

This equation is a second-order differential equation that can be solved exactly. The solutions are of the form

$$\boldsymbol{D} = \boldsymbol{\mathcal{A}}\cos(\omega t + \phi) \tag{8.9}$$

where the vector $\boldsymbol{\mathcal{A}}$ and the scalars ω and ϕ are to be determined. After substitution of this expression into Equation (8.8) and cancelling out of the cosine factors, the equation becomes

$$\mathbf{H}\boldsymbol{\mathcal{A}} = \omega^2\mathbf{M}\boldsymbol{\mathcal{A}} \tag{8.10}$$

It is normal to rewrite this equation by using mass-weighted Cartesian coordinates so that the dependence on the mass matrix, $\mathbf{M}$, is removed from the right-hand side. This can be done by introducing the inverse square root of the mass matrix, $\mathbf{M}^{-\frac{1}{2}}$, which is equal to the diagonal matrix of the inverse square roots of the atomic masses. With this matrix, after some rearrangement, Equation (8.10) becomes

$$(\mathbf{M}^{-\frac{1}{2}}\mathbf{H}\mathbf{M}^{-\frac{1}{2}})(\mathbf{M}^{\frac{1}{2}}\boldsymbol{\mathcal{A}}) = \omega^2(\mathbf{M}^{\frac{1}{2}}\boldsymbol{\mathcal{A}}) \tag{8.11}$$

or

$$\mathbf{H}'\boldsymbol{\mathcal{A}}' = \lambda\boldsymbol{\mathcal{A}}' \tag{8.12}$$

where

$$\mathbf{H}' = \mathbf{M}^{-\frac{1}{2}}\mathbf{H}\mathbf{M}^{-\frac{1}{2}} \tag{8.13}$$

$$\boldsymbol{\mathcal{A}}' = \mathbf{M}^{\frac{1}{2}}\boldsymbol{\mathcal{A}} \tag{8.14}$$

$$\lambda = \omega^2 \tag{8.15}$$

Equation (8.12) is a secular equation for the mass-weighted second-derivative matrix and can be solved to obtain the eigenvalues, λ, and the eigenvectors, $\boldsymbol{\mathcal{A}}'$. Because the matrix has dimensions $3N \times 3N$ there will be $3N$ different solutions, each of which represents an independent displacement that the system can make. These displacements are called the *normal modes*. Associated with each mode is a *frequency* that is the square root of the mode's eigenvalue. As stated before, the eigenvalues of a symmetric matrix are all real, so the frequencies associated with

each mode can be real (for a positive eigenvalue) or imaginary (for a negative eigenvalue). The motion produced by each mode with a positive eigenvalue is a simple oscillation at a characteristic frequency.

Because the normal modes are independent, the most general solution for the displacement vector, $\boldsymbol{D}$, is a linear combination of all the modes:

$$\boldsymbol{D} = \mathbf{M}^{-\frac{1}{2}} \sum_{k=1}^{3N} \alpha_k \boldsymbol{\mathcal{A}}'_k \cos(\omega_k t + \phi_k) \tag{8.16}$$

where the α_k are linear expansion coefficients and the ϕ_k are arbitrary phases. Both of these sets of parameters are determined by imposing additional constraints on the solution, such as specifying initial conditions for the motion of the atoms.

The expression for the potential energy can be rewritten in terms of the normal mode vectors by substituting the expression for the displacement, Equation (8.16), into Equation (8.2). Remembering the fact that the various eigenvectors are orthonormal gives

$$\Delta \mathcal{V} = \frac{1}{2} \sum_{k=1}^{3N} \alpha_k^2 \omega_k^2 \cos^2(\omega_k t + \phi_k) \tag{8.17}$$

The analysis above is important because it provides detailed information about the dynamics of a system around a stationary point. For minima, it is often true that the frequencies of motion, ω_k, can be obtained experimentally, most notably by using vibrational infrared spectroscopy, so a direct link between experiment and theory can be made.

The harmonic normal mode analysis discussed above is an important tool but there are other related techniques that are sometimes used, especially when analysing spectroscopic data. For example, in some cases the harmonic analysis outlined above is not sufficiently precise and so extra third- and fourth-order terms are included in Equation (8.2). These terms are obviously more onerous to calculate insofar as they involve the third and fourth derivatives of the energy with respect to the atomic positions. There are also other derivatives that it is possible to relate to experimental data. One of the simpler of these is the derivatives of the dipole moment whose squares are related to the *infrared intensities* of the vibrational motions of a molecule. The intensities for each vibrational mode are proportional to the square of the dipole derivative vectors projected onto the normal mode vectors. Writing the intensity of the kth mode as $\mathcal{I}_k$, we have

$$\boldsymbol{\mathcal{I}}_k \propto \sum_{\alpha=x,y,z} \left(\boldsymbol{\mathcal{A}}_k^{\mathrm{T}} \frac{\partial \mu_\alpha}{\partial \boldsymbol{R}} \right)^2 \tag{8.18}$$

Two functions have been provided in pDynamo for carrying out a normal mode analysis of a system. One performs the analysis and the other allows inspection of the calculated frequencies and normal mode vectors. Their definitions are:

Function `NormalModes_SystemGeometry`

Perform a normal mode analysis for a system.

Usage:
```
NormalModes_SystemGeometry ( system,
                             log    = logfile,
                             modify = None )
```

`system` is the system for which a normal mode analysis is to be performed.

`log` is the log file to which output is to occur. By default, some information about the normal mode analysis, including the calculated frequencies, is printed.

`modify` is an optional string argument that specifies various modifications that can be performed on the second-derivative matrix before the normal mode analysis is carried out. These options are explained in more detail at the end of the following section.

Remarks: This function first calculates the second-derivative matrix and then performs the normal mode analysis. The second-derivative matrix, the frequencies and the normal mode vectors are stored in `system` for subsequent analysis but they are kept only as long as the geometry of the system remains unchanged.

Function `NormalModesPrint_SystemGeometry`

Print the normal mode frequencies and vectors for a system.

Usage:
```
NormalModesPrint_SystemGeometry ( system,
                                  log       = logfile,
                                  modes     = None,
                                  selection = None )
```

`system` is the system for which a normal mode analysis has recently been performed.

`log` is the log file to which output is to occur.

`modes` specifies the indices of the modes that are to be printed. All modes are printed if the argument is absent.

`selection` gives the indices of the atoms for which the elements of the normal mode vectors are to be printed. The vector elements for all atoms are printed by default.

Remarks: The arguments `modes` and `selection` are usually instances of the class `Selection`.

8.3 Rotational and translational modes

We have already mentioned several times that the set of Cartesian coordinates is redundant. This is because, in general, only $3N-6$ parameters are needed to determine completely the geometry of a molecule with N atoms, but there are $3N$ Cartesian coordinates. The difference between the two representations is that the Cartesian set has six extra *degrees of freedom* that define the position and the orientation of the system in space.

For a system in vacuum, which category comprises all those we have looked at so far, the position and orientation of the system are unimportant because its potential energy and other properties will be *invariant* with respect to rotations and translations applied to the system as a whole. If, however, there is a preferred direction in space, due to the presence of an external field (such as an electric field) or some other environment, the system's absolute position and orientation in space will be important and its properties will no longer be invariant with respect to rotations and translations.

In those cases in which the absolute position and orientation of a system are not of interest, it is possible to remove the redundancy inherent to the Cartesian description by defining six constraint conditions that are functions of the coordinates and reduce the number of degrees of freedom available to the system to $3N-6$. There are several forms for the constraints but the most commonly used is the *Eckart conditions* which relate the coordinates of the atoms in a system, $\boldsymbol{r}_i$, to their values in a reference structure, $\boldsymbol{r}_i^0$. The Eckart condition on the translational motion is

$$\sum_{i=1}^{N} m_i(\boldsymbol{r}_i - \boldsymbol{r}_i^0) = \boldsymbol{0} \tag{8.19}$$

and that on the rotational motion is

$$\sum_{i=1}^{N} m_i \boldsymbol{r}_i \wedge (\boldsymbol{r}_i^0 - \boldsymbol{r}_i) = \sum_{i=1}^{N} m_i \boldsymbol{r}_i \wedge \boldsymbol{r}_i^0 = \boldsymbol{0} \tag{8.20}$$

These constraints are derived by considering the dynamics of the atoms in the system when they are displaced away from their positions in the reference structure. First a Hamiltonian that describes the dynamics is defined and then it is manipulated so that the motions due to the rotations and translations of the entire system are separated out from those due to its internal vibrations. It turns out that it is possible to separate off completely the translational motion if the condition

of Equation (8.19) is satisfied. The rotational and vibrational motions, however, cannot be completely separated but their coupling can be reduced by requiring that Equation (8.20) holds.

The derivation described above is a general one in that the overall rotational and translational motions of one structure with respect to another can be constrained if Equations (8.19) and (8.20) are obeyed. We came across such an example when discussing the self-avoiding walk algorithm for calculating reaction paths in Section 7.9. There we saw that it was important to prevent rotational and translational motions of the structures along the chain during the optimization process. The Eckart conditions provide a means of doing this. The idea is to select one of the initial structures along the path as the reference structure and then to ensure that all the other structures along the path obey a set of Eckart conditions with respect to the reference at each iteration in the subsequent optimization.

This procedure is straightforward to implement. The reason is that the constraints in Equations (8.19) and (8.20) are linear functions of the coordinates of the atoms in the non-reference structure (note that the coordinates of the reference structure are treated as constants) and so their derivatives with respect to the coordinates will be constants. This allows us to project out of the gradient vector for each structure at each iteration the contributions that lie in the subspace spanned by the constraint derivative vectors. If we denote the gradient vector for a structure, I, as $\boldsymbol{G}_{\mathrm{I}}$ and the orthogonalized constraint derivative vectors as $\boldsymbol{\Lambda}_{\alpha}$ ($\alpha = 1, 6$), the modified gradient vector $\boldsymbol{G}'_{\mathrm{I}}$, has the form

$$\boldsymbol{G}'_{\mathrm{I}} = \boldsymbol{G}_{\mathrm{I}} - \sum_{\alpha=1}^{6} \left(\frac{\boldsymbol{\Lambda}_{\alpha}^{\mathrm{T}} \boldsymbol{G}_{\mathrm{I}}}{\boldsymbol{\Lambda}_{\alpha}^{\mathrm{T}} \boldsymbol{\Lambda}_{\alpha}} \right) \boldsymbol{\Lambda}_{\alpha} \tag{8.21}$$

The modified gradients can now be used in the optimization process as normal but, because of the projection, they will not produce displacements that induce rotations and translations.

Let us now return to the vibrational problem. The invariance of the potential energy of a system with respect to rotations and translations is manifested in a normal mode analysis at a stationary point by the presence of six modes with zero frequencies. Three of these modes correspond to translations and three to rotations. It can be seen from Equation (8.17) that a displacement along the mode leaves the potential energy of the system unchanged if the frequency of a mode is zero.

The forms for the vectors of the rotational and translational modes can be derived from the Eckart conditions. To do this properly, though, requires that the formulae in Equations (8.19) and (8.20) be re-expressed in mass-weighted coordinates because these are the natural coordinates to use when dealing with eigenvectors of the mass-weighted Hessian matrix (see Equation (8.12)). If we

take the Eckart condition for translation along the x axis as an example, we can write the part of the condition that depends upon the non-reference structure as

$$\sum_{i=1}^{N} m_i x_i = \sum_{i=1}^{N} \sqrt{m_i} q_i^x \tag{8.22}$$

where q_i^x is the mass-weighted x coordinate for atom i defined in Equation (7.8). Equation (8.22) can be re-expressed in vector form as $\boldsymbol{\mathcal{T}}_x^{\mathrm{T}}\boldsymbol{Q}$, where $\boldsymbol{Q}$ is the $3N$-dimensional vector of mass-weighted Cartesian coordinates for the system and $\boldsymbol{\mathcal{T}}_x$ is the mode vector for translation in the x direction. The latter has the form

$$\boldsymbol{\mathcal{T}}_x \propto \mathbf{M}^{\frac{1}{2}} \begin{pmatrix} 1 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ \vdots \\ 1 \\ 0 \\ 0 \end{pmatrix} \tag{8.23}$$

That this vector is equivalent to a translation of the whole system may be confirmed by replacing $\boldsymbol{\mathcal{A}}'$ by the expression for $\boldsymbol{\mathcal{T}}_x$ in Equation (8.16).

The remaining vectors for translation and those for rotation are constructed in the same fashion. The translation vectors, $\boldsymbol{\mathcal{T}}_y$ and $\boldsymbol{\mathcal{T}}_z$, have the same form as $\boldsymbol{\mathcal{T}}_x$ except they have non-zero elements for their y and z components, respectively. The vector for rotation about the x axis, $\boldsymbol{\mathcal{R}}_x$, is

$$\boldsymbol{\mathcal{R}}_x \propto \mathbf{M}^{\frac{1}{2}} \begin{pmatrix} 0 \\ -z_1 \\ y_1 \\ 0 \\ -z_2 \\ y_2 \\ \vdots \\ 0 \\ -z_N \\ y_N \end{pmatrix} \tag{8.24}$$

The $\boldsymbol{\mathcal{R}}_y$ and $\boldsymbol{\mathcal{R}}_z$ modes are similar but with the arrangements $(z, 0, -x)$ and $(-y, x, 0)$ for each atom, respectively.

There are circumstances under which it is important to be able to manipulate the rotational and translational modes that arise from a Hessian matrix. One such case is when normal mode analyses are done at points that are not stationary. In these instances, the arguments of Section 8.2 are invalid because the gradient vector is not zero and there will no longer be six zero eigenvalues but only the three that correspond to translational motion. A second case is when performing optimizations with algorithms that employ second-derivative information, such as the Baker algorithm discussed in Section 7.5. It is obviously uninteresting to search along the modes that rotate or translate the entire system and so it is preferable to have some means of removing them from the optimization process.

We shall mention two methods that can be used to modify the Hessian matrix. The first technique, which is used in the Baker algorithm, changes the eigenvalues of the rotational and translational modes. To do this for the translation in the x direction, for example, requires adding a term to the mass-weighted Hessian matrix as follows:

$$\mathbf{H}' \rightarrow \mathbf{H}' + \lambda \boldsymbol{\mathcal{T}}_x \boldsymbol{\mathcal{T}}_x^{\mathrm{T}} \tag{8.25}$$

where λ is the new (usually large) value for the mode's eigenvalue.

In the second technique a projection matrix, $\mathbf{P}$, is defined with the form

$$\mathbf{P} = \mathbf{I} - \boldsymbol{\mathcal{R}}_x \boldsymbol{\mathcal{R}}_x^{\mathrm{T}} - \boldsymbol{\mathcal{R}}_y \boldsymbol{\mathcal{R}}_y^{\mathrm{T}} - \boldsymbol{\mathcal{R}}_z \boldsymbol{\mathcal{R}}_z^{\mathrm{T}} - \boldsymbol{\mathcal{T}}_x \boldsymbol{\mathcal{T}}_x^{\mathrm{T}} - \boldsymbol{\mathcal{T}}_y \boldsymbol{\mathcal{T}}_y^{\mathrm{T}} - \boldsymbol{\mathcal{T}}_z \boldsymbol{\mathcal{T}}_z^{\mathrm{T}} \tag{8.26}$$

and the Hessian matrix is modified as follows:

$$\mathbf{H}' \rightarrow \mathbf{P}\mathbf{H}'\mathbf{P} \tag{8.27}$$

In contrast to the first technique, this method guarantees that the six translational and rotational modes will have zero eigenvalues even if the normal mode analysis is to be carried out away from a stationary point.

Both these techniques have been implemented for use in a normal mode analysis. They are invoked by the optional argument `modify` of the function `NormalModes_SystemGeometry` described in the last section. If `modify` is set to the string, `"raise"`, the technique of Equation (8.25) is used and the eigenvalues of rotational and translational modes are increased to very large values. If it is `"project"`, Equation (8.27) is used instead. If the modify option is not used the second-derivative matrix is left unchanged.

8.4 Generating normal mode trajectories

The motion of the atoms in the system within the harmonic approximation is given by the solution for the displacement vector of Equation (8.16). To complete the determination of the solution the parameters α_k and ϕ_k remain to be specified.

The values of the phases, ϕ_k, are arbitrary and do not alter the general behaviour of the dynamics, but the values of the linear expansion coefficients, α_k, affect the relative importance of each mode in the displacement.

These parameters can be characterized more precisely by investigating the *total* energy of the system. Up to now, when we have discussed energies, we have been concerned exclusively with the value of the potential energy. The total energy of a system is the sum of the potential energy and its kinetic energy, which is the energy that arises due to the motions of the atoms in the system. In classical mechanics, the kinetic energy, $\mathcal{K}$, has the form

$$\mathcal{K} = \frac{1}{2}\sum_{i=1}^{N} m_i \boldsymbol{v}_i^2 \tag{8.28}$$

where $\boldsymbol{v}_i$ is the velocity of the atom and is equal to the first time derivative of the atom position, i.e.

$$\boldsymbol{v}_i = \frac{\mathrm{d}\boldsymbol{r}_i}{\mathrm{d}t} = \dot{\boldsymbol{r}}_i \tag{8.29}$$

Using these definitions, we can write down the total energy of the system within the harmonic approximation as a function of the displacement vector, $\boldsymbol{D}$. It is

$$E = \frac{1}{2}\dot{\boldsymbol{D}}^{\mathrm{T}}\mathbf{M}\dot{\boldsymbol{D}} + \frac{1}{2}\boldsymbol{D}^{\mathrm{T}}\mathbf{H}\boldsymbol{D} \tag{8.30}$$

The atoms' velocities can be obtained by differentiating Equation (8.16) with respect to time. Doing this, substituting into the above equation and simplifying produces the final result for the total energy:

$$E = \frac{1}{2}\sum_{k=1}^{3N} \alpha_k^2 \omega_k^2 \tag{8.31}$$

To identify the α_k we can make use of a result from statistical thermodynamics which says that, at equilibrium, the total energy of a system of independent harmonic oscillators will be equal to the number of oscillators multiplied by $k_{\mathrm{B}}T$, where k_{B} is Boltzmann's constant and T is the absolute temperature. To apply this result to Equation (8.31), we equate the energy contributed by each oscillator to $k_{\mathrm{B}}T$, giving $\alpha_k = \sqrt{2k_{\mathrm{B}}T}/\omega_k$. Thus, the size of the amplitude of each mode will be inversely proportional to its frequency. The lower the frequency, the larger the amplitude of the motion. It is important to note that the above analysis is invalid for modes with zero or imaginary frequencies, for which other arguments have to be used.

With this result we can now investigate the motion due to a particular mode at a given temperature. This is done with the following function:

Function `NormalModesTrajectory_SystemGeometry`

Calculate a trajectory corresponding to a normal mode of a system.

Usage:

```
NormalModesTrajectory_SystemGeometry (
                    system,
                    trajectory,
                    cycles      = 10,
                    frames      = 21,
                    mode        = 1,
                    temperature = 300.0)
```

`system` is an instance of `System`. It must recently have undergone a normal mode analysis.

`trajectory` is the trajectory object to which structures are to be written.

`cycles` specifies for how many complete vibrational cycles structures should be calculated.

`frames` gives the number of frames to generate for each cycle.

`mode` is the index of the mode for which the trajectory is to be generated. A simple check on the frequency of the mode is made. If the absolute value of the frequency is too small then the function returns without doing a calculation. If the frequency is imaginary, but with a magnitude that is large enough, a trajectory will be generated making the assumption that the mode has a *real* frequency. This is, of course, physically invalid but is done so that at least the displacements generated by the mode can be studied or visualized.

`temperature` is the temperature, in kelvins, at which the trajectory is to be calculated.

Remarks: The total number of coordinate sets stored on the trajectory will be the product of `cycles` and `frames`.

8.5 Example 14

The example in this section calculates the normal modes for the chair form of cyclohexane that was introduced in the last chapter. The program is

```
1 """Example 14."""
2
3 from Definitions import *
4
```

```
5 # . Define the molecule and its QC model.
6 molecule = XYZFile_ToSystem ( \
                os.path.join ( xyzpath, "cyclohexane_chair.xyz" ) )
7 molecule.DefineQCModel ( QCModelMNDO ( "am1" ) )
8 molecule.Summary ( )
9
10 # . Calculate the normal modes.
11 NormalModes_SystemGeometry ( molecule, modify = "project" )
12
13 # . Create an output trajectory.
14 trajectory = SystemGeometryTrajectory ( \
      os.path.join ( scratchpath, "cyclohexane_chair_mode7.trj" ), \
                                              molecule, mode = "w" )
15
16 # . Generate a trajectory for one of the modes.
17 NormalModesTrajectory_SystemGeometry ( molecule,           \
                                         trajectory,         \
                                         mode   =  7,        \
                                         cycles = 10,        \
                                         frames = 21,        \
                                         temperature = 600.0 )
```

Lines 6–8 define the molecule and its energy model.

Line 11 performs the normal mode analysis. The function calculates the second-derivative matrix and then carries out the analysis after it has projected out the motions corresponding to the rotational and translational degrees of freedom from the second-derivative matrix.

Lines 14–17 generate a trajectory for the mode with the lowest non-zero frequency at a temperature of 600 K. The trajectory contains 10 cycles, each of which has 21 frames, giving 210 frames in total.

The frequencies arising as a result of this calculation and for exactly equivalent ones on the twist-boat and saddle-point structures of cyclohexane are shown in Figure 8.1. Some of the modes for the chair conformer are *degenerate*, which means that there are two (or more in the general case) modes with exactly the same frequency. This normally arises because the structure has a particular symmetry.

The imaginary mode for the saddle-point structure is illustrated in Figure 8.2. The arrows on the atoms give the direction and the magnitude of the displacement induced by the mode. It can be seen that the mode induces displacements that lead to the chair and boat structures, respectively.

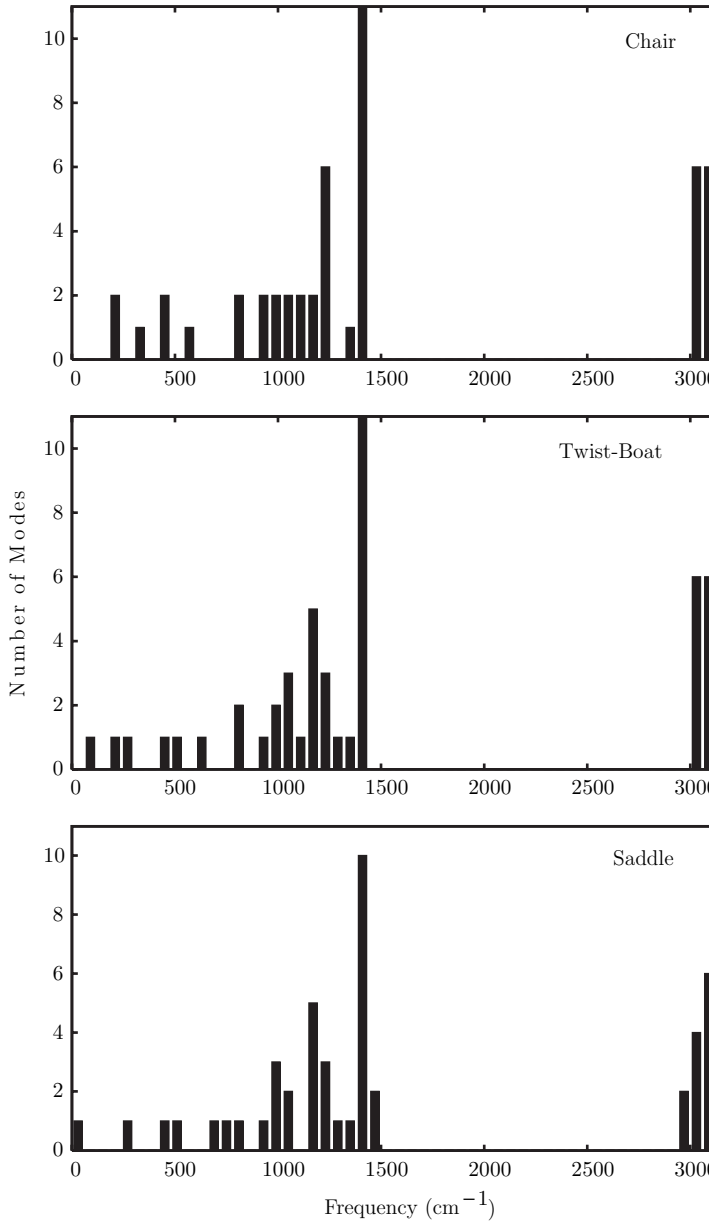


Fig. 8.1. Histograms of the normal mode frequencies for the chair, twist-boat and saddle-point structures of cyclohexane. The width of each histogram bin is 60 cm^{-1}. The imaginary frequency for the saddle-point structure, which is not shown, has a value of 183i cm^{-1}.

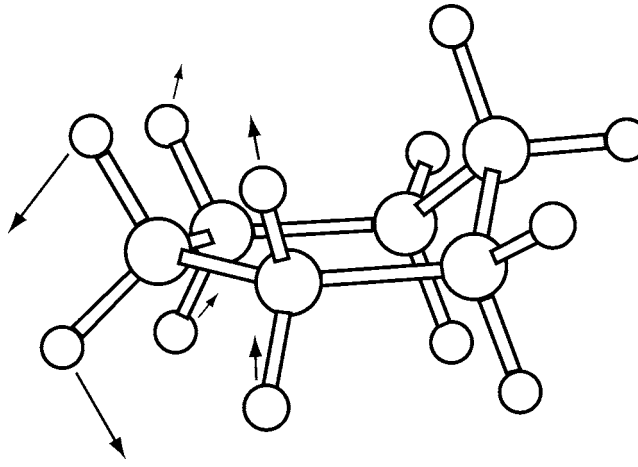


Fig. 8.2. Atomic displacements generated by the normal mode with the imaginary frequency for the saddle-point structure of cyclohexane. Only the largest displacements are shown for clarity.

8.6 Calculation of thermodynamic quantities

One of the principal aims of doing molecular simulations is to be able to compare the results of the calculations with those obtained from experiment. We have already met one way in which this can be done in this chapter – we can calculate the vibrational frequencies for a system. In this section we consider an additional approach for obtaining macroscopic properties from the atomic properties that we calculate when we do a simulation. To do this we use some well-known results from statistical thermodynamics, which is the branch of physics that links the microscopic and the macroscopic worlds. It is impractical in the space that we have here to provide any but the most cursory overview of this subject but a brief description is provided, whose principal purpose will be to define precisely the quantities that are calculated by the pDynamo function that is to be introduced later.

There are both quantum and classical formulations of statistical mechanics, but, in the quantum case, the theory is developed in terms of the various *quantum states* that the system can occupy. A fundamental quantity in the theory is the *partition function*, z, which can be regarded as a measure of the effective number of states that are accessible to the system. Its expression is

$$z = \sum_i \mathrm{e}^{-\epsilon_i/(k_\mathrm{B}T)} \tag{8.32}$$

where the sum runs over all the different states of the system, each of which has an energy, ϵ_i. To make progress in calculating the partition function from molecular quantities it is necessary to make some approximations. One of the most useful

is to hypothesize that the energy for a state can be written as a sum of energies, each of which comes from a different property of the system. In particular, it can be assumed that the state energy is the sum of electronic, nuclear, rotational, translational and vibrational energies, i.e.

$$\epsilon_i = \epsilon_i^{\text{elec}} + \epsilon_i^{\text{nucl}} + \epsilon_i^{\text{rot}} + \epsilon_i^{\text{trans}} + \epsilon_i^{\text{vib}} \tag{8.33}$$

where the superscripts refer to electronic, nuclear, rotational, translational and vibrational terms, respectively. This separation of the state energies into different terms and, in particular, the separation of the rotational and vibrational energies of the system is called the *rigid-rotor, harmonic oscillator approximation*.

Within the rigid-rotor, harmonic oscillator approximation the partition function for a molecule can be written as a product of the partition functions pertaining to the different types of energy. Thus, the total partition function is

$$z = z_{\text{elec}} z_{\text{nucl}} z_{\text{rot}} z_{\text{trans}} z_{\text{vib}} \tag{8.34}$$

This partition function is for a single molecule. The partition function, Z, for a collection of N_{mol} molecules will be equal to the product of the partition functions of the individual molecules, $z^{N_{\text{mol}}}$, if the molecules are *distinguishable* (as in a solid) or equal to $z^{N_{\text{mol}}}/N_{\text{mol}}!$ if the molecules are *indistinguishable* (as in a gas).

To calculate the electronic and nuclear partition functions the values of the electronic and nuclear energy levels are required. For most cases in which we will be interested we can assume that it is only the electronic and nuclear lowest-energy or *ground states* that are important, the remaining *excited states* being so much higher in energy that their contribution to the total partition function will be negligible. Thus, these terms can be considered to have values of 1.

The other three components of the total partition function can be readily calculated, either exactly, for the case of the vibrational function, or by approximating the sum over states by an integral for the rotational and translational functions. The results are

$$z_{\text{rot}} = \sqrt{\frac{\pi}{\sigma^2}\left(\frac{8\pi^2 \mathcal{I}_{\text{A}} k_{\text{B}} T}{h^2}\right)\left(\frac{8\pi^2 \mathcal{I}_{\text{B}} k_{\text{B}} T}{h^2}\right)\left(\frac{8\pi^2 \mathcal{I}_{\text{C}} k_{\text{B}} T}{h^2}\right)} \tag{8.35}$$

$$z_{\text{trans}} = \left(\frac{2\pi M_{\text{T}} k_{\text{B}} T}{h^2}\right)^{\frac{3}{2}} V \tag{8.36}$$

$$z_{\text{vib}} = \prod_{i=1}^{3N-6} \frac{\exp[-h\omega_i/(2k_{\text{B}}T)]}{1-\exp[-h\omega_i/(k_{\text{B}}T)]} \tag{8.37}$$

In these equations, $\mathcal{I}_{\text{A}}$, $\mathcal{I}_{\text{B}}$ and $\mathcal{I}_{\text{C}}$ are the moments of inertia of the system, M_{T} is its total mass, V is the volume it occupies and h is Planck's constant. σ is what is known as the *symmetry number* of the molecule. For a molecule without any

symmetry its value is 1. For molecules at minima that have symmetry, it is the number of different ways the molecule can be rotated into configurations that are indistinguishable from the original configuration.

The expressions for the rotational and vibrational partition functions need to be modified when the molecule is linear. The upper limit of the sum in Equation (8.37) becomes $3N-5$ and Equation (8.35) must be replaced by

$$z_{\text{rot}} = \frac{8\pi^2 \mathcal{I}_{\text{A}} k_{\text{B}} T}{h^2 \sigma} \tag{8.38}$$

since a linear molecule has only one unique moment of inertia, $\mathcal{I}_{\text{A}}$. For atomic systems the rotational and vibrational partition functions have a value of 1.

The partition function can be related to thermodynamic quantities using standard relations from statistical thermodynamics. For an ideal gas, which is the case in which we are interested, the equations are

$$U = RT^2 \left(\frac{\partial \ln z}{\partial T} \right)_V \tag{8.39}$$

$$S = R \ln z + RT \left(\frac{\partial \ln z}{\partial T} \right)_V - R \ln N_{\text{av}} + R \tag{8.40}$$

$$A = U - TS \tag{8.41}$$

$$H = U + PV \tag{8.42}$$

$$G = H - TS \tag{8.43}$$

$$C_V = \left(\frac{\partial U}{\partial T} \right)_V \tag{8.44}$$

$$C_P = \left(\frac{\partial H}{\partial T} \right)_P \tag{8.45}$$

$$PV = RT \tag{8.46}$$

where U is the *internal energy*, S is the *entropy*, A is the *Helmholtz free energy*, H is the *enthalpy*, G is the *Gibbs free energy* and C_V and C_P are the *heat capacities at constant volume and pressure*, respectively. P, T and V denote the pressure, temperature and volume of the system. R is the molar gas constant and is equal to the product of Avogadro's constant, N_{av}, and Boltzmann's constant, k_{B}. The subscripts on the brackets surrounding the partial derivatives indicate that these quantities are assumed to be constant for the differentiation.

The values of these quantities can be determined experimentally in many cases and it is found that the rigid-rotor, harmonic oscillator approximation often gives values that are in good agreement with experiment. It is also possible to combine the values for several structures to calculate *equilibrium constants* and *rate*

constants. For example, if we consider the equilibrium between two states of a system, A and B,

$$\mathrm{A} \overset{K_{\mathrm{eq}}}{\rightleftharpoons} \mathrm{B} \tag{8.47}$$

the equilibrium constant (at constant pressure), K_{eq}, can be written as

$$K_{\mathrm{eq}} = \exp\left[-\left(\frac{G_{\mathrm{B}} - G_{\mathrm{A}}}{RT}\right)\right] \tag{8.48}$$

where G_{A} and G_{B} are the Gibbs free energies of states A and B, respectively.

To calculate rate constants for a reaction it is possible to use *transition state theory*. In its simplest version, this theory assumes that there is an *activated complex* or *transition state structure* that is in thermodynamic equilibrium with the reactant molecules and is transformed into products. If $\mathrm{A}^{\ddagger}$ is this complex, the reaction can be written as

$$\mathrm{A} \overset{K^{\ddagger}}{\rightleftharpoons} \mathrm{A}^{\ddagger} \overset{k'}{\rightarrow} \text{products} \tag{8.49}$$

The rate for this process, k_{f}, is then written as

$$k_{\mathrm{f}} = k' K^{\ddagger} = \frac{k_{\mathrm{B}} T}{h} \exp\left[-\left(\frac{G_{\mathrm{A}^{\ddagger}} - G_{\mathrm{A}}}{RT}\right)\right] \tag{8.50}$$

where the factor $k_{\mathrm{B}}T/h$ gives the rate at which the activated complex goes to products and the exponential factor is the equilibrium constant for the equilibrium between the activated complex and reactants. It is normal to equate the activated complex structure to the saddle-point structure on the reaction path between reactants and products. In these cases, the imaginary frequency of the saddle point is omitted from the calculation of the vibrational partition function of Equation (8.37).

Note that, for the determination both of the equilibrium and of the rate constants in Equations (8.48) and (8.50), the free energy values are calculated with respect to the same reference value on the potential energy surface. In other words, the differences between the free energy values include the difference in potential energy between the two structures in addition to the terms depending directly upon the partition function.

To calculate the thermodynamic quantities defined in Equations (8.39)–(8.46), a function has been provided, `ThermodynamicsRRHO_SystemGeometry`, with the definition:

Function `ThermodynamicsRRHO_SystemGeometry`

Calculate some thermodynamic quantities for a system within the rigid-rotor, harmonic oscillator approximation.

Usage:

```
results = ThermodynamicsRRHO_SystemGeometry (
                    system,
                    pressure       = 1.0,
                    symmetrynumber = 1,
                    temperature    = 300.0 )
```

`system` is an instance of `System`.

`pressure` is the pressure, in atmospheres. It is used to determine the volume of the system from Equation (8.46), the value of which is needed for the calculation of the translational partition function in Equation (8.36).

`symmetrynumber` is the symmetry number of the system.

`temperature` is the temperature, in kelvins.

`results` is a dictionary that holds the results of the calculation. The dictionary keys are the names of the quantities calculated and include `"Enthalpy"`, `"Entropy"`, `"Gibbs Free Energy"`, `"Helmholtz Free Energy"` and `"Internal Energy"`. The units for A, G, H and U are kJ mol^{-1} and those for C_P, C_V and S are kJ mol^{-1} K^{-1}.

Remarks: The function requires the vibrational frequencies and so `system` should have undergone a recent normal mode analysis. It should also be noted that, for this function to return sensible values, the system should be at a stationary point because the calculation of the vibrational partition function leaves out the six (five for a linear system) frequencies with the lowest absolute magnitudes as well as any imaginary frequencies.

8.7 Example 15

We can use the function presented in the last section to calculate the thermodynamic quantities for the chair and twist-boat forms of cyclohexane at a series of temperatures and then to determine the equilibrium constant for the process

$$\text{Chair} \rightleftharpoons \text{Twist-boat} \tag{8.51}$$

The program is:

```
1 """Example 15."""
2
3 from Definitions import *
4
```

```
# . Methods.
def FreeEnergies ( filename, temperatures, symmetrynumber = 1 ):
    """Calculate the potential energy for a system and its
       Gibbs free energies at several temperatures."""

    # . Define the molecule and its QC model.
    molecule = XYZFile_ToSystem ( \
                   os.path.join ( xyzpath, filename + ".xyz" ) )
    molecule.DefineQCModel ( QCModelMNDO ( "am1" ) )
    molecule.Summary ( )

    # . Calculate the energy and normal modes.
    e = molecule.Energy ( )
    NormalModes_SystemGeometry ( molecule, modify = "project" )

    # . Loop over the temperatures.
    g = []
    for T in temperatures:
        tdics = ThermodynamicsRRHO_SystemGeometry ( molecule, \
                                pressure       = 1.0,           \
                                symmetrynumber = symmetrynumber, \
                                temperature    = T              )
        g.append ( tdics["Gibbs Free Energy"] )

    # . Return the energies.
    return ( e, g )

# . Create a sequence of temperatures.
temperatures = [ 100.0 * i for i in range ( 1, 11 ) ]

# . Get the energies for the boat and chair structures.
( eb, gboat  ) = FreeEnergies ( "cyclohexane_twistboat", \
                                temperatures, symmetrynumber = 4 )
( ec, gchair ) = FreeEnergies ( "cyclohexane_chair",     \
                                temperatures, symmetrynumber = 6 )
deltae = ( eb - ec )

# . Output the equilibrium constants.
table = logfile.GetTable ( columns = [ 25, 25 ] )
table.Start   ( )
table.Title   ( "Equilibrium Constants (Chair -> Twist Boat)" )
table.Heading ( "Temperature" )
table.Heading ( "Log K"       )
for ( T, gc, gb ) in zip ( temperatures, gchair, gboat ):
```

```
43     RT     = ( CONSTANT_MOLAR_GAS * T ) / 1000.0
44     log10K = math.log10 ( math.e ) * ( - ( gb-gc+deltae ) / RT )
45     table.Entry ( "%.4f" % ( T,      ) )
46     table.Entry ( "%.6g" % ( log10K, ) )
47 table.Stop ( )
```

Line 6 starts the definition of a function whose purpose is to calculate the potential energy of a system and its Gibbs free energies at a series of temperatures. The arguments to the function are `filename` which gives the name of the XYZ file used for the definition of the system, `temperatures` which is a list of temperature values and `symmetrynumber` which specifies the symmetry number to be employed for the calculation of the free energy.

Lines 11–13 define the system and its energy model.

Lines 16–17 calculate the potential energy and perform a normal mode analysis for the system. The potential energy is stored as the variable `e`.

Lines 20–23 generate a list of Gibbs free energies, `g`, one for each of the temperatures in the argument `temperatures`. The loop is rather wasteful as all thermodynamical quantities are evaluated even though only the Gibbs free energy is required.

Line 26 returns the calculated quantities, `e` and `g`.

Line 29 starts the main body of the program by creating a list of ten temperature values in the range 100–1000 K in 100 K increments.

Lines 32–33 calculate the potential and Gibbs free energies for the chair and twist-boat forms of cyclohexane using the function defined on *lines 6–26*. The chair form of cyclohexane has point group symmetry, D_{3d}, so its symmetry number is 6 whereas the symmetry number of the twist-boat structure is 4 because it has D_2 symmetry.

Line 34 determines the difference in the potential energies between the two conformations of cyclohexane.

Lines 37–41 set up the table for output of the equilibrium constants as a function of temperature.

Line 42 starts a loop over the lists of temperature and free energy values. The Python built-in function `zip` is employed which ensures that, at each iteration, a temperature and its corresponding chair and twist-boat free energies are extracted.

Line 43 calculates the product of the temperature, T, and the molar gas constant, R, in $\mathrm{kJ\,mol^{-1}}$. Many constants are predefined in the pDynamo library. The gas constant is stored as `CONSTANT_MOLAR_GAS` and has units of $\mathrm{J\,mol^{-1}\,K^{-1}}$.

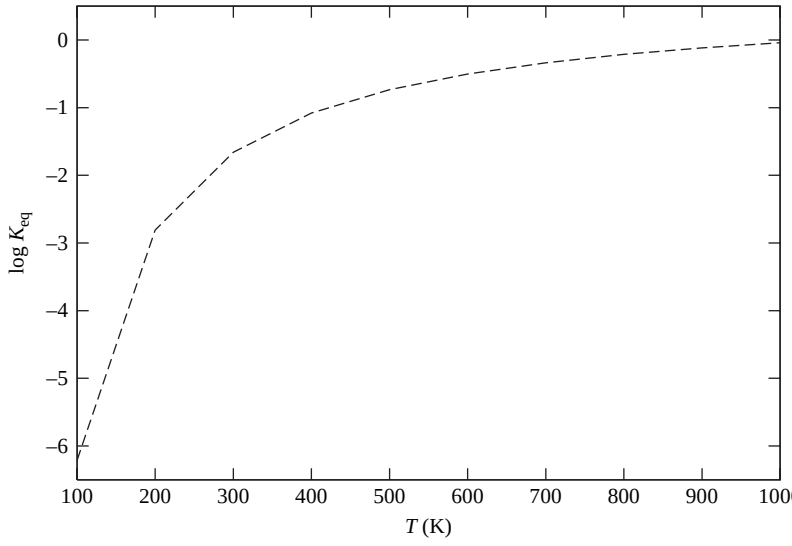


Fig. 8.3. A plot of the equilibrium constant for the equilibrium between the chair and twist-boat forms of cyclohexane as a function of temperature.

Line 44 calculates the logarithm to base 10 of the equilibrium constant. The equilibrium constant itself is defined as

$$K_{eq} = \exp\left[-\left(\frac{G_{\text{twist-boat}} - G_{\text{chair}}}{RT}\right)\right] \tag{8.52}$$

noting that the free energy difference must be corrected for the difference, `deltae`, in the potential energy between the two structures. Two items from the module `math` are employed on this line – `math.log10` which calculates the logarithm and `math.e` which is the mathematical constant e.

The results of the calculation are displayed in Figure 8.3. As expected, the equilibrium constant for the two species increases with the temperature. It should be remarked that the results of these calculations are somewhat fictitious and have been presented for illustrative purposes only. In particular, note that this molecule liquefies at about 350 K and so the gas-phase equilibrium constants below this will be unattainable experimentally.

Exercises

8.1 Calculate the rate constants for the interconversion of the chair and twist-boat forms of cyclohexane using the transition state theory expression given in Equation (8.50). The program should be similar to that of Section 8.7 except

that a third normal mode calculation needs to be done for the saddle-point structure. Calculate both the forwards and the reverse rate constants.

8.2 The results of a normal mode analysis depend upon the masses of the atoms. Thus, if different isotopes are used for atoms, the frequencies of vibration will shift. These *isotope effects* can be an important analytic tool for the investigation of the mechanisms of reactions. Choose a molecule, such as cyclohexane, and investigate the effect of using different isotopes (exchanging H for D, for example) on the vibrational frequencies of each structure and the values of the equilibrium and rate constants.

9

Molecular dynamics simulations I

9.1 Introduction

We saw in Chapter 7 how it was possible to explore relatively small parts of a potential energy surface and in Chapter 8 how to use some of this information to obtain approximate dynamical and thermodynamic information about a system. These methods, though, are local – they consider only a limited portion of the potential energy surface and the dynamics of the system within it. It is possible to go beyond these 'static' approximations to study the dynamics of a system directly. Some of these techniques will be introduced in the present chapter.

9.2 Molecular dynamics

As we discussed in Chapter 4, complete knowledge of the behaviour of a system can be obtained, in principle, by solving its time-dependent Schrödinger equation (Equation (4.1)), which governs the dynamics of all the particles in the system, both electrons and nuclei. To progress in the solution of this equation we introduced the Born–Oppenheimer approximation, which allows the electronic and the nuclear problems to be treated separately. This separation leads to the concept of a potential energy surface, which is the effective potential that the nuclei experience once the electronic problem has been solved. In principle, it is possible to study the dynamics of the nuclei under the influence of the effective electronic potential using an equivalent equation to Equation (4.1) but for the nuclei only. This can be done for systems consisting of a very small number of particles but proves impractical otherwise.

Fortunately, whereas it is difficult or impossible to study the dynamics of the system quantum mechanically, a classical dynamical study is relatively straightforward and provides much useful information. Although they constitute an approximation to the real dynamics, classical dynamical simulation techniques are believed to provide accurate descriptions in many cases. They do, however, omit

a number of effects that can be important in certain circumstances. For example, they may fail in the treatment of the dynamics of light particles (especially hydrogen), for which quantum mechanical *tunneling effects* can be important, and they do not include *zero-point motion*, which is the vibrational motion that all quantum mechanical systems undergo even at the absolute zero of temperature (0 K).

There are various formulations for the classical dynamical analysis of a system but we adopt the description that starts off by defining the *classical Hamiltonian*, $\mathcal{H}$, for the system. This is the sum of kinetic and potential energy terms and can be written as

$$\mathcal{H}(\boldsymbol{p}_i, \boldsymbol{r}_i) = \sum_{i=1}^{N} \frac{1}{2m_i} \boldsymbol{p}_i^2 + \mathcal{V}(\boldsymbol{r}_i) \tag{9.1}$$

where $\boldsymbol{p}_i$ is the momentum of particle i and $\mathcal{V}$ is the potential (in our case obtained by QC, MM or QC/MM techniques). The Hamiltonian is a function of $6N$ independent variables, the $3N$ particle momenta and the $3N$ particle positions.

It is possible to derive equations of motion for the variables from Equation (9.1) using Hamilton's equations, which are

$$\begin{aligned} \dot{\boldsymbol{p}}_i &= -\frac{\partial \mathcal{H}}{\partial \boldsymbol{r}_i} \\ &= -\frac{\partial \mathcal{V}}{\partial \boldsymbol{r}_i} \\ &= \boldsymbol{f}_i \end{aligned} \tag{9.2}$$

$$\begin{aligned} \dot{\boldsymbol{r}}_i &= \frac{\partial \mathcal{H}}{\partial \boldsymbol{p}_i} \\ &= \frac{\boldsymbol{p}_i}{m_i} \end{aligned} \tag{9.3}$$

These equations are first-order differential equations. A second-order differential equation can be obtained by noting from Equation (9.3) that the momentum of a particle is equal to the product of the mass of the particle and its velocity (the time derivative of its position). Substitution of this expression, Equation (9.3), into Equation (9.2) gives Newton's equation of motion for the particle:

$$m_i \ddot{\boldsymbol{r}}_i = \boldsymbol{f}_i \tag{9.4}$$

To study the dynamics of a system, or to perform a molecular dynamics simulation, the equations of motion (either Equations (9.2) and (9.3) or Equation (9.4)) must be solved for each particle. This is an example of a well-studied mathematical problem, that of the integration of a set of ordinary differential equations, for which a large variety of algorithms exist. The choice of algorithm depends

upon the exact nature of the equations of motion and the accuracy of the solution required, but the great majority of algorithms are *initial value algorithms*, which means that they start off with initial values for the particles' positions and momenta (or velocities) and integrate the equations for a specific length of time.

For very-high-accuracy solutions of the equations of motion it is usually advantageous to solve the system of first-order differential equations for each particle (Equations (9.2) and (9.3)). Suitable algorithms are the *predictor–corrector integrators* and the method due to R. Burlisch and J. Stoer. Owing to the special form of Newton's equation, it turns out that it is more efficient for normal use to solve the set of second-order differential equations directly (Equation (9.4)). Methods for solving these equations, such as Stoermer's rule, have been known for a long time, but they are generally called *Verlet methods* after L. Verlet, who was one of the first people to apply them to molecular simulations.

The standard Verlet method is easy to derive. If, at a time t, the positions of the atoms in the system are $\boldsymbol{R}(t)$, then the positions of the atoms at a time $t+\Delta$ can be obtained from a Taylor expansion in terms of the *timestep*, Δ, and the positions and their derivatives at time t. The expansion is

$$\boldsymbol{R}(t+\Delta) = \boldsymbol{R}(t) + \Delta\dot{\boldsymbol{R}}(t) + \frac{\Delta^2}{2}\ddot{\boldsymbol{R}}(t) + O(\Delta^3) \tag{9.5}$$

Similarly, the positions at a time $t-\Delta$ are obtained from the expansion

$$\boldsymbol{R}(t-\Delta) = \boldsymbol{R}(t) - \Delta\dot{\boldsymbol{R}}(t) + \frac{\Delta^2}{2}\ddot{\boldsymbol{R}}(t) - O(\Delta^3) \tag{9.6}$$

Adding these equations and rearranging gives an expression for the positions of the particles at $t+\Delta$ in terms of the positions and forces on the particles at earlier times:

$$\begin{aligned}\boldsymbol{R}(t+\Delta) &= 2\boldsymbol{R}(t) - \boldsymbol{R}(t-\Delta) + \Delta^2\ddot{\boldsymbol{R}}(t) + O(\Delta^4)\\ &\simeq 2\boldsymbol{R}(t) - \boldsymbol{R}(t-\Delta) + \Delta^2\mathbf{M}^{-1}\boldsymbol{F}(t)\end{aligned} \tag{9.7}$$

where on going from the first to the second equation we have made use of Newton's equations for the particles (Equation (9.4)).

Subtracting Equation (9.6) from Equation (9.5) gives an equation for the velocities of the particles, $\boldsymbol{V}$, at the current time, t:

$$\begin{aligned}\boldsymbol{V}(t) &= \dot{\boldsymbol{R}}(t)\\ &\simeq \frac{1}{2\Delta}\left(\boldsymbol{R}(t+\Delta) - \boldsymbol{R}(t-\Delta)\right)\end{aligned} \tag{9.8}$$

Equations (9.7) and (9.8) are sufficient to integrate the equations of motion but they are slightly inconvenient. This is because the velocities at time t are

available only once the positions at time $t+\Delta$ have been calculated, which means that, at the start of the simulation, i.e. when $t=0$, it is necessary to use another formula. A slight modification of these equations produces an algorithm called the *velocity Verlet method* that avoids these problems and can be shown to produce results that are equivalent to those of the standard Verlet method. The equations are

$$\boldsymbol{R}(t+\Delta) = \boldsymbol{R}(t) + \Delta \boldsymbol{V}(t) + \frac{\Delta^2}{2}\mathbf{M}^{-1}\boldsymbol{F}(t) \tag{9.9}$$

$$\boldsymbol{V}(t+\Delta) = \boldsymbol{V}(t) + \frac{\Delta}{2}\mathbf{M}^{-1}(\boldsymbol{F}(t) + \boldsymbol{F}(t+\Delta)) \tag{9.10}$$

It should be noted that a wide variety of Verlet-type algorithms are in use, including the so-called *leapfrog* methods, which calculate the positions on the full step (i.e. at $t+\Delta$) and the velocities on the half step (i.e. at $t+\Delta/2$). We shall use the velocity Verlet algorithm in this chapter because it will be sufficient for our needs, but in other situations one of the other algorithms may be more appropriate.

It is important to be able to check the accuracy of any integration algorithm. Some of the more useful measures of the precision of a simulation are the *conservation conditions* on certain properties of the system, notably the momentum, angular momentum and the energy. The total momentum, $\mathcal{M}$, and angular momentum, $\mathcal{L}$, of a system are defined as

$$\mathcal{M} = \sum_{i=1}^{N} \boldsymbol{p}_i \tag{9.11}$$

$$\mathcal{L} = \sum_{i=1}^{N} \boldsymbol{r}_i \wedge \boldsymbol{p}_i \tag{9.12}$$

It is straightforward to show that the total energy of a system described by a classical Hamiltonian that is independent of time, such as the one given in Equation (9.1), is conserved or constant. This can be done by differentiating Equation (9.1) with respect to time and then substituting the expressions for Hamilton's equations of motion (Equations (9.2) and (9.3)) to show that the total derivative is zero. This means that throughout the simulation the total energy should be the same as the energy at the beginning. Of course, this will not exactly be so, but the size of the deviations in the total energy or the drift away from its initial value will give a measure of the precision of the integration algorithm. The conservation conditions for the momentum and angular momentum can be derived in a similar way by differentiating Equations (9.11) and (9.12). On doing this it can be seen that there will be conservation of momentum and angular momentum

if there is no nett force, $\sum_{i=1}^{N} \boldsymbol{f}_i$, or torque, $\sum_{i=1}^{N} \boldsymbol{r}_i \wedge \boldsymbol{f}_i$, on the system, respectively. This will be the case for a system in vacuum but it will not necessarily hold if the system experiences some exterior influence due to an external field, for example.

Having defined the algorithm for the integration of the equations of motion it is straightforward to devise a scheme to perform a molecular dynamics simulation. The one that we shall use is as follows:

(i) Define the composition of the system, including the number and type of atoms, their masses and their interaction potential.
(ii) Assign initial values ($t = 0$) to the particles' positions, $\boldsymbol{R}$, and velocities, $\boldsymbol{V}$.
(iii) Define the timestep, Δ, for the integration and the number of integration steps (i.e. the duration of the simulation).
(iv) Perform the simulation. Initially the positions and the velocities of the particles are known, but the forces at $t = 0$ must be calculated. At subsequent integration steps do the following:
  (a) Calculate the positions at the current step $t + \Delta$ using Equation (9.9).
  (b) Calculate the forces at $t + \Delta$.
  (c) Calculate the velocities at $t + \Delta$ using Equation (9.10).
  (d) Do any analysis that is required with the positions and velocities at the current step $t + \Delta$. This can include the calculation and the printing of intermediate results or the storage of the position and velocity data on an external file to create a molecular dynamics trajectory.
  (e) Increment the time by the timestep.
(v) Analyse the results.

There are a number of issues raised by this scheme that need elaboration. The first is how to choose the value of the timestep. In general, we would like a timestep that is as large as possible so that the simulation is as long as possible, but not so large that the accuracy of the integration procedure is jeopardized. The factor that normally limits the upper size of the timestep is the nature of the highest frequency motions in the system. In organic molecules, these are typically motions involving hydrogen (because it is light) and include, for example, the stretching associated with the vibrations of carbon–hydrogen bonds. To integrate accurately over these motions the timestep needs to be small with respect to the period of the vibration. So, for example, if the highest frequency vibrations have values of around 3000 cm^{-1}, their characteristic timescales will be of the order of a few femtoseconds (1 fs = 10^{-15} s), which means that the timestep will need to be less than this. In practice, values of about 1 fs are found to be the largest reasonable for systems possessing these types of motion when Verlet algorithms are employed.

The second point that needs discussion is how to determine the initial values of the velocities for the atoms (we shall assume that a starting set of coordi-

nates is available). One of the most convenient ways is to choose the velocities so that the system will have a particular temperature at the start of the simulation. From statistical thermodynamics it is known that the velocities of the atoms in a classical system are distributed according to the *Maxwell–Boltzmann distribution*. This says that, if the temperature of the system is T, the probability of each component of the velocity of the ith atom having a value between v and $v + \mathrm{d}v$ is

$$f(v)\,\mathrm{d}v = \sqrt{\frac{m_i}{2\pi k_\mathrm{B} T}} \exp\left(-\frac{m_i}{2k_\mathrm{B}T} v^2\right) \mathrm{d}v \tag{9.13}$$

The values of the velocities of the atoms can be assigned by treating them as independent *Gaussian random variables* drawn from the distribution defined in Equation (9.13) which has a mean value of zero and a standard deviation of $\sqrt{k_\mathrm{B}T/m_i}$. If this is done the temperature of the system will not be exactly T because the values are assigned randomly, but it is easy to scale the velocities obtained in this way uniformly so that the *instantaneous temperature* of the system does correspond to the value desired. There is a well-known result from statistical thermodynamics that relates the average of the kinetic energy of the system to the temperature. It is

$$T = \frac{2}{N_\mathrm{df} k_\mathrm{B}} \langle \mathcal{K} \rangle \tag{9.14}$$

The average in this equation is a thermodynamic, *ensemble* average that must be done over all the configurations that are accessible to the system. An instantaneous temperature, $\mathcal{T}$, can be defined using the same equation but by removing the average. Thus

$$\mathcal{T} = \frac{2}{N_\mathrm{df} k_\mathrm{B}} \mathcal{K} \tag{9.15}$$

This expression allows the instantaneous temperature to be defined exactly once the initial velocity values have been chosen.

The quantity N_df in Equations (9.14) and (9.15) is the number of degrees of freedom accessible to the system. As we saw in Section 8.3, the number of internal degrees of freedom that a molecule has is $3N - 6$ ($3N - 5$ if the molecule is linear). Classically, as can be deduced from Equation (9.14), each degree of freedom contributes on average $k_\mathrm{B}T/2$ to the kinetic energy. The remaining degrees of freedom, six for non-linear and five for linear molecules, correspond to the overall rotational and translational degrees of freedom for the system. These will contribute to the kinetic energy – the translational motion giving an overall momentum to the system and the rotational motion an angular momentum. When assigning velocities, it is possible not only to scale the velocities such that the correct instantaneous temperature is obtained, but also to ensure that the overall

translational and rotational motions are removed. If this is done, the number of degrees of freedom used in the equation for the temperature will be $3N-6$ (or $3N-5$) whereas, if the overall translational and rotational motions are left in, the number of degrees of freedom will be $3N$. In any case, for large systems the difference between the two will normally be small.

During a simulation it is often useful to be able to control the temperature of the system. This is particularly so during the initial stages of a simulation study and can be done straightforwardly by scaling the velocities to obtain the required instantaneous temperature after they have been calculated in step (iv)(c) of the molecular dynamics scheme outlined above. This scaling procedure is simple, although not entirely rigorous, and more sophisticated temperature control schemes that perturb the dynamics of the system less will be discussed in a later chapter.

A function has been provided to perform molecular dynamics simulations using the velocity Verlet algorithm. It has the following definition:

Function `VelocityVerletDynamics_SystemGeometry`

Perform a molecular dynamics simulation with the velocity Verlet algorithm.

Usage:

```
VelocityVerletDynamics_SystemGeometry (
        system,
        logfrequency               = 1,
        rng                        = None,
        steps                      = 1000,
        temperature                = None,
        temperaturescalefrequency  = 0,
        temperaturescaleoption     = None,
        temperaturestart           = None,
        temperaturestop            = None,
        timestep                   = 0.001,
        trajectories               = None )
```

`system` is the system whose dynamics is to be simulated.

`logfrequency` is the frequency at which data concerning the simulation are to be printed. The default is to print at every step.

`rng` defines an instance of a *random number generator* which is employed when assigning velocities from the Maxwell–Boltzmann distribution to the atoms. This argument is redundant if no assignment is needed. The advantage of supplying one's own generator is that it can be prepared in a given *state* which means that the same velocities and,

hence, dynamics trajectories will be produced if the program is run more than once. If `rng` is not defined, and velocities are to be assigned, the function creates an instance of a generator in an arbitrary state.

`steps` is the number of simulation steps.

`temperature` gives the temperature at which the simulation is to be run. This argument is required both for the initial assignment of velocities and to specify the target temperature if the constant temperature scaling option is being used.

`temperaturescalefrequency` is the frequency at which temperature scaling is to be carried out.

`temperaturescaleoption` is the argument that specifies which temperature scaling option to use. The default is to do no scaling. Otherwise allowed options are: `"constant"` which maintains the temperature constant throughout the simulation at the value specified by the argument `temperature`; `"linear"` which changes the temperature linearly from `"temperaturestart"` to `"temperaturestop"` during the simulation; and `"exponential"` which is similar to the `"linear"` option except that the temperature is changed exponentially.

`temperaturestart` gives the starting temperature for exponential or linear temperature scaling.

`temperaturestop` gives the stopping temperature for exponential or linear temperature scaling.

`timestep` is the value of the timestep in picoseconds (1 ps is equivalent to 10^{-12} s or 1000 fs). The total length of the simulation will be `steps` $\times$ `timestep` ps.

`trajectories` defines trajectory objects to which data are to be written during the simulation. The only means of analysing data from a molecular dynamics simulation is by storing intermediate information in a trajectory and performing the analysis separately afterwards. `trajectories` is a combination of the `savefrequency` and `trajectory` arguments that we have met for the functions described in Sections 7.7, 7.9 and 8.4. It should be a sequence consisting of pairs of items, the first a trajectory object and the second an integer that gives the frequency at which data are to be saved to the trajectory. The reason for allowing multiple trajectories in this way is so

that data of different types, such as coordinates, velocities and energy values, can be obtained from the same simulation.

The assignment and handling of velocities for the simulated system is done automatically by the velocity Verlet function. Instances of the class `System` have an attribute which stores the atomic velocities. If, on entry to `VelocityVerletDynamics_SystemGeometry`, this attribute does not exist, the function generates velocities at the temperature specified by the argument `temperature` (for no or constant temperature scaling) or by the argument `temperaturestart` (for linear or exponential scaling). Velocities are assigned to the atoms from a Maxwell–Boltzmann distribution at the desired temperature with, if appropriate, removal of rotational and translational motion. If the instance of `System` already has velocities, due, for example, to a previous call to the velocity Verlet function, these will be employed instead to start the simulation and no new velocities will be assigned.

9.3 Example 16

In this section we illustrate the use of the velocity Verlet dynamics function by performing a short simulation on the blocked alanine molecule. The program is:

```
"""Example 16."""

from Definitions import *

# . Define the energy models.
mmmodel = MMModelOPLS ( "booksmallexamples" )
nbmodel = NBModelFull ( )

# . Generate the molecule.
molecule = MOLFile_ToSystem ( \
                      os.path.join ( molpath, "bala_c7eq.mol" ) )
molecule.DefineMMModel ( mmmodel )
molecule.DefineNBModel ( nbmodel )
molecule.Summary ( )
molecule.Energy  ( )

# . Optimization.
ConjugateGradientMinimize_SystemGeometry ( \
                                    molecule,                   \
                                    maximumiterations    = 2000, \
                                    logfrequency         =  100, \
```

```
                                    rmsgradienttolerance =  0.1  )
18
19 # . Define a random number generator in a given state.
20 rng = Random ( )
21 rng.seed ( 175189 )
22
23 # . Heating.
24 VelocityVerletDynamics_SystemGeometry ( \
                            molecule,                              \
                            logfrequency              =      100, \
                            rng                       =      rng, \
                            steps                     =     1000, \
                            timestep                  =    0.001, \
                            temperaturescalefrequency =      100, \
                            temperaturescaleoption    = "linear", \
                            temperaturestart          =     10.0, \
                            temperaturestop           =    300.0  )
25
26 # . Equilibration.
27 VelocityVerletDynamics_SystemGeometry ( \
                          molecule,                                \
                          logfrequency              =        500, \
                          steps                     =       5000, \
                          timestep                  =      0.001, \
                          temperaturescalefrequency =        100, \
                          temperaturescaleoption    = "constant", \
                          temperature               =      300.0  )
28
29 # . Data-collection.
30 trajectory = SystemGeometryTrajectory ( \
                  os.path.join ( scratchpath, "bala_c7eq.trj" ), \
                                       molecule, mode = "w" )
31 VelocityVerletDynamics_SystemGeometry ( \
                           molecule,                               \
                           logfrequency =                   500, \
                           steps        =                 10000, \
                           timestep     =                 0.001, \
                           trajectories = [ ( trajectory, 100 ) ] )
```

Lines 6–14 define the bALA molecule and its OPLS MM energy model and calculate the energy at the starting configuration.

Line 17 optimizes the molecule's coordinates so that the RMS gradient for the system is not too high. This is standard procedure before starting a molecular dynamics simulation study because the integration algorithm can

become unstable if the forces on some of the atoms are large due to strain in the molecule or unfavourable non-bonding contacts.

Lines 20–21 create an instance of Python's default random number generator class, `Random`, and set its state by calling the instance's `seed` method with an arbitrary integer argument.

Lines 24–31 perform the dynamics simulation with three separate calls to the function `VelocityVerletDynamics_SystemGeometry`. The timestep for each call is 1 fs (10^{-3} ps). Each call does a different phase of a dynamics calculation. In the *heating phase* on *line 24*, the temperature of the system is increased from an initial value of 10 K to a final value of 300 K. In the *equilibration phase* on *line 27*, the temperature of the system is maintained constant at 300 K. In the *data-collection phase* on *line 31*, no temperature modification is performed. The random number generator instance, `rng`, created on *line 20*, is only needed for the first call to the velocity Verlet function as this is the only one in which velocities are assigned.

The heating and equilibration phases of the dynamics are done to prepare the system for the data-collection phase and are necessary to ensure that the kinetic energy in the system is partitioned roughly equally between all the available degrees of freedom. For a small system, such as blocked alanine, a heating period of 1 ps and then an equilibration period of 5 ps (1000 and 5000 steps, respectively) are probably adequate. For larger systems longer periods will be necessary. The scaling of the velocities in the heating and equilibration phases is done at 100-step intervals and is performed so that the temperature of the system increases linearly between 10 and 300 K in the heating phase but stays constant at 300 K in the equilibration phase. For the data-collection stage, 10 ps of dynamics is performed and the coordinates for the molecule are saved at 100-step intervals (i.e. every 0.1 ps) in the trajectory object which is defined on *line 30*. No velocity modification is done during this phase.

In all three calls information about the dynamics is printed out at reasonable intervals just to check that there are no anomalies in the integration. The information consists of the total energy, the kinetic and potential energies and the temperature of the system. In addition, at the end of the simulation the averages and the RMS deviations of these quantities from their averages for the complete run are printed. These values are especially useful when no velocity modification has been done because the average total energy and its RMS deviation will give an indication of how well the energy was conserved during the simulation.

The values of the energies and temperature from the data-collection phase of the simulation are plotted in Figures 9.1 and 9.2, respectively. The total energy

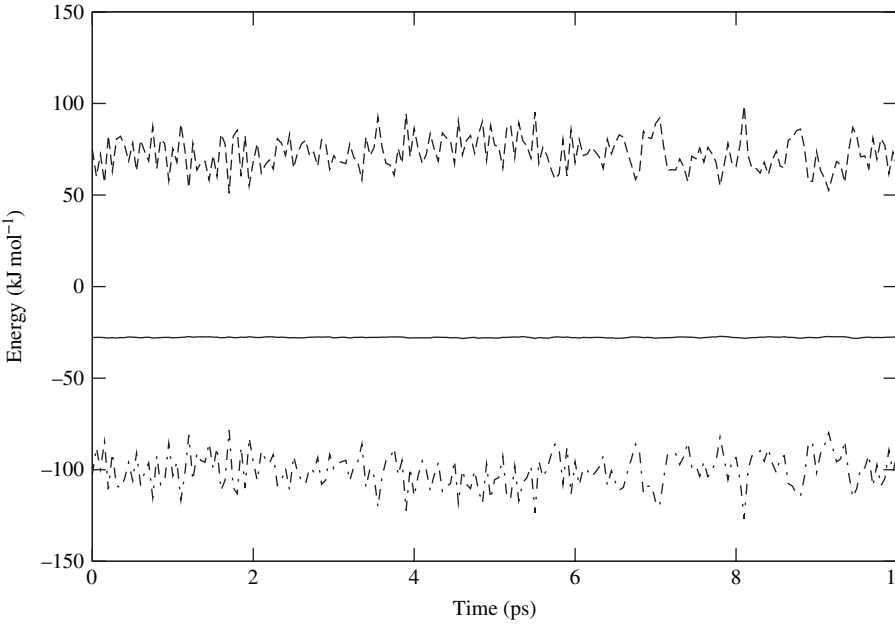


Fig. 9.1. A plot of the kinetic, potential and total energies of the bALA system during the data-collection phase of the dynamics of Example 16. Kinetic energy, dashed line; potential energy, dash–dot line; total energy, solid line.

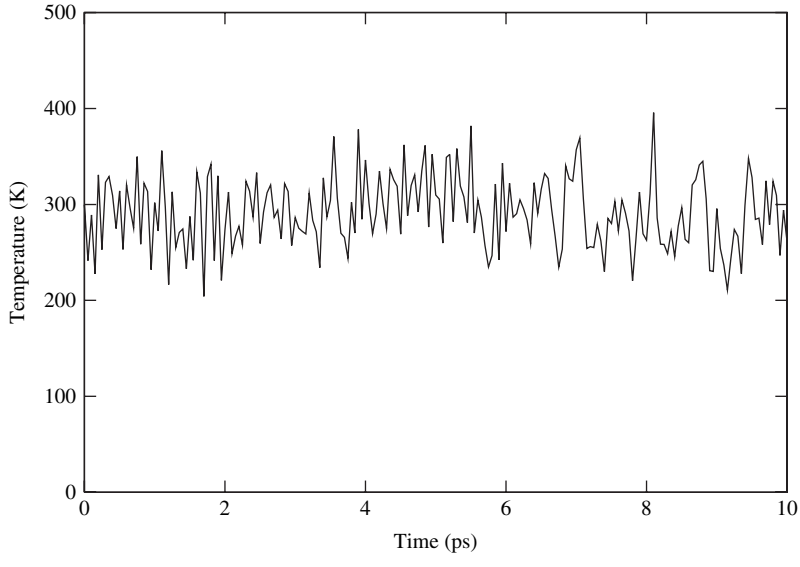


Fig. 9.2. A plot of the temperature of the bALA system during the data-collection phase of the dynamics of Example 16.

is reasonably well conserved with an RMS deviation for the entire simulation of about 0.2 kJ mol^{-1}. Owing to the fact that the total energy is conserved, there is a constant transfer of energy between the kinetic and the potential degrees of freedom. This can be seen to have a significant effect on the value of the instantaneous temperature, which overall has an average of about 293 K but varies in the range 220–390 K.

9.4 Trajectory analysis

We have seen how to generate trajectories of coordinate data for a system from a molecular dynamics simulation. A crucial part of most simulation studies is the analysis of these trajectories, either because we want to use the data to calculate properties of the system that can be related to those observable experimentally, or for some other reason. In this section only some very simple analyses of trajectory data are described. More advanced techniques will be left to Section 11.2.

At the most basic level, the analysis of a dynamics trajectory consists of calculating a property for each frame of the trajectory and seeing how it changes as a function of time. The sequence of data created in this way is called a *time series* for the property. Direct inspection of time-series data can be useful, for example, to chemists wanting a qualitative view of the change in the structural or other properties of a system. Usually, though, a more rigorous analysis needs to be undertaken if meaningful conclusions are to be extracted.

Here we consider two of the most useful statistical quantities that can be calculated from simulation data. These are *averages* and *fluctuations*. If $\mathcal{X}$ is the property under consideration, $\mathcal{X}_n$ is the nth value of the property in the time series and n_t is the total number of elements in the series, the average of the property is

$$\langle \mathcal{X} \rangle = \frac{1}{n_t} \sum_{n=1}^{n_t} \mathcal{X}_n \tag{9.16}$$

and the fluctuation is

$$\langle (\delta \mathcal{X})^2 \rangle = \langle (\mathcal{X} - \langle \mathcal{X} \rangle)^2 \rangle \tag{9.17}$$

$$= \langle \mathcal{X}^2 \rangle - \langle \mathcal{X} \rangle^2 \tag{9.18}$$

The importance of these two types of function is that, when they are calculated for specific properties, they can be related to experimentally observable quantities for the system. The formulae which link the two – the microscopic calculated data and the macroscopic observable data – are all derivable from statistical thermodynamics. We shall discuss this aspect in more detail in a later chapter,

but we have already met one such relation in Equation (9.14), which equates the temperature of the system to the average of its kinetic energy.

The straightforward statistical analyses presented in this section are implemented in pDynamo with the class `Statistics`. Instances of this class take a sequence of floating-point data and return many of the data's standard statistical quantities as instance attributes. The class definition is:

Class `Statistics`

A class to perform statistical analyses of a sequence of floating-point data.

Constructor

Construct an instance of the `Statistics` class given some floating-point data.

Usage: `new = Statistics ( data )`

`data` is a sequence of floating-point data. This argument is often a Python `list` but other sequence types, such as `Vector`, are also acceptable.

`new` is the new instance of `Statistics`.

Method `Count`

Count the number of data elements with a particular value.

Usage: `n = statistics.Count ( value, tolerance = 0.0 )`

`value` is the value whose frequency in the data set is to be determined.

`tolerance` is a tolerance that indicates how far data elements can be away from the target value before they are considered unequal. A data element `datum` is taken to be equal to `value` if $|\texttt{datum} - \texttt{value}| \leq \texttt{tolerance}$. The default value for `tolerance` is zero which means that only 'exact' matches will be counted.

`statistics` is the instance of `Statistics` for which counting is to be performed.

`n` is the number of occurrences.

Attributes

`maximum`	is the maximum value in the data set.
`mean`	is the mean of the data.
`minimum`	is the minimum value in the data set.

`size`	is the number of data elements.
`standarddeviation`	is the standard deviation of the data determined as the square root of the variance.
`sum`	is the sum of the data.
`variance`	is the variance of the data. This quantity is equivalent to the fluctuation of Equation (9.18).

Some of the more routine or sophisticated analyses of trajectory data in pDynamo employ special functions, some of which will be introduced in later chapters. However, in this chapter we shall restrict ourselves to the case in which the data in a frame of a trajectory are restored to the instance of `System` associated with the trajectory. This is done by extending the definition of the class `SystemGeometryTrajectory` that was introduced in Section 7.9 with another method whose definition is:

Class `SystemGeometryTrajectory`

Methods for extracting trajectory data.

Method `RestoreOwnerData`

Restore the data from a frame of the trajectory to the instance of `System` associated with the trajectory.

Usage: `QOK = trajectory.RestoreOwnerData ( )`

`trajectory`	is the instance of `SystemGeometryTrajectory` from which data are being transferred.
`QOK`	is a Boolean variable that takes the value `True` if data from the frame were successfully transferred but `False` if there were no more frames on the trajectory.
Remarks:	The trajectory object has a counter, initially set to zero, that indicates the position of the current frame and which is incremented every time a frame is read.

9.5 Example 17

In this section we perform an analysis of a molecular dynamics trajectory using the trajectory file that was generated in Example 16 of Section 9.3. The program calculates the values of two of the dihedral angles, ϕ and ψ, in the blocked alanine molecule for each frame of the molecular dynamics trajectory and is:

```
"""Example 17."""

from Definitions import *

# . Read the molecule definition.
molecule = MOLFile_ToSystem ( \
                   os.path.join ( molpath, "bala_c7eq.mol" ) )
molecule.Summary ( )

# . Define the trajectory.
trajectory = SystemGeometryTrajectory ( \
              os.path.join ( scratchpath, "bala_c7eq.trj" ), \
                                      molecule, mode = "r" )

# . Loop over the frames in the trajectory.
phi = []
psi = []
while trajectory.RestoreOwnerData ( ):
    phi.append ( molecule.coordinates3.Dihedral ( 4, 6,  8, 14 ) )
    psi.append ( molecule.coordinates3.Dihedral ( 6, 8, 14, 16 ) )

# . Set up the statistics calculation.
phistatistics = Statistics ( phi )
psistatistics = Statistics ( psi )

# . Output the results.
table = logfile.GetTable ( columns = [ 20, 20, 20 ] )
table.Start   ( )
table.Title   ( "Phi/Psi Angles" )
table.Heading ( "Frame" )
table.Heading ( "Phi"   )
table.Heading ( "Psi"   )
for ( i, ( h, s ) ) in enumerate ( zip ( phi, psi ) ):
    table.Entry ( `i` )
    table.Entry ( "%.2f" % ( h, ) )
    table.Entry ( "%.2f" % ( s, ) )
table.Entry ( "Mean:",               alignment = "l" )
table.Entry ( "%.2f" % ( phistatistics.mean, ) )
table.Entry ( "%.2f" % ( psistatistics.mean, ) )
table.Entry ( "Standard Deviation:", alignment = "l" )
table.Entry ( "%.2f" % ( phistatistics.standarddeviation, ) )
table.Entry ( "%.2f" % ( psistatistics.standarddeviation, ) )
table.Stop ( )
```

Lines 6–7 define the bALA molecule that was simulated in Example 16. No energy model is specified as no energies are to be calculated in this program.

Line 10 creates a trajectory object for the trajectory file generated in Example 16.

Lines 13–14 initialize lists that will hold the ϕ and ψ angles for each frame in the trajectory.

Line 15 uses a Python `while` statement to restore the data on the trajectory to `molecule` one frame at a time. The only data restored in this case are the `coordinates3` attribute of `molecule`. The loop stops when there are no more data on the trajectory.

Lines 16–17 calculate the ϕ and ψ angles for the current frame using the method `Dihedral` from the `Coordinates3` class. The arguments to the methods give the indices of the atoms that define the ϕ and ψ angles.

Lines 20–21 create instances of the class `Statistics` that will be used to analyse the ϕ and ψ angle data.

Lines 24–33 output the values of the ϕ and ψ angles for each frame in the trajectory to a table. The loop on *line 30* uses a combination of the Python built-in functions `enumerate` and `zip` to return an integer that gives the current loop index and pairs of ϕ and ψ values.

Lines 34–40 terminate the table by printing out the means and standard deviations of the ϕ and ψ angles generated during the simulation.

The results of this program for the dihedrals are plotted in Figure 9.3. Each point in the plot represents a single point along the trajectory and gives the values of both dihedrals. In this example the range of angles sampled is relatively small because of the limited length of the trajectory.

In Figure 9.4 the RMS coordinate deviations between the starting structure for bALA and the subsequent structures in the trajectory are shown. Each structure has been oriented using the methods discussed in Section 3.6 so as to minimize the value of the RMS deviation. The average value of the coordinate deviation is about 0.4 Å but there are deviations from this average of up to 0.2 Å along the trajectory.

As a final point in this section we note that the framework developed for trajectory analysis in this section is a general one and can be used to analyse trajectories generated in other applications. We have already met two of these, for normal modes (Section 8.4) and for reaction paths (Sections 7.7 and 7.9).

9.6 Simulated annealing

We leave for a moment the use of molecular dynamics simulation as a tool for the investigation of the equilibrium and dynamical properties of a system and

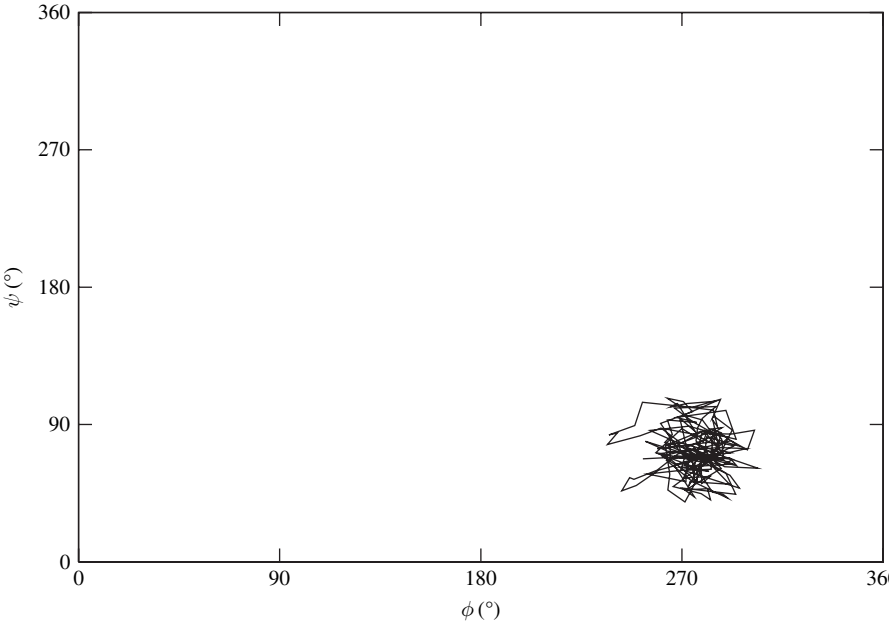


Fig. 9.3. A plot of the values of the ϕ and ψ angles as functions of time for the simulation of bALA of Example 16.

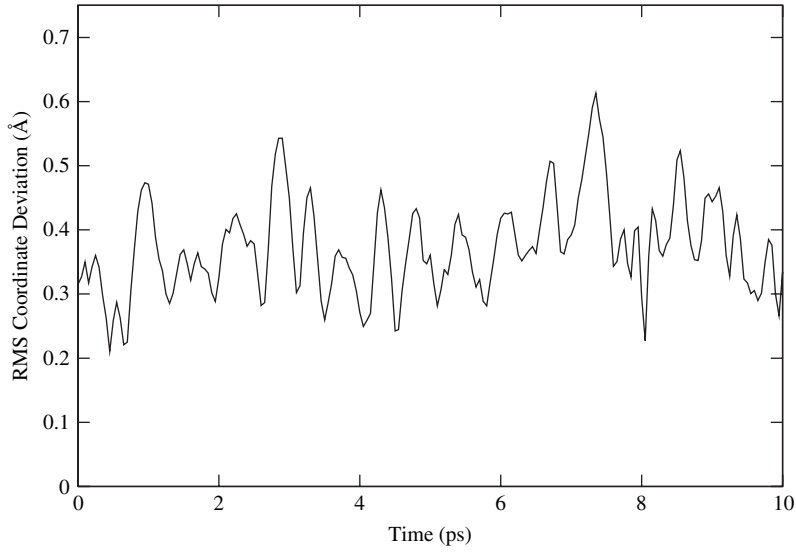


Fig. 9.4. A plot of the RMS coordinate deviations between the starting and the subsequent bALA structures along the trajectory generated in the dynamics simulation of Example 16.

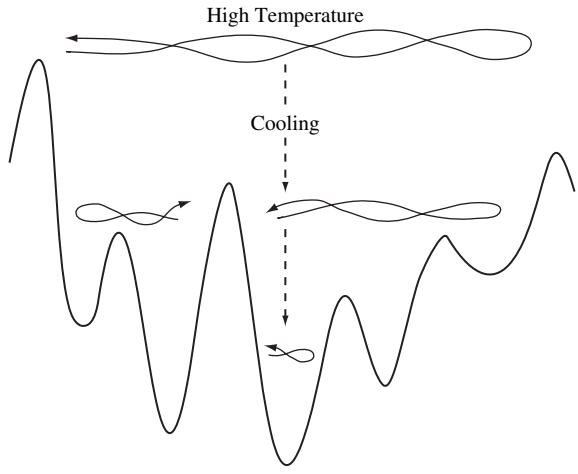


Fig. 9.5. A schematic diagram of a simulated annealing calculation on a potential energy surface.

discuss its application in another context. In Chapter 7, algorithms for exploring the potential energy surface of a system were discussed. These algorithms had in common the fact that they were local and searched the region of the surface in the neighbourhood of the starting configuration. Such procedures are useful in many cases but in other applications knowledge of the *global minimum* or near-global minima is required, for which local search algorithms are inappropriate.

Because of their practical importance, *global optimization algorithms* have been the subject of intense research and several global search strategies have been developed. One of the earliest and still one of the most useful is the method of *simulated annealing*, which was introduced by S. Kirkpatrick, C. D. Gelatt and M. P. Vecchi during the early 1980s. This method is based upon the correspondence between a statistical mechanical system and an optimization problem in which a minimum of a function that depends on many parameters is to be found.

The essential idea behind simulated annealing and the fact that distinguishes it from local optimization algorithms is the consideration of the temperature of the system. When a system has a non-zero temperature its total energy is no longer just the potential energy because there is a kinetic energy component too. An image of this is shown in Figure 9.5, in which the addition of the temperature 'lifts' the system off the potential energy surface and makes a much larger number of configurations accessible. The higher the temperature the larger the number of configurations that are accessible because the system has energy to surmount larger barriers.

A simulated annealing calculation proceeds by giving the system a high temperature, allowing it to equilibrate and then cooling until the system has been

annealed to the potential energy surface, i.e. until the temperature is zero. The way in which the cooling is done, the *cooling schedule*, determines the effectiveness of the simulated annealing method. In general, the cooling needs to be done slowly so that the system can thoroughly explore the potential energy surface and avoid becoming trapped in regions of high potential energy. Of course, there is no guarantee that this will not happen, but it is known from statistical mechanics that the probability that a particular configuration will be favoured is proportional to its Boltzmann factor, $\exp[-\mathcal{V}/(k_B T)]$, where $\mathcal{V}$ is the potential energy of the configuration. Thus, the lower the potential energy of the configuration the more probable it is. The optimal cooling schedule cannot be found in most problems of interest and so the choice of schedule is to some extent a matter of experimentation. Fortunately, even relatively crude schedules can give good results.

Molecular dynamics methods, because they employ a temperature, can be used for simulated annealing optimization calculations and it is this approach that we shall illustrate in the next example. It should be noted, though, that Monte Carlo algorithms, which are discussed later, are equally viable for simulated annealing calculations. In fact, it was with these that the first simulated annealing applications were performed.

9.7 Example 18

As an example of a simulated annealing calculation we consider Lennard-Jones clusters of size 13. There are two reasons for this. First, it is relatively easy to obtain the global minimum for the bALA molecule using local minimization techniques (try it and see!) so this does not represent a particularly interesting test case. Second, the Lennard-Jones clusters, as mentioned in Section 7.2, represent something of a benchmark for the evaluation of global optimization methods for molecular systems.

The program is one of the most complicated that we shall meet in this book. What it does is to generate 100 structures for the cluster and then optimize each of them in two different ways, first by a local conjugate-gradient minimization procedure and, second, using a simulated annealing protocol. The energies of the two sets of optimized structures are compared and printed at the end.

```
1 """Example 18."""
2
3 from Definitions import *
4
5 # . Define various parameters.
```

```
CLUSTERSIZE     = 3.0
ENERGYTOLERANCE = 0.01
FORCECONSTANT   = 5.0
NTRIALS         = 100

# . Set up the system.
cluster = MOLFile_ToSystem ( os.path.join ( molpath, "argon13.mol" ) )
cluster.DefineMMModel ( MMModelOPLS ( "lennardjones" ) )
cluster.DefineNBModel ( NBModelFull ( ) )
cluster.Summary ( )

# . Define tether constraints for each atom.
origin            = Vector3 ( value = 0.0 )
tetherenergymodel = SoftConstraintEnergyModelHarmonicRange ( 0.0, \
                                  0.5 * CLUSTERSIZE, FORCECONSTANT )
tethers           = SoftConstraintContainer ( )
for i in range ( len ( cluster.atoms ) ):
    tethers[‘i‘] = SoftConstraintTether ( i, origin, tetherenergymodel )

# . Initialize the random number generator.
rng = Random ( )

# . Initialize lists to keep energies.
pe0 = []
pe1 = []
pe2 = []
pe3 = []

# . Loop over the trials.
for i in range ( NTRIALS ):

    # . Reset the cluster coordinates and cluster constraints.
    cluster.coordinates3 = MOLFile_ToCoordinates3 ( \
                                    os.path.join ( molpath, "argon13.mol" ) )
    cluster.DefineSoftConstraints ( tethers )

    # . Initialize some variables for the trial.
    rng.seed ( 957612 + i )
    tstart = 300.0 * ( rng.random ( ) + 1.0 )

    # . Do a short dynamics to generate a new structure.
    VelocityVerletDynamics_SystemGeometry ( cluster,                        \
                                    log                     =       None, \
                                    rng                     =        rng, \
```

```
                                    steps                     =      10000, \
                                    timestep                  =      0.001, \
                                    temperaturescalefrequency =        100, \
                                    temperaturescaleoption    = "constant", \
                                    temperaturestart          =      tstart )
46
47  # . Save the starting coordinates and energy.
48  temporary3 = Clone ( cluster.coordinates3 )
49  cluster.DefineSoftConstraints ( None )
50  pe0.append ( cluster.Energy ( log = None ) )
51
52  # . Minimization.
53  cluster.DefineSoftConstraints ( tethers )
54  ConjugateGradientMinimize_SystemGeometry ( cluster,                    \
                                               log                  =  None, \
                                               maximumiterations    = 10000, \
                                               rmsgradienttolerance = 1.0e-4 )
55  cluster.DefineSoftConstraints ( None )
56  ConjugateGradientMinimize_SystemGeometry ( cluster,                    \
                                               log                  =  None, \
                                               maximumiterations    = 10000, \
                                               rmsgradienttolerance = 1.0e-4 )
57  pe1.append ( cluster.Energy ( log = None ) )
58
59  # . Simulated annealing from the starting coordinates.
60  cluster.coordinates3 = temporary3
61  cluster.DefineSoftConstraints ( tethers )
62  VelocityVerletDynamics_SystemGeometry ( cluster,           \
                  log                       =          None, \
                  steps                     =         40000, \
                  timestep                  =         0.001, \
                  temperaturescalefrequency =            10, \
                  temperaturescaleoption    = "exponential", \
                  temperaturestart          = tstart,        \
                  temperaturestop           = tstart * math.exp ( - 10.0 ) )
63  cluster.DefineSoftConstraints ( None )
64  pe2.append ( cluster.Energy ( log = None ) )
65
66  # . Minimization of the annealed structure.
67  ConjugateGradientMinimize_SystemGeometry ( cluster,                    \
                                               log                  =  None, \
                                               maximumiterations    = 10000, \
                                               rmsgradienttolerance = 1.0e-4 )
68  pe3.append ( cluster.Energy ( log = None ) )
69
```

```
70 # . Prepare the energies for statistics.
71 stpe1 = Statistics ( pe1 )
72 stpe2 = Statistics ( pe2 )
73 stpe3 = Statistics ( pe3 )
74
75 # . Output the results.
76 table = logfile.GetTable ( columns = [ 10, 20, 20, 20, 20 ] )
77 table.Start   ( )
78 table.Title   ( "Optimization Results" )
79 table.Heading ( "Attempt"          )
80 table.Heading ( "Initial Energy"   )
81 table.Heading ( "Minimized Energy" )
82 table.Heading ( "Annealed Energy"  )
83 table.Heading ( "Final Energy"     )
84 for i in range ( NTRIALS ):
85     table.Entry ( `i` )
86     table.Entry ( "%20.3f" % ( pe0[i], ) )
87     table.Entry ( "%20.3f" % ( pe1[i], ) )
88     table.Entry ( "%20.3f" % ( pe2[i], ) )
89     table.Entry ( "%20.3f" % ( pe3[i], ) )
90 table.Entry ( "Minimum Energies:", alignment = "l", columnspan = 2 )
91 table.Entry ( "%20.3f" % ( stpe1.minimum, ) )
92 table.Entry ( "%20.3f" % ( stpe2.minimum, ) )
93 table.Entry ( "%20.3f" % ( stpe3.minimum, ) )
94 table.Entry ( "Frequencies:",      alignment = "l", columnspan = 2 )
95 table.Entry ( `stpe1.Count ( stpe1.minimum, tolerance = ENERGYTOLERANCE )` )
96 table.Entry ( `stpe2.Count ( stpe2.minimum, tolerance = ENERGYTOLERANCE )` )
97 table.Entry ( `stpe3.Count ( stpe3.minimum, tolerance = ENERGYTOLERANCE )` )
98 table.Stop ( )
```

Lines 6–9 set the values of various parameters for use later in the program.

Lines 12–15 define the cluster and its energy model. To construct a system, pDynamo requires an elemental type for each of its atoms and so, for the purposes of this program, the cluster is taken to consist of 13 argon atoms. However, to enable comparison between the cluster energies calculated by the program and those listed in Table 7.1, the Lennard-Jones radius and well depth parameters in the OPLS file `"lennardjones"` are both defined as 1.

Lines 18–22 create a set of tether constraints for each of the atoms in the cluster (see Section 5.6). These are employed in parts of the optimization process to ensure that the atoms stay together in the same region of space and do not split up into smaller clusters. The tether energy model is such that the particles are constrained only if they are further than `0.5 * CLUSTERSIZE` or 1.5 Å away from the origin.

Lines 25–31 create an instance of Python's random number generator and initialize some lists that will hold potential energy values.

Line 34 starts the loop in which structures will be locally and globally optimized. In all 100 trials are performed.

Lines 37–45 prepare a starting structure for the subsequent optimizations by carrying out a short molecular dynamics simulation with tether constraints on the atoms and with `"constant"` temperature scaling. The same structure (from the file `"argon13.mol"`) is always used as input to the simulation but the temperature of the simulation, `tstart`, and the state of the random number generator, `rng`, are different.

Line 48 clones the starting coordinates for later use.

Lines 49–50 evaluate and save the energy of the starting, unoptimized cluster structure in the absence of constraints.

Lines 53–57 locally optimize the cluster structure using a conjugate-gradient algorithm, first in the presence and then in the absence of constraints.

Lines 60–64 perform the simulated annealing calculation starting with the same structure (from `temporary3`) as the local optimization. The molecular dynamics simulation employs an exponential cooling schedule.

Lines 67–68 further refine the annealed structure using a local optimization method. This is typical practice because the annealed structures will not usually be minima on the potential energy surface, although they will often be in regions of low potential energy.

Lines 71–98 analyse and output the results of the optimizations. For each trial, the energies of the starting, the locally optimized, the annealed and the optimized-annealed structures are printed out. The output terminates with the lowest energy found for each set of structures and the number of times that it occurs.

It is important to emphasize that multiple trials, with different starting conditions, are essential when doing simulated annealing if low-energy structures are to be obtained. The global minimum will certainly not be found in one attempt! For the 13-atom cluster studied in this example, the global minimum has an energy of −44.327 reduced units (see Table 7.1). If Example 18 is run several times (with different initial states for the random number generator), the simulated annealing method finds the global minimum approximately 40% of the time which is about twice as often as the local optimization technique.

Exercises

9.1 In Example 16 a timestep of 1 fs was used to integrate the dynamics equations. Repeat the simulations using different timesteps, but for the same total

simulation time to see how the results change. Is energy conservation substantially better with a shorter timestep? How do the values of the dihedral angles change? Are there better ways of comparing the different trajectories?

9.2 Do several long simulations of bALA starting from different structures and repeat the analysis of the ϕ and ψ angles of Section 9.5. How do the ϕ–ψ maps compare with the schematic potential energy surface for bALA of Exercise 7.1?

9.3 The simulated annealing calculations in Example 18 were done with a Lennard-Jones cluster of 13 atoms. Repeat the calculations with other cluster sizes and using different annealing schemes. In particular try the 'magic number' clusters with sizes of 19, 55 and 147. Does the simulated annealing procedure stay as efficient as the size of the cluster increases? One property of interest in cluster studies is the value of the energies of the states which are less stable than the ground state. How do these spectra of cluster energies compare when obtained with local optimization and simulated annealing approaches?

10

More on non-bonding interactions

10.1 Introduction

Up until now we have encountered a variety of standard techniques for the simulation of molecular systems. All the systems we have looked at, however, have been in vacuum and we have not, as yet, considered any extended condensed phase systems such as liquids, solvated molecules or crystals. This is because special techniques are needed to evaluate the non-bonding interactions in such systems. In the present chapter we introduce methods for determining these interactions which will allow us to treat some condensed phase problems. For simplicity we focus upon MM energy functions but similar principles are applicable to QC and hybrid potential energy models.

10.2 Cutoff methods for the calculation of non-bonding interactions

As we discussed in detail in Section 5.2.2, the non-bonding energy for a molecular system with the types of MM force fields that we are using can be written as a sum of electrostatic and Lennard-Jones contributions. The expression for the energy, $\mathcal{V}_{\text{nb}}$, is

$$\mathcal{V}_{\text{nb}} = \sum_{ij \text{ pairs}} \left(\frac{q_i q_j}{4\pi\epsilon_0 \epsilon r_{ij}} + \frac{A_{ij}}{r_{ij}^{12}} - \frac{B_{ij}}{r_{ij}^{6}} \right) \tag{10.1}$$

The crucial aspect of this equation is that the sum runs over all pairs of interacting atoms in the system. These comprise all possible pairs of atoms except those 1–2, 1–3 and (possibly) 1–4 interactions that are specifically excluded. In all the simulations we have done to date we have used instances of the class `NBModelFull` that evaluate the non-bonding energy in the simplest way possible, by calculating the interaction for all pairs of atoms explicitly. Because the number of pairs increases as $O(N^2)$, where N is the number of atoms in the system, the calculation of the non-bonding energy using this technique rapidly

becomes unmanageable. In particular, it becomes difficult or impossible to use for condensed phase systems.

To overcome this problem and to increase the efficiency of the non-bonding energy evaluation, a number of different techniques of varying sophistication can be used. They can be broadly divided into two categories – those that attempt to evaluate Equation (10.1) exactly (or, at least, to within a certain estimated precision) and those that modify the form of the expression for the non-bonding interaction in some way so that it is more readily evaluated. We shall first discuss the latter, approximate methods because, although they are less rigorous, they are easily implemented and they have been extensively employed for condensed phase simulations. We shall return to the more exact methods at the end of the chapter.

The principal problem with the non-bonding energy is the long-range electrostatic interaction which decays as the reciprocal of the distance between the atoms. The long-range nature of the interaction means that many pairs have to be included in the sum of Equation (10.1) to obtain a non-bonding energy of a given precision. The most widely used approximate methods for the evaluation of the non-bonding energy overcome the long-range nature of the electrostatic interaction by modifying its form so that the interactions between atoms are zero after some finite distance. These are the *cutoff* or *truncation* methods. The fact that the interactions are truncated means that the complexity of the calculation is formally reduced from $O(N^2)$ to $O(N)$. That this is so can be seen by the following argument. Suppose that a spherical truncation scheme is used and the cutoff distance for the interaction is r_c. Then each atom within the system will interact with all the atoms within a volume of $4\pi r_c^3/3$. If the mean number density of atoms within the system is ρ, the total number of interactions for the system will be $4\pi r_c^3 \rho N/3$ or $\propto N$. Obviously the cost of the calculation will depend upon the size of the cutoff. The trick is to use as small a cutoff as possible while still providing an adequate treatment of the non-bonding interactions.

There are several subtleties that have to be addressed when using cutoff schemes. The first is that of how the truncation is to be effected. The easiest way is to use an abrupt truncation and simply ignore all interactions that are beyond the cutoff distance. This is equivalent to multiplying each term in the non-bonding energy expression (Equation (10.1)) by a truncation function, $S(r)$, of the form

$$S(r) = \begin{cases} 1 & r \leq r_c \\ 0 & r > r_c \end{cases} \tag{10.2}$$

The problem with this type of truncation is that the energy and its derivatives are no longer continuous functions of the atomic coordinates and that there will be jumps in the energy during a minimization or a dynamics simulation as atoms

move in and out of each other's cutoff distance. These can disrupt a minimization process or lead to unwanted effects (such as heating) in a dynamics simulation.

An alternative to abrupt truncation is to use a smoothing function, $S(r)$, that tapers the interaction continuously to zero at a given distance. Many smoothing functions have been proposed. One example is a *switch function* that is cubic in the square of the interaction distance, r^2:

$$S(r) = \begin{cases} 1 & r \leq r_{\text{on}} \\ \frac{\left(r_{\text{off}}^2 - r^2\right)^2 \left(r_{\text{off}}^2 + 2r^2 - 3r_{\text{on}}^2\right)}{\left(r_{\text{off}}^2 - r_{\text{on}}^2\right)^3} & r_{\text{on}} < r \leq r_{\text{off}} \\ 0 & r > r_{\text{off}} \end{cases} \tag{10.3}$$

It has the property that the interaction is not modified for distances less than an inner cutoff distance, r_{on}, and is smoothed to zero at the outer cutoff, r_{off}. The function is constructed so that its first derivative is continuous in the full range $r \leq r_{\text{off}}$, which is necessary if problems in minimizations and in dynamics simulations are to be avoided. The second derivatives, though, are discontinuous.

A second example of a smoothing function is a *shift function*:

$$S(r) = \begin{cases} \left[1 - \left(\frac{r}{r_{\text{c}}}\right)^2\right]^2 & r \leq r_{\text{c}} \\ 0 & r > r_{\text{c}} \end{cases} \tag{10.4}$$

Like the switch function, the shift function has continuous first derivatives but, unlike the switch function, it relies on just one cutoff distance and it modifies the form of the interaction throughout its entire range.

Graphs of these functions and their first and second derivatives are illustrated in Figures 10.1, 10.2 and 10.3, respectively. In Figure 10.4, the products of the smoothing functions and a Coulomb interaction between two unit positive charges are shown. The first derivatives of these interactions are plotted in Figure 10.5. It can be seen that the use of truncation techniques can introduce radical differences in the form of the interaction potential.

The second problem that needs to be addressed is that of how the truncation or smoothing function is to be employed. The simplest way is to apply the smoothing procedure to each interaction separately. This means that, for each pair of atoms, ij, the smoothing function is calculated and the interaction for that pair is the product of the function and the pair's non-bonding energy, $S(r_{ij})\mathcal{V}_{\text{nb}}^{ij}$. This is adequate for uncharged systems, in which there are only Lennard-Jones interactions, or for systems with small charges but, for charged systems, an *atom-based* truncation scheme can lead to a problem that is colloquially known as *splitting of the dipoles*. For such systems *group-based* truncation schemes can provide better behaviour. In these methods atoms are partitioned into groups and

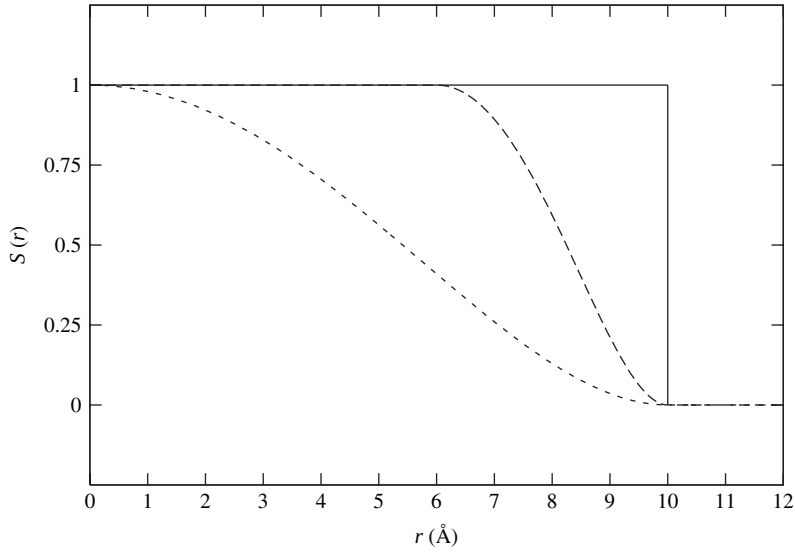


Fig. 10.1. Plots of various truncation functions, $S(r)$, as functions of distance. Direct truncation, solid line; switch function, long-dash line; shift function, short-dash line. The values of the cutoffs are $r_c = r_{off} = 10$ Å and $r_{on} = 6$ Å.

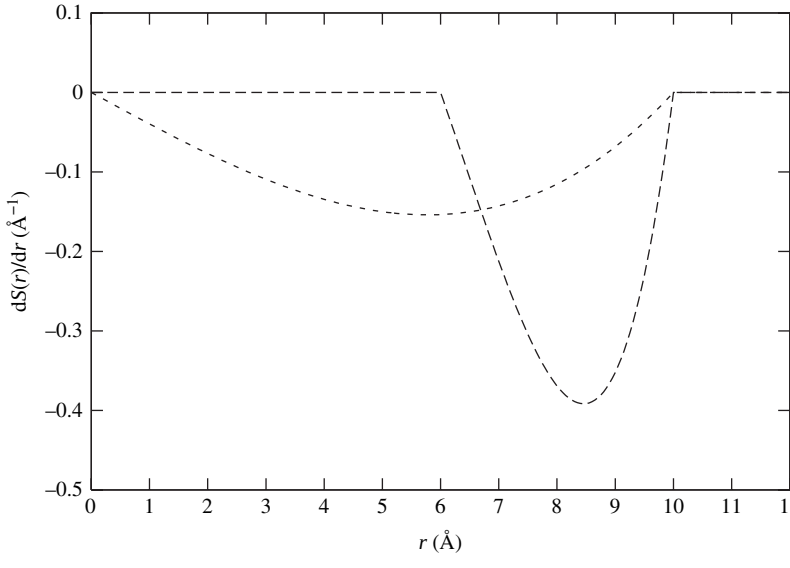


Fig. 10.2. Plots of the first derivatives of the switch and shift truncation functions displayed in Figure 10.1.

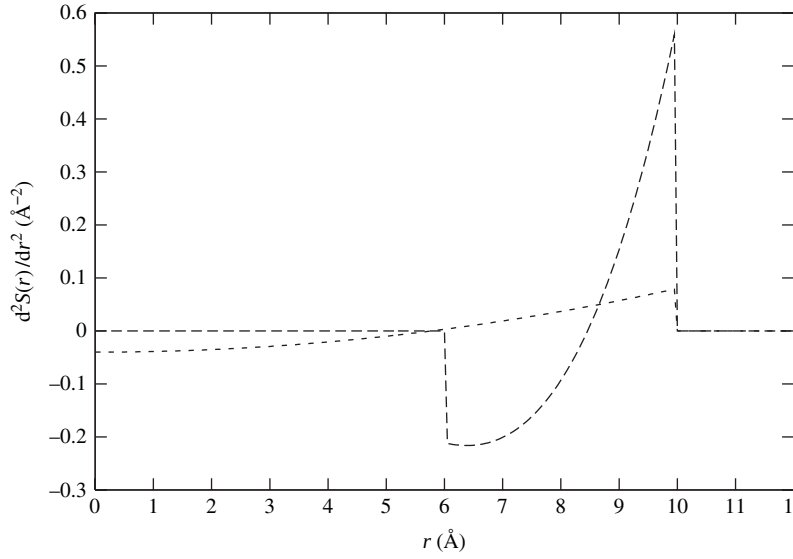


Fig. 10.3. Plots of the second derivatives of the switch and shift truncation functions displayed in Figure 10.1.

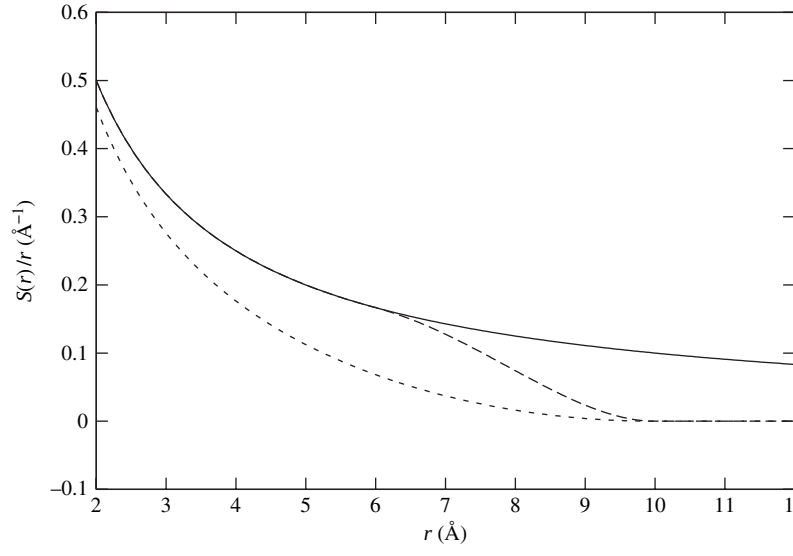


Fig. 10.4. The Coulomb interaction between two unit positive charges as a function of distance and as modified by the application of the switch and shift truncation functions. Full interaction, solid line; switched interaction, long-dash line; shifted interaction, short-dash line.

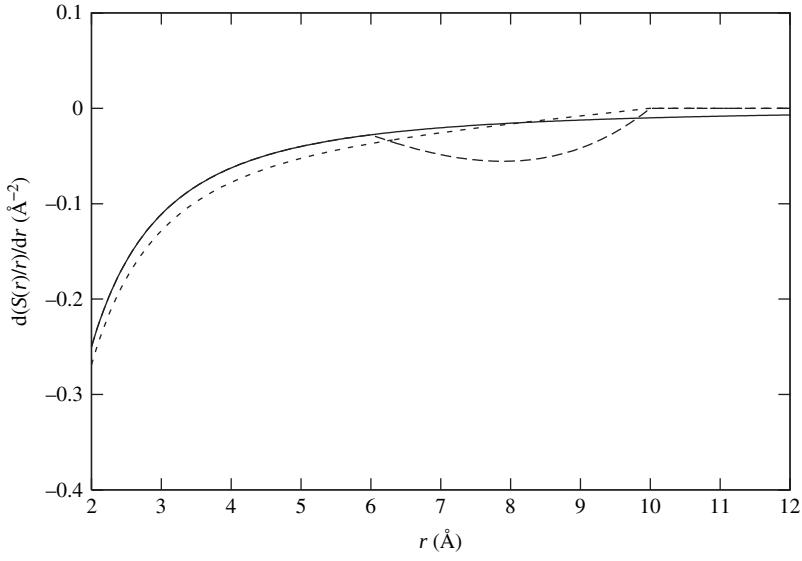


Fig. 10.5. The derivatives of the Coulomb interactions displayed in Figure 10.4.

the non-bonding interactions between all the atoms of two groups are calculated in full. The full group–group interaction energy is then multiplied by a single truncation or smoothing function that is calculated using a characteristic distance between the two sets of atoms such as, for example, the distance between their centres of geometry.

To illustrate the dipole-splitting problem, consider the interaction between an ion with a unit charge and a simple point-charge model for a water molecule. The water molecule is neutral overall but has a dipole moment with a value of, say, μ. The energy of the interaction between the charge and the dipole is proportional to the charge, the dipole and the inverse of the distance between the two species squared. Thus, the interaction decays more rapidly with distance than does a charge–charge interaction. Now, within a simple point-charge model, the water dipole can be represented by charges of $-2q$ on the oxygen and $+q$ on each of the hydrogens and so the charge–dipole interaction will be represented by three charge–charge interactions. To reproduce this interaction accurately it is necessary to include *all* these interactions fully within the calculation. It is also important to note that each charge–charge interaction is of a larger magnitude and is of longer range than the total charge–dipole interaction. Thus, if a cutoff model that splits these interactions is used (if, for example, two of the interactions are within the cutoff and one is not) large distortions in the energy and forces are likely to take place. Of course, these effects will be reduced if a smoothing function is employed rather than straight truncation but they will persist nevertheless. Although the

dipole-splitting effect was illustrated with a charge–dipole interaction, it occurs generally. For example, the interaction of two water molecules is a dipole–dipole interaction that scales as the inverse cube of the distance between the dipoles. Thus, splitting of the dipoles in this case could lead to larger errors than those for the charge–dipole interaction.

The final point to be tackled in connection with cutoff schemes is that of how to evaluate which interactions are within the cutoff and which are not. After all, determining the distances between all atom pairs in order to find which to calculate and which not is exactly the operation that we are trying to avoid! Clearly, a procedure similar to that outlined in Section 3.2 for the estimation of a system's bonds would be appropriate because it scales as $O(N)$. In principle, such a method could be employed for determining the interactions afresh each time an energy calculation is performed. In practice, though, it is observed that it is normally more efficient to use the method intermittently by creating a temporary list of non-bonding interactions that is valid for several energy calculations. This list is generated using a cutoff distance, the list cutoff, r_{list}, that is greater than the interaction cutoff, r_{c} or r_{off}. This means that the list will contain more interactions than are necessary for a single energy calculation, but it also means that the list does not need to be regenerated every time an energy is required. This is done only when the atoms have moved by an amount of the order of $r_{\text{list}}-r_{\text{c}}$ or $r_{\text{list}}-r_{\text{off}}$ so that the current list is invalidated.

Of the many possible truncation schemes that exist, the method that will be used in this work for the evaluation of the non-bonding interactions is one that has been described by P. Steinbach and B. R. Brooks and is called the *atom-based force-switching truncation scheme*. In this method it is not the interaction energy that is truncated directly, but its first derivative (and, hence, its force). Thus, if $f_{\text{true}}(r)$ is the force between two particles, the modified force, $f(r)$, has the form

$$f(r) = S(r) f_{\text{true}}(r) \tag{10.5}$$

For the electrostatic interactions, Steinbach and Brooks used the same switching function as that in Equation (10.3). Because this function has continuous first derivatives, it implies that the second derivatives of the energy will be continuous throughout the range of the modified interaction as well. The potential energy of each interaction is obtained by integrating Equation (10.5). If $\mathcal{V}(r)$ is the modified interaction and $\mathcal{V}_{\text{true}}$ the true interaction ($\equiv c/r$), one has

$$\mathcal{V}(r) = \begin{cases} \mathcal{V}_{\text{true}} + \frac{8c}{\gamma}\left[r_{\text{on}}^2 r_{\text{off}}^2 (r_{\text{off}} - r_{\text{on}}) - \frac{1}{5}(r_{\text{off}}^5 - r_{\text{on}}^5)\right] & r \le r_{\text{on}} \\ c\left[A\left(\frac{1}{r} - \frac{1}{r_{\text{off}}}\right) + B(r_{\text{off}} - r) + C(r_{\text{off}}^3 - r^3) + D(r_{\text{off}}^5 - r^5)\right] & r_{\text{on}} < r \le r_{\text{off}} \\ 0 & r > r_{\text{off}} \end{cases} \tag{10.6}$$

where the following constants have been defined:

$$\gamma = (r_{\text{off}}^2 - r_{\text{on}}^2)^3$$

$$A = \frac{r_{\text{off}}^4 (r_{\text{off}}^2 - 3r_{\text{on}}^2)}{\gamma}$$

$$B = \frac{6r_{\text{on}}^2 r_{\text{off}}^2}{\gamma}$$

$$C = -\frac{r_{\text{off}}^2 + r_{\text{on}}^2}{\gamma}$$

$$D = \frac{2}{5\gamma}$$

For the Lennard-Jones interactions, a simpler truncation function is used. This does not ensure continuity of the second derivatives of the energy but this is deemed acceptable because the magnitudes of the Lennard-Jones terms are so much smaller. If the form of each part of the Lennard-Jones interaction is c/r^n, where $n = 6$ or 12, the modified interaction energy is

$$\mathcal{V}(r) = \begin{cases} \mathcal{V}_{\text{true}} - \frac{c}{(r_{\text{on}} r_{\text{off}})^{n/2}} & r \le r_{\text{on}} \\ \frac{c r_{\text{off}}^{n/2}}{r_{\text{off}}^{n/2} - r_{\text{on}}^{n/2}} \left[\left(\frac{1}{r}\right)^{n/2} - \left(\frac{1}{r_{\text{off}}}\right)^{n/2} \right]^2 & r_{\text{on}} < r \le r_{\text{off}} \\ 0 & r > r_{\text{off}} \end{cases} \tag{10.7}$$

It can be seen, in both cases, that the interaction energies at less than the inner cutoff distance, r_{on}, are unaltered except that a constant which ensures that the potential is continuous at $r = r_{\text{on}}$ has been added. The interaction energy of two unit charges given by the scheme of Equation (10.6) and its derivative are plotted in Figures 10.6 and 10.7, respectively. Evidently the forces are only slightly distorted.

Steinbach and Brooks found that this scheme was one of the most effective for calculating non-bonding interactions – i.e. it reproduced well the results of calculations in which the full non-bonding interaction was calculated – as long as a reasonably long inner cutoff distance and a broad switching region were used (the latter to minimize the dipole-splitting effect). They suggested that minimum values of 8 and 12 Å were suitable for the inner and outer cutoffs, respectively. The method, although atom-based, was found to give results that were as good as or better than those from group-based methods even for highly charged systems.

The method of calculating the non-bonding interaction energy described above has been implemented in the class, `NBModelABFS`. This class is interchangeable

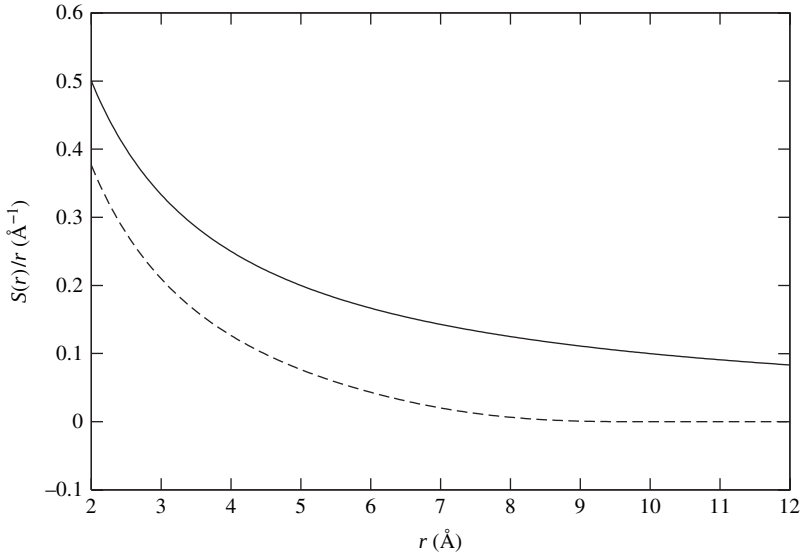


Fig. 10.6. The Coulomb interaction between two unit positive charges as a function of distance and as modified by the application of the force-switching function. Full interaction, solid line; force-switched interaction, dashed line. The values of the cutoffs are $r_{\text{off}} = 10$ Å and $r_{\text{on}} = 6$ Å and the constant $c = 1$.

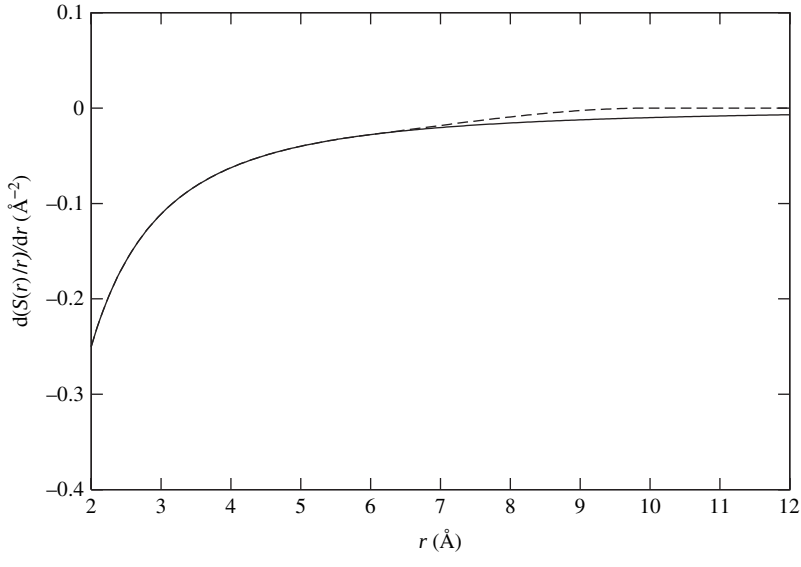


Fig. 10.7. The derivative of the Coulomb interaction displayed in Figure 10.6.

with the class `NBModelFull` which was described in Section 5.3.3 and behaves in a similar fashion. Its definition is:

Class `NBModelABFS`

A class to calculate non-bonding interactions within an atom-based force-switching approximation.

Constructor

Construct an instance of the `NBModelABFS` class.

Usage:
```
new = NBModelABFS ( )
```
`new` is the new instance of `NBModelABFS`.

Method `SetOptions`

Define the values of the cutoffs to be used in the calculation of the non-bonding interactions and in the generation of the non-bonding interaction lists.

Usage:
```
nbmodel.SetOptions ( innercutoff = 8.0,
                     outercutoff = 12.0,
                     listcutoff  = 13.5 )
```
`innercutoff` is the value of the inner cutoff, r_{on}.
`outercutoff` is the value of the outer cutoff, r_{off}.
`listcutoff` is the value of the list cutoff, r_{list}.
`nbmodel` is an instance of `NBModelABFS`.

Instances of this class generate the atom non-bonding interaction lists automatically and store them internally using a procedure similar to that described in Section 3.2. The list is generated when an energy is first required and subsequently when any atom has moved by more than half the difference between the cutoff distance used to generate the non-bonding energy interaction list (r_{list}) and the outer cutoff distance (r_{off}). This ensures that the list is always up to date. If the cutoff distances are set correctly, the update procedure is done only once every 10–20 energy calculations. A value for the list cutoff of 1–2 Å greater than the outer cutoff distance should give a reasonable update frequency. List generation and calculation of the energy both scale as $O(N)$ for a given cutoff distance.

To finish this section we emphasize a couple of general points. First, it is the electrostatic interactions that cause the biggest problems because of their long range and large size. The effects of truncation on the Lennard-Jones energies and forces are less crucial, although they can still be sizeable. The second point is

that no truncation method is ideal. The one that has been chosen here should give reasonable results in many cases, if it is properly used, but there will be others that will give results that are as valid in particular circumstances. According to Steinbach and Brooks, the most important lesson is that the cutoff should be as large as possible. The differences among the best alternative truncation methods are then less significant.

10.3 Example 19

To illustrate how the use of a truncation method can alter the calculation of the energy of a system it is interesting to calculate the energy for a large system using different cutoff schemes. The example program in this section does this for a small protein, crambin.

The program is:

```
"""Example 19."""

from Definitions import *

# . Define various parameters.
BUFFER     = 4.0
CINCREMENT = 1.0
CSTART     = 0.0
NENERGIES  =  40

# . Define the energy models.
mmmodel = MMModelOPLS ( "protein" )
nbfull  = NBModelFull ( )
nbabfs  = NBModelABFS ( )

# . Set up the system.
molecule = PDBFile_ToSystem ( \
                   os.path.join ( pdbpath, "crambin.pdb" ), \
                                  QUSERESIDUELIBRARY = True )
molecule.DefineMMModel ( mmmodel )
molecule.DefineNBModel ( nbfull  )
molecule.Summary ( )

# . Get the energy with a full model.
ef = molecule.Energy ( log = None )

# . Reset the NB model for the molecule.
molecule.DefineNBModel ( nbabfs )
```

```
27
28 # . Initialize the cutoff.
29 cut = CSTART
30
31 # . Output the energy difference for each cutoff.
32 table = logfile.GetTable ( columns = [ 20, 20 ] )
33 table.Start   ( )
34 table.Title   ( "Cutoff/Full Energy Difference" )
35 table.Heading ( "Inner Cutoff" )
36 table.Heading ( "Difference"   )
37 for i in range ( NENERGIES ):
38     cut += CINCREMENT
39     nbabfs.SetOptions ( innercutoff = cut,          \
                           outercutoff = cut + BUFFER, \
                           listcutoff  = cut + BUFFER  )
40     et   = molecule.Energy ( log = None )
41     table.Entry ( "%.1f" % ( cut, ) )
42     table.Entry ( "%.4f" % ( et - ef, ) )
43 table.Stop ( )
```

Lines 6–9 set the values of various quantities that are used later in the program.

Line 12 defines an instance of an OPLS MM energy model with the `"protein"` parameter set.

Lines 13–14 define instances of the two different NB energy models that we have encountered so far.

Lines 17–20 create an instance of `System` for the protein. On *line 17* the option `QUSERESIDUELIBRARY` to `PDBFile_ToSystem` is set so that the complete connectivity is generated for the molecule. As discussed in Section 5.3.3, this is needed in order to properly define the MM energy model.

Line 23 calculates and stores the potential energy for the protein with the full NB energy model.

Line 26 replaces the full NB energy model for the protein with the cutoff one.

Line 29 initializes the variable `cut`. This is the value of the inner cutoff, r_{on}, with which the cutoff non-bonding energies are to be calculated.

Lines 32–36 start the table that will be used for output of the energies.

Line 37 is the first line of the loop over cutoff distances. In this example energies will be calculated at 40 different cutoff distances.

Lines 38–39 increment `cut` by 1.0 Å and use it to define the new cutoffs in the non-bonding energy model. In all cases the outer cutoff is 4.0 Å greater than the inner cutoff.

Line 40 calculates the potential energy of the molecule, including the non-bonding energy terms with the current values of the cutoffs.

Lines 41–42 print the inner cutoff value and the difference between the full, `ef`, and cutoff, `et`, potential energies.

To illustrate the results of this example, the electrostatic energies as a function of the cutoff distance, r_{off}, are shown in Figure 10.8. Also shown are the energies produced with straight truncation (when the inner cutoff distance in Example 19 has the same value as the outer cutoff, i.e. $r_{\text{on}} = r_{\text{off}}$). The importance of a smoothing region is clear insofar as the energy tends smoothly to its limiting value if one is employed and oscillates wildly otherwise. This is a manifestation of the dipole-splitting problem. The effect of straight truncation is much less marked for the Lennard-Jones energies, which are shown in Figure 10.9. The number of non-bonding interaction pairs is plotted versus the cutoff distance in Figure 10.10. Above 32 Å all the possible interactions between atoms within the protein are included in the energy calculation.

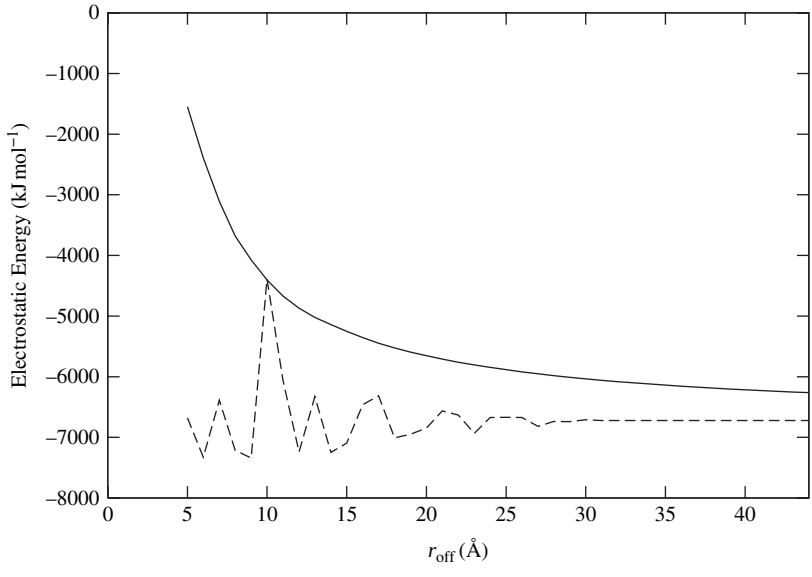


Fig. 10.8. The electrostatic energies as a function of the cutoff distance, r_{off}. With a smoothing region, solid line; straight truncation, dotted line.

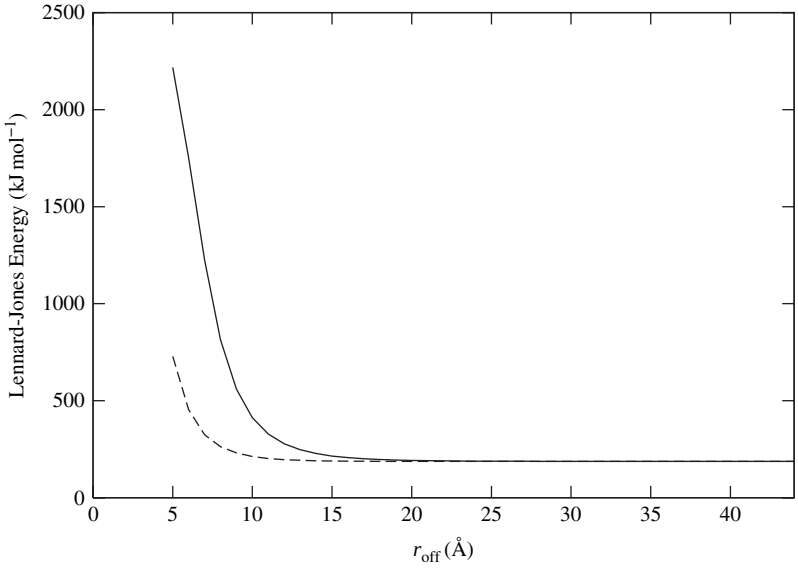


Fig. 10.9. The Lennard-Jones energies as a function of the cutoff distance, r_{off}. With a smoothing region, solid line; straight truncation, dotted line.

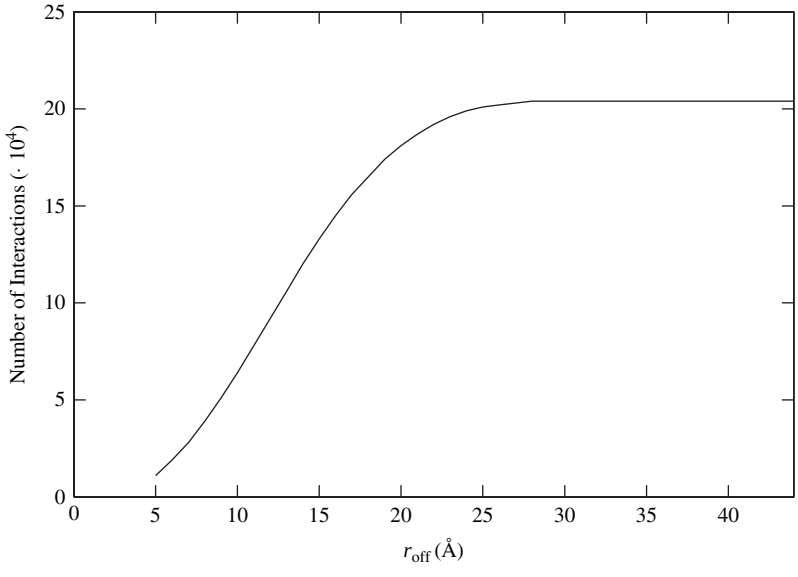


Fig. 10.10. The number of non-bonding interaction pairs as a function of the cutoff distance, r_{off}.

10.4 Including an environment

One of the most interesting and important applications of molecular simulations is the study of systems in the condensed phase. After all, it is in the condensed phase that the great majority of chemical and biochemical processes occur. Unlike the systems that we have been studying up to now, condensed phase systems are effectively infinite in extent. It is obviously impractical to try to simulate such systems directly because, no matter how large and fast the computer, the limits of its computational ability would be attained very rapidly! Currently, the only really feasible way of simulating condensed phase systems at an atomic level is to select a small part of the system to study in detail – for example, a small volume of a liquid or a crystal or a single solvated protein molecule – and then use methods that imitate the effect of the remainder of the system or the *environment*. The fact that an infinite system is being studied by using a finite one means, of necessity, that there are limitations to the types of process that can be studied. It is evident, for example, that one needs to be careful about drawing conclusions for properties of a system that have length scales larger than the size of the finite simulation system. However, if these concerns are borne in mind, simulation approaches can be powerful tools for investigating processes in the condensed phase.

There is a wide range of approximations in use to model the environment of a system and we shall discuss only a few. Probably the most widely used model and arguably the most reliable, if not the cheapest, is the method of *periodic boundary conditions* (PBCs). In this technique, a complete condensed phase system is modeled as an infinitely and periodically repeated series of copies of a small, but representative, part of the full system (see Figure 10.11). The assumption of periodicity immediately makes the simulation of such a system tractable because equivalent atoms in each of the copies behave identically and so do not need to be treated distinctly during a simulation. Because of its importance the PBC method is the one that we shall use and it will be described in more detail in the remaining sections of this chapter.

The PBC method works well in many cases but it has some drawbacks. First, an order is imposed that would not normally be present because the system is assumed to be periodic. This can lead to artefactual results for structural and dynamical properties obtained from simulation. Second, the method is often expensive, especially for large molecules. To see this, consider a large molecule, such as a protein, in solution (see, for example, Figure 2.2). It takes only a little reflection to realize that, to immerse the molecule fully in the solvent, a volume of solvent much larger than that of the molecule itself will be required. This adds considerably to the size of the system and means that most of the time during a simulation will be spent dealing with the solvent rather than with the solvated molecule which is the object of principal interest.

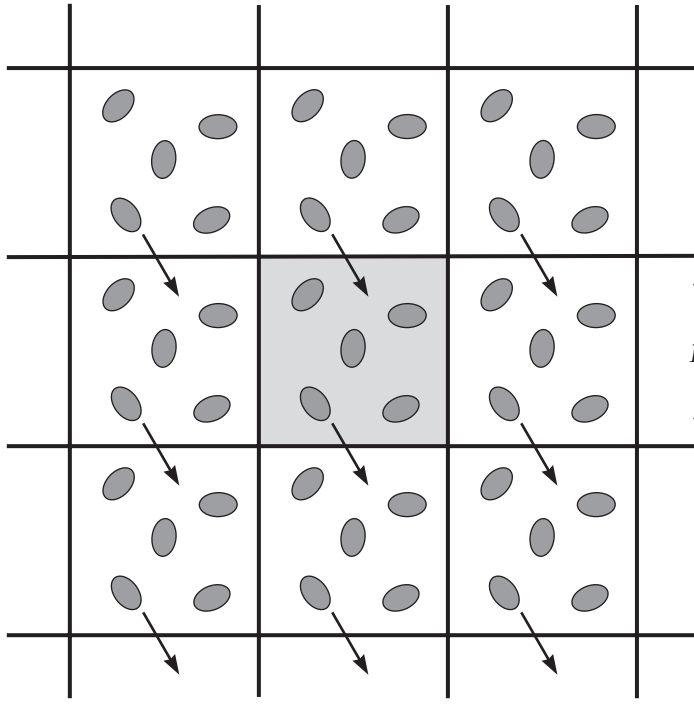


Fig. 10.11. An example of the PBC approximation in two dimensions in which the central, square, shaded box of side L is replicated in both dimensions.

As a result of the limitations of the PBC method, alternative techniques have been sought. One series of methods has been developed primarily to mimic effects of solvent on molecules. These *implicit solvent methods* replace the *explicit* description of the solvent molecules of the PBC approach by simpler models. It is usual to use different strategies to model the electrostatic and non-polar (Lennard-Jones) interactions between the solute and the solvent because these interactions are different in nature.

For the electrostatic interactions a common approach is to use a *reaction field model* in which the solvent is replaced by a medium that has a dielectric constant that is appropriate to the solvent being modeled. The solute is assumed to be located in a cavity (of a different dielectric – often unity) within the continuum. The solute's charge distribution polarizes the solvent which in turn acts back upon the solute's charges with a reaction field. In this model the energy of the interaction between the solute and solvent is determined by first solving the *Poisson–Boltzmann equation* for the electric potential, ϕ, in the system. This equation has the form

$$\nabla^{\mathrm{T}}(\epsilon\nabla\phi) - \epsilon\kappa^2 \sinh\phi + 4\pi\rho = 0 \tag{10.8}$$

where ϵ is the dielectric constant, κ is the *Debye–Hückel parameter*, which is related to the type and concentration of ions in the solution, and ρ is the charge distribution in the system. It is to be noted that all these parameters, as well as the potential, are functions of position. If the κ function is everywhere zero the Poisson–Boltzmann equation reduces to the *Poisson equation*. Once the potential has been obtained, the total electrostatic energy for the system, $\mathcal{V}_{\text{el}}$, can be calculated. For a simple point-charge model of the solute's charge distribution and for the case in which the potential is small (so that the $\sinh\phi$ term in Equation (10.8) can be *linearized* to ϕ) this is

$$\mathcal{V}_{\text{el}} = \frac{1}{2}\sum_{i=1}^{N} q_i \phi(\boldsymbol{r}_i) \tag{10.9}$$

Although the solution to the Poisson–Boltzmann equation appears to give good results for solvation energies, it is expensive to solve. As a result it has been employed almost exclusively to calculate the potentials and the energies of single structures and not in minimization or molecular dynamics calculations.

For these longer types of calculations other more approximate, ad hoc methods have been developed. The simplest include models that reduce the charges on charged groups to account for the screening of charges by the solvent and the use of a dielectric constant other than unity in the calculation of the electrostatic interactions (Equation (10.1)). This can be a constant with a larger value or it can be a function of the distance between the particles. Thus, for example a *distance-dependent dielectric function*, $\epsilon(r) \propto r$, has often been used in the simulation of biomacromolecules. These methods are, however, of dubious accuracy and are better avoided if viable alternatives are available. More precise methods, albeit still approximate, are based upon the *Born expression* for the electrostatic solvation energy of a charged sphere in a medium of a different dielectric constant. This energy, ΔG_{Born}, is

$$\Delta G_{\text{Born}} \propto \frac{q^2}{2a}\left(\frac{1}{\epsilon_{\text{o}}} - \frac{1}{\epsilon_{\text{i}}}\right) \tag{10.10}$$

where q is the charge on the sphere, a is its radius and ϵ_{i} and ϵ_{o} are the dielectric constants inside and outside the sphere, respectively. To account for interactions between charged spheres, Equation (10.10) can be generalized to

$$\Delta G_{\text{solv}} \propto \left(\frac{1}{\epsilon_{\text{o}}} - \frac{1}{\epsilon_{\text{i}}}\right)\sum_{i=1}^{N}\sum_{j=1}^{N}\frac{q_i q_j}{f(r_{ij}, a_i, a_j)} \tag{10.11}$$

where $f(r_{ij}, a_i, a_j)$ is a function such that $f \to 1/r_{ij}$ as $r_{ij} \to \infty$ (i.e. when the spheres are very far apart and do not overlap) and $f \to a_i$ or a_j as $r_{ij} \to 0$ (i.e. when the spheres coalesce). The accuracy of the representation depends upon

the form of the function chosen for f and upon the way in which the *effective Born radii*, a_i, are calculated. These will obviously be different depending upon whether the atom is completely buried in the interior of a molecule (and so has no exposure to solvent) or is at the surface.

The continuum dielectric models and their equivalents account for the electrostatic interactions between solute and solvent. To treat the non-polar interactions (dispersion and repulsion) other models are necessary. The commonest is the *surface free energy description*, which relates the interaction energy to the accessible surface area of the molecule. Like the Born model, it has been used mostly for solvation studies. The non-polar energy, $\mathcal{V}_{\text{np}}$, is

$$\mathcal{V}_{\text{np}} = \sum_{i=1}^{N} \gamma_i A_i \tag{10.12}$$

where A_i is the surface area of atom i that is accessible to solvent and γ_i is a constant that depends upon the chemical type of the atom and must be parametrized to reproduce the non-polar surface free energy. At the present time the combination of *generalized Born* and *solvent-accessible surface area* models probably provides the most accurate and cost-effective implicit solvation models for molecular simulations. One of the more widely used MM models of this type is the one developed by W. C. Still and co-workers.

An alternative class of methods mixes elements of explicit and implicit models. These methods will be more reliable than purely continuum approaches when a correct description of the interaction between solute and solvent depends upon the solvent's molecular structure. Examples of these methods include the various *boundary approximations* which select a small, often spherical, region of the system of interest and treat that with an atomic level model. The remainder of the system is removed and replaced by a *boundary potential*. In the simplest cases the potential can be neglected or it can be a hard wall, but in more sophisticated algorithms it will have a form that accounts for non-polar interactions as well as reaction-field-type terms. Taking the solvation of a solute as an example again, the solute molecule would be placed at the centre of the sphere, surrounded with a few shells of solvent molecules and then the boundary potential would be placed outside this. These methods can reduce quite substantially the time required for a simulation with explicit solvent molecules, but the choice of boundary potential is not always evident and can significantly affect the results obtained.

10.5 Periodic boundary conditions

In the PBC approximation an infinite system is constructed by periodically replicating a finite system. For the infinite system to be continuous, the finite system

must be of a sufficiently regular shape that it can fill space when it is copied. The most common option, by far, in three dimensions is to use finite systems that are *cubic*. Also common are *orthorhombic* boxes whose angles are all right angles but whose sides are of different lengths. Other shapes that are possible are *triclinic*, *hexagonal*, *truncated octahedral* and various sorts of *dodecahedral* boxes. These shapes are geometrically more complicated but they can be required when studying certain types of system, such as crystals.

In pDynamo, use of the PBC approximation first of all requires that the type of periodic box and its shape be defined. The presence of periodicity imposes a symmetry on a system (in this case translational symmetry) and so these operations are handled by various symmetry-related classes. Although we do not make use of symmetry in this book, other than its role in PBC simulations, a system's symmetry properties are fundamental and rank in importance with, for example, details of its atomic composition.

A range of box shapes are permitted in pDynamo which are all implemented as subclasses of the class `CrystalClass`. The only one that we shall need is one for a cubic box, `CrystalClassCubic`, whose definition is:

Class `CrystalClassCubic`

A class to represent cubic translational symmetry.

Constructor

Construct an instance of `CrystalClassCubic`.

Usage: `new = CrystalClassCubic ( )`
`new` is the new instance of `CrystalClassCubic`.

Several extensions of the `System` class are needed for symmetry handling. A limited definition of these extensions that is sufficient for the examples in this book is:

Class `System`

Symmetry-related methods and attributes.

Method `DefineSymmetry`

Define the symmetry properties of a system.

Usage: `system.DefineSymmetry(crystalclass = None, a = None)`

`crystalclass` defines the system's crystal class. It must be an instance of `CrystalClass` or `None`.

`a` is a floating-point number that gives the length of the side of a cubic box. The edges of the box will be along the Cartesian axes.

`system` is the instance of `System` for which symmetry is being defined.

Remarks: The definition given above is only appropriate for assignment of cubic symmetry. A much more extensive set of keywords is available for other symmetries.

Attributes

`symmetry` contains the part of the system's symmetry definition that is independent of physical variables that specify such quantities as box size. For a cubic system, this attribute would include an instance of `CrystalClassCubic`.

`symmetryparameters` holds the lengths, angles and other variables needed to complete the symmetry definition. For a cubic system, this attribute would consist of the length of the box side.

Remarks: The role of these two attributes resembles, in many ways, the roles played by the attributes that define the connectivity for a system, particularly `atoms` and `bonds`, and its coordinates, `coordinates3`. The former encapsulate a conceptual model of the system whereas the latter permit its physical realization in three-dimensional space.

Once a symmetry has been assigned to a system, all simulations are performed in the normal way. No extra classes, method calls or keywords are needed as pDynamo automatically detects that symmetry is present and adjusts its algorithms accordingly. The largest difference between vacuum (or non-PBC) and PBC calculations arises in the evaluation of the non-bonding interaction energy. If we focus, for the moment, upon the truncation method implemented in the class `NBModelABFS`, extra work arises because atoms in the central box no longer interact just amongst themselves but also with atoms of the neighbouring boxes or *images* that are within cutoff range.

As in the vacuum case, the `NBModelABFS` class employs a two-step procedure to determine these extra interactions. The first step is to generate lists of potential non-bonding interactions between image and central box atoms. This is done with

the cutoff, r_{list}, and involves a preliminary search to locate all images that are within the cutoff distance of the central box before the atom-pair lists themselves are created. The second step is to evaluate the interaction energies between the images and central box using the atom lists and the interaction cutoff, r_{off}. As before, the use of two cutoffs, r_{list} and r_{off}, ensures that the non-bonding image lists need not be re-created every time an energy is calculated but only when the atoms have moved by a distance of the order of $r_{\text{list}} - r_{\text{off}}$.

To finish this section, we note that a common approximation that is made when combining truncation methods with PBCs is the *minimum image convention*. The major assumption of this convention is that a particle will interact with, at most, the nearest copy of another particle in the system. This makes the method relatively inexpensive and easy to implement but it means that the cutoff distance for truncation of the non-bonding interactions must be less than or equal to half the length of the side of the periodic box. This imposes a serious (lower) limit to the size of the system that can be studied especially if a reasonable non-bonding cutoff is desired. Because of these disadvantages, the class `NBModelABFS` does not employ the minimum image convention but the more general strategy described above in which there is no restriction upon how many images the atoms of the central box can interact with.

10.6 Example 20

As an example of a simulation with PBCs, we study a liquid system consisting of a cubic box of 216 ($\equiv 6^3$) water molecules. The program is:

```
1  """Example 20."""
2
3  from Definitions import *
4
5  # . Define the box side (in Angstroms).
6  BOXSIDE = 18.641
7
8  # . Define the MM and NB models.
9  mmmodel = MMModelOPLS ( "booksmallexamples" )
10 nbmodel = NBModelABFS ( )
11
12 # . Generate the solvent.
13 solvent = MOLFile_ToSystem ( \
                os.path.join ( molpath, "water216_cubicbox.mol" ) )
14 solvent.DefineSymmetry ( crystalclass = CrystalClassCubic ( ), \
                           a           = BOXSIDE               )
```

```
15 solvent.DefineMMModel  ( mmmodel )
16 solvent.DefineNBModel  ( nbmodel )
17 solvent.Summary ( )
18
19 # . Save the system for later use.
20 XMLPickle ( \
          os.path.join ( scratchpath, "water216_cubicbox.xpk" ), \
                                                        solvent )
21
22 # . Define a random number generator in a given state.
23 rng = Random ( )
24 rng.seed ( 491831 )
25
26 # . Equilibration.
27 VelocityVerletDynamics_SystemGeometry (
                          solvent,                                \
                          logfrequency              =        500, \
                          rng                       =        rng, \
                          steps                     =       5000, \
                          timestep                  =      0.001, \
                          temperaturescalefrequency =        100, \
                          temperaturescaleoption    = "constant", \
                          temperaturestart          =      300.0  )
28
29 # . Data-collection.
30 trajectory = SystemGeometryTrajectory ( \
          os.path.join ( scratchpath, "water216_cubicbox.trj" ), \
                                             solvent, mode = "w" )
31 VelocityVerletDynamics_SystemGeometry (
                            solvent,                              \
                            logfrequency =                  500, \
                            steps        =                10000, \
                            timestep     =                0.001, \
                            trajectories = [ ( trajectory, 50 ) ] )
```

This program is very similar to that of Example 16 of Section 9.3. Major differences and other points to note are as follows:

Lines 13–14 define the cubic system of 216 water molecules. *Line 14* assigns the symmetry to the system and sets the dimension of the box to be the variable `BOXSIDE`. The value of 18.641 Å on *line 6* gives a volume for the system that is consistent with the experimental value for the density of water at 300 K which is about 996 kg m^{-3}.

The use of this combination of two commands to define a system with translational symmetry has been done primarily for illustrative purposes.

In practice, it will be more usual to read a file in a format that includes all the necessary data, including the symmetry definition. Standard formats that fall into this category include the PDB format described in Section 2.4.3. Another convenient alternative is to employ files in pDynamo's XPK format. Ways in which XPK files can be generated for periodic systems are detailed in Appendix 3.

Line 20 saves the complete definition of the system to an XPK file. This will be needed by the examples in later chapters.

Line 27 performs a short dynamics to equilibrate the system. In this program no minimization or heating is performed because the coordinates in the MOL file correspond to a system that has already been partially equilibrated (in a previous molecular dynamics simulation).

Line 30 creates a trajectory object in which geometrical data about the system will be saved. As the system now has symmetry this will include not only atomic coordinates but also the variable from the attribute `symmetryparameters` that determines the size of the cubic box.

Line 31 performs a 10 ps dynamics simulation. Data are saved to the trajectory every 50 steps, making 201 frames in total. A fuller discussion of ways in which trajectories like this can be analysed will be left to the next chapter.

The most notable thing about this program will become apparent only when it is executed. Up to now the examples we have considered have needed relatively low resources in terms of memory or CPU time. In contrast, this program performs a simulation of 648 atoms and at each timestep about 3×10^5 non-bonding interactions are processed. It demands a lot more time than the previous examples, although it should run in a few hours on a reasonably fast personal computer or workstation.

10.7 Ewald summation techniques

In the previous sections we discussed methods for the calculation of non-bonding interactions that truncate the interactions beyond a certain cutoff distance. This is an approximation that could (and does!) have important consequences for the behaviour of a system during a molecular simulation. It would obviously be better to have methods that allow the non-bonding interactions to be calculated fully for a periodic system. Such approximations exist, one class of which is called *Ewald lattice summation techniques* in honour of one of their originators. These methods were originally developed for studying such systems as ionic crystals, but algorithmic advances in the 1990s (see the next section) have meant that they are

now routinely applied for simulations of periodic molecular and macromolecular systems.

The derivation of the Ewald summation formulae involves a number of subtleties and so only a brief description will be presented here to give a flavour of what is required. The literature on Ewald and related techniques is large, but the exposition below mostly follows that due to D. E. Williams. Consider a periodic atomic system within which the particles interact with a potential of the form $\lambda_i\lambda_j/r_{ij}^p$. This is appropriate for the Coulomb interaction and both for the repulsive and for the dispersive parts of the Lennard-Jones interaction if geometrical mean combination rules are used. The non-bonding interaction energy of a single box of the periodic system will be the sum of the interactions between the atoms within the box and between its atoms and those of all the remaining boxes. The expression for this sum, S_p, is

$$S_p = \frac{1}{2}\sum_{\boldsymbol{n}}{}' \sum_i \sum_j \frac{\lambda_i\lambda_j}{|\boldsymbol{r}_i - \boldsymbol{r}_j + \boldsymbol{t}_{\boldsymbol{n}}|^p} \tag{10.13}$$

where the sums over $\boldsymbol{n}$, i and j indicate summations over all periodic boxes (including $\boldsymbol{n} = \boldsymbol{0}$, the central box), the atoms in box $\boldsymbol{n}$ and the atoms in the central box, respectively. The prime on the summation over boxes means that the self-interaction (i.e. with $i = j$ and $\boldsymbol{n} = \boldsymbol{0}$) is omitted because this is divergent. The vector $\boldsymbol{t}_{\boldsymbol{n}}$ is the vector that indicates the displacement between the interacting boxes. If the sides of the boxes are along the Cartesian axes it will have the form $(n_x a, n_y b, n_z c)$, where n_x, n_y and n_z are the integer components of the vector $\boldsymbol{n}$ and a, b and c are the lengths of the box in the directions of the x, y and z axes, respectively. For a cubic box $a = b = c$.

The trick common to the Ewald summation techniques is to employ a *convergence function*, $\phi(r)$, to split the sum given in Equation (10.13) into two parts. This function has the property that it decays rapidly to zero as r increases and takes the value 1 for $r = 0$. Using it, the sum S_p can be rewritten as

$$S_p = S_p^{(1)} + S_p^{(2)} \tag{10.14}$$

$$S_p^{(1)} = \frac{1}{2}\sum_{\boldsymbol{n}ij}{}' \frac{\lambda_i\lambda_j\phi(r_{\boldsymbol{n}ij})}{r_{\boldsymbol{n}ij}^p} \tag{10.15}$$

$$S_p^{(2)} = \frac{1}{2}\sum_{\boldsymbol{n}ij}{}' \frac{\lambda_i\lambda_j\left(1 - \phi(r_{\boldsymbol{n}ij})\right)}{r_{\boldsymbol{n}ij}^p} \tag{10.16}$$

where the shorthand $r_{\boldsymbol{n}ij}$ has been used to denote $|\boldsymbol{r}_i - \boldsymbol{r}_j + \mathbf{t}_{\boldsymbol{n}}|$. Because of the properties of the convergence function the sum $S_p^{(1)}$ involves only short-range interactions and so it can be calculated using techniques similar to those described

in Section 10.5. The cost of this summation scales as $O(N)$ where N is the number of atoms in the central box. The second sum involves long-range interactions but it can be evaluated if its *Fourier transform* is taken and the sum performed in *reciprocal space*. Details of this transformation will not be given here but the final form of the expression depends crucially upon the form chosen for ϕ. It is usual to follow a suggestion of B. R. A. Nijboer and F. W. De Wette and use

$$\phi(r) = \frac{\Gamma(p/2, \kappa^2 r^2)}{\Gamma(p/2)} \tag{10.17}$$

where κ is an arbitrary parameter, and $\Gamma(x)$ and $\Gamma(x, y)$ are the *complete* and *incomplete gamma functions*, respectively:

$$\Gamma(x) = \int_0^\infty t^{x-1} \exp(-t)\, \mathrm{d}t \tag{10.18}$$

$$\Gamma(x, y) = \int_y^\infty t^{x-1} \exp(-t)\, \mathrm{d}t \tag{10.19}$$

The sums S_p for $p > 3$ are *absolutely convergent*, which means that they converge no matter what the values of λ and no matter in which order the summation is done. Using the form for ϕ given in Equation (10.17) for $p = 6$ results in an expression that is suitable for the evaluation of the dispersive part of the Lennard-Jones interaction. It is

$$\begin{aligned} S_6 = {} & \frac{1}{2} {\sum_{\boldsymbol{n}ij}}' \frac{\lambda_i \lambda_j}{r_{\boldsymbol{n}ij}^6} \left[1 + (\kappa r_{\boldsymbol{n}ij})^2 + \frac{(\kappa r_{\boldsymbol{n}ij})^4}{2} \right] \exp[-(\kappa r_{\boldsymbol{n}ij})^2] \\ & + \frac{(\sqrt{\pi}\kappa)^3}{6V} \sum_{\boldsymbol{k}} \left\{ [1 - 2(k/2\kappa)^2] \exp[-(k/2\kappa)^2] + 2(k/2\kappa)^3 \sqrt{\pi}\, \mathrm{erfc}[k/(2\kappa)] \right\} \\ & \times \left\{ \left[\sum_i \lambda_i \cos(\boldsymbol{k}^{\mathrm{T}} \boldsymbol{r}_i) \right]^2 + \left[\sum_i \lambda_i \sin(\boldsymbol{k}^{\mathrm{T}} \boldsymbol{r}_i) \right]^2 \right\} \\ & - \frac{\kappa^3}{12} \sum_i \lambda_i^2 \end{aligned} \tag{10.20}$$

In this equation, V is the volume of a periodic box ($V = abc$) and erfc is the *complementary error function* ($\mathrm{erfc}(x) = \Gamma(\frac{1}{2}, x^2)/\sqrt{\pi}$). The second summation is over a set of vectors, $\boldsymbol{k}$, which are called the *reciprocal space vectors* or the $\boldsymbol{k}$ *vectors*. For an orthorhombic box with the box sides along the Cartesian axes, they take the form $\boldsymbol{k} = 2\pi(n_x/a, n_y/b, n_z/c)$. This sum will be rapidly convergent as k ($= |\boldsymbol{k}|$) increases in size due to the presence of the exp and erfc terms.

The sums S_p for $p \leq 3$ are only *conditionally convergent*, which means that the value of the sum can depend upon the way in which the summation is done. In

addition, the sum will not converge at all unless the condition $\sum_i \lambda_i = 0$ is satisfied. For the Coulomb interaction this implies that the simulation box must have no nett charge. The derivation of the summation formula is more complicated in this case, but it is

$$\begin{aligned} S_1 = & \frac{1}{2}\sum_{\boldsymbol{n}ij}{}' \frac{\lambda_i \lambda_j \operatorname{erfc}(r_{\boldsymbol{n}ij})}{r_{\boldsymbol{n}ij}} \\ & + \frac{2\pi}{V}\sum_{\boldsymbol{k}\neq\boldsymbol{0}} \frac{\exp[-(k/2\kappa)^2]}{k^2} \left\{ \left[\sum_i \lambda_i \cos(\boldsymbol{k}^{\mathrm{T}}\boldsymbol{r}_i)\right]^2 + \left[\sum_i \lambda_i \sin(\boldsymbol{k}^{\mathrm{T}}\boldsymbol{r}_i)\right]^2 \right\} \\ & - \frac{\kappa}{\sqrt{\pi}}\sum_i \lambda_i^2 \\ & + J(\boldsymbol{\mu}) \end{aligned} \tag{10.21}$$

It is to be noted that the sum over the $\boldsymbol{k}$ vectors in this expression explicitly excludes the $\boldsymbol{k} = \boldsymbol{0}$ term because this term is divergent. There is also an extra term in the sum, $J(\boldsymbol{\mu})$, often called the *surface correction* term, which is a function of the dipole moment, $\boldsymbol{\mu}$, of the periodic box. It turns out, when deriving this term, that it is necessary to specify various macroscopic boundary conditions for the ensemble of periodic boxes for which the electrostatic energy is being computed. In particular, the shape of the macroscopic crystal and the dielectric constant of the medium surrounding the crystal are important. If the medium has an infinite dielectric constant the term disappears, which corresponds to *tin-foil boundary conditions*. For a spherical crystal in vacuum (with a dielectric constant of 1) the expression is

$$J(\boldsymbol{\mu}) = \frac{2\pi}{3V}|\boldsymbol{\mu}|^2 \tag{10.22}$$

where the box's dipole-moment vector is

$$\boldsymbol{\mu} = \sum_{i=1}^{N} \lambda_i \boldsymbol{r}_i \tag{10.23}$$

The first three terms on the right-hand sides of the expressions for the energies in Equations (10.20) and (10.21) are often called the *real space*, the *reciprocal space* and the *self-energy* terms, respectively. The derivatives of all these terms with respect to the atomic positions are straightforward to determine by direct differentiation.

There is an extra complication that is not apparent in the formulae for the non-bonding energies given in Equations (10.20) and (10.21). Both these formulae apply to the case in which *all* the interactions between particles

are calculated. As we have seen, this is not the case for empirical force fields, which often exclude certain interactions, notably those between bonded atoms. For these excluded interactions (the 1–2, 1–3 and sometimes the 1–4 terms) it is necessary to calculate their energies using the normal expressions (i.e. without the convergence functions) and subtract them from the sums in Equations (10.20) and (10.21). Care should also be taken to ensure that the 1–4 interactions are treated properly, for these are not always fully excluded but sometimes only partially so (such as with the OPLS force field used in this book).

To calculate the non-bonding interaction energy for a periodic system using the Ewald algorithm, we introduce a third NB model class `NBModelEwald`. It has the same structure as the other classes that calculate the non-bonding energy and, hence, can be used interchangeably with them. The major difference is that additional keywords can be supplied to the method `SetOptions`. The class definition is:

Class `NBModelEwald`

A class to calculate non-bonding interactions with an Ewald summation technique.

Constructor

Construct an instance of the `NBModelEwald` class.

Usage:
```
new = NBModelEwald ( )
```
`new` is the new instance of `NBModelEwald`.

Method `SetOptions`

Define the values of various options necessary for the Ewald algorithm.

Usage:
```
nbmodel.SetOptions ( innercutoff  =  8.0,
                     kappa        =  0.2,
                     listcutoff   =  13.5,
                     nmaximum     =  0,
                     outercutoff  =  12.0,
                     QTINFOIL     =  True )
```
`kappa` is an argument that sets the value of the parameter κ to be used in the Ewald summation procedure (see Equation (10.17)).

`nmaximum` gives the maximum and minimum values of the components of the vector, $\boldsymbol{n}$, to use in the calculation of the $\boldsymbol{k}$ vectors. `nmaximum` can be an integer in which case this value will be

applied to all three dimensions or it can be a sequence of three integers that give the limits for each dimension separately. Supposing that the value of `nmaximum` for a given dimension is $n_{\max}$, the $\boldsymbol{k}$-vector sum in this dimension would run from $-n_{\max}$ to $+n_{\max}$ and include $2n_{\max}+1$ terms.

`QTINFOIL` is a Boolean that determines whether or not tin-foil boundary conditions are to be used in the calculation of the electrostatic interactions. This implies that the surface term of Equation (10.22) is not calculated if the argument is `True` but is included if the argument is `False`.

`nbmodel` is an instance of `NBModelEwald`.

Remarks: The keywords that set the values of the various cutoffs are the same as for the class `NBModelABFS` and behave similarly. `listcutoff` and `outercutoff` are only employed for the evaluation of the short-range sums ($S_p^{(1)}$ in Equation (10.15)) whereas `innercutoff` is also required for those parts of the Lennard-Jones potential that are being calculated with a truncation approximation.

Two points should be made about the implementation of the Ewald method by this class. First, the electrostatic interactions are always calculated using an Ewald technique (Equation (10.21)). The dispersive part of the Lennard-Jones interactions will also be calculated by the Ewald method (Equation (10.20)) if geometric combination rules are used for the atomic radii (as in the OPLS force field). Otherwise, these will be calculated using a truncation approximation. The interactions due to the repulsive, r^{-12}, part of the Lennard-Jones potential are always calculated using a truncation approximation as it assumes, which should be reasonable in most cases, that the value of the interaction cutoff, r_{off}, is large enough (> 10 Å) that all the interactions that are left out in this way will be negligible in size.

Second, the precision and speed of the Ewald algorithm depend upon a careful balancing of the parameters κ, r_{off} and $n_{\max}$. A large value of κ reduces the number of real space interactions but means that the number of reciprocal space terms that need to be evaluated should be increased to obtain equivalent accuracy in the energy and forces. Likewise a small value of κ increases the number of real space terms but permits a decrease in the size of the reciprocal space summation. It is possible to show, although we do not do so here, that the Ewald algorithm scales as $O(N^{3/2})$ if the optimal set of parameters is chosen so as to have calculated energies of a given precision.

10.8 Fast methods for the evaluation of non-bonding interactions

Owing to the importance of the non-bonding interactions, especially the long-range electrostatic interactions, much research has gone into algorithms that can be used to evaluate these terms exactly in as efficient a way as possible. The aim is to obtain methods that scale linearly with the size of the system, i.e. as $O(N)$, rather than as the $O(N^2)$ of the direct summation techniques or the $O(N^{3/2})$ of the Ewald methods. Such methods are not used in this book but they are becoming standard for many types of molecular simulation. We consider algorithms of two widely used classes of technique below. Most, although not all, of them exist in versions that can handle interactions involving both discrete and continuous charge distributions. The former are normally easier to implement and are appropriate for use with MM energy functions whereas the latter are necessary for QC potentials.

The *fast multipole methods* rely on the fact that the electrostatic potential at long distance due to a charge distribution can be well approximated as a limited *multipole expansion*, i.e. as a charge, dipole, quadrupole, octupole, etc. Thus, instead of calculating the electrostatic interaction between distant charge distributions directly, it can be approximated to within a certain precision as an interaction between their respective multipoles. Algorithms using this principle divide space into a hierarchy of cells of different sizes within each of which the multipole expansions due to the charge distributions are evaluated. The interactions between cells that are near to each other are calculated directly whereas those that are at longer range are determined using the multipole approximation. L. F. Greengard and V. Rokhlin were the first to do systematic work upon these approaches and it was they who developed an algorithm, appropriate for both vacuum and periodic systems, that scales as $O(N)$. These methods are quite complex to implement because they require a judicious grouping of cells when calculating the long-range interactions to ensure that linear scaling is preserved. Nevertheless, they are employed for simulations both with MM and with *ab initio* QC potentials.

The second class of techniques is the *fast Ewald methods* that work for periodic systems. The gain in efficiency arises from the evaluation of the long-range interactions as the short-range terms are handled in the same way as in the original Ewald method. Although details vary, a typical scheme for treatment of the long-range energy involves the following steps: (i) a grid or mesh is introduced into the simulation box; (ii) a representation of the charge distribution in the box is created on the grid; (iii) the electrostatic potential at each mesh point is calculated by solving the Poisson equation either in reciprocal space with a *fast Fourier transform* (FFT) technique or in real space, often with a *multigrid* method; (iv) the potentials on the atoms are generated from the mesh point values; and (v) the energy is calculated from the potentials on the atoms using an expression such

as that in Equation (10.9). The efficiency of the algorithm arises from step (iii) as solution of the Poisson equation scales as $O(n \ln n)$ for FFT and as $O(n)$ for multigrid solvers where n is the number of mesh points. This number is, in turn, proportional to the number of atoms in the system, N, for a given mesh size. One of the most popular fast Ewald methods is the *particle mesh Ewald* method introduced by T. Darden, D. York and L. Pedersen. Currently, it appears to be the method of choice for the simulation with MM potentials of condensed phase molecular systems consisting of 10^4–10^5 particles.

Exercises

10.1 In Section 10.3 the non-bonding energies were calculated as a function of the cutoff distance for the non-bonding interactions. Perform a similar analysis but look at properties other than energies. For example, how do the minimized structures of a molecule change as the cutoff distance is increased and how are the forms of the normal modes altered? Is the cutoff approximation a reasonable one?

10.2 Repeat the simulation of Example 20 but this time using the other NB energy models. What happens when the class `NBModelFull` is used and why? With the Ewald technique, try different combinations of parameters (κ, r_{off} and n_{max}) to see how the values of the non-bonding interaction energy change. Which set of values permits the most efficient (i.e. fastest) simulation time for a given precision in the energy?

11

Molecular dynamics simulations II

11.1 Introduction

In Chapter 9 we saw how to perform molecular dynamics simulations, although they were not very sophisticated because there was no means of including the effect of the environment. Ways of overcoming this limitation were introduced in the last chapter, in which we discussed methods for calculating the energy and its derivatives for a system within the PBC approximation. As an example, a molecular dynamics simulation for a periodically replicated cubic box of water molecules was performed. Here, we shall start by describing in more detail the type of information that can be computed from molecular dynamics trajectories and also by indicating how the quality of that information can be assessed. Later we shall talk about more advanced molecular dynamics techniques including those that allow simulations to be carried out in various thermodynamic ensembles and those that permit the calculation of free energies.

11.2 Analysis of molecular dynamics trajectories

We have seen how to generate trajectories of data for a system, either in vacuum or with an environment, with the molecular dynamics technique. The point of performing a simulation is, of course, that we want to use these data to calculate some interesting quantities, preferably those which can be compared with experimentally measured ones. The aim of this section is to give a brief overview of some of the techniques that can be used to analyse molecular dynamics trajectories and some of the types of quantities that can be calculated.

In Section 9.4 we defined a time series for a property as a sequence of values for the property obtained from successive frames of a molecular dynamics trajectory. Of the many possible statistical quantities that can be calculated from a time series, we shall focus on three types. These are averages and fluctuations, which we met in Section 9.4, and *time correlation functions*. Let $\mathcal{X}$ be the property,

$\mathcal{X}_n$ the nth value of the property in the time series and n_t the total number of elements in the series. The average of the property, denoted by $\langle \mathcal{X} \rangle$, is

$$\langle \mathcal{X} \rangle = \frac{1}{n_t} \sum_{n=1}^{n_t} \mathcal{X}_n \tag{11.1}$$

The fluctuation is the average of the square of the deviation of the property from its average. If the deviation is denoted by $\delta\mathcal{X} = \mathcal{X} - \langle \mathcal{X} \rangle$, the fluctuation is

$$\begin{aligned} \langle (\delta\mathcal{X})^2 \rangle &= \langle \mathcal{X}^2 \rangle - \langle \mathcal{X} \rangle^2 \\ &= \frac{1}{n_t} \sum_{n=1}^{n_t} \mathcal{X}_n^2 - \langle \mathcal{X} \rangle^2 \end{aligned} \tag{11.2}$$

The RMS deviation of the property, denoted $\sigma(\mathcal{X})$, is the square root of the fluctuation, i.e. $\sigma^2(\mathcal{X}) = \langle (\delta\mathcal{X})^2 \rangle$.

The method of calculation of the averages and fluctuations of a time series is immediately obvious by inspection of Equations (11.1) and (11.2). The calculation of correlation functions is more complicated because they are functions of time. The *autocorrelation* function for the property $\mathcal{X}$ is denoted by $\mathcal{C}_{\mathcal{X}\mathcal{X}}(t)$ and has the expression

$$\begin{aligned} \mathcal{C}_{\mathcal{X}\mathcal{X}}(t) &= \langle \delta\mathcal{X}(t)\, \delta\mathcal{X}(0) \rangle \\ &= \langle \mathcal{X}(t)\mathcal{X}(0) \rangle - \langle \mathcal{X} \rangle^2 \end{aligned} \tag{11.3}$$

It is common to normalize the function by dividing it by the fluctuation of the property. The normalized function, $\hat{\mathcal{C}}_{\mathcal{X}\mathcal{X}}(t)$, is

$$\hat{\mathcal{C}}_{\mathcal{X}\mathcal{X}}(t) = \mathcal{C}_{\mathcal{X}\mathcal{X}}(t)/\sigma^2(\mathcal{X}) \tag{11.4}$$

A *cross-correlation function* for two different properties, $\mathcal{X}$ and $\mathcal{Y}$, can be defined as

$$\begin{aligned} \mathcal{C}_{\mathcal{X}\mathcal{Y}}(t) &= \langle \delta\mathcal{X}(t)\, \delta\mathcal{Y}(0) \rangle \\ &= \langle \mathcal{X}(t)\mathcal{Y}(0) \rangle - \langle \mathcal{X} \rangle \langle \mathcal{Y} \rangle \end{aligned} \tag{11.5}$$

It can be normalized in the same way as the autocorrelation function by dividing it by the product of the RMS deviations of properties $\mathcal{X}$ and $\mathcal{Y}$, i.e. by $\sigma(\mathcal{X})\sigma(\mathcal{Y})$.

Calculation of a correlation function relies upon the fact that it obeys the *stationarity condition*:

$$\langle \mathcal{X}(t)\mathcal{X}(0) \rangle = \langle \mathcal{X}(\tau + t)\mathcal{X}(\tau) \rangle \tag{11.6}$$

which implies that, for a given time, t, products $\mathcal{X}(\tau + t)\mathcal{X}(\tau)$, for all possible values of τ, will contribute to the average used to calculate $\mathcal{C}_{\mathcal{X}\mathcal{X}}(t)$. For the autocorrelation function, this translates into a discretized equation of the form

$$\langle \mathcal{X}(t_n)\mathcal{X}(0)\rangle = \frac{1}{n_{\max}} \sum_{i=1}^{n_{\max}} \mathcal{X}_i \mathcal{X}_{i+n} \tag{11.7}$$

where t_n is the time corresponding to the interval between n elements in the time series and $n_{\max} = n_t - n$ is the number of intervals used to calculate the average. Note that, if $n = 0$, the expression reduces to that for the fluctuation given in Equation (11.2), as it should. Note too that, as n gets large, $n_{\max}$ becomes small, so that for $n \simeq n_t$ there are very few intervals that can be used for the calculation. In practice, this means that, for large n, the values of the calculated correlation function are unreliable because there are not enough products contributing to the average to get a statistically significant result.

The importance of these three statistical quantities is that they are necessary for many of the formulae, derivable from statistical mechanics, that allow the microscopic properties calculated from a simulation and the macroscopic quantities that can be obtained experimentally to be linked. We have already come across some of these formulae, notably the one equating the temperature of the system to the average of its kinetic energy (Equation (9.14)) and those that permit the calculation of some thermodynamic quantities for a molecule within the rigid-rotor, harmonic oscillator approximation (Section 8.6). Another important relation involving an average, which we shall discuss in detail in Section 11.4, allows the pressure for a condensed phase system to be obtained from a simulation. Examples of expressions involving fluctuations are those that relate the specific heats (either at constant volume or at constant pressure) to fluctuations in the potential and kinetic energies. Time correlation functions are important because they are fundamental to the derivation of formulae that permit *transport coefficients* for a system to be calculated. Examples include the *diffusion coefficient*, the *bulk* and *shear viscosities* and the *thermal conductivity*.

We shall consider the formulae for two properties in detail because these will be calculated in the example program of the next section. The first property is the diffusion coefficient for a species i, D_i, which is proportional to the time integral of its velocity autocorrelation function:

$$D_i = \frac{1}{3}\int_0^\infty \mathrm{d}t\, \langle \boldsymbol{v}_i^{\mathrm{T}}(t)\boldsymbol{v}_i(0)\rangle \tag{11.8}$$

This equation can be integrated by parts to give the following expression which is valid at long times, t:

$$6tD_i = \langle (\boldsymbol{r}_i(t) - \boldsymbol{r}_i(0))^2\rangle \tag{11.9}$$

Equation (11.9) is an example of an *Einstein relation* for a transport coefficient. The average on the right-hand side of Equation (11.9) is closely related to that of a time correlation function and it can be calculated in a very similar fashion. The only difference is that, instead of taking the average of the product of the property at two different times as in Equation (11.7), the averaging is performed over the square of the difference of the property at the two times.

The second property is the *pair distribution function* or the *radial distribution function*, which is important in the theory of simple fluids (gases and liquids) because many thermodynamic quantities can be determined from it. The function, often denoted $g(r)$, can be thought of as a measure of the structure in a system because it gives the probability of finding a pair of particles a distance r apart, relative to the probability that would be expected for a random distribution with the same density. In practice $g(r)$ is not calculated for single values of the distance r, but for discrete intervals with a width of, say, δr. Denoting the value of the radial distribution function in the range $[r, r+\delta r]$ as $g(r+\frac{1}{2}\delta r)$ (which is the mid-point of the interval), one has

$$g\left(r+\frac{1}{2}\,\delta r\right)=\frac{n_{\text{sim}}([r, r+\delta r])}{n_{\text{random}}([r, r+\delta r])} \tag{11.10}$$

where the functions n give the average number of particles whose distances from a given particle lie within the range $[r, r+\delta r]$ and the subscripts 'sim' and 'random' denote the simulation and the random values, respectively. The expression for n_{random} is

$$n_{\text{random}}([r, r+\delta r])=\frac{4\pi N}{3V}[(r+\delta r)^3-r^3] \tag{11.11}$$

where N is the number of particles for which $g(r)$ is to be calculated and V is the system's volume. To determine n_{sim} it is necessary that all the distances between particles be calculated for each frame in the trajectory and a histogram kept that records the number of distances that fall within any particular range $[r, r+\delta r]$. The value of n_{sim} is then equal to the number of distances found for the interval from the histogram divided by the number of frames in the trajectory, n_{f}, and by N. The division by n_{f} gives the average number of distances per frame and that by N the average number of distances per particle.

A crucial part of any simulation study is how to estimate the accuracy of the results that have been obtained. Errors can be introduced at several stages. At the most basic level, they arise when a model is chosen to describe the system. We know, for example, that using semi-empirical QC or MM potentials to calculate energies and the assumption that atoms are classical particles for the dynamics are approximations that will affect the generality of the simulation methodology. There is not much we can do about these errors except improve the physical basis

of the model. At the next level, errors arise due to the way in which the model we have chosen is applied. Examples include whether the truncation scheme for the determination of the non-bonding interactions is adequate, whether the size of the central box for a PBC simulation is large enough that the effects of imposing periodicity are unimportant and whether the starting configuration of the system (the atomic positions and velocities) has been sufficiently well prepared. These types of errors are characterized by the fact that, for a given model, it is possible to investigate them by changing some of the parameters of the simulation and then repeating the study to see how the results change. In practice, how well this can be done will depend upon the type of system being studied. It will often be feasible to determine systematically the importance of these effects for a system comprising relatively few atoms or simple molecules, but it can be difficult if the system is so large that each simulation is expensive or if the system is a complicated one containing, for example, flexible molecules with multiple conformations.

A third type of errors is statistical errors, which occur in any quantity that is calculated from a simulation of finite length. The problem is to estimate how close the quantities that have been calculated from the simulation are to the values that would have been obtained from a simulation of infinite length. A related question, although one that is more difficult to answer, is that we would like to determine, if possible, the minimum length of a simulation that is necessary to obtain results of a given precision. A great variety of sophisticated statistical techniques is available for such estimates, although we shall not deal with any of them in this book. Instead, we shall limit ourselves to a number of *very basic* points. First, it is always a good idea to monitor the values of averages and fluctuations as a simulation proceeds to see whether they converge to a limiting value. It is also usually necessary to repeat the simulation several times, with different starting configurations, to see whether the same limiting values are obtained. Second, for quantities that depend on some variable, such as time correlation functions and radial distribution functions, the curves obtained should be smooth. Any roughness indicates that not enough data have been used in their determination. Finally, a useful approximate rule of thumb is that the length of simulation, t, required to obtain sufficient data for the calculation of a particular property, $\mathcal{X}$, must be much longer than the *correlation time*, $\tau_{\mathcal{X}}$, associated with that property, i.e. $t \gg \tau_{\mathcal{X}}$. This rule comes from the fact that many correlation functions have an exponential form, i.e. $\mathcal{C}_{\mathcal{X}\mathcal{X}}(t) \propto \exp(-t/\tau_{\mathcal{X}})$, where the exponent of the exponential is the inverse of the correlation time. For short times, $t \simeq \tau_{\mathcal{X}}$, the value of $\mathcal{X}$ is correlated to its initial value and so will not contribute independently to the average or fluctuation. To do so, the simulation needs to be several periods of length $\tau_{\mathcal{X}}$ long so that several independent *blocks* of $\mathcal{X}$ values have been generated. It should be noted that by no means all correlation functions decay exponentially

and that some decay very slowly at long times. They are said to have *long-time tails*. The calculation of the correlation functions for such properties can be a computationally demanding task.

To finish the discussion on errors we summarize by emphasizing that it can take data generated from several or even many *long* simulations to obtain results to within a reasonable precision. Even then, readers should be warned that great care must always be exercised when interpreting the significance of the results of any numerical simulation!

Two functions and an extension to the class `Statistics` that can help in performing some of the analyses described above are introduced in this section. The function definitions are:

Function `RadialDistributionFunction`

Calculate the radial distribution function from a trajectory using Equation (11.10).

Usage:

```
RadialDistributionFunction ( trajectory,
                             selection1, selection2,
                             bins     = 100,
                             log      = logfile,
                             maximumr = None )
```

`trajectory` is the instance of `SystemGeometryTrajectory` to be analysed.

`selection1` and `selection2` are optional positional arguments that give the indices of the atoms for which the radial distribution is to be determined. They should be instances of the class `Selection`. If two selections are present, the radial distribution function is determined between the sets of selected atoms. If one is present, the function is calculated for the selected atoms only, whereas the absence of any selection means the function will be calculated for all atoms.

`bins` is the number of intervals or *bins* to use in the calculation of the function.

`log` is the instance of `LogFileWriter` to which the function will be written.

`maximumr` is the maximum value of the distance for which the function is to be evaluated. All interparticle distances greater than this are discarded. The width of each bin (equivalent to δr in Equation (11.10)) is the value of this argument divided by the number

of bins. If the argument is absent a suitable value will be estimated.

Remarks: The function only works if the system whose trajectory is being analysed has translational symmetry, in which case the interparticle distances are determined using the minimum image convention. For a given pair of particles, this is the smallest distance that exists between any two of their images in the extended system. This condition means that the largest value of `maximumr` that is reasonable is half the length of the simulation box.

Function `SelfDiffusionFunction`

Calculate the self-diffusion function for a set of particles from a trajectory using the Einstein relation of Equation (11.9).

Usage:

```
SelfDiffusionFunction(
    trajectory, selection,
    log          = logfile,
    maximumtime  = None )
```

`selection` is an optional positional argument that gives the indices of the atoms for which the function of Equation (11.9) is to be calculated. If multiple atoms are selected, the functions of Equation (11.9) are evaluated for each atom and then averaged. Absence of the argument leads to all atoms being selected.

`maximumtime` is the maximum time difference for which the Einstein relation function is to be calculated. The maximum value of this argument is determined by the duration of the data stored on the trajectory.

Remarks: The remaining arguments to the function are similar to those of `RadialDistributionFunction`.

Correlation functions can be determined with methods in the `Statistics` class. Their use requires that the pertinent data first be extracted from a trajectory and then assigned to a `Statistics` instance. The method definitions are:

Class `Statistics`

Methods for calculating correlation functions.

Method `AutoCorrelation`

Calculate an autocorrelation function.

Usage:
```
acf = statistics.AutoCorrelation (
                    maximuminterval  =  None,
                    QNORMALIZED      =  True )
```

`maximuminterval` is an integer that specifies the maximum interval for which the autocorrelation is to be determined. The minimum value of this argument is zero and the maximum value is $n-1$ where n is the length of the data. Absence of this argument or a value of `None` implies that the autocorrelation will be determined for all possible $n-1$ intervals.

`QNORMALIZED` is a Boolean that specifies whether or not the autocorrelation is to be normalized.

`statistics` is the instance of `Statistics` for which the autocorrelation is being calculated.

`acf` is the autocorrelation function returned as an instance of `Vector`. The length of `acf` will be one more than the maximum interval for which it has been evaluated.

Remarks: This method assumes that the interval between successive data elements is constant and will not give sensible results otherwise.

Method `CrossCorrelation`

Calculate a cross-correlation function.

Usage:
```
ccf = statistics.CrossCorrelation (
                    other,
                    maximuminterval  = None,
                    QNORMALIZED      = True )
```

`other` is an instance of `Statistics` that contains the data against which the cross-correlation is to be evaluated.

`statistics` is the instance of `Statistics` for which the cross-correlation is being calculated.

`ccf` is the cross-correlation function returned as an instance of `Vector`.

Remarks: The remaining arguments of this method behave similarly to those of the method `AutoCorrelation`. The maximum interval for which the cross-correlation can be calculated will be one less than the minimum number of data elements in

`statistics` or `other`. To give sensible results, data in these two instances must have the same origin and have been generated using the same (constant) interval.

Determination of a correlation function using an expression that is directly based upon Equation (11.7) will scale as $O(n_c(2n_t - n_c)) \simeq O(n_c n_t)$, where n_t is the number of elements in the time series and n_c is the number of elements for which the autocorrelation function is to be calculated. This implies that the time required for evaluation of the full function is $O(n_t^2)$. It turns out, however, that a much more efficient algorithm can be formulated that employs FFT techniques and that scales as $O(n_t \ln n_t)$. No details will be given here, but the methods described above will use the faster algorithm when appropriate. Similar remarks apply to the self-diffusion function of Equation (11.9) as its computation can be formulated in terms of correlation functions.

11.3 Example 21

The example program presented in this section is a simple one that illustrates the use of two of the functions described in the last section for analysing the molecular dynamics trajectory that was generated in Example 20 of Section 10.6. The program is:

```
1  """Example 21."""
2
3  from Definitions import *
4
5  # . Read the system definition.
6  solvent = XMLUnpickle ( \
          os.path.join ( scratchpath, "water216_cubicbox.xpk" ) )
7  solvent.Summary ( )
8
9  # . Select all oxygens.
10 indices = []
11 for ( i, atom ) in enumerate ( solvent.atoms ):
12     if atom.atomicnumber == 8: indices.append ( i )
13 oxygens = Selection ( indices )
14
15 # . Analyse the trajectory data.
16 trajectory = SystemGeometryTrajectory ( \
         os.path.join ( scratchpath, "water216_cubicbox.trj" ), \
                                              solvent, mode = "r" )
17
```

```
18 # . Self-diffusion function.
19 SelfDiffusionFunction ( trajectory, oxygens )
20
21 # . Radial distribution function.
22 RadialDistributionFunction ( trajectory, oxygens )
```

Line 6 reads in the system definition from the XPK file that was created in Example 20.

Lines 10–13 create an instance of `Selection` that contains the indices of all the oxygens in the system.

Line 16 defines the trajectory object for the data that are to be analysed. These data were generated in Example 20.

Lines 19 and 22 calculate the self-diffusion and radial distribution functions for the oxygens in the system using the trajectory data. The results are printed to pDynamo's default log file.

Plots of the simulation results are shown in Figures 11.1 and 11.2. The value of the diffusion coefficient can be calculated from the slope of the line in Figure 11.1 at large times, giving a value of 0.43 $Å^2$ ps^{-1} or 4.3×10^{-9} m^2 s^{-1}, which is large relative to the experimental value of 2.3×10^{-9} m^2 s^{-1} at 25 °C. To verify this result fully, a similar analysis would have to be carried out on a longer trajectory to ensure that the function plotted in Figure 11.1 had indeed reached its limiting value at long times.

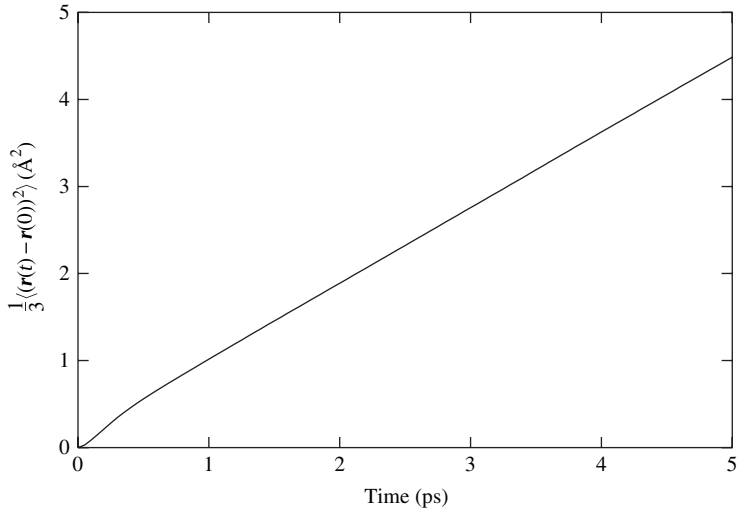


Fig. 11.1. The function $\frac{1}{3}\langle(\boldsymbol{r}(t)-\boldsymbol{r}(0))^2\rangle$ calculated for the oxygen atoms of the water molecules using the molecular dynamics trajectory generated in Example 20.

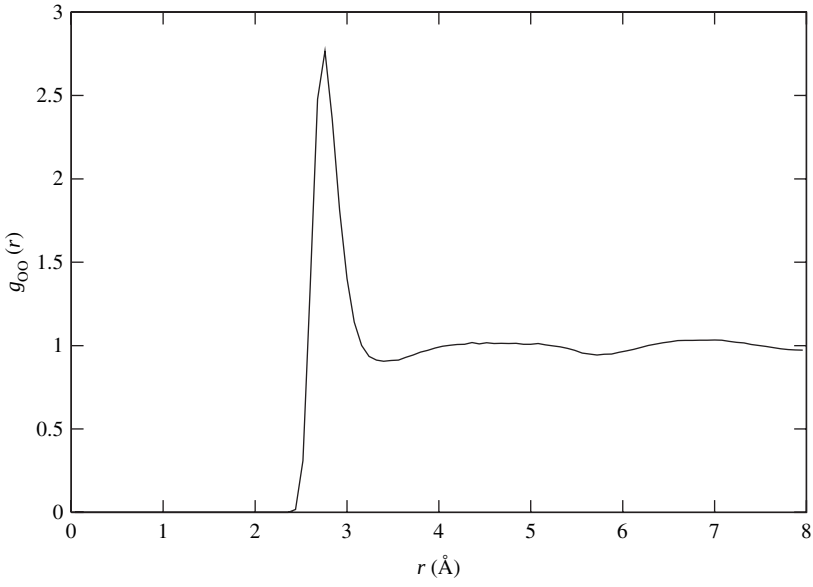


Fig. 11.2. The radial distribution function calculated for the oxygen atoms of the water molecules using the molecular dynamics trajectory generated in Example 20.

The radial distribution function of Figure 11.2 reproduces reasonably closely the experimental neutron and X-ray results, although the experimental curve has a lower first peak and has more pronounced oscillations at longer range. The fact that $g_{OO}(r) = 0$ for $r \leq 2.4$ Å indicates that the probability of finding two oxygens at these distances apart is negligible whereas the presence of oscillations after this implies that the water molecules are preferentially located in particular regions that correspond to the various 'coordination' shells in the liquid. The structure in the first coordination shell is especially marked.

11.4 Temperature and pressure control in molecular dynamics simulations

Three of the example programs described up to now have involved molecular dynamics simulations. The general procedure when doing a simulation has been to assign velocities to the atoms in the system at a low temperature, heat the system up to the temperature desired during the course of a short simulation and then equilibrate it at this temperature for another short period before starting the simulation for which a trajectory is to be generated. In the heating and equilibration phases of the simulations described previously, the temperature of the system was set to the required value by the simple, ad hoc procedure of scaling the velocities of all the atoms every few steps. When no temperature modification is done, we

have seen that one measure of the accuracy of the integration of the equations of motion for the particles is that the total energy for the system is conserved. In thermodynamic terms, the simulations are said to have been performed in the *microcanonical* or *NVE ensemble*. In other words, the number of particles, the volume and the energy of the system are constants.

Molecular dynamics simulations within the microcanonical ensemble are the easiest to perform, but, if the aim of our simulations is to mimic the conditions under which systems are investigated experimentally, the microcanonical ensemble is not necessarily the most appropriate. In particular, it is common to do experiments under conditions in which the ambient temperature and/or pressure are constants. The thermodynamic ensembles that correspond to such conditions are the *canonical* or *NVT ensemble*, the *isothermal–isobaric* or *NPT ensemble* and the *isobaric–isoenthalpic* or *NPH ensemble*, respectively.

Before we go on to discuss methods that allow molecular dynamics simulations to be performed for these ensembles, we shall introduce an algorithm developed by H. J. C. Berendsen, J. P. M. Postma, W. F. van Gunsteren, A. Di Nola and J. R. Haak for the control of the temperature and pressure in a molecular dynamics simulation. It should be emphasized that this algorithm does *not* generate trajectories drawn from the NVT, NPT or NPH ensembles, although the results need not be too different in particular cases. The reason that we use it here is that it is simple, robust and widely used. It also acts as an introduction to certain concepts that will be needed in the discussion of the more precise algorithms that are to be mentioned later.

An essential idea behind the method is that the system that we want to simulate is, in fact, not isolated but interacts, or is *coupled*, with an *external bath*. Let us consider temperature control first. Remember that, in a simulation in the microcanonical ensemble, the energy remains constant and the temperature fluctuates. A coupling to an external system means that energy can be transferred into and out of the system that we are simulating and so its energy will fluctuate. It is this transfer, properly formulated, that allows the algorithm to control the temperature.

Berendsen *et al.* proposed the following modification of the equation of motion for the velocities of the atoms, $\boldsymbol{V}$, to accomplish the coupling:

$$\dot{\boldsymbol{V}} = \mathbf{M}^{-1}\boldsymbol{F} + \frac{1}{2\tau_T}\left(\frac{T_\mathrm{B}}{\mathcal{T}} - 1\right)\boldsymbol{V} \tag{11.12}$$

where T_B is the reference temperature, which is the temperature of the external *thermal bath*, $\mathcal{T}$ is the instantaneous temperature defined in Equation (9.15) and τ_T is a coupling constant (with units of time). The extra term added to the equations of motion acts like a *frictional force*. When the actual temperature of the system is higher than the desired temperature, the force is negative. This results in the motions of the atoms being damped and the kinetic energy and, hence,

the temperature being reduced. If the temperature is too low, the reverse happens because the frictional force is positive and energy is supplied to the system. The coupling constant, τ_T, determines the strength of the coupling to the external bath. For large values of τ_T the coupling is weak and the temperature of the system will be steered only slowly towards the temperature of the bath. For small values the coupling is stronger and the dynamics of the system will be more strongly perturbed.

The principles behind the control of pressure are similar to those for the control of temperature. The major difference is that the control of pressure manifests itself as a modification of the equation of motion for the coordinates *and* the volume of the system, V. Note that, because volume changes are involved, simulations that control the pressure only make sense in the condensed phase. The pressure-control equations of motion proposed by Berendsen *et al.* are

$$\dot{\boldsymbol{R}} = \boldsymbol{V} - \frac{\beta}{3\tau_P}(P_{\mathrm{B}} - \mathcal{P})\boldsymbol{R} \tag{11.13}$$

$$\dot{V} = -\frac{\beta}{\tau_P}(P_{\mathrm{B}} - \mathcal{P})V \tag{11.14}$$

where P_{B} is the reference pressure and $\mathcal{P}$ is the *instantaneous pressure*. The parameters β and τ_P are the *isothermal compressibility* of the system, which has units of inverse pressure, and the pressure coupling constant, which has units of time. It is the ratio of these two parameters, β/τ_P, that determines the size of the coupling of the system to the external *pressure bath*. Equations (11.13) and (11.14) behave in a similar way to Equation (11.12). If the actual pressure is less than the desired pressure the system contracts whereas if the actual pressure is too large the system expands.

An instantaneous pressure can be defined in the same way as an instantaneous temperature. Thus, the thermodynamic pressure, P, is the average of the instantaneous pressure:

$$P = \langle \mathcal{P} \rangle \tag{11.15}$$

The expression for the thermodynamic pressure, P, in the canonical ensemble is

$$P = -\left(\frac{\partial A}{\partial V}\right)_T \tag{11.16}$$

where A is the Helmholtz free energy and V is the volume. It can be shown that this is equivalent to

$$P = \frac{1}{3V}\langle 2\mathcal{K} + \mathcal{W} \rangle \tag{11.17}$$

where $\mathcal{K}$ and $\mathcal{W}$ are the kinetic energy and (instantaneous) *virial* for the system, respectively. The latter is proportional to the derivative of the potential energy, $\mathcal{V}$, with respect to the volume:

$$\mathcal{W} = -3V\frac{\partial \mathcal{V}}{\partial V} \tag{11.18}$$

More explicit expressions for the virial can be derived for particular forms of the potential energy (for example, functions consisting of pairwise interactions) but, in the general case, it is better to evaluate it directly from Equation (11.18).

The above formulae are valid only for an isotropic system in which the pressure acts equally no matter what the direction. In the general case of an anisotropic system, the pressure is written as a tensor, $\mathbf{\Pi}$. This tensor, which has nine components, is sometimes known as the *stress tensor*. Its form is

$$\mathbf{\Pi} = \begin{pmatrix} \Pi_{xx} & \Pi_{xy} & \Pi_{xz} \\ \Pi_{yx} & \Pi_{yy} & \Pi_{yz} \\ \Pi_{zx} & \Pi_{zy} & \Pi_{zz} \end{pmatrix} \tag{11.19}$$

The pressure in the isotropic case is given by one-third of the trace of the stress tensor:

$$\mathcal{P} = \frac{1}{3}\left(\Pi_{xx} + \Pi_{yy} + \Pi_{zz}\right) \tag{11.20}$$

It is possible to generalize Equations (11.13)–(11.18) so that simulations can be performed with all components of the stress tensor. This results in the shape of the simulation box changing as well as its size. Such techniques are particularly useful for studying crystals and other solids in which there are changes of phase.

Berendsen *et al.* formulated their algorithm with a leapfrog version of the Verlet dynamics integrator and so the function that implements this technique has the following definition:

Function `LeapFrogDynamics_SystemGeometry`

Perform a molecular dynamics simulation with a leapfrog Verlet algorithm.

Usage:

```
LeapFrogDynamics_SystemGeometry (
    system,
    logfrequency          = 1,
    pressure              = 1.0,
    pressurecoupling      = 2000.0,
    rng                   = None,
    steps                 = 1000,
    temperature           = 300.0,
    temperaturecoupling   = 0.1,
    timestep              = 0.001,
    trajectories          = None )
```

`pressure` is the value of the reference pressure, P_{B}, in atmospheres.

`pressurecoupling` gives the value of the ratio τ_P/β in Equations (11.13) and (11.14) in units of ps atm. Pressure control is deactivated if the system is not a periodic one or if this argument is set to zero or to `None`.

`temperature` is the value of the reference pressure, T_{B}, in kelvins.

`temperaturecoupling` is the value of the temperature coupling constant, τ_T, from Equation (11.12) in picoseconds. Temperature control is deactivated by setting this argument to zero or to `None`.

Remarks: The remaining arguments behave like those of the function `VelocityVerletDynamics_SystemGeometry` described in Section 9.2.

The implementation of the algorithm is straightforward. The essential difference from a standard dynamics algorithm is that, at appropriate points in the integration, the velocities, coordinates and box size are scaled by factors determined by Equations (11.12)–(11.14). For temperature control the velocity scale factor, ζ_T, is

$$\zeta_T = \sqrt{1 + \frac{\Delta}{\tau_T}\left(\frac{T_{\mathrm{B}}}{\mathcal{T}} - 1\right)} \tag{11.21}$$

while for pressure control the coordinate scale factor, ζ_P, is

$$\zeta_P = \sqrt[3]{1 - \frac{\Delta\beta}{\tau_P}(P_{\mathrm{B}} - \mathcal{P})} \tag{11.22}$$

The volume is scaled by the factor ζ_P^3. In both equations, Δ is the timestep for the integration of the equations of motion.

The algorithm described above is the one that we shall use in this book. As already remarked, though, it does not generate trajectories in the NVT, NPT or

NPH ensembles (depending upon whether temperature and/or pressure control is being used). There are methods that do this but before we describe some of them it is important to define exactly what is meant by generating trajectories in an appropriate ensemble. To do this we need the *probability density distribution functions* for the various ensembles. These are crucial to the theory of statistical thermodynamics because they can be used to calculate the properties of any equilibrium system. Thus, if the probability density distribution function in a particular ensemble is ρ, the average of a property, $\mathcal{X}$, can be written as an integral:

$$\langle \mathcal{X} \rangle = \int \mathrm{d}\Gamma \, \mathcal{X}(\Gamma)\rho(\Gamma) \tag{11.23}$$

where Γ are the ensemble variables which will include the coordinates and momenta of the particles and $\mathrm{d}\Gamma$ indicates the volume element for a multidimensional integration over these variables.

The probability density is easiest to define for the microcanonical ensemble. If we suppose that the energy in the ensemble has a value E, then the probability of a configuration that does not have this energy will be zero for, by definition, the energy is a constant in the ensemble. Hence, the probability of a configuration that has this energy will be simply the reciprocal of the total number of states with an energy E. Mathematically, the number of states with an energy E must be written as an integral over the *phase space* of the system because the system's coordinates, $\boldsymbol{R}$, and momenta, $\boldsymbol{P}$, are continuous variables. Taking this into account, the probability density for the microcanonical ensemble, ρ_{NVE}, is

$$\rho_{\mathrm{NVE}} = \frac{\delta(\mathcal{H}(\boldsymbol{P}, \boldsymbol{R}) - E)}{\int \mathrm{d}\boldsymbol{P} \, \mathrm{d}\boldsymbol{R} \, \delta(\mathcal{H}(\boldsymbol{P}, \boldsymbol{R}) - E)} \tag{11.24}$$

where $\mathcal{H}$ is the Hamiltonian for the system and the Dirac delta functions have the effect of selecting only those configurations with total energy E.

In the canonical ensemble, it is the thermodynamic temperature that is constant and the energy fluctuates. The probability density is written as

$$\rho_{\mathrm{NVT}} = \frac{\exp[-\mathcal{H}(\boldsymbol{P}, \boldsymbol{R})/(k_{\mathrm{B}}T)]}{\int \mathrm{d}\boldsymbol{P} \, \mathrm{d}\boldsymbol{R} \, \exp[-\mathcal{H}(\boldsymbol{P}, \boldsymbol{R})/(k_{\mathrm{B}}T)]} \tag{11.25}$$

where k_{B} is Boltzmann's constant and T is the temperature. Note that this equation is a statement of the familiar *Boltzmann distribution law* which says that the probability of a configuration is proportional to its *Boltzmann factor*, $\exp[-\mathcal{H}/(k_{\mathrm{B}}T)]$.

In the isobaric–isothermal ensemble the density function is similar to the canonical function but the volume of the system is also a variable:

$$\rho_{\mathrm{NPT}} = \frac{\exp[-(\mathcal{H}(\boldsymbol{P}, \boldsymbol{R}) + PV)/(k_{\mathrm{B}}T)]}{\int \mathrm{d}\boldsymbol{P} \, \mathrm{d}\boldsymbol{R} \, \mathrm{d}V \, \exp[-(\mathcal{H}(\boldsymbol{P}, \boldsymbol{R}) + PV)/(k_{\mathrm{B}}T)]} \tag{11.26}$$

What we would like is a molecular dynamics method that generates configurations that are representative either of the canonical or of the isobaric–isothermal ensembles, i.e. one that generates states with a probability distribution appropriate for the ensemble. Many methods that do this have been proposed and we shall only briefly mention a few of them here. As was implicit with the Berendsen algorithm, it is usual to divide the methods into those that keep the temperature constant and those that maintain the pressure. Simulations in the NPT ensemble are then performed by combining algorithms for each type of control separately.

We consider constant-temperature algorithms for molecular dynamics simulations first. One technique was originally proposed by S. Nosé and later extended by W. G. Hoover and by G. J. Martyna, M. L. Klein and M. E. Tuckerman. The basis of the method is to introduce an extra, *thermostating* degree of freedom that represents the external thermal bath to which the system is coupled. In the original Nosé–Hoover algorithm, there is a single bath coordinate, η, and an associated momentum, p_η, in addition to the atomic coordinates and momenta. The modified equations of motion for the combined or *extended system* are

$$\dot{\boldsymbol{R}} = \mathbf{M}^{-1}\boldsymbol{P} \tag{11.27}$$

$$\dot{\boldsymbol{P}} = \boldsymbol{F}(\boldsymbol{R}) - \frac{p_\eta}{Q}\boldsymbol{P} \tag{11.28}$$

$$\dot{\eta} = \frac{p_\eta}{Q} \tag{11.29}$$

$$\dot{p_\eta} = \boldsymbol{P}^{\mathrm{T}}\mathbf{M}^{-1}\boldsymbol{P} - N_{\mathrm{df}}k_{\mathrm{B}}T \tag{11.30}$$

where N_{df} is the number of coordinate degrees of freedom. The parameter Q is the 'mass' of the thermostat (with units of mass times length squared) which determines the size of the coupling. A good choice of Q is crucial. Large values result in equations that approximate Newton's equations (and, hence, a constant-energy simulation), whereas small values produce large couplings and, as Nosé showed, lead to dynamics with poor equilibration. Physically, the momentum variable, p_η, in Equation (11.28) acts like a friction coefficient. When its value is positive, the kinetic energy of the system is damped, and, when the value is negative, it is increased. The value of p_η is determined by Equation (11.30) whose right-hand side is proportional to the difference between the actual temperature and the desired temperature. If the current temperature is too high, energy is removed from the system because the friction coefficient increases, whereas the reverse happens if the temperature is too low.

To perform a constant-temperature simulation, these equations are integrated in the normal way (with the extra two degrees of freedom). Like calculations in the microcanonical ensemble, it is important to be able to verify the precision of the simulation by monitoring conserved quantities. In the microcanonical ensemble

it is the energy defined by the classical Hamiltonian (Equation (9.1)) which is perhaps the most important. The equivalent in the Nosé–Hoover algorithm is defined by the Hamiltonian, $\mathcal{H}_{\mathrm{NH}}$:

$$\mathcal{H}_{\mathrm{NH}} = \frac{1}{2}\boldsymbol{P}^{\mathrm{T}}\mathbf{M}^{-1}\boldsymbol{P} + \mathcal{V}(\boldsymbol{R}) + \frac{p_\eta^2}{2Q} + N_{\mathrm{df}}k_{\mathrm{B}}T\eta \tag{11.31}$$

Nosé and Hoover showed that the system of Equations (11.27)–(11.30) produced trajectories of atomic coordinates and momenta drawn from a canonical distribution. It was observed, however, that in some cases the control of temperature was inadequate or poor. This led Martyna *et al.* to propose adding more thermostating degrees of freedom by introducing a 'chain' of thermostats. Their equations were

$$\dot{\boldsymbol{R}} = \mathbf{M}^{-1}\boldsymbol{P} \tag{11.32}$$

$$\dot{\boldsymbol{P}} = \mathbf{F}(\boldsymbol{R}) - \frac{p_{\eta_1}}{Q_1}\boldsymbol{P} \tag{11.33}$$

$$\dot{\eta}_i = \frac{p_{\eta_i}}{Q_i} \quad \forall \quad i = 1, \ldots, M \tag{11.34}$$

$$\dot{p_{\eta_1}} = \boldsymbol{P}^{\mathrm{T}}\mathbf{M}^{-1}\boldsymbol{P} - N_{\mathrm{df}}k_{\mathrm{B}}T - p_{\eta_1}\frac{p_{\eta_2}}{Q_2} \tag{11.35}$$

$$\dot{p_{\eta_j}} = \frac{p_{\eta_{j-1}}^2}{Q_{j-1}} - k_{\mathrm{B}}T - p_{\eta_j}\frac{p_{\eta_{j+1}}}{Q_{j+1}} \ \forall \ j = 2, \ldots, M-1 \tag{11.36}$$

$$\dot{p_{\eta_M}} = \frac{p_{\eta_{M-1}}^2}{Q_{M-1}} - k_{\mathrm{B}}T \tag{11.37}$$

where M is the total number of thermostats. Note that it is possible to formulate equivalent equations for the case in which there are several different chains, each of which is coupled to a different part of the system.

Nosé–Hoover thermostating is not the only way in which the temperature can be controlled. An early algorithm was developed by H. C. Andersen, who suggested that, at intervals during a normal simulation, the velocities of a randomly chosen particle or molecule could be reassigned from a Maxwell–Boltzmann distribution. This is equivalent to the particle 'colliding' with one of the particles in a heat bath. The algorithm produces trajectories in the canonical ensemble but, because of the reassignment of velocities, they are discontinuous. A related approach, which is both elegant and widely used, is one in which a stochastic analogue of Newton's equation of motion, the *Langevin equation*, is employed to describe the dynamics of a particle interacting with a thermal bath. The Langevin equation has two extra force terms arising from this interaction – a *random force*

that buffets the particle about and a frictional force, proportional to the particle's velocity, that dissipates excess kinetic energy. Another method that is based upon a different concept is one in which the kinetic energy and, hence, the temperature, are constrained to be constant at each step. The modification of Newton's equations that achieves this is straightforward and uses *Gauss's principle of least constraint*. The problem with this technique is that the coordinate configurations produced are drawn from a canonical ensemble, but the momentum configurations are not.

There is a smaller variety of algorithms for performing constant-pressure molecular dynamics simulations. The most common types are extended system algorithms, although algorithms based upon Gauss's principle of least constraint have also been developed. All methods change the volume of the simulation box.

One of the first extended system algorithms to be proposed was that of Andersen, who introduced the volume of the simulation box, V, as an additional dynamical variable. His equations are

$$\dot{\boldsymbol{R}} = \mathbf{M}^{-1}\boldsymbol{P} + \frac{1}{3}\frac{\dot{V}}{V}\boldsymbol{R} \tag{11.38}$$

$$\dot{\boldsymbol{P}} = \boldsymbol{F}(\boldsymbol{R}) - \frac{1}{3}\frac{\dot{V}}{V}\boldsymbol{P} \tag{11.39}$$

$$\ddot{V} = \frac{1}{W}(\mathcal{P} - P_{\mathrm{B}}) \tag{11.40}$$

where W is the 'mass' of the volume or *barostat* degree of freedom with units of mass times length to the fourth power. The Hamiltonian corresponding to this system of equations, $\mathcal{H}_{\mathrm{A}}$, is conserved and is

$$\mathcal{H}_{\mathrm{A}} = \mathcal{H} + \frac{1}{2}W\dot{V}^2 + P_{\mathrm{B}}V \tag{11.41}$$

Andersen showed that these equations generate trajectories consistent with the isobaric–isoenthalpic (NPH) ensemble. Hoover later modified them by adding thermostat variables of the types found in Equations (11.27)–(11.30) so that the NPT ensemble could be sampled. More recent improvements to these algorithms have been suggested by Martyna and co-workers.

All the constant-pressure algorithms described above can be generalized to allow the shape as well as the size of the simulation box to change during the course of a simulation. In these cases extra degrees of freedom must be introduced, which correspond to the position vectors of the sides of the box and add considerably to the complexity of the equations. M. Parrinello and A. Rahman and Nosé and Klein did early work to adapt Andersen's equations for simulations of this sort.

11.5 Example 22

To illustrate the pressure and temperature coupling algorithm implemented in the function `LeapFrogDynamics_SystemGeometry`, we perform a simulation of water. The program is:

```
"""Example 22."""

from Definitions import *

# . Read the system definition.
solvent = XMLUnpickle ( \
           os.path.join ( scratchpath, "water216_cubicbox.xpk" ) )
solvent.Summary ( )

# . Define a random number generator in a given state.
rng = Random ( )
rng.seed ( 917133 )

# . Equilibration.
LeapFrogDynamics_SystemGeometry ( solvent,                        \
                                  logfrequency        =    500, \
                                  pressure            =    1.0, \
                                  pressurecoupling    = 2000.0, \
                                  rng                 =    rng, \
                                  steps               =   5000, \
                                  temperature         =  300.0, \
                                  temperaturecoupling =    0.1, \
                                  timestep            =  0.001  )

# . Data-collection.
trajectory = SystemGeometryTrajectory ( \
       os.path.join ( scratchpath, "water216_cubicbox_cpt.trj" ), \
                                           solvent, mode = "w" )
LeapFrogDynamics_SystemGeometry ( solvent,                         \
                     logfrequency        =                   500, \
                     pressure            =                   1.0, \
                     pressurecoupling    =                2000.0, \
                     steps               =                 10000, \
                     temperature         =                 300.0, \
                     temperaturecoupling =                   0.1, \
                     timestep            =                 0.001, \
                     trajectories        = [ ( trajectory, 50 ) ] )
```

The program is very similar to that of Example 20 of Section 10.6. The major difference is that the leapfrog algorithm is being used instead of the velocity

Verlet technique. As before there is an initial equilibration of 5 ps (*line 14*) followed by a data-collection phase of 10 ps (*line 18*). Both are performed with pressure and temperature control at a reference pressure of 1 atm and a reference temperature of 300 K. The temperature coupling constant is 0.1 ps while the pressure coupling constant is 2000 ps atm, which corresponds to having values of β and τ_P from Equations (11.13) and (11.14) of 5×10^{-5} atm^{-1} and 0.1 ps, respectively. The values of 0.1 ps for the parameters τ_P and τ_T are the minimum values recommended by Berendsen *et al.*

The simulation above produces results for the static and dynamical properties of water that do not differ markedly from the simulation performed within the microcanonical ensemble in Example 20. Likewise, the values for the averages of the temperature are very similar for the two simulations, although the average from the simulation with the Berendsen algorithm is almost exactly 300 K (as it should be). In both cases the RMS deviations of the temperature are small, with values of 3–4 K.

In contrast, the pressure exhibits much greater fluctuations. The instantaneous pressure as a function of time from the simulation is plotted in Figure 11.3. Although the average at the end of the simulation is of the order of 1 atm, the instantaneous pressure can deviate by several hundred atmospheres from this. The volume as a function of time is shown in Figure 11.4. The changes here are also quite marked.

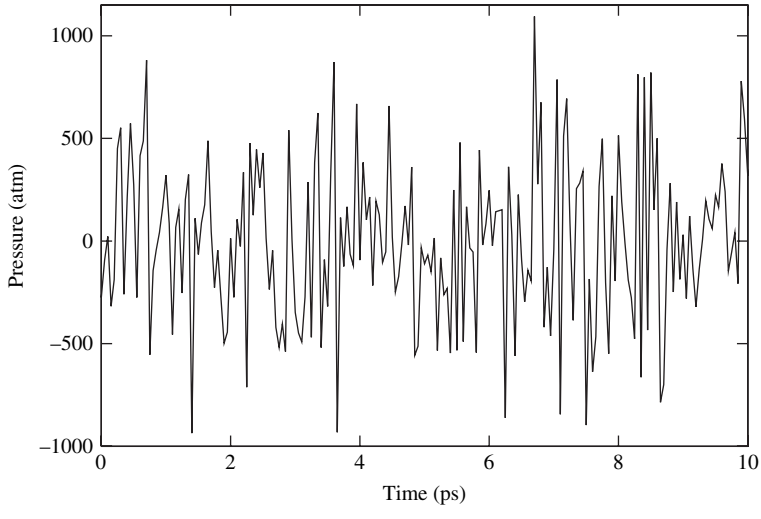


Fig. 11.3. The instantaneous pressure as a function of time from the simulation of Example 22.

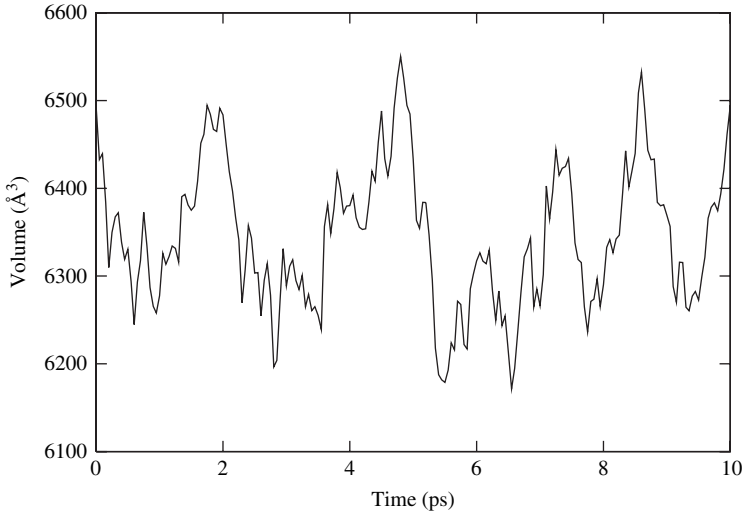


Fig. 11.4. The volume as a function of time from the simulation of Example 22.

11.6 Calculating free energies: umbrella sampling

In Section 8.6 we saw how it was possible to estimate various thermodynamic quantities for a gas-phase system within the rigid-rotor, harmonic oscillator approximation. This approximation, although it can give useful results, is limited in that it relies on data from a very limited part of the potential energy surface, namely a stationary point. Molecular dynamics techniques, because they explore the phase space more fully, can be used to determine thermodynamic quantities more rigorously. In this section, we introduce this topic by considering a method that can be employed to calculate the free energies of certain types of processes either in the gas or in condensed phases. Additional techniques for the calculation of free energies will be left to Section 12.5.

In what follows we shall limit the discussion to the canonical ensemble, although the arguments generalize to the isobaric–isothermal case. In classical statistical thermodynamics the partition function, Z_{NVT}, for a system of N indistinguishable particles at constant temperature and volume is

$$Z_{\mathrm{NVT}} = \frac{1}{h^{3N}N!}\int \mathrm{d}\boldsymbol{P}\int \mathrm{d}\boldsymbol{R}\,\exp[-\mathcal{H}(\boldsymbol{P},\boldsymbol{R})/(k_{\mathrm{B}}T)] \tag{11.42}$$

For Hamiltonians of the form given in Equation (9.1), it is possible to perform the integration over the momentum variables, leaving an integral, called the *configuration integral*, of the position coordinates only. The probability density

distribution function of Equation (11.25) can be expressed in terms of the partition function as

$$\rho_{\mathrm{NVT}} = \frac{1}{h^{3N}N!}\frac{\exp[-\mathcal{H}(\boldsymbol{P},\boldsymbol{R})/(k_{\mathrm{B}}T)]}{Z_{\mathrm{NVT}}} \tag{11.43}$$

Once the partition function and the density distribution function are known, the thermodynamic quantities for the system can be determined. Thus, for example, the Helmholtz free energy, A, is given by

$$A = -k_{\mathrm{B}}T\ln Z_{\mathrm{NVT}} \tag{11.44}$$

It is probably not evident from this expression how the free energy for a system can be calculated from a simulation but we can, with a little manipulation, rewrite the partition function of Equation (11.42) as an ensemble average of the form found in Equation (11.23). The argument is as follows:

$$\begin{aligned} Z_{\mathrm{NVT}} &\propto \int \mathrm{d}\boldsymbol{R}\exp[-\mathcal{V}/(k_{\mathrm{B}}T)] \\ &\propto \frac{\int \mathrm{d}\boldsymbol{R}\,\exp[-\mathcal{V}/(k_{\mathrm{B}}T)]}{\int \mathrm{d}\boldsymbol{R}\,\exp[-\mathcal{V}/(k_{\mathrm{B}}T)]\exp[\mathcal{V}/(k_{\mathrm{B}}T)]} \\ &\quad\times \int \mathrm{d}\boldsymbol{R}\,\exp[-\mathcal{V}/(k_{\mathrm{B}}T)]\exp[\mathcal{V}/(k_{\mathrm{B}}T)] \\ &\propto \frac{1}{\langle\exp[\mathcal{V}/(k_{\mathrm{B}}T)]\rangle} \end{aligned} \tag{11.45}$$

In this derivation we have neglected the integrals over the atomic momenta, which can be treated separately. In the second step we note that the integral introduced in the numerator and denominator reduces to $\int \mathrm{d}\boldsymbol{R}$ which evaluates to a constant equal to V^N, where V is the volume of the system and N is the number of particles.

It might be thought that a possible way to determine the free energy is to perform a molecular dynamics simulation, evaluate the average $\langle\exp[\mathcal{V}/(k_{\mathrm{B}}T)]\rangle$ along the trajectory and thus calculate the partition function and hence the free energy. However, this approach turns out to be impractical because it is extremely difficult to get reliable values for the average. The reason is that the simulation will preferentially sample configurations with large negative potential energies because these are the configurations that have a higher probability. The configurations that contribute significantly to the average, though, will be those with large potential energies because their factors, $\exp[\mathcal{V}/(k_{\mathrm{B}}T)]$, will be large.

Although the problem of adequate sampling may be especially acute when we are trying to calculate free energies and related thermodynamic properties, it must be borne in mind generally whenever any average is being calculated from a simulation. Several strategies are employed for tackling this problem. One is to

use techniques that *enhance* sampling either for the phase space as a whole or in certain regions of it during a simulation. It is one of these methods that we shall discuss in more detail in the remainder of the section. Another approach is to be less ambitious and define quantities that can be calculated without encountering the same convergence problems. These techniques will be left to Section 12.5.

One way to enhance sampling in a particular region of phase space is the method of *umbrella sampling*, which was suggested by J. P. Valleau and G. M. Torrie. In this technique a positive *biasing function*, $\mathcal{B}(\boldsymbol{R})$, is introduced and the ensemble average for a property, $\mathcal{X}$, becomes

$$\begin{aligned}\langle \mathcal{X} \rangle &= \frac{\int \mathrm{d}\boldsymbol{R}\, \mathcal{X}(\boldsymbol{R}) \exp\left[-\mathcal{V}(\boldsymbol{R})/(k_\mathrm{B}T)\right]}{\int \mathrm{d}\boldsymbol{R} \exp\left[-\mathcal{V}(\boldsymbol{R})/(k_\mathrm{B}T)\right]} \\ &= \frac{\int \mathrm{d}\boldsymbol{R}\, (\mathcal{X}(\boldsymbol{R})/\mathcal{B}(\boldsymbol{R}))\, \mathcal{B}(\boldsymbol{R}) \exp\left[-\mathcal{V}(\boldsymbol{R})/(k_\mathrm{B}T)\right]}{\int \mathrm{d}\boldsymbol{R}\, \mathcal{B}(\boldsymbol{R}) \exp\left[-\mathcal{V}(\boldsymbol{R})/(k_\mathrm{B}T)\right]} \\ &\quad \times \frac{\int \mathrm{d}\boldsymbol{R}\, \mathcal{B}(\boldsymbol{R}) \exp\left[-\mathcal{V}(\boldsymbol{R})/(k_\mathrm{B}T)\right]}{\int \mathrm{d}\boldsymbol{R}\, (1/\mathcal{B}(\boldsymbol{R}))\, \mathcal{B}(\boldsymbol{R}) \exp\left[-\mathcal{V}(\boldsymbol{R})/(k_\mathrm{B}T)\right]} \\ &= \frac{\langle \mathcal{X}(\boldsymbol{R})/\mathcal{B}(\boldsymbol{R}) \rangle_\mathrm{biased}}{\langle 1/\mathcal{B}(\boldsymbol{R}) \rangle_\mathrm{biased}} \end{aligned} \tag{11.46}$$

In the derivation of this equation, it has been assumed that the property is independent of the atomic momenta and so the integrals concerning them cancel out. The notation $\langle \cdots \rangle_\mathrm{biased}$ indicates an ensemble average determined with the biased distribution function, ρ_biased:

$$\rho_\mathrm{biased} = \frac{\mathcal{B}(\boldsymbol{R}) \exp\left[-\mathcal{V}(\boldsymbol{R})/(k_\mathrm{B}T)\right]}{\int \mathrm{d}\boldsymbol{R}\, \mathcal{B}(\boldsymbol{R}) \exp\left[-\mathcal{V}(\boldsymbol{R})/(k_\mathrm{B}T)\right]} \tag{11.47}$$

Equation (11.46) states that the ensemble average for a property can be rewritten as the ratio of the averages of the properties $\mathcal{X}(\boldsymbol{R})/\mathcal{B}(\boldsymbol{R})$ and $1/\mathcal{B}(\boldsymbol{R})$ calculated within the biased ensemble. It is easier to see what this means if we write an expression for the biasing function, $\mathcal{B}(\boldsymbol{R})$, in terms of an *umbrella potential*, $\mathcal{V}_\mathrm{umb}$:

$$\mathcal{B}(\boldsymbol{R}) = \exp\left[-\mathcal{V}_\mathrm{umb}(\boldsymbol{R})/(k_\mathrm{B}T)\right] \tag{11.48}$$

With this definition, the distribution function of Equation (11.47) corresponds to that of a system whose Hamiltonian has been modified by adding to it an extra term, $\mathcal{V}_\mathrm{umb}$. Ensemble averages in the biased ensemble are calculated by performing molecular dynamics simulations for the system with the modified Hamiltonian and then the results for the normal, unbiased ensemble are obtained by applying the formula of Equation (11.46).

In this section we shall apply the technique of umbrella sampling to the calculation of a particular type of free energy, the *potential of mean force* (PMF),

which is central to many statistical thermodynamics theories. It is, for example, required in some versions of transition-state theory to calculate the rate of reaction between two different states of a system.

The PMF can be obtained as a function of one or more of the system's degrees of freedom, although for clarity we shall restrict our attention to the unidimensional case. This degree of freedom, ξ, can be a simple function of the Cartesian coordinates of the atoms, such as a distance or an angle, or it can take a more complicated form depending upon the process being studied. The expression for the PMF is the same as that for a free energy (Equation (11.44)) except that the averaging is done over all degrees of freedom apart from the one corresponding to the variable, ξ. Let us denote ξ_0 as the value of the degree of freedom, ξ, for which the PMF is being calculated and $\xi(\boldsymbol{R})$ as the function which relates ξ to the atomic coordinates, $\boldsymbol{R}$. The PMF, $\mathcal{U}(\xi_0)$, can then be written as

$$\mathcal{U}(\xi_0) = c - k_\mathrm{B}T \ln\left(\int \mathrm{d}\boldsymbol{R}\ \delta(\xi(\boldsymbol{R}) - \xi_0) \exp\left[-\mathcal{V}(\boldsymbol{R})/(k_\mathrm{B}T)\right]\right) \tag{11.49}$$

In this equation, the parts of the partition function, such as the prefactor and the integrals over the atomic momenta, that are independent of ξ have been separated off into the arbitrary constant, c. The removal of the degree of freedom, ξ, from the averaging procedure is ensured by the Dirac delta function, which selects only those combinations of the atomic coordinates, $\boldsymbol{R}$, that give the reference value of the PMF coordinate, ξ_0.

Although Equation (11.49) illustrates the connection between the PMF and Equation (11.44), it is more usual to write the PMF in terms of the ensemble average of the probability distribution function of the coordinate, $\langle\rho(\xi_0)\rangle$, which has the expression

$$\langle\rho(\xi_0)\rangle = \frac{\int \mathrm{d}\boldsymbol{R}\ \delta(\xi(\boldsymbol{R}) - \xi_0) \exp\left[-\mathcal{V}(\boldsymbol{R})/(k_\mathrm{B}T)\right]}{\int \mathrm{d}\boldsymbol{R}\ \exp\left[-\mathcal{V}(\boldsymbol{R})/(k_\mathrm{B}T)\right]} \tag{11.50}$$

The PMF is then

$$\mathcal{U}(\xi_0) = c' - k_\mathrm{B}T \ln\langle\rho(\xi_0)\rangle \tag{11.51}$$

where c' is another arbitrary constant.

In principle, the average, $\langle\rho(\xi_0)\rangle$, can be calculated by performing a simulation in the canonical ensemble and then constructing a histogram of the frequencies of occurrence of the different values of the variable ξ_0 along the trajectory, using a similar technique to that for the calculation of the radial distribution function of Section 11.2. In practice, though, this will normally not be a feasible approach because of the difficulties of sampling sufficiently the various configurations that are accessible to the system.

One way to tackle this problem is to use umbrella sampling. The difficulty with this method, however, is that we need to choose a form for the umbrella potential, $\mathcal{V}_{\text{umb}}$, that allows efficient sampling throughout the range of the variable, ξ_0, that we are studying. The optimum choice of this potential, or the biasing function, requires a knowledge of the distribution function that we are trying to find which is, of course, unknown beforehand. The solution is to perform a series of calculations, instead of one, with umbrella potentials that concentrate the sampling in different, but overlapping, regions of phase space. The trajectories for each simulation or *window* are then used to calculate a series of biased distribution functions, in the form of histograms, which are pieced together to obtain a distribution function that is valid for the complete range of the coordinate, ξ_0.

The form of the umbrella potential that restricts sampling to a limited range of values of ξ_0 is arbitrary but a common choice, and the one that we shall make, is a harmonic form, i.e.

$$\mathcal{V}_{\text{umb}}(\xi_0) = k_{\text{umb}}(\xi_0 - \xi_{\text{ref}})^2 \tag{11.52}$$

where k_{umb} is the force constant for the potential and ξ_{ref} is the reference value of the coordinate whose value is changed at each window.

The reconstruction of the full distribution function from the separate distributions for each window is the crucial step in an umbrella sampling calculation. An efficient procedure for doing this is the *weighted histogram analysis method* (WHAM) which was developed by S. Kumar and co-workers from a technique originally due to A. M. Ferrenberg and R. H. Swendsen. The WHAM method aims to construct an optimal estimate for the average distribution function in the unbiased ensemble, $\langle \rho(\xi_0) \rangle$, from the biased distribution functions for each window. Suppose that there are N_{w} windows, each of which has an umbrella potential, $\mathcal{V}^{\alpha}_{\text{umb}}$, and an associated biased distribution function, $\langle \rho(\xi_0) \rangle^{\alpha}_{\text{biased}}$. The unbiased distribution functions for each window are determined by applying the arguments used in the derivation of Equation (11.46) to the expression for the distribution function in Equation (11.50). The result is

$$\langle \rho(\xi_0) \rangle^{\alpha}_{\text{biased}} = \frac{\exp\left[-\mathcal{V}^{\alpha}_{\text{umb}}(\xi_0)/(k_{\text{B}}T)\right] \langle \rho(\xi_0) \rangle^{\alpha}_{\text{unbiased}}}{\langle \exp\left[-\mathcal{V}^{\alpha}_{\text{umb}}(\xi_0)/(k_{\text{B}}T)\right] \rangle_{\text{unbiased}}} \tag{11.53}$$

Note that the unbiased distribution functions would be equivalent to the full distribution function if the form of the umbrella potential allowed a complete sampling of the range of the coordinate, ξ_0. Because this is not the case, the unbiased distributions are likely to provide useful information only for values of ξ_0 around the reference value of the coordinate, $\xi^{\alpha}_{\text{ref}}$, for each window.

Kumar *et al.* supposed that the full distribution function could be written as a weighted sum of the unbiased window distribution functions and then derived an

expression for the weights that minimized the statistical error in the estimate of the function. The equation for the distribution is

$$\langle \rho(\xi_0) \rangle = \sum_{\alpha=1}^{N_\mathrm{w}} \langle \rho(\xi_0) \rangle^{\alpha}_{\mathrm{unbiased}} \left\{ \frac{n_\alpha \exp\left[-(\mathcal{V}^{\alpha}_{\mathrm{umb}}(\xi_0) - \mathcal{F}_\alpha)/(k_\mathrm{B}T)\right]}{\sum_{\beta=1}^{N_\mathrm{w}} n_\beta \exp\left[-(\mathcal{V}^{\beta}_{\mathrm{umb}}(\xi_0) - \mathcal{F}_\beta)/(k_\mathrm{B}T)\right]} \right\} \tag{11.54}$$

where n_α is the number of independent data points employed for the generation of the distribution function for a window and $\mathcal{F}_\alpha$ is a window free energy that is related to the denominator of Equation (11.53) by the following expression:

$$\exp\left[-\mathcal{F}_\alpha/(k_\mathrm{B}T)\right] = \langle \exp\left[-\mathcal{V}^{\alpha}_{\mathrm{umb}}(\xi_0)/(k_\mathrm{B}T)\right] \rangle_{\mathrm{unbiased}} \tag{11.55}$$

Equation (11.54) can be rewritten in terms of the biased distribution functions using Equation (11.53) as

$$\langle \rho(\xi_0) \rangle = \frac{\sum_{\alpha=1}^{N_\mathrm{w}} n_\alpha \langle \rho(\xi_0) \rangle^{\alpha}_{\mathrm{biased}}}{\sum_{\beta=1}^{N_\mathrm{w}} n_\beta \exp\left[-(\mathcal{V}^{\beta}_{\mathrm{umb}}(\xi_0) - \mathcal{F}_\beta)/(k_\mathrm{B}T)\right]} \tag{11.56}$$

To complete the derivation an estimate of the constants, $\mathcal{F}_\alpha$, is needed. These can be determined from Equation (11.55) using the expression for the full distribution function given in Equation (11.56), i.e.

$$\exp\left[-\mathcal{F}_\alpha/(k_\mathrm{B}T)\right] = \int \mathrm{d}\xi_0 \, \langle \rho(\xi_0) \rangle \exp\left[-\mathcal{V}^{\alpha}_{\mathrm{umb}}(\xi_0)/(k_\mathrm{B}T)\right] \tag{11.57}$$

Equations (11.56) and (11.57) provide the means of calculating the average distribution function, $\langle \rho(\xi_0) \rangle$, and, hence, the PMF from a set of window distribution functions. The equations must be solved iteratively because both the distribution function, $\langle \rho(\xi_0) \rangle$, and the N_w free energies, $\mathcal{F}_\alpha$, are unknown initially. The procedure is to start by guessing a set of values for the free energies of each window (usually zero) and, with these, calculate $\langle \rho(\xi_0) \rangle$ for the complete range of ξ_0 from Equation (11.56). This estimate of the distribution function is then used to determine the window free energies from Equation (11.57) and the process is repeated until the values both of $\langle \rho(\xi_0) \rangle$ and of the set of $\mathcal{F}_\alpha$ no longer change. Experience has shown that this procedure is stable but that accurate results will be obtained only if the histograms corresponding to neighbouring distribution functions overlap to a reasonable extent.

The first requirement when carrying out molecular dynamics–umbrella sampling calculations using the scheme presented above is a way of specifying the umbrella potentials, $\mathcal{V}^{\alpha}_{\mathrm{umb}}$. In pDynamo, this is conveniently done using the various subclasses of `SoftConstraint` discussed in Section 5.6. The second requirement is that the values of ξ_0 and of $\mathcal{V}^{\alpha}_{\mathrm{umb}}(\xi_0)$ that are sampled during the window simulations need to be saved. This is achieved with a new class

called `SystemSoftConstraintTrajectory`. Instances of this class behave exactly as those of `SystemGeometryTrajectory` except that data concerning soft constraints are written instead of geometrical data. The third requirement is that the WHAM equations, Equations (11.56) and (11.57), need to be solved for the PMF given the window data. A function suitable for this task is:

Function `WHAMEquationSolver`

Solve the WHAM equations given a set of soft-constraint trajectories.

Usage:

```
WHAMEquationSolver (
                    trajectories,
                    bins           = 100,
                    log            = logfile,
                    temperature    = 300.0 )
```

`trajectories` is a list of `SystemSoftConstraintTrajectory` instances containing the umbrella potential data.

`bins` specifies the number of bins to use in the calculation of the unbiased distribution function and PMF. The function works by first scanning the range of values that are spanned by the umbrella function variable and then dividing this range into `bins` divisions of equal width for construction of the data histograms.

`log` is the instance of `LogFileWriter` to which the results will be written.

`temperature` is the temperature at which the PMF is to be calculated. This is usually the temperature at which the simulations were performed.

In this book we only consider examples in which a single umbrella potential is applied during a simulation. However, PMFs can be calculated that are functions of more than one variable and the WHAM procedure generalizes straightforwardly to these cases.

11.7 Examples 23 and 24

In this section, the potential of mean force between two conformations of the bALA molecule in vacuum is computed. The reaction coordinate is chosen as the distance between the carbonyl oxygen of the N-terminal acetyl group and the amide hydrogen of the C-terminal *N*-methyl group. At short distances there is an intramolecular hydrogen bond but this is broken as the distance increases.

There are two example programs. The first performs the molecular dynamics simulations to generate the umbrella sampling data for a series of windows and the second uses these data to calculate the PMF by solving the WHAM equations. The first program is:

```
"""Example 23."""

from Definitions import *

# . Define some parameters.
DINCREMENT    =   1.0
DMINIMUM      =   1.5
DNAME         = "dOH"
FORCECONSTANT =  20.0
NWINDOWS      =     5

# . Define the atom indices.
OXYGEN   =  5
HYDROGEN = 17

# . Define the MM and NB models.
mmmodel = MMModelOPLS ( "booksmallexamples" )
nbmodel = NBModelFull ( )

# . Generate the molecule.
molecule = MOLFile_ToSystem ( \
                       os.path.join ( molpath, "bala_c7eq.mol" ) )
molecule.DefineMMModel ( mmmodel )
molecule.DefineNBModel ( nbmodel )
molecule.Summary ( )

# . Read in the starting coordinates.
molecule.coordinates3 = XYZFile_ToCoordinates3 ( \
                      os.path.join ( xyzpath, "bala_1pt5.xyz" ) )

# . Define a constraint container and assign it to the system.
constraints = SoftConstraintContainer ( )
molecule.DefineSoftConstraints ( constraints )

# . Save the molecule definition.
XMLPickle ( \
            os.path.join ( scratchpath, "bala_example23.xpk" ), \
                                                    molecule )

# . Define a random number generator.
```

```
rng = Random ( )

# . Loop over the values of the distance.
for i in range ( NWINDOWS ):

    # . Reset the random number generator.
    rng.seed ( 291731 + i )

    # . Calculate the new constraint distance.
    distance = DINCREMENT * float ( i ) + DMINIMUM

    # . Define a new constraint.
    scmodel    = SoftConstraintEnergyModelHarmonic ( distance, \
                                                     FORCECONSTANT )
    constraint = SoftConstraintDistance ( OXYGEN, HYDROGEN, \
                                                       scmodel )
    constraints[DNAME] = constraint

    # . Equilibration.
    LeapFrogDynamics_SystemGeometry ( molecule,                  \
                                  logfrequency        =   1000, \
                                  rng                 =    rng, \
                                  steps               =  50000, \
                                  temperature         =  300.0, \
                                  temperaturecoupling =    0.1, \
                                  timestep            =  0.001  )

    # . Data-collection.
    trajectory = SystemSoftConstraintTrajectory ( \
    os.path.join ( scratchpath, "bala_window" + `i` + ".trj" ), \
                                           molecule, mode = "w" )
    LeapFrogDynamics_SystemGeometry ( molecule,                  \
                    logfrequency        =                 1000, \
                    steps               =               100000, \
                    temperature         =                300.0, \
                    temperaturecoupling =                  0.1, \
                    timestep            =                0.001, \
                    trajectories        = [ ( trajectory, 1 ) ] )
```

Lines 6–14 set the values of various parameters that will be needed later in the program.

Lines 17–24 define the system corresponding to the bALA molecule. Note that the molecule is small enough that the `NBModelFull` NB model is appropriate.

Line 27 reads in a set of starting coordinates for the simulations. These coordinates correspond to a minimum-energy structure for the bALA molecule in which the distance between the N-terminal carbonyl oxygen and the C-terminal amide hydrogen is constrained to be 1.5 Å. The constraint was imposed by geometry optimizing the molecule with a soft constraint on the O–H distance.

Lines 30–31 create an empty soft constraint container and assign it to the system.

Line 34 saves the molecule definition in an XPK file for use in Example 24.

Line 40 starts the loop within which the window simulations are performed.

Lines 46–50 create the soft constraint that will be applied to the O–H distance for the window. The definitions for each window are the same except that the equilibrium distances change, going from 1.5 to 5.5 Å in increments of 1 Å. The constraint force constant has a value of $20\,\mathrm{kJ\,mol^{-1}\,Å^{-2}}$ and has been chosen because it gives good results for this application. The aim is to have values that enable the full range of O–H distances covered by each window (in this case, about 0.5 Å either side of the equilibrium distance) to be adequately sampled and to have some, but not too much, overlap with the sampling of adjacent windows.

In practice suitable force constants are most often chosen by trial and error. A few, short simulations are done with guess values and then the values are adjusted upwards if the sampled range is too great or downwards if it is too small. One of the advantages of umbrella sampling simulations is that no data are wasted. If it appears after several simulations that sampling is lacking in some areas of the constraint space, additional simulations targeting these areas can be done with appropriate constraints and the new data added to the old.

Line 51 assigns the new constraint to the system's constraint container with the name `"dOH"`. At the same time, any existing constraints of the same name are replaced.

It should be remarked here that the same name must be used for the constraint for each window. This is because it is by name that data are identified as belonging to equivalent constraints when the window trajectories are analysed by the WHAM function.

Line 54 performs a constant-temperature simulation to equilibrate the system with the new constraint. There is no heating phase as this is less critical for this size of system when the Berendsen algorithm is employed.

Line 57 creates a trajectory object to store the soft-constraint data.

Line 58 carries out the data-collection dynamics, also at constant temperature. Soft-constraint data – in this case, the constraint O–H distance and its energy – are written to the trajectory at every step.

The second program is much more straightforward and is:

```
1  """Example 24."""
2
3  from Definitions import *
4
5  # . Read the molecule definition.
6  molecule = XMLUnpickle ( \
           os.path.join ( scratchpath, "bala_example23.xpk" ) )
7
8  # . Get the list of trajectory file names.
9  filenames = glob.glob (
             os.path.join ( scratchpath, "bala_window*.trj" ) )
10
11 # . Create the list of trajectory objects.
12 trajectories = []
13 for filename in filenames:
14     trajectories.append ( SystemSoftConstraintTrajectory ( \
                             filename, molecule, mode = "r" ) )
15
16 # . Calculate the PMF.
17 WHAMEquationSolver ( trajectories,       \
                       bins        =  100, \
                       temperature = 300.0 )
```

Line 6 reads in the system definition from the XPK file that was created in Example 23.

Line 9 uses the `glob` function from Python's `glob` module to create a list of all trajectory file names that are to be analysed. The asterisk (`*`) in the file name is matched against any character or set of characters so all names starting with `"bala_window"` and finishing with `".trj"` in the directory `scratchpath` will be returned.

Lines 12–14 create a list of instances of soft-constraint trajectory objects from the list of trajectory file names.

Line 17 employs `WHAMEquationSolver` to determine the PMF from the trajectory data.

The results of these calculations are shown in Figures 11.5 and 11.6. Figure 11.5 shows a histogram analysis of the O—H distance data generated for the windows of the umbrella sampling calculation. Each distribution is relatively smooth and there is a large overlap between the adjacent curves. The corresponding PMF is shown in Figure 11.6 as the solid line. The more stable minimum is at an O—H distance of about 2.1 Å whereas the second minimum has an energy 3 kJ mol^{-1}

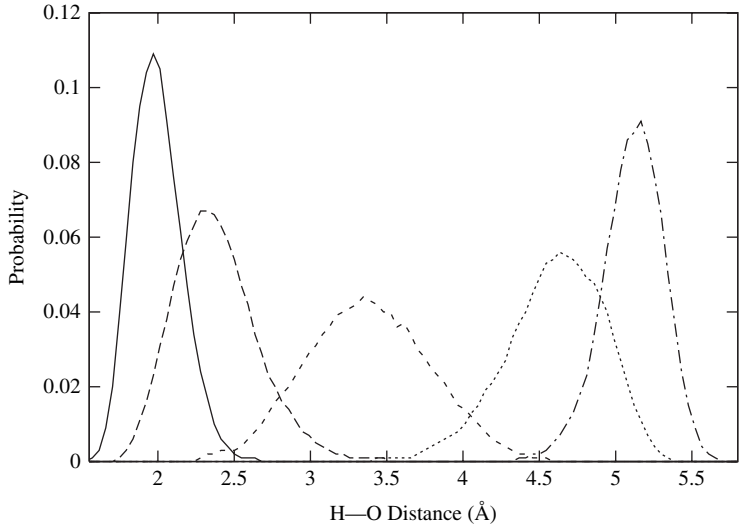


Fig. 11.5. The distribution of O—H distances for each window of the umbrella sampling simulation of Example 23.

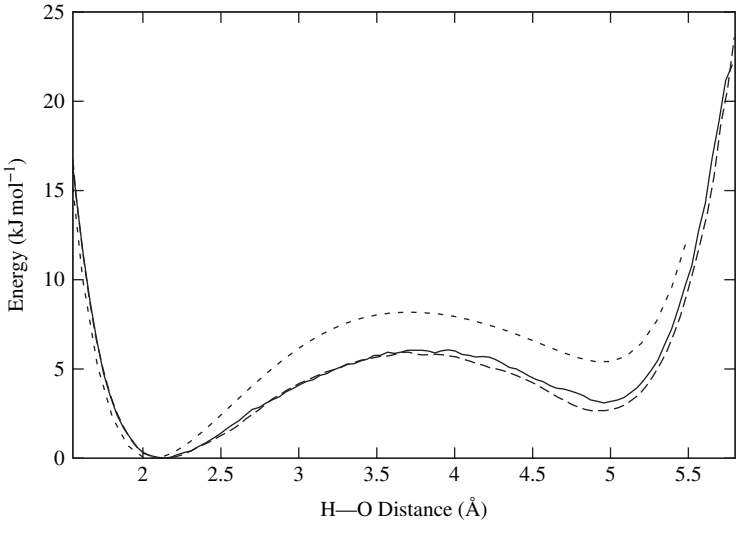


Fig. 11.6. The energy profiles for the bALA molecule as a function of the O—H distance. The PMF calculated with Examples 23 and 24 is shown as a solid line. The profile calculated using energy minimization is given by the dotted line and the PMF determined using 17 windows, instead of 5, is the dashed line.

higher and is at a distance of 5 Å. This means that the hydrogen-bonded structure is the more stable. The barrier to the interconversion of the two forms is about $6\,\mathrm{kJ\,mol^{-1}}$ starting from the hydrogen-bonded form. Note that the minima in the PMF correspond closely to the configurations of highest probability in Figure 11.5.

Also plotted in Figure 11.6 are two more curves. There is another PMF (the dashed line) calculated using the same basic procedure as above but with 17 windows at intervals of 0.25 Å and a larger force constant for the umbrella potential of 30 kJ $\mathrm{mol^{-1}}$ $\mathrm{Å^{-2}}$. Each window simulation was run for 50 ps. It can be seen that the curves for the PMF are in close agreement. The second curve (the dotted line) plots the energy profile that results if the O—H distance is constrained at certain values (using soft constraints) and the structure geometry optimized. The differences between the PMFs and the profile obtained from energy minimization are quite large. The minima are in roughly the same place but the energy of the more unstable minimum is over twice as large at about 6 kJ $\mathrm{mol^{-1}}$.

11.8 Speeding up simulations

We conclude our presentation of molecular dynamics techniques by giving a brief overview of approaches that are designed to increase the speed and efficiency of simulations (the necessity for which readers will no doubt have become aware!). We can divide these approaches into three (somewhat arbitrary) categories, namely those that modify a system's potential energy surface, those that modify its dynamics, and alternative dynamics algorithms.

We have already met an example of the first category. In umbrella sampling, biases are added to the potential energy function to permit sampling in those regions of phase space that would not otherwise be sampled. The effect of the biases is then corrected for when the simulation data are analysed to compute quantities in the appropriate, unbiased thermodynamic ensemble.

Techniques in the second category alter the dynamics of a system by removing or suppressing its high-frequency motions which permits a larger timestep to be employed when integrating its equations of motion. A very simple, if crude, way of doing this is to increase the mass of all the light atoms, especially hydrogens, in the system. This will radically change the system's dynamic properties but this is irrelevant when, for example, the simulations are being performed to calculate thermodynamic averages.

More sophisticated techniques identify the degrees of freedom that are responsible for the high-frequency motions and deal with them directly. The most general way of doing this, in principle, is to define a new set of independent geometrical variables, called *generalized coordinates*, in terms of which the dynamics of the system can be formulated. The number of such coordinates will be $3N$ ($3N-6$ if

the rotational and translational degrees of freedom are removed) minus the number of high-frequency motions, N_c. The drawback of this approach, though, is that the generalized coordinates are, except in the simplest of cases, highly non-trivial functions of the Cartesian coordinates. This gives rise to two difficulties. First, the formulae that involve them are invariably more complicated to handle (we shall see an example of this in Section 12.3 of the next chapter) and, second, quantities that involve them, such as their potential energy derivatives, are more expensive to evaluate. For these reasons, generalized coordinates have been employed less widely than Cartesian coordinates in molecular dynamics simulations.

Another approach to the problem is to retain Cartesian coordinates as the variables but constrain the degrees of freedom generating the high-frequency motions using the method of Lagrange multipliers. Suppose that we have a set of constraints that are functions of the coordinates of the atoms only (and perhaps the time). Such constraints are called *holonomic* and can be written as

$$\Lambda_k(\boldsymbol{R}) = 0 \;\forall\; k = 1, \ldots, N_c \tag{11.58}$$

The dynamics of a system change in the presence of the constraints. From classical mechanics, the modified equations of motion are

$$\mathbf{M}\ddot{\boldsymbol{R}} = -\frac{\partial \mathcal{V}}{\partial \boldsymbol{R}} - \sum_{k=1}^{N_c} \lambda_k \frac{\partial \Lambda_k}{\partial \boldsymbol{R}} \tag{11.59}$$

The first term on the right-hand side of the equation corresponds to the forces arising from the potential whereas the second term is the force due to the constraints. The variables λ_k are the Lagrange multipliers which are a function of time and have values chosen to ensure that the constraints of Equation (11.58) are satisfied at all points along the trajectory. It is worth pointing out here that the constraint forces contribute terms that need to be included when calculating certain properties of a system. These include its virial (or pressure) and its free energy.

The way in which these equations are solved varies depending upon the numerical method chosen to integrate the equations of motion. If the standard Verlet algorithm is being used, the first step would be to obtain the unconstrained positions for each atom, $\boldsymbol{R}'(t+\Delta)$, using the formula in Equation (9.7) and the forces arising from the potential, $\mathcal{V}$. Note that it is assumed that the previous points along the trajectory satisfy the constraint conditions. The positions for the atoms that satisfy the constraints, $\boldsymbol{R}(t+\Delta)$, are then written as

$$\boldsymbol{R}(t+\Delta) = \boldsymbol{R}'(t+\Delta) - \Delta^2 \mathbf{M}^{-1} \sum_{k=1}^{N_c} \lambda_k \frac{\partial \Lambda_k}{\partial \boldsymbol{R}} \tag{11.60}$$

and satisfy the conditions

$$\Lambda_k(\boldsymbol{R}(t+\Delta)) = 0 \;\forall\; k = 1, \ldots, N_c \tag{11.61}$$

Substitution of Equation (11.60) into Equation (11.61) gives a set of N_c equations for the N_c unknown Lagrange multipliers, λ_k, which can be solved fairly straightforwardly by a variety of iterative techniques. These methods are often quite efficient, converging in a small number of steps, but they can have a limited *radius of convergence* for certain types of constraint, which means that they will not work if the unconstrained atom positions deviate too much from their target values.

One of the earliest applications of the above techniques was the *SHAKE* algorithm developed by J. P. Ryckaert, G. Ciccotti and H. J. C. Berendsen. SHAKE, and other algorithms like it, is most commonly used for fixing bond lengths in a simulation, especially those involving hydrogens, in which cases timesteps of 2 fs can be routinely employed. The methods can also be used for other types of constraint, such as bond angles, although studies have shown that fixing these types of degree of freedom can significantly alter the dynamics of the system.

The last category of approach that we mention in this section is alternative dynamics algorithms, prominent examples of which are the *multiple timestep integration schemes*. The rationale underlying these methods is that timesteps of different lengths are employed to integrate the forces arising from different parts of the potential energy function. Short timesteps are required for the integration of the force terms that give rise to the high-frequency motions whereas longer timesteps can be used to integrate the force terms that change more slowly. Efficiency will be enhanced as long as the short-timestep forces are cheap to evaluate and the expensive long-timestep forces can be calculated less often than in the equivalent single timestep integration schemes.

Although methods of this type existed before, it was the work of M. Tuckerman, B. J. Berne and G. J. Martyna in the early 1990s that permitted the development of consistent and robust multiple timestep integrators. Since then, methods of this type have become quite widely used for simulations with MM potentials. Although details vary, the covalent and short-range non-bonding forces, especially the repulsive part of the Lennard-Jones potential, change most rapidly and so are integrated with shorter timesteps, typically 0.5–2 ps. By contrast, the long-range non-bonding forces, which are the most expensive to evaluate, change more slowly and are integrated with longer timesteps, often in the range 4–8 ps. Versions of these algorithms have also been developed for use in conjunction with QC potentials but principally for DFT methods with plane-wave basis sets.

Exercises

11.1 The diffusion coefficient for an atom or molecule can be calculated from the integral of the velocity autocorrelation function (Equation (11.8)). Do this calculation for the water box system used in Example 22. This will require: (i) repeating the molecular dynamics simulation so that a trajectory containing the velocities is generated. This can be done with the class `SystemVelocityTrajectory`; (ii) calculating the correlation functions and their integrals for each of the oxygens in the system; and (iii) averaging the resulting values. How do the results for the diffusion coefficients compare with those obtained in Example 21? By looking at the form of the velocity autocorrelation function estimate the length of a simulation that is needed to calculate this property – is 10 ps reasonable, is a longer simulation needed or would a shorter simulation be adequate?

11.2 Repeat the constant-pressure and -temperature calculations for water using different sets of coupling parameters, both smaller and larger than 0.1 ps. How do the static and dynamic quantities calculated from the simulation differ from each other and from those calculated in the microcanonical ensemble and at constant volume and constant temperature?

11.3 Calculate a PMF for a problem different from that of Examples 23 and 24. Possible examples include the determination of a PMF as a function of an internal torsion angle of a molecule (such as butane) or a PMF as a function of the distance between two atoms or molecules (to simulate an association or dissociation). Do the calculation in vacuum and, if possible, in solution. Note that these calculations will be much more expensive! Is it possible to estimate how accurate the results are? pDynamo also includes other algorithms for performing constant-temperature simulations. Try one of these, such as `LangevinDynamics_SystemGeometry`, instead of the leapfrog method. Are the calculations more efficient?

12

Monte Carlo simulations

12.1 Introduction

In the previous chapters a variety of techniques for the simulation of molecular systems have been covered. These have included methods such as energy minimization and reaction-path-finding algorithms, which explore a relatively limited portion of the potential energy surface of a system, and the molecular dynamics method, which makes accessible a much larger region of the potential energy surface and with which time-dependent events and properties can be studied. The ability of molecular dynamics simulations to sample a large region of the phase space of the system is important, as we have seen, for locating global potential energy minima and for calculating thermodynamic quantities.

There is an alternative technique, the *Monte Carlo method*, that is distinct from the molecular dynamics method but can also sample the phase space of the system and, hence, is appropriate for calculating thermodynamic quantities or for performing simulated annealing calculations. Unlike the molecular dynamics method it cannot be used to study time-dependent properties but it does have other features that are advantageous in some circumstances.

12.2 The Metropolis Monte Carlo method

Consider the integral, $\mathcal{I}$, of a function, $f(x)$, over a region $[a, b]$:

$$\mathcal{I} = \int_a^b f(x)\,\mathrm{d}x \tag{12.1}$$

A normal way to estimate the integral, for well-behaved functions, would be to divide the region $[a, b]$ into n equally spaced slices, each of width $\Delta = (b-a)/n$ and then use a standard integration formula of the type

$$\mathcal{I} \simeq \Delta \sum_{i=0}^{n} w_i f(a + i\Delta) \tag{12.2}$$

where the w_i are weights whose values depend upon the formula being used. For the well-known Euler formula these would be 1 except at the end points, where they would be $\frac{1}{2}$.

The basis of the Monte Carlo approach is to realize that, instead of using a regular discretization of the integration variable, as in Equation (12.2), it is possible to use a stochastic method in which the values of the integration variable are chosen randomly. Let n denote the *number of trials* and x_i (with $i = 1, \ldots, n$) the values of the integration variable that are chosen at random from a uniform distribution with values between a and b. The integral can be evaluated as

$$\mathcal{I} \simeq \Delta \sum_{i=1}^{n} f(x_i) \tag{12.3}$$

where, as before, $\Delta = (b-a)/n$, but this time it represents the average distance between integration points rather than the exact distance.

This formula works reasonably well for functions whose values do not change too much from one place to another in the integration range. For functions whose values vary greatly or are peaked in certain areas, the formula in Equation (12.3) will be inefficient because the values of the function at many of the randomly chosen integration points will contribute negligibly to the integral. In these cases, it is more useful to be able to choose values of x that are concentrated in areas in which the function will be large. To do this the integral of Equation (12.1) can be rewritten as

$$\mathcal{I} = \int_a^b \left(\frac{f(x)}{\rho(x)} \right) \rho(x)\, \mathrm{d}x \tag{12.4}$$

where $\rho(x)$ is a probability density function that is large where it is thought that the function will be large. The integral can now be approximated by choosing values of the integration variable randomly from the function $\rho(x)$ in the range $[a, b]$, instead of from the uniform distribution, and averaging over the values of $f(x_i)/\rho(x_i)$ that are obtained, i.e.

$$\mathcal{I} \simeq \frac{1}{n} \sum_{i=1}^{n} \frac{f(x_i)}{\rho(x_i)} \tag{12.5}$$

That this formula is the same as Equation (12.3) in the case of a uniform distribution follows because the probability distribution function for the uniform distribution is $1/(b-a)$. The use of a function ρ in this way to enhance sampling in certain regions of space is known as *importance sampling*.

The stochastic method outlined above cannot usually compete with numerical methods of the type given in Equation (12.2) if there is a small number of integration variables. However, the number of function evaluations required by simple discretization schemes for the estimation of an integral becomes prohibitively

large as the number of dimensions, N_{dim}, increases. To see this, suppose that n points are chosen for the discretization in each direction, then the number of function evaluations required is $n^{N_{\text{dim}}}$. It is in these cases that stochastic methods are often the only realistic approaches for tackling the problem.

As we have already seen in Section 11.6, the integrals that are of interest in thermodynamics are almost always multidimensional. As an example, consider a property, $\mathcal{X}$, of the system that is a function of the $3N$ coordinates of the atoms, $\boldsymbol{R}$, only. The average of the property in the canonical ensemble is then the ratio of two multidimensional integrals:

$$\langle \mathcal{X} \rangle = \frac{\int \mathrm{d}\boldsymbol{R}\, \mathcal{X}(\boldsymbol{R}) \exp[-\mathcal{V}(\boldsymbol{R})/(k_{\text{B}}T)]}{\int \mathrm{d}\boldsymbol{R}\, \exp[-\mathcal{V}(\boldsymbol{R})/(k_{\text{B}}T)]} \tag{12.6}$$

where $\mathcal{V}$ is the potential energy of the system. This equation can be rewritten in a form reminiscent of Equation (12.4) by employing the probability density distribution function for the canonical ensemble, ρ_{NVT}. Thus

$$\langle \mathcal{X} \rangle = \int \mathrm{d}\boldsymbol{R}\, \mathcal{X}(\boldsymbol{R})\, \rho_{\text{NVT}}(\boldsymbol{R}) \tag{12.7}$$

If, somehow (and this is the difficult part!), it is possible to choose configurations for the system drawn from the function, ρ_{NVT}, then the average, $\langle \mathcal{X} \rangle$, can be calculated using a formula analogous to Equation (12.5), i.e.

$$\langle \mathcal{X} \rangle \simeq \frac{1}{n} \sum_{I=1}^{n} \mathcal{X}(\boldsymbol{R}_I) \tag{12.8}$$

where n is the number of configurations generated in the simulation and $\boldsymbol{R}_I$ is a vector of the coordinates of the atoms at each configuration. Owing to the presence of the exponential in the Boltzmann factor, it is crucial to employ importance sampling for these integrals because, for most properties, only the configurations of lowest potential energy will contribute significantly.

The Monte Carlo method for integration was formalized in the late 1940s by N. Metropolis, J. von Neumann and S. Ulam whereas the extension for generating configurations drawn from a canonical distribution was introduced by Metropolis and co-workers in the early 1950s to study atomic systems. In outline, it is as follows:

(i) Choose an initial configuration for the system, $\boldsymbol{R}_0$, and calculate its potential energy, $\mathcal{V}_0$. Set $I = 0$.

(ii) Generate, at random, a new configuration for the system, $\boldsymbol{R}_J$, from the current configuration. How to generate the new configuration is unimportant for the moment and will be discussed in detail later. Metropolis *et al.* used a recipe in which the probability, p_{IJ}, of generating a state J from a state I was equal to the probability, p_{JI}, of generating the state I from state J. They also insisted that the method should

allow, in principle, every state to be accessible from all other possible states, if not as a result of a single change, then as a result of a sequence of changes.

(iii) Calculate the potential energy of the new state, $\mathcal{V}_J$.

(iv) If the difference in the potential energies of the two states, $\mathcal{V}_J - \mathcal{V}_I$, is less than zero, choose state J as the new configuration, i.e. set $\boldsymbol{R}_J$ to $\boldsymbol{R}_{I+1}$.

(v) If $\mathcal{V}_J - \mathcal{V}_I > 0$ fetch a random number from a uniform distribution in the range [0, 1]. If the number is less than $\exp[-(\mathcal{V}_J - \mathcal{V}_I)/(k_B T)]$ *accept* the new configuration, otherwise *reject* it and keep the old one.

(vi) Accumulate any averages that are required using Equation (12.8) and the new configuration, $\boldsymbol{R}_{I+1}$. Note that, even if the 'new' configuration is the same as the old one, it still must be re-used if proper averages are to be obtained.

(vii) Increment I to $I+1$ and return to step (ii) for as many steps as are desired in the simulation.

The above scheme is an ingenious one because it avoids any explicit reference to the configuration integral or to the partition function for the system. It is possible to prove rigorously that the scheme generates configurations drawn from a canonical ensemble using the theory of *Markov chains*, because technically what the Metropolis algorithm does is to produce a Markov chain of configurations with the limiting distribution of the canonical ensemble. However, in their paper, Metropolis *et al.* argued as follows. Suppose that there exists an ensemble of identical systems in various states and that n_I is the number of systems in the ensemble in state I. Consider two states, I and J with $\mathcal{V}_I > \mathcal{V}_J$. During a simulation of the *entire* ensemble using the Metropolis algorithm, the nett transformation of states I to states J will be $p_{IJ}\, n_I$ and of states J to I, $p_{JI}\, n_J \exp[-(\mathcal{V}_I - \mathcal{V}_J)/(k_B T)]$. Thus, the nett flow from states J to states I will be

$$p_{JI}\, n_J \exp[-(\mathcal{V}_I - \mathcal{V}_J)/(k_B T)] - p_{IJ}\, n_I = p_{IJ}\{n_J \exp[-(\mathcal{V}_I - \mathcal{V}_J)/(k_B T)] - n_I\} \tag{12.9}$$

because it has been assumed that $p_{IJ} = p_{JI}$. After sufficiently many configurations the nett flow and, hence, the term in brackets will tend to zero, so

$$\frac{n_I}{n_J} \simeq \frac{\exp[-\mathcal{V}_I/(k_B T)]}{\exp[-\mathcal{V}_J/(k_B T)]} \tag{12.10}$$

This is exactly what we are seeking because Equation (12.10) gives the ratios of the populations of two states in the canonical ensemble. As an aside, it should now be clear why the calculation of correct averages requires that 'old' configurations be re-used. If this were not the case, it would mean that every time a configuration was rejected and a system was left in its original state it would be eliminated from the ensemble.

There is a further point worth discussing about the Metropolis scheme. The condition on the method of generation of new configurations that all states be

accessible from all others (see step (ii)) means that the full phase space of the system can, in principle, be explored. This property, which is called *ergodicity*, is important because the values of any properties calculated from a simulation are likely to be significantly in error if the method becomes trapped in a small region of phase space. Of course, even if a method is theoretically ergodic, little is implied about how long a simulation needs to be or how many configurations need to be sampled to obtain averages to within a given precision.

The Metropolis Monte Carlo method has the great advantage that it can be easily extended to generate chains of configurations from other ensembles. W. W. Wood first showed how this could be done in the isothermal–isobaric (NPT) ensemble. In this ensemble the average of a property, analogous to the average in the NVT ensemble of Equation (12.6), is

$$\langle \mathcal{X} \rangle = \frac{\int \mathrm{d}\boldsymbol{R}\, \mathrm{d}V\, \mathcal{X}(\boldsymbol{R}, V) \exp[-(\mathcal{V}(\boldsymbol{R}) + PV)/(k_\mathrm{B}T)]}{\int \mathrm{d}\boldsymbol{R}\, \mathrm{d}V \exp[-(\mathcal{V}(\boldsymbol{R}) + PV)/(k_\mathrm{B}T)]} \tag{12.11}$$

where V is the volume of the system and P is the pressure. It is to be noted that in these integrals the coordinates of the atoms, $\boldsymbol{R}$, and the volume are not independent variables. To ensure a correct derivation, as Wood showed, it is necessary to transform the absolute coordinates of the atoms, $\boldsymbol{R}$, to fractional atomic coordinates, $\boldsymbol{S}$. For a cubic box of side L, $\boldsymbol{R} = L\boldsymbol{S}$ and the volume element for the integral $\mathrm{d}V\, \mathrm{d}\boldsymbol{R}$ becomes $\mathrm{d}V\, \mathrm{d}\boldsymbol{S}\, V^N$ where $V = L^3$. Putting the V^N factor into the exponential gives a probability density distribution function, ρ_{NPT}, for the ensemble which is proportional to $\exp[-(\mathcal{V} + PV)/(k_\mathrm{B}T) + N \ln V]$. The procedure for performing a Monte Carlo simulation in the NPT ensemble can now be formulated and it turns out to be identical to that for simulations in the NVT ensemble except for the following differences:

(i) When generating new configurations, the fractional coordinates of the atoms and the volume of the system are changed instead of just the atomic coordinates.
(ii) The criterion for accepting a configuration is no longer based on the difference between the potential energies of the old and new configurations but on the difference between the quantities in the exponential of ρ_{NPT}, i.e. on $\mathcal{V}_J - \mathcal{V}_I + P(V_J - V_I) - Nk_\mathrm{B}T \ln(V_J/V_I)$, which reduces to the difference of potential energies when $V_I = V_J$. With this new quantity the Metropolis tests are applied in exactly the same way as before.

12.3 Monte Carlo simulations of molecules

The Metropolis algorithm, although it was the first of a number of Monte Carlo algorithms to have been developed, is still the most successful and widely used Monte Carlo method for the simulation of molecular systems. The Monte Carlo

technique itself has a number of features that can make it preferable to the molecular dynamics method in some circumstances. These include the following:

(i) Only the energy is required whereas molecular dynamics methods require the forces.
(ii) It is easy to perform simulations in the NVT and NPT ensembles whereas to do so with molecular dynamics simulations requires more complicated techniques.
(iii) It is straightforward to 'constrain' various degrees of freedom in the system. As we shall see below, this is done by keeping these degrees of freedom fixed when new configurations for the system are generated.
(iv) In principle, very different configurations of a system can be sampled during a Monte Carlo simulation if efficient schemes for the construction of new states can be devised. In the molecular dynamics method the prescription for the generation of new states is inherent to the formulation and is determined by the integration of the equations of motion of the atoms. The difference between successive structures, which is governed by the timestep, is small and so it can take very long simulations to probe very different regions of the system's phase space. Likewise, it may be difficult or even impossible to explore certain regions of phase space if the intervening energy barriers are large.

The last advantage of the Monte Carlo technique is also its disadvantage because recipes to generate new configurations for the system must be conceived. It happens that this is relatively simple for systems composed of atoms or of small molecules. For large, flexible molecules it has proved more difficult to come up with an efficient method and so the application of the Monte Carlo technique in these cases has been relatively limited (with the caveat that an atomic model of the molecule is being used – the Monte Carlo method has been employed widely for polymer studies with simplified molecular models).

Because of the added complexity in dealing with large molecules, the applications of the Monte Carlo method in this book will be more limited than those of the molecular dynamics methods that have already been described. The latter can be used to study almost any molecular system with certain limitations, such as those that are due to the applicability of the energy function (whether QC, MM or QC/MM). For Monte Carlo simulations, however, we shall consider systems composed of atoms or of small, rigid molecules and that have MM energy models only.

With these restrictions, there are three types of change that have to be considered in order to generate new configurations for a system. These are translations and rotations of the molecules and changes in the volume of the simulation box. How we deal with each of these types of moves is arbitrary but the ways described below are commonly found and are the ones that we shall employ.

Translations are the easiest to deal with. Figure 12.1 shows schematically how this is done. Suppose that there is a cube, of side $2\,\delta t$, centred at the centre of

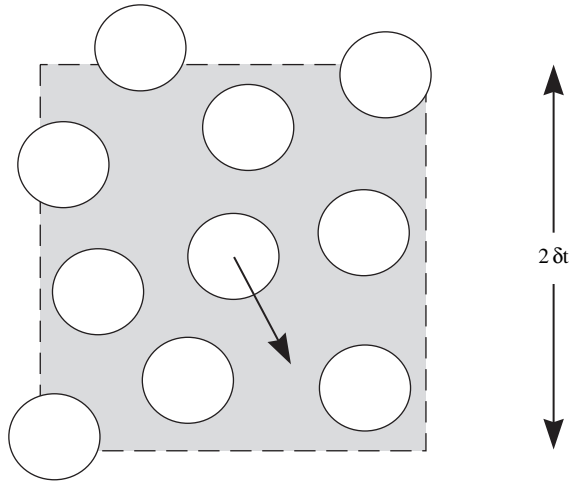


Fig. 12.1. A schematic diagram of how a molecule is translated in a Monte Carlo simulation. The two-dimensional representation is easily generalized to three dimensions.

mass of the molecule. To translate the molecule, choose a vector of three random numbers, $\boldsymbol{u}$, uniformly in the range $[-1, 1]$ and then translate the molecule by $\delta t\, \boldsymbol{u}$. This type of move satisfies both of the conditions for the generation of new configurations outlined in step (ii) of the Metropolis scheme. First of all, the translation of a molecule from one centre ('state'), I, to another, J, will have the same probability as the translation from J to I because the selection of the translation is independent of direction and is uniform within the cube. Second, all states (or molecule centres) will be accessible from all others because a molecule can be translated in a single move to 'any' position within its cube (limited by the precision of the computer) and, hence, after a succession of moves, to any position in the simulation box.

The length of a possible translation is governed by the parameter δt. It is obvious that, if δt is large, there will be a high probability that the atoms of two molecules will overlap, making the molecules possess a large, positive interaction energy due to the repulsive part of the Lennard-Jones potential. Such configurations are likely to be rejected by the Metropolis criterion for deciding whether to accept new states. If the parameter δt is small then the probability of accepting the state will be high but it will take a long time for the configuration space of the system to be sampled effectively. The aim, therefore, is to choose a value of δt that is as large as possible while giving a reasonable *acceptance ratio* for new configurations. No comprehensive study seems to have been done to decide what the best value of this ratio is – indeed, it is likely to depend on the nature of the system being studied – but most workers appear to prefer values of the order of

50%. Although the exact value of δt required to attain a certain acceptance ratio might not be known in advance it is straightforward to modify the value of δt in the course of a simulation so that the desired acceptance ratio is approached. This can be done by scaling δt by a small amount every few steps, up if the acceptance ratio is too high and down if it is too small.

Until now no mention of whether all or, if not, which molecules are to be moved during the generation of new states has been made. Because the Metropolis scheme imposes no constraints of its own per se, we are free to move one molecule at a time, several molecules or all of them at once, whichever gives the most efficient scheme for sampling the phase space of the system. Once again, no comprehensive study of which method is best appears to have been done, but the preferred choice in the literature and, incidentally, the one that Metropolis *et al.* used in their original work, is the one in which only a *single* molecule is moved at a time. The molecule to be moved can be chosen either at random or by cycling through all the molecules in a given order. It should be noted that it is not necessary to recalculate the complete energy of a system that is described with a pairwise additive potential if only a single molecule is moved – it is necessary to recalculate only the *energy of interaction* between the moved molecule and the rest of the system. The difference in potential energies between the old and new states required for the Metropolis test is then simply the difference between the moved molecule's interaction energies in the old and new states. This makes the cost of n single-molecule moves of the same order as the cost of a move in which n molecules are moved at the same time.

Single-molecule moves or moves in which all the molecules are translated at once are appropriate for homogeneous systems but it may be preferable to use other schemes for other systems. Consider, for example, the simulation of a solute molecule in a solvent. In this case, it may be more efficient to move the solute molecule more often than the solvent molecules and to move those solvent molecules that are closer to the solute more frequently than solvent molecules that are further away. Several such *preferential sampling* schemes have been developed but they require a modification of the normal Metropolis procedure and will not be discussed further here.

It is common to rotate a molecule at the same time as it is translated when a new configuration is generated, although some subtleties arise in effecting the rotation. This is because the argument presented in Section 12.2 to justify the Metropolis scheme stated that it led to the correct ratio of the populations of two states (Equation (12.10)). This is true for Cartesian coordinates and for other coordinate systems whose volume elements are independent of the values of the current coordinates. To see this notice that the probabilities of states I and J are proportional to $\exp[-\mathcal{V}_I/(k_\mathrm{B}T)]\,\mathrm{d}\boldsymbol{R}$ and $\exp[-\mathcal{V}_J/(k_\mathrm{B}T)]\,\mathrm{d}\boldsymbol{R}$, respectively.

The volume elements in both cases are $d\boldsymbol{R}$ and so cancel out when the ratio of probabilities is taken. For a general set of coordinates, the cancellation does not occur and the implementation of the Metropolis algorithm must be modified.

Many sets of coordinates that specify orientation, such as the *Euler angles*, do not have volume elements that will be the same for two states. It is feasible to alter the Metropolis scheme to use such coordinates but it is also possible to select an appropriate set of coordinates that leaves the Metropolis scheme unchanged. One such set, the set that will be used below, involves choosing one of the Cartesian axes, x, y or z, at random and then rotating the molecule about this axis by a random angle chosen in the range, $[-\delta\eta, \delta\eta]$, where the parameter $\delta\eta$ plays exactly the same role for the rotation as δt does when translating a molecule. If we suppose that the centre of mass of the molecule is chosen as the origin of the rotation, then the new coordinates of the atoms, $\boldsymbol{r}_i'$, in a molecule will be generated from the old ones, $\boldsymbol{r}_i$, using the following transformation:

$$\boldsymbol{r}_i' = \mathbf{U}(\boldsymbol{r}_i - \boldsymbol{R}_\mathrm{c}) + \boldsymbol{R}_\mathrm{c} \tag{12.12}$$

Here $\boldsymbol{R}_\mathrm{c}$ is the centre of mass of the molecule and $\mathbf{U}$ is the rotation matrix, which for a rotation about the x axis by an angle, η, will have the form

$$\mathbf{U}_x = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\eta & \sin\eta \\ 0 & -\sin\eta & \cos\eta \end{pmatrix} \tag{12.13}$$

Because rotational and translational moves are often performed together, it is best if the values of the parameters $\delta\eta$ and δt are compatible so that the rotational space and the translational space available to the molecules are sampled with roughly the same efficiency.

The final type of move is a volume move that is required for Monte Carlo simulations in the NPT ensemble. It is possible to combine moves in which molecules are translated and rotated with those in which the volume is changed simultaneously. A commoner strategy, though, is to intersperse moves in which only the volume is changed with moves in which only molecule rotations and translations are performed. A volume change can be done in exactly the same way as a rotation or a translation. A random number, υ, in the range $[-1, 1]$ is generated and the volume of the system is changed by $\delta V\,\upsilon$, where δV is the maximum change in the volume that is allowed in any single move. Once a new value for the volume has been selected, it is necessary to change the coordinates of all the atoms in the simulation box in an appropriate manner. Note that, if this were not done, an increase in volume would lead to cavities forming at the boundaries of the simulation box, whereas a decrease in volume would lead to overlap of molecules in the same regions. For a system composed of atoms, the

coordinates of the atoms are changed such that their fractional coordinates, $\boldsymbol{S}$, remain the same. Thus, for a cubic box, the coordinates of the atoms in the new configuration, $\boldsymbol{R}'$, are generated by scaling the coordinates of the atoms in the old configuration, $\boldsymbol{R}$, by the ratio of the new to the old box lengths, L'/L. Such a scaling for a molecular system, in which molecules are supposed to be rigid, will not work because it will lead to changes in their internal geometries. In this case, the same scaling factor is applied instead to the coordinates of the centres of mass of each molecule. Thus, the molecule as a whole is moved but its internal geometry is left unchanged. The transformation relating the new, $\boldsymbol{r}_i'$, and old, $\boldsymbol{r}_i$, coordinates for an atom is then

$$\boldsymbol{r}_i' = \boldsymbol{r}_i + \left(\frac{L'}{L} - 1\right)\boldsymbol{R}_\mathrm{c} \tag{12.14}$$

The value of the parameter determining the size of the volume moves, δV, can be adjusted during the simulation in exactly the same way as for the rotational and translational move parameters so that the desired acceptance ratio is obtained.

It is feasible to vary the internal geometry of a molecule in a Monte Carlo simulation. The simplest way, which leaves the Metropolis scheme described above unchanged, is to move the atoms by a small amount. This is easy to do but it suffers from the same disadvantage as the molecular dynamics technique in that the geometry of the molecule will probably change by only a small amount, so it may take many moves, or may even be impossible, for the system to exit from its local region of phase space and explore neighbouring configurations.

A more attractive method, in principle, which allows larger conformational changes and, thus, a more efficient exploration of the phase space of a system, is to alter the internal coordinates and, in particular, the dihedral angles of the molecule directly. The problem with this approach is the one alluded to earlier in the discussion of the rotational moves of molecules, namely that a set of generalized coordinates, $\boldsymbol{Q}$, must now be used instead of Cartesian coordinates. It is possible to express integrals of the type given in Equation (12.6) in terms of the generalized coordinates, but the equation that results is the following:

$$\langle \mathcal{X} \rangle = \frac{\int \mathrm{d}\boldsymbol{Q}\, \mathcal{X}(\boldsymbol{Q})\sqrt{\|\mathbf{J}\|}\exp[-\mathcal{V}(\boldsymbol{Q})/(k_\mathrm{B}T)]}{\int \mathrm{d}\boldsymbol{Q}\sqrt{\|\mathbf{J}\|}\exp[-\mathcal{V}(\boldsymbol{Q})/(k_\mathrm{B}T)]} \tag{12.15}$$

where $\mathbf{J}$ is a matrix whose elements are related to the transformation between the two sets of coordinates:

$$J_{\alpha\beta} = \sum_{i=1}^{N} m_i \left(\frac{\partial \boldsymbol{r}_i}{\partial Q_\alpha}\right)^\mathrm{T} \frac{\partial \boldsymbol{r}_i}{\partial Q_\beta} \tag{12.16}$$

Here Q_α and Q_β are components of the generalized coordinate vector, $\boldsymbol{Q}$. It is the square root of the determinant of the matrix, $\sqrt{\|\mathbf{J}\|}$, that causes the problem

because its value, in general, will be dependent upon the conformation and so must be correctly accounted for in the Metropolis procedure. For simple molecules, the determinant is relatively straightforward to evaluate but for larger molecules it is much more complicated and it is this that has limited the application of the Monte Carlo technique to these types of systems. Having said all this, it should be noted that some workers have done Monte Carlo simulations with a 'normal' Metropolis algorithm in which some of the internal degrees of freedom of the molecules are altered. The assumption is that the errors introduced by the neglect of terms like those involving the matrix **J** in Equation (12.16) are small.

The implementation and the use of the Monte Carlo methods in this chapter owe much to the work of W. Jorgensen and co-workers, who have been some of the principal exponents of the Monte Carlo method for the simulation of condensed phase systems. Indeed, the OPLS-AA force field that was chosen for the examples in this book was developed and tested extensively with the aid of Monte Carlo simulations.

A new class and three new functions are introduced in this chapter to do Monte Carlo simulations. They are `NBModelMonteCarlo`, which is a non-bonding model class for Monte Carlo calculations, `MonteCarlo_SystemGeometry`, which is a function for performing the simulations, and two ancillary functions that are useful for analysis and are required by Example 26 at the end of the chapter. In keeping with the complexities of devising an algorithm for large molecules mentioned above, pDynamo's Monte Carlo procedure works uniquely on *isolates*. An isolate is a group of atoms (or a single atom) that have bonds between each other but with no other atoms outside their group. The internal geometry of each isolate is rigid and only its relative orientation and position can be changed. No check is done on the geometry of the isolates and so it is up to the user to ensure beforehand that all isolates of the same type have the same values for their internal coordinates.

Because of the focus on isolates, the potential energy of a system in a Monte Carlo simulation is due solely to non-bonding terms – no internal terms contribute. The energy calculated by the class `NBModelMonteCarlo` is a sum of all the non-bonding interaction energies (electrostatic and Lennard-Jones) between isolates. If N_{iso} is the number of isolates, I and J denote different isolates and i and j the atoms in those isolates, the energy can be written as

$$\mathcal{V}_{\mathrm{MC}} = \sum_{I=2}^{N_{\mathrm{iso}}} \sum_{J=1}^{I-1} S(R_{IJ}) \left\{ \sum_{i \in I} \sum_{j \in J} \frac{q_i q_j}{4\pi\epsilon_0 \epsilon r_{ij}} + 4\epsilon_{ij} \left[\left(\frac{s_{ij}}{r_{ij}} \right)^{12} - \left(\frac{s_{ij}}{r_{ij}} \right)^{6} \right] \right\} \quad (12.17)$$

The function $S(R_{IJ})$ is a non-bonding truncation function like those discussed in Section 10.2, except that it is a function of the distance between the centres of mass of each isolate, R_{IJ}. This means that *all* the individual interactions between

two isolates will be scaled by the *same* amount. The form of the function that is used is the same as that chosen by Jorgensen and his co-workers in much of their work and is

$$S(r) = \begin{cases} 1 & r < r_{\mathrm{L}} \\ \left(\frac{r_{\mathrm{U}}^2 - r^2}{r_{\mathrm{U}}^2 - r_{\mathrm{L}}^2}\right) & r_{\mathrm{L}} \leq r < r_{\mathrm{U}} \\ 0 & r \geq r_{\mathrm{U}} \end{cases} \tag{12.18}$$

where r_{U} and r_{L} are upper and lower cutoff distances, respectively.

There are two significant differences between the classes `NBModelMonteCarlo` and `NBModelABFS`. The first is that the truncation function, $S(R_{IJ})$, unlike the one used by the class `NBModelABFS`, does not have continuous first derivatives because the Monte Carlo method makes no use of the derivatives. It is, however, a good idea to have some smoothing to avoid artefacts that arise when similar configurations of a system have very different energies because some isolates happen to lie within the truncation distance and some do not. This will be especially important for systems composed of isolates which have a non-zero charge. The second difference is that `NBModelMonteCarlo` treats systems with periodic symmetry using the minimum image convention, which means that the upper cutoff distance, r_{U}, must be less than or equal to half the length of the periodic box.

The definition of the class is very similar to those of the other NB model classes. The major difference is in the arguments to the method `SetOptions`. These are now:

Class `NBModelMonteCarlo`

A class to calculate non-bonding interactions during a Monte Carlo simulation.

Method `SetOptions`

Define the values of the cutoffs to be used in the calculation of the non-bonding interactions.

Usage:

```
nbmodel.SetOptions ( cutoff = 8.5,
                     smooth = 0.5 )
```

`cutoff` is the value of the upper cutoff, r_{U}.

`smooth` is the value of the difference between the upper and lower cutoffs, $r_{\mathrm{U}} - r_{\mathrm{L}}$.

`nbmodel` is an instance of `NBModelMonteCarlo`.

The definition of the function that performs the simulations is:

Function `MonteCarlo_SystemGeometry`

Perform a simulation with the Metropolis Monte Carlo algorithm.

Usage:

```
MonteCarlo_SystemGeometry (
  system,
  acceptanceratio  = 0.4,
  adjustfrequency  = 1000,
  blocks           = 10,
  logfrequency     = 1,
  moves            = 10000,
  pressure         = 1.0,
  rng              = None,
  rotation         = 15.0,
  temperature      = 300.0,
  trajectories     = None,
  volumechange     = 400.0,
  volumefrequency  = 500 )
```

`system` is the system to be simulated.

`acceptanceratio` is a floating-point number indicating the acceptance ratio that it is desired to achieve in the Monte Carlo simulation. The default value is 0.4 (i.e. 40%).

`adjustfrequency` is an integer argument that specifies the move frequency at which the maximum values for changing the volume of the simulation box and for rotating and translating a molecule are to be adjusted. Each time these parameters are checked they are scaled by 0.95 if the current acceptance ratio is less than the desired value or by 1.05 if it is greater than the desired value. Note that the box and molecule move parameters (rotation and translation) are scaled independently. The default value for this argument is 1000.

`blocks` is the number of blocks of Monte Carlo moves to perform.

`logfrequency` is the frequency, in terms of blocks, at which data concerning the simulation are to be printed. The default is to print after each block.

`moves` gives the number of Monte Carlo moves to perform per block.

`pressure` gives the value of the pressure in atmospheres for the simulation. The default value is 1 atm.

`rng` defines the random number generator for the simulation. The function will create one itself if this argument is absent.

`rotation` is the maximum allowable rotation of a molecule in degrees about the x, y or z axis that is to be made during a molecule's move. The rotation will be chosen uniformly in the range [−`rotation`, `rotation`]. The default rotation is 15°.

`temperature` is the argument specifying the simulation temperature in kelvins and has a default value of 300 K.

`trajectories` specifies the trajectory objects, and their save frequencies, that are to be used for storing information during the simulation. This argument behaves identically to its equivalents in the functions employed for performing molecular dynamics simulations.

`translation` is the maximum allowable translation of a molecule in ångströms along each coordinate axis during a molecule's move. It is applied in the same way as the `rotation` argument and has a default value of 0.15 Å.

`volumechange` is the maximum allowable value in $Å^3$ by which the volume of the simulation box can be changed during a volume move. The volume change will be selected uniformly in the range [−`volumechange`, `volumechange`]. The default value is 400 $Å^3$.

`volumefrequency` is an integer argument giving the frequency at which volume, as opposed to molecule, moves are to be attempted. The default value of the argument is 500, which means that simulations will automatically be performed in the NPT ensemble. To generate results appropriate for the NVT ensemble this parameter must be set to 0.

Remarks: The maximum allowable rotations, translations and volume changes will fluctuate during a simulation if a non-zero `adjustfrequency` has been specified. In these cases, the values of the arguments `rotation`, `translation` and `volumechange` only apply to the first few Monte Carlo moves of the simulation.

The total number of Monte Carlo moves in a simulation is given by the product `blocks` × `moves`. The reason for using blocks of moves in a simulation rather than just specifying a total number of moves is primarily for convenience, for it allows the progress of a simulation to be monitored closely by looking

at the evolution of the averages of various properties for each block. Also, as mentioned in Section 11.2 when we discussed the analysis of molecular dynamics trajectories, the division of a simulation into statistically independent blocks provides a more accurate measure of the convergence of the values of any calculated quantities.

Two additional functions are described in this section. One of these is `MonteCarlo_IsolateInteractionEnergy`, which returns the energy of interaction between an isolate and the rest of the system, and the other is `MonteCarlo_ScaleIsolateInteractionParameters`, which scales the parameters in the MM energy function for an isolate. Their definitions are:

Methods

[Function `MonteCarlo_IsolateInteractionEnergy`] Calculate the interaction energy between an isolate and the rest of the system.

Usage:
```
energy = MonteCarlo_IsolateInteractionEnergy (
                                system, isolate )
```

`system`	is the system for which the interaction energy is to be calculated.
`isolate`	is an integer giving the index of the isolate in the system.
`energy`	is the interaction energy.

Function `MonteCarlo_ScaleIsolateInteractionParameters`

Scale the parameters for an isolate that are used in the calculation of its interaction energy with the rest of the system.

Usage:
```
MonteCarlo_ScaleIsolateInteractionParameters (
                    system, isolate,
                    chargescale  = 1.0,
                    epsilonscale = 1.0,
                    sigmascale   = 1.0 )
```

`system`	is the system containing the isolate.
`isolate`	is an integer giving the index of the isolate whose parameters are to be scaled.
`chargescale`	is the scaling factor for charges (the q_i in Equation (12.17)).
`epsilonscale`	is the scaling factor for Lennard-Jones well-depths (the ϵ_{ij} in Equation (12.17)).

`sigmascale` is the scaling factor for Lennard-Jones radii (the s_{ij} in Equation (12.17)).

12.4 Example 25

The program in this section uses the class and the simulation function described above to perform a Monte Carlo simulation of a small solute, methane, in water. The program is a simple one and is:

```
1  """Example 25."""
2
3  from Definitions import *
4
5  # . Read in the system.
6  solution = XMLUnpickle ( \
         os.path.join ( xpkpath, "ch4_water215_cubicbox_mc.xpk" ) )
7  solution.Summary ( )
8
9  # . Define a random number generator.
10 rng = Random ( )
11 rng.seed ( 899311 )
12
13 # . Do a Monte Carlo simulation.
14 trajectory = SystemGeometryTrajectory ( \
    os.path.join ( scratchpath, "ch4_water215_cubicbox_mc.trj" ), \
                                            solution, mode = "w" )
15 MonteCarlo_SystemGeometry ( solution,                          \
                           blocks       =                   20, \
                           moves        =                100000, \
                           rng          =                  rng, \
                           trajectories = [ ( trajectory, 100 ) ] )
```

Line 6 reads the XPK file that defines the system that is to be simulated. There are several ways in which solvated systems can be generated but this one was prepared using techniques very similar to those described in Section A3.2. The system itself comprises a methane molecule solvated in a cubic box of 215 water molecules. The energy model consists of the OPLS-AA force field along with the Monte Carlo NB model described above. The cutoff length for interaction is 8.5 Å and all intermolecular interactions falling between 8.0 Å and 8.5 Å are 'smoothed' using the function defined in Equation (12.18). Because the dimension of the simulation box is about 18.8 Å this cutoff value is small enough that the minimum image convention can be applied throughout the simulation.

The file also contains the atomic coordinates for the system which have been equilibrated by means of a previous Monte Carlo simulation. Because rigid molecules are being used in the simulations, the geometries of all the water molecules are the same with H—O bond distances of 0.9572 Å and H—O—H bond angles of 104.52°. The methane molecule has tetrahedral symmetry, with C—H bond lengths of 1.090 Å and H—C—H bond angles of 109.47°. These geometries were chosen because they are close to those observed experimentally and they also correspond to those that would result by optimizing the geometries of a single methane or water molecule in vacuum with the OPLS-AA force field.

Line 14 defines the trajectory object that is to be used for saving coordinate data during the Monte Carlo simulation.

Line 15 performs a Monte Carlo simulation of 2×10^6 steps in 20 blocks of 10^5 steps. Configurations are saved every 100 configurations on the trajectory giving 20 001 configurations in all for later analysis. The default parameters are used for all the remaining options, which means that the simulation is done in the NPT ensemble at a pressure of 1 atm and a temperature of 300 K. Volume moves are performed every 500 moves. The desired acceptance ratio is 40% and the move sizes are adjusted every 1000 moves so that this ratio is approached.

Simulations of methane and other small alkanes in water have proved popular because they provide simple models for investigating effects such as *hydrophobicity*, which are necessary for understanding the solvation of more complicated molecules such as lipids and proteins. Insights into the solvation of these molecules can be obtained by analysing the structure of the solvent around the solute and the energy of interaction between the solute and the solvent.

The determination of radial distribution functions for the solute and solvent atoms gives preliminary information about the structure of the solute around the solvent. This can be done in exactly the same way as for the molecular dynamics simulation of water in Section 11.3. Figure 12.2 shows the radial distribution function for the carbon of the methane and the oxygens of the water molecules. The function peaks at 3.7 Å with a height of about 1.8, then falls to a minimum at 5.4 Å before tending to a value of 1. It is to be noted that the curve is much rougher than the radial distribution function of Figure 11.2, which is due to the more limited set of data that is available for its calculation.

The number of neighbours, $n_n(r)$, may be determined by integrating the radial distribution function using the following formula:

$$n_n(r) = \frac{4\pi N_O}{V} \int_0^r s^2 g(s)\, ds \tag{12.19}$$

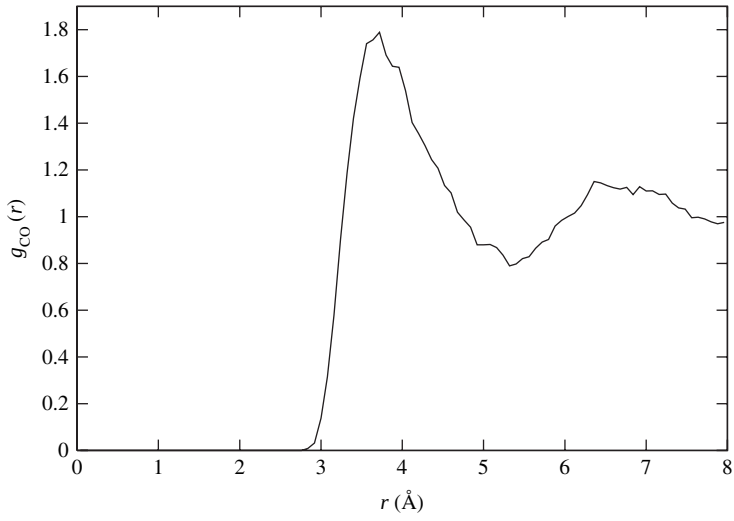


Fig. 12.2. The radial distribution function, $g_{CO}(r)$, calculated from the trajectory generated by the program of Example 25.

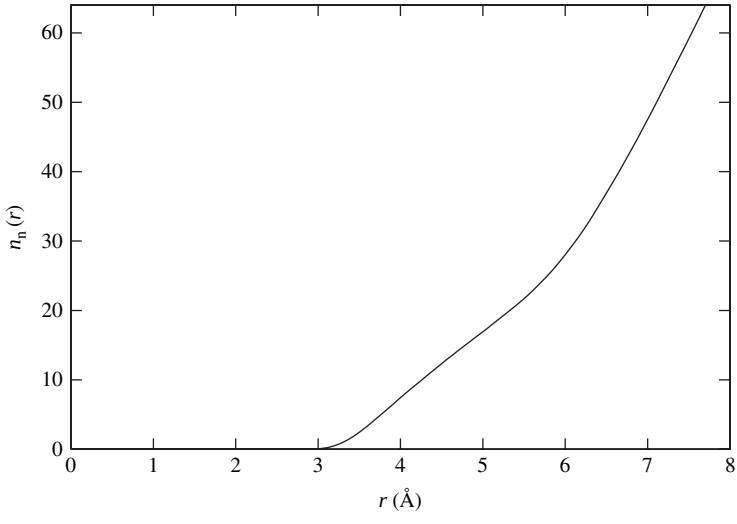


Fig. 12.3. The neighbour function, $n_n(r)$, calculated from the radial distribution function $g_{CO}(r)$.

where N_O is the number of oxygens in the simulation box and V is the average box volume. The values produced are shown in Figure 12.3. The number of water molecules in the first solvation shell of the methane molecule may be estimated by taking the value of the neighbour function at 5.4 Å, which is the distance corresponding to the first trough in the radial distribution function. The value is approximately 21.

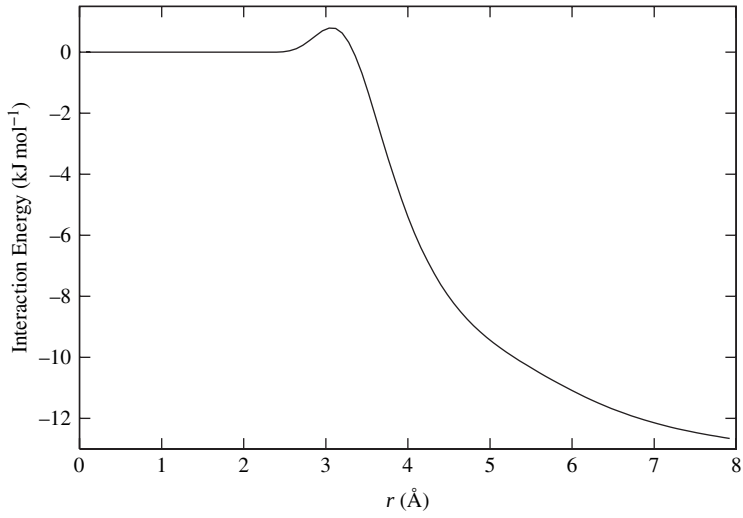


Fig. 12.4. The average energy of interaction between methane and the surrounding water molecules as a function of cutoff distance calculated from the trajectory generated by the program of Example 25.

The trajectory can also be analysed for the energetics of interaction between the solute and solvent. Figure 12.4 shows the average energy of interaction between methane and the water as a function of distance calculated from the same trajectory as the radial distribution function. The energy is zero up to about 2.5 Å and is then positive due to short-range repulsive interactions. At about 3.4 Å it becomes negative as the attractive interactions dominate and the value decreases, rapidly at first and then more slowly. At the cutoff distance, the interaction energy is approximately -13 kJ mol^{-1}. At 5.4 Å, which is the limit of the first solvation shell, the interaction energy is about $-10\,\mathrm{kJ\,mol^{-1}}$, which shows that the water molecules in the first solvation shell contribute the bulk of the interaction energy in the model. Because there are about 21 molecules in the shell the interaction energy per solvent molecule is about $-0.48\,\mathrm{kJ\,mol^{-1}}$.

12.5 Calculating free energies: statistical perturbation theory

In Section 11.6, we saw how to calculate the potential of mean force for processes in gas and condensed phase systems using the umbrella sampling technique. This is by no means the only method that is available for calculating free energies and in this section we broach other algorithms for tackling this problem. We shall limit our discussion to the free energy because this is the quantity that is critical for the interpretation of many chemical and physical phenomena. Methods for the

calculation of other thermodynamic properties, such as the entropy, exist but they can be more difficult to apply.

Recall that the Helmholtz free energy for a system is written as

$$A = -k_{\mathrm{B}}T \ln Z_{\mathrm{NVT}} \tag{12.20}$$

where the partition function, Z_{NVT}, has the form

$$Z_{\mathrm{NVT}} = \frac{1}{h^{3N}N!}\int \mathrm{d}\boldsymbol{P}\int \mathrm{d}\boldsymbol{R}\, \exp\left[-\mathcal{H}(\boldsymbol{P},\boldsymbol{R})/(k_{\mathrm{B}}T)\right] \tag{12.21}$$

and $\mathcal{H}$ is the Hamiltonian for the system. In principle, it might be possible to calculate this quantity directly from a simulation but, as noted in Section 11.6, this proves impossible in practice because extremely long simulation times are required in order to obtain results of acceptable accuracy. This problem was solved with the umbrella sampling method by using a biasing function to restrict a simulation to a certain, smaller region of configuration space that could be sampled adequately. If the property being calculated, which in our case was a PMF, required sampling from a larger region of space, this was achieved by carrying out simulations with slightly different biasing functions and collating the results for each simulation afterwards.

In this section, we consider alternative approaches for overcoming the sampling problem. They are distinct from the umbrella sampling method, but, like it, they work by restricting the region of phase space that need be sampled in a simulation. The way this is done is to compute the *differences* between the free energies of two very similar systems rather than the absolute free energy for a system given by Equation (12.20). We introduce two classes of methods to calculate free-energy differences in this section. They are the *thermodynamic integration* and *thermodynamic* or *statistical perturbation* methods. The perturbation methods will be described first.

The free-energy difference between two states, I and J, of a system with partition functions Z_I and Z_J, respectively, is

$$\begin{aligned}\Delta A_{I\to J} &= A_J - A_I \\ &= -k_{\mathrm{B}}T \ln\left(\frac{Z_J}{Z_I}\right)\end{aligned} \tag{12.22}$$

To simplify the derivation a little we suppose that the kinetic energy parts of the Hamiltonians are the same (i.e. the numbers and the masses of the particles are identical) and that only the potential energy terms, $\mathcal{V}_I$ and $\mathcal{V}_J$, differ. The kinetic energy terms in the equation cancel out and we are left with

$$\Delta A_{I\to J} = -k_{\mathrm{B}}T \ln \frac{\int \mathrm{d}\boldsymbol{R}\, \exp\left[-\mathcal{V}_J(\boldsymbol{R})/(k_{\mathrm{B}}T)\right]}{\int \mathrm{d}\boldsymbol{R}\, \exp\left[-\mathcal{V}_I(\boldsymbol{R})/(k_{\mathrm{B}}T)\right]} \tag{12.23}$$

It is possible to rearrange this equation and express the free-energy difference as an average over an ensemble of configurations for the state, I. To do this, we add and subtract terms involving the potential energy of state I to the exponential in the numerator of Equation (12.23) and then notice that the resulting equation resembles Equations (12.6)–(12.8). Denoting the probability density function for state I ρ_I and the ensemble average with respect to configurations of the state I $\langle \ldots \rangle_I$ gives

$$\begin{aligned}\Delta A_{I\to J} &= -k_\mathrm{B}T \ln \frac{\int \mathrm{d}\boldsymbol{R} \exp[-(\mathcal{V}_J(\boldsymbol{R}) - \mathcal{V}_I(\boldsymbol{R}))/(k_\mathrm{B}T)] \exp[-\mathcal{V}_I(\boldsymbol{R})/(k_\mathrm{B}T)]}{\int \mathrm{d}\boldsymbol{R} \exp[-\mathcal{V}_I(\boldsymbol{R})/(k_\mathrm{B}T)]} \\ &= -k_\mathrm{B}T \ln \int \mathrm{d}\boldsymbol{R}\, \rho_I(\boldsymbol{R}) \exp[-(\mathcal{V}_J(\boldsymbol{R}) - \mathcal{V}_I(\boldsymbol{R}))/(k_\mathrm{B}T)] \\ &= -k_\mathrm{B}T \ln \langle \exp[-(\mathcal{V}_J(\boldsymbol{R}) - \mathcal{V}_I(\boldsymbol{R}))/(k_\mathrm{B}T)] \rangle_I \end{aligned} \tag{12.24}$$

Equation (12.24) is the main formula for statistical perturbation theory. It states that the difference in free energy between two states, I and J, can be calculated by generating a trajectory for state I (using either a molecular dynamics or a Monte Carlo technique) and calculating the average of the exponential of the difference between the potential energies of states I and J for each configuration divided by $k_\mathrm{B}T$.

The average in Equation (12.24) will not converge very rapidly unless the differences between the energies of the two states I and J are very small (of the order of $k_\mathrm{B}T$). In actual applications this will not normally be the case, so it is usual to break the problem down into smaller pieces. To do this, we introduce a new Hamiltonian that is a function of a *coupling* or *perturbation parameter*, λ. This Hamiltonian, $\mathcal{H}(\boldsymbol{P}, \boldsymbol{R}, \lambda)$, is such that, when $\lambda = 0$ or 1, it is equal to the Hamiltonians of the end states, $\mathcal{H}_I$ and $\mathcal{H}_J$, but at other values it defines a series of *intermediate states* for the transition between I and J.

Many different *coupling schemes* have been used to define the intermediate states. Probably the simplest is a coupling scheme that interpolates linearly between the two states:

$$\mathcal{H}(\boldsymbol{P}, \boldsymbol{R}, \lambda) = (1-\lambda)\mathcal{H}_I(\boldsymbol{P}, \boldsymbol{R}) - \lambda \mathcal{H}_J(\boldsymbol{P}, \boldsymbol{R}) \tag{12.25}$$

More complicated coupling schemes have been proposed and may be advantageous in some cases. For example, a straightforward extension of Equation (12.25) gives a non-linear scheme:

$$\mathcal{H}(\boldsymbol{P}, \boldsymbol{R}, \lambda) = (1-\lambda)^n \mathcal{H}_I(\boldsymbol{P}, \boldsymbol{R}) - \lambda^n \mathcal{H}_J(\boldsymbol{P}, \boldsymbol{R}) \tag{12.26}$$

In other formulations, it is not the Hamiltonians that are scaled directly but parameters in the energy function. Thus, for example, a parameter, p, from an

MM energy function in an intermediate state could be written as a linear function of λ, i.e.

$$p_\lambda = (1-\lambda)p_I - \lambda p_J \tag{12.27}$$

Such coupling can, of course, lead to a very complicated functional dependence of the energy of the system on the coupling parameter, λ.

Once the new Hamiltonian has been defined, the free-energy difference between two intermediate states with different values of the perturbation parameter, λ_i and λ_j, can be calculated. Assuming, as before, that the kinetic energy terms cancel out, the difference is

$$\begin{aligned}\Delta A_{i\to j} &= A_j - A_i \\ &= -k_\mathrm{B}T \ln\langle \exp\left[-(\mathcal{V}(\boldsymbol{R},\lambda_j) - \mathcal{V}(\boldsymbol{R},\lambda_i))/(k_\mathrm{B}T)\right]\rangle_{\lambda_i}\end{aligned} \tag{12.28}$$

The total free-energy difference is, then, the sum of the individual free-energy differences between the intermediate states on going from state I to state J:

$$\Delta A_{I\to J} = \sum_{i=0}^{N_\mathrm{w}} \Delta A_{i\to(i+1)} \tag{12.29}$$

where N_w is the total number of intermediate states or windows and the states $i=0$ and $i=N_\mathrm{w}+1$ refer to the end states, I and J, respectively.

Up to now, we have focused on the free-energy difference, $A_J - A_I$. An expression for the difference in the reverse direction, $A_I - A_J$, can be obtained from Equation (12.24) by interchanging the states I and J. The values of the two differences should be equal in magnitude and opposite in sign but the relationship between the perturbation formulae is not so straightforward. In one case, the ensemble average is for a trajectory generated for state I and in the other it is for state J. The fact that the same quantity can be calculated in two independent ways provides a very useful check on the accuracy of a statistical perturbation theory calculation. If the difference between the changes in free energy for the *forwards* and *backwards* perturbations (or the *hysteresis* of the simulation) is large then the sampling in the simulation has been inadequate and the runs need to be longer or carried out with more intermediate states. It should be noted that it is unnecessary to perform two distinct sets of simulations to obtain the two free-energy differences. This is because the trajectory generated by the simulation of an intermediate state, i, can be used to determine simultaneously the free-energy differences for the windows in the forwards direction, $i \to i+1$, and in the backwards direction, $i \to i-1$.

So far the discussion has been a little abstract and it may be unclear to some readers exactly what form the perturbation from one state to another can take in real applications. Perhaps the most important point to note is that one of the

major advantages of this technique (as well as the thermodynamic integration method to be discussed below) is that the perturbation does not have to correspond to a physically realizable process. Consider the example studied using the umbrella sampling technique in Section 11.7, in which the free-energy profile was calculated for the bALA molecule as a function of one of the intramolecular hydrogen-bonding distances, but this time for the same process in a solvent. If the perturbation technique were applied to this problem, it would be physically most reasonable (and it might also be most efficient) to define the intermediate states as structures having a particular value of the distance. It is also possible, though, to employ one of the coupling formulae, either Equation (12.25) or Equation (12.26), in which case the intermediate states would correspond to a weighted superposition of the two end states. Suppose that the linear coupling formula were used. This would mean that both structures would be present in the simulation but their interactions with the solvent (and their internal interactions) would be scaled by $1-\lambda$ and λ, respectively. There would be no interactions between the solute molecules.

Because physically realizable changes do not have to be studied, it is possible to calculate the free-energy differences between 'states' of a system in which the number and the type of atoms are altered. These types of changes are sometimes called *alchemical perturbations*. Thus, for example, the relative hydration free energies of two different solute molecules can be calculated by performing a simulation in which one solute molecule is transformed into another. As we shall see in the next section, we can also obtain absolute free energies of hydration if a solute is made to vanish entirely during a simulation (i.e. the two states of the system correspond to those with the solute molecule present and with it absent). Another common application is the calculation of the relative stability of binding of two different ligand molecules to another molecule, such as a protein. In this case, the transformation would be effected by simulating the transformation between the two ligand molecules as they are bound to the host molecule.

One of the most important aids in the formulation of problems that involve the calculation of free-energy differences is the concept of a *thermodynamic cycle*. Let us take as an example the process of calculating the relative binding affinities of two ligands, A and B, to a host molecule, H. The thermodynamic cycle for this is

$$\begin{array}{ccc} & \xrightarrow{\Delta A_{\mathrm{AH}}} & \\ \mathrm{H}+\mathrm{A} & & \mathrm{HA} \\ \Delta A_{\mathrm{AB}} \downarrow & & \downarrow \Delta A_{\mathrm{AHBH}} \\ \mathrm{H}+\mathrm{B} & & \mathrm{HB} \\ & \xrightarrow{\Delta A_{\mathrm{BH}}} & \end{array} \tag{12.30}$$

There are four free-energy contributions to the cycle, the free energies of binding of the ligands to the host in solution, ΔA_{AH} and ΔA_{BH}, the free energy for the conversion of the two ligand molecules in solution, ΔA_{AB}, and that for the conversion between the two ligand complexes, ΔA_{AHBH}. Because the free energy is a thermodynamic state function this means that any free-energy difference depends only upon the nature of the end states and is independent of the path over which the change occurs. Thus, the sum of the individual free energies around the thermodynamic cycle is zero and so

$$\Delta A_{\mathrm{AH}} + \Delta A_{\mathrm{AHBH}} - \Delta A_{\mathrm{BH}} - \Delta A_{\mathrm{AB}} = 0 \tag{12.31}$$

The relative binding affinity of the two ligands is determined by the difference $\Delta A_{\mathrm{AH}} - \Delta A_{\mathrm{BH}}$. For some problems it may be possible to calculate these terms directly, although, according to Equation (12.31), the same difference can also be written as $\Delta A_{\mathrm{AB}} - \Delta A_{\mathrm{AHBH}}$. In many cases, it is easier to compute the free-energy differences for the unphysical conversion of the ligands (in solution and bound to the host) than it is to compute those for the physical process of each ligand binding to the host. Thermodynamic cycles of this sort can be formulated for many other problems of interest.

The other class of methods that we shall mention here is the thermodynamic integration methods. These also calculate the differences between the free energies of two states of a system and employ the same coupling parameter approach as the thermodynamic perturbation theory methods. Instead of Equation (12.22), however, they rely on the following identity:

$$\Delta A_{I \to J} = \int_0^1 \mathrm{d}\lambda \, \frac{\partial A}{\partial \lambda} \tag{12.32}$$

Determination of the derivatives of the free energy is straightforward. Making use of Equations (12.20) and (12.21) and the fact that the Hamiltonian depends upon the coupling parameter, λ, gives

$$\Delta A_{I \to J} = \int_0^1 \mathrm{d}\lambda \left\langle \frac{\partial \mathcal{H}}{\partial \lambda} \right\rangle_{\lambda} \tag{12.33}$$

The derivative of the Hamiltonian with respect to the perturbation parameter is easily evaluated once the coupling scheme has been defined. The integral itself is determined by performing simulations to calculate the ensemble average in the integrand at various values of λ and then applying a standard numerical integration technique to the values that result. With thermodynamic integration methods there is no equivalent of forwards and backwards perturbations because the ensemble averages depend upon only one value of λ. Instead, the precision of the calculations can be judged by estimating the error arising in the calculation of each of the ensemble averages.

We have briefly reviewed some of the principles behind the thermodynamic perturbation and thermodynamic integration methods for the calculation of free-energy differences. It has not been possible in the space available here to describe the many variations of each of these techniques that exist or any alternative methods that may be better for certain problems. It should be borne in mind though, when calculating free-energy differences, that the choice of the most appropriate method and coupling scheme need not be at all obvious and that it will be necessary to experiment to obtain the best approach. Whatever the method, extreme care should be taken to ensure that enough simulations are done and done for long enough that the results are of the precision that is desired. Free-energy calculations of all sorts are notorious for providing pitfalls for the unwary user!

12.6 Example 26

The problem addressed by the program in this section is the estimation of the free energy of water using statistical perturbation theory. The free energy is determined by taking a box of water molecules and incrementally making one of them vanish. The transformation is effected in two steps. In the first the charges on the atoms in a single water molecule are made to disappear and in the second the Lennard-Jones parameters for the oxygen are gradually reduced to zero (the Lennard-Jones parameters for the hydrogens are already zero). In each case, the parameter values are changed by linearly scaling them with the perturbation parameter, λ, and performing several simulations with values for λ between 0 and 1. In the step in which the charges are made to disappear, 21 simulations are performed, with increments in λ of 0.05. In the second step, in which the Lennard-Jones parameters are removed, 11 simulations are done and the change in λ at each step is 0.1.

The program for the step in which the charges are reduced to zero is:

```
1  """Example 26."""
2
3  from Definitions import *
4
5  # . Set some parameters.
6  NAME        = "example26.trj"
7  NLAMBDAS    =    21
8  SOLUTE      =     0
9  TEMPERATURE = 300.0
10 DLAMBDA     =   1.0 / float ( NLAMBDAS - 1 )
11 RT          = CONSTANT_MOLAR_GAS * TEMPERATURE / 1000.0
12
```

```
# . Read in the system.
solution = XMLUnpickle ( \
               os.path.join ( xpkpath, "water216_cubicbox_mc.xpk" ) )
solution.Summary ( )

# . Define a random number generator.
rng = Random ( )

# . Initialize the dictionary that will hold the free energy values.
dg = {}

# . Perform simulations at different coupling constants.
for i in range ( NLAMBDAS - 1, -1, -1 ):

    # . Reset the random number generator.
    rng.seed ( 622199 + i )

    # . Get the value of the coupling parameter.
    LAMBDA = float ( i ) * DLAMBDA

    # . Scale the solute's charge parameters.
    MonteCarlo_ScaleIsolateInteractionParameters ( \
                            solution, SOLUTE, chargescale = LAMBDA )

    # . Equilibration.
    MonteCarlo_SystemGeometry ( solution,                \
                                blocks      =         10, \
                                moves       =     100000, \
                                rng         =        rng, \
                                temperature = TEMPERATURE )

    # . Data-collection.
    mcdata = SystemGeometryTrajectory ( \
                               os.path.join ( scratchpath, NAME ), \
                                              solution, mode = "w" )
    MonteCarlo_SystemGeometry ( solution,                          \
                                blocks       =                20, \
                                moves        =            100000, \
                                rng          =               rng, \
                                temperature  =       TEMPERATURE, \
                                trajectories = [ ( mcdata, 100 ) ] )

     # . Define a trajectory object for reading.
    mcdata = SystemGeometryTrajectory ( \
```

```
                                os.path.join ( scratchpath, NAME ), \
                                             solution, mode = "r" )
44
45      # . Initialize the accumulators.
46      gb = gf = 0.0
47
48      # . Loop over the frames in the trajectory.
49      while mcdata.RestoreOwnerData ( ):
50
51          # . Get the interaction energy at i.
52          MonteCarlo_ScaleIsolateInteractionParameters ( solution, SOLUTE, \
                                                           chargescale = LAMBDA )
53          ei = MonteCarlo_IsolateInteractionEnergy ( solution, SOLUTE )
54
55          # . Calculate the energy at i-1.
56          if i > 0:
57              MonteCarlo_ScaleIsolateInteractionParameters ( solution, SOLUTE, \
                                                 chargescale = LAMBDA - DLAMBDA )
58              ej = MonteCarlo_IsolateInteractionEnergy ( solution, SOLUTE )
59              gb += math.exp ( - ( ej - ei ) / RT )
60
61          # . Calculate the energy at i+1.
62          if i < ( NLAMBDAS - 1 ):
63              MonteCarlo_ScaleIsolateInteractionParameters ( solution, SOLUTE, \
                                                 chargescale = LAMBDA + DLAMBDA )
64              ej = MonteCarlo_IsolateInteractionEnergy ( solution, SOLUTE )
65              gf += math.exp ( - ( ej - ei ) / RT )
66
67      # . Scale and save the values.
68      gb /= float ( len ( mcdata ) )
69      gf /= float ( len ( mcdata ) )
70      if i > 0:                dg[(i,i-1)] = - RT * math.log ( gb )
71      if i < ( NLAMBDAS - 1 ): dg[(i,i+1)] = - RT * math.log ( gf )
72
73  # . Output the results.
74  table = logfile.GetTable ( columns = [ 12, 12, 16, 16, 16 ] )
75  table.Start   ( )
76  table.Title   ( "Calculated Free Energies" )
77  table.Heading ( "Lambda I"     )
78  table.Heading ( "Lambda J"     )
79  table.Heading ( "dG (I->J)"    )
80  table.Heading ( "dG (I<-J)"    )
81  table.Heading ( "dG (average)" )
82  dgijtot = dgjitot = 0.0
83  for j in range ( NLAMBDAS - 2, -1, -1 ):
```

```
84      i = j + 1
85      dgij = dg[(i,j)]
86      dgji = dg[(j,i)]
87      dgijtot += dgij
88      dgjitot += dgji
89      table.Entry ( "%12.4f" % ( float ( i ) * DLAMBDA, ) )
90      table.Entry ( "%12.4f" % ( float ( j ) * DLAMBDA, ) )
91      table.Entry ( "%16.4e" % ( dgij,                  ) )
92      table.Entry ( "%16.4e" % ( dgji,                  ) )
93      table.Entry ( "%16.4e" % ( 0.5 * ( dgij - dgji ), ) )
94  table.Entry ( "Total:", alignment = "l", columnspan = 2 )
95  table.Entry ( "%20.3f" % ( dgijtot, ) )
96  table.Entry ( "%20.3f" % ( dgjitot, ) )
97  table.Entry ( "%20.3f" % ( 0.5 * ( dgijtot - dgjitot ), ) )
98  table.Stop ( )
```

Lines 6–11 define various parameters that will be needed by the program. These include the number of different values of λ to simulate, `NLAMBDAS`, the index of the water molecule that will be made to disappear, `SOLUTE`, and the temperature at which the simulations are to be performed, `TEMPERATURE`.

Line 14 reads the XPK file that defines the system that is to be simulated. This is a box of 216 water molecules prepared so that the internal geometries of all the water molecules are identical and, hence, suitable for Monte Carlo simulation. Ways in which solvent boxes of this type can be generated are described in Section A3.1.

Line 21 creates a Python dictionary that will be employed for storing the free-energy results.

Lines 24–30 start the loop over different values of λ. The loop is arranged so that the simulations at $\lambda = 1$ are carried out first and those at $\lambda = 0$ last.

Line 33 sets the scale factor for the charges of the water molecule that is being made to disappear.

Lines 36–40 perform two Monte Carlo simulations, the first to equilibrate the system with the modified energy function and the second for data collection. In all, 40 001 structures are saved for later analysis.

Line 43 creates a trajectory object that is appropriate for reading the data generated in the previous Monte Carlo simulation.

Lines 49–65 loop over each of the configurations stored in the trajectory in turn and calculate the interaction energy between the water molecule being removed and the remaining molecules in the system. In the most general case (i.e. when λ is not equal to zero or to unity), the interaction

energy is calculated for *three* different sets of parameters, which are those of the previous, the current and the subsequent simulations. Denoting these energies $\mathcal{V}_{i-1}$, $\mathcal{V}_i$ and $\mathcal{V}_{i+1}$, respectively, the free energies in the backwards, $\Delta G_{i-1\leftarrow i}$, and forwards, $\Delta G_{i\rightarrow i+1}$, directions are calculated as

$$\Delta G_{i-1\leftarrow i} = -RT \ln \left\langle \exp\left(-\frac{(\mathcal{V}_{i-1} - \mathcal{V}_i)}{RT}\right)\right\rangle_{\lambda_i} \tag{12.34}$$

$$\Delta G_{i\rightarrow i+1} = -RT \ln \left\langle \exp\left(-\frac{(\mathcal{V}_{i+1} - \mathcal{V}_i)}{RT}\right)\right\rangle_{\lambda_i} \tag{12.35}$$

Lines 68–71 terminate calculation of the free energies for the window. This is done by dividing the accumulated energy terms by the number of frames in the trajectory and applying the formula of Equation (12.28) to the resulting average.

Lines 74–98 output the results of the calculation. The free energies for each window in the forwards and backwards directions are printed as well as the average values. Output terminates with the free-energy changes for the complete transformation.

The program for performing the second step in which the Lennard-Jones parameters of the oxygen are reduced to zero is essentially the same as the program given above. The major differences lie in the number of simulations performed and the arguments that are passed to the function that scales the disappearing water molecule's interaction parameters. Reasonable results are obtained by carrying out 11 simulations and scaling both the ϵ_{ij} and s_{ij} parameters of the disappearing water molecule by λ. It should be remembered that the charge scaling factor is always zero for this part of the calculation.

Free-energy values obtained from these programs as functions of the perturbation parameter are shown in Tables 12.1 and 12.2. There is reasonable agreement between the free-energy differences calculated in the forwards and backwards directions. The total free energies for the step in which the charges are removed differ by less than 0.2 kJ mol^{-1} while there is a difference of about 1.8 kJ mol^{-1} between the values for the second step. The charges make the biggest contribution to the free-energy change and the charge and Lennard-Jones terms have opposite contributions – the removal of the electrostatic interactions between a water molecule and its neighbours requires energy whereas the removal of the Lennard-Jones interactions is favourable. The average of the forwards and backwards total free-energy changes from the simulations is 25.6 kJ mol^{-1}, which is in good agreement with the experimental value of 26.5 kJ mol^{-1} (at 25 °C).

Table 12.1 *The free-energy change as a function of the perturbation parameter, λ, for the first step of the statistical perturbation calculation of Example 26 in which the electrostatic interactions between a single water molecule and the remainder of the molecules in the system are eliminated. Energies are in kJ mol^{-1}.*

λ_i	λ_j	$\Delta G_{i\to j}$	$\Delta G_{i\leftarrow j}$	$\Delta G_{\text{average}}$
1.00	0.95	4.73	−4.22	−4.48
0.95	0.90	3.74	−3.99	−3.87
0.90	0.85	3.70	−3.86	−3.78
0.85	0.80	3.57	−3.34	−3.45
0.80	0.75	2.99	−3.17	−3.08
0.75	0.70	2.76	−2.63	−2.69
0.70	0.65	2.40	−2.42	−2.41
0.65	0.60	2.11	−1.93	−2.02
0.60	0.55	1.66	−1.66	−1.66
0.55	0.50	1.42	−1.52	−1.47
0.50	0.45	1.32	−1.47	−1.40
0.45	0.40	1.30	−1.13	−1.22
0.40	0.35	0.96	−1.06	−1.01
0.35	0.30	0.89	−0.58	−0.74
0.30	0.25	0.45	−0.61	−0.53
0.25	0.20	0.46	−0.56	−0.51
0.20	0.15	0.45	−0.43	−0.44
0.15	0.10	0.30	−0.32	−0.31
0.10	0.05	0.18	−0.28	−0.23
0.05	0.00	0.15	−0.10	−0.12
Total		35.55	−35.27	−35.41

Finally we note that, in both these programs, the whole perturbation is carried out at once. In a real study it is more likely that the perturbation would be broken up into several different jobs so that the results of one calculation could be verified before those of the next were begun.

Exercises

12.1 Analyse in more detail the trajectory generated in the program described in Section 12.4.

(a) Write a program to calculate the average of the energies of interaction between methane and the surrounding water molecules. In addition to reproducing the data in Figure 12.4, determine the relative importance of the electrostatic and the Lennard-Jones contributions to the interaction. How does the size of these inter-

Table 12.2 *The free-energy change as a function of the perturbation parameter, λ, for the second step of the statistical perturbation calculation of Example 26 in which the Lennard-Jones interactions between a single, chargeless water molecule and the remainder of the molecules in the system are eliminated. Energies are in kJ mol^{-1}.*

λ_i	λ_j	$\Delta G_{i\rightarrow j}$	$\Delta G_{i\leftarrow j}$	$\Delta G_{\text{average}}$
1.0	0.9	−0.73	1.25	0.99
0.9	0.8	−1.85	2.21	2.02
0.8	0.7	−1.70	1.31	1.50
0.7	0.6	−1.11	1.21	1.16
0.6	0.5	−1.17	1.18	1.18
0.5	0.4	−0.74	1.43	1.09
0.4	0.3	−0.81	0.95	0.88
0.3	0.2	−0.56	0.63	0.60
0.2	0.1	−0.01	0.34	0.17
0.1	0.0	−0.22	0.16	0.19
Total		−8.90	10.70	9.78

actions compare with, for example, the interactions between two water molecules in solution?

(b) What is the structure of the water about the methane molecule? For example, are there hydrogen bonds?

(c) Estimate the size of the effect of the truncation of the intermolecular interactions on the values of the electrostatic and Lennard-Jones energies.

12.2 The programs in Section 12.6 calculated the free energy of a water molecule in water. Do similar calculations using the same programs to calculate the free energies of hydration of other small solutes, such as methane (see Section 12.4) and the chloride anion. A good strategy in both cases is to transform the solute molecules into the Lennard-Jones particle that acted as an intermediate in the calculations of Section 12.6. Try various approaches for changing the parameters. Which are the most effective? Note that, for the negatively charged chloride anion, a correction that accounts for the neglect of the electrostatic interactions beyond the cutoff distance will have to be made in order to obtain reasonable agreement with experimental values. Try using the Born expression of Equation (10.10) to estimate the size of this effect.

Another point to investigate concerns the formula for the calculation of the free-energy differences. Example 26 employed Equation (12.28) for the forwards and backwards transformations and averaged the two values, but other approaches have been proposed. One of these is the *simple overlap*

sampling method which is based on the following expression:

$$\Delta A_{i\to j} = -k_{\mathrm{B}}T \ln \left\{ \frac{\langle \exp[-(\mathcal{V}(\boldsymbol{R}, \lambda_j) - \mathcal{V}(\boldsymbol{R}, \lambda_i))/(2k_{\mathrm{B}}T)] \rangle_{\lambda_i}}{\langle \exp[+(\mathcal{V}(\boldsymbol{R}, \lambda_j) - \mathcal{V}(\boldsymbol{R}, \lambda_i))/(2k_{\mathrm{B}}T)] \rangle_{\lambda_j}} \right\} \quad (12.36)$$

Modify Example 26 to use this equation. How do the results compare?

Appendix 1
The pDynamo library

The algorithms and capabilities of the pDynamo library discussed in this book represent only a portion of those that are available. The choice of which to include has been highly subjective and, due to space restrictions and the perseverance of the author (!), relatively small. The topics that I most regret omitting or skimping upon include density functional theory, the calculation of surfaces and volumes, continuum methods for including solvation effects, *non-equilibrium methods*, especially those for the determination of free energies, and enhanced methods, such as *transition path sampling*, for the investigation of chemical reactions and other *rare events*. In any case, readers are encouraged to investigate these and alternative methods themselves, for many of the techniques presented in the book are the subject of active research and are undergoing continual improvement.

pDynamo itself and the example programs described in the text are available on the World Wide Web. At the time of publication, the appropriate addresses were www.ibs.fr and www.pdynamo.org. The websites give full details about how to download, install and use the library and the types of machines upon which it has been tested.

For convenience, we include here tables of the methods and attributes of the `System` class and of the other classes and functions that were encountered in the book along with the section in which they were first described or in which significant new capabilities were introduced. These are in Tables A1.1, A1.2 and A1.3, respectively. The `System` class is, in many ways, the central class of the library although it has several other important features which could not be treated in the text.

Table A1.1 *The pDynamo* `System` *class*

Name	Section
Attributes	
`angles`	3.2
`atoms`	2.2
`bonds`	3.2
`coordinates3`	2.2
`dihedrals`	3.2
`electronicstate`	4.7
`symmetry`	10.5
`symmetryparameters`	10.5
Methods	
`AtomicCharges`	4.7
`BondsFromCoordinates`	3.2
`DefineMMModel`	5.3.3
`DefineNBModel`	5.3.3
`DefineQCModel`	4.7, 6.3
`DefineSoftConstraints`	5.6
`DefineSymmetry`	10.5
`DipoleMoment`	4.7
`Energy`	4.7
`Summary`	2.2

Table A1.2 *pDynamo classes*

Name	Section
pBabel	
`CMLFileReader`, `CMLFileWriter`	2.5
`MOLFileReader`, `MOLFileWriter`	2.5
`PDBFileReader`, `PDBFileWriter`	2.5
`SMILESReader`, `SMILESWriter`	2.5
`SystemGeometryTrajectory`	7.9, 9.4
`SystemSoftConstraintTrajectory`	11.6
`SystemVelocityTrajectory`	Exercise 11.1
`XYZFileReader`, `XYZFileWriter`	2.5
pCore	
`Coordinates3`	3.3, 3.5, 3.6
`LogFileWriter`	2.4.4, 3.4
`Matrix33`	3.5, 3.7
`ObjectFunction`	4.9

Table A1.2 (cont.)

Name	Section
`Selection`	2.3
`Statistics`	9.4, 11.2
`TextLogFileWriter`	3.4
`TextTable`	3.4
`Vector`	3.5
`Vector3`	3.5, 4.7
`XHTMLLogFileWriter`	3.4
pDynamo	
`Angle`	3.3
`Atom`	2.2
`AtomContainer`	2.2, 3.7
`Bond`	3.3
`CrystalClass`	10.5
`CrystalClassCubic`	10.5
`Dihedral`	3.3
`ElectronicState`	4.7
`MMModel`	5.3.3
`MMModelOPLS`	5.3.3
`NBModel`	5.3.3
`NBModelABFS`	10.2
`NBModelEwald`	10.7
`NBModelFull`	5.3.3
`NBModelMonteCarlo`	12.3
`QCModel`	4.7
`QCModelMNDO`	4.7
`SoftConstraint`	5.6
`SoftConstraintContainer`	5.6, 9.7, 11.7
`SoftConstraintDistance`	5.6
`SoftConstraintEnergyModel`	5.6
`SoftConstraintEnergyModelHarmonic`	5.6
`SoftConstraintEnergyModelHarmonicRange`	5.6
`SoftConstraintTether`	5.6
`System`	2.2, 3.2, 4.7, 5.3.3, 5.6, 10.5
`SystemGeometryObjectFunction`	4.9

Table A1.3 *pDynamo functions*

Name	Section
`BakerOptimize_SystemGeometry`	7.5
`BuildCubicSolventBox`	A3.1
`Clone`	2.2
`CMLFile_FromSystem`	2.5
`CMLFile_ToCoordinates3`	3.7
`CMLFile_ToSystem`	2.5
`ConjugateGradientMinimize_SystemGeometry`	7.3
`LangevinDynamics_SystemGeometry`	Exercise 11.3
`LeapFrogDynamics_SystemGeometry`	11.4
`MergeByAtom`	2.2
`MOLFile_FromSystem`	2.5
`MOLFile_ToCoordinates3`	3.7
`MOLFile_ToSystem`	2.5
`MonteCarlo_IsolateInteractionEnergy`	12.3
`MonteCarlo_ScaleIsolateInteractionParameters`	12.3
`MonteCarlo_SystemGeometry`	12.3
`NormalModesPrint_SystemGeometry`	8.2
`NormalModesTrajectory_SystemGeometry`	8.4
`NormalModes_SystemGeometry`	8.2
`PDBFile_FromSystem`	2.5
`PDBFile_ToCoordinates3`	3.7
`PDBFile_ToSystem`	2.5, 5.3.3
`PruneByAtom`	2.2
`RadialDistributionFunction`	11.2
`SAWOptimize_SystemGeometry`	7.9
`SMILES_FromSystem`	2.5
`SMILES_ToSystem`	2.5
`SelfDiffusionFunction`	11.2
`SolvateSystemBySuperposition`	A3.2
`SteepestDescentPath_SystemGeometry`	7.7
`ThermodynamicsRRHO_SystemGeometry`	8.6
`VelocityVerletDynamics_SystemGeometry`	9.2
`WHAMEquationSolver`	11.6
`XMLPickle`	2.5
`XMLUnpickle`	2.5
`XYZFile_FromSystem`	2.5
`XYZFile_ToCoordinates3`	3.7
`XYZFile_ToSystem`	2.5

Appendix 2

Mathematical appendix

A2.1 The eigenvalues and eigenvectors of a matrix

The eigenvalues, λ, and eigenvectors, $\boldsymbol{v}$, of an $N \times N$ matrix, $\mathbf{A}$, are defined by the following equation:

$$\mathbf{A}\boldsymbol{v} = \lambda \boldsymbol{v} \tag{A2.1}$$

$\mathbf{A}$ can be any type of square matrix but we shall restrict our discussion to symmetric matrices because these encompass all the examples that are required in the book. Symmetric matrices are those for which the elements are all real and whose transpose equals itself, i.e. $\mathbf{A}^{\mathrm{T}} = \mathbf{A}$.

The eigenvalues of a matrix are determined by solving its *characteristic* or *secular equation*. This is

$$\|\mathbf{A} - \lambda \mathbf{I}\| = 0 \tag{A2.2}$$

where $\mathbf{I}$ is the $N \times N$ identity matrix and the double straight lines denote the determinant. Expanding the determinant gives an Nth order polynomial whose N roots are the eigenvalues λ.

The eigenvalues and eigenvectors of symmetric matrices have some important properties. One of these is that the eigenvalues are real, either negative, zero or positive, but not complex. Another is that the eigenvectors, which are also real, form an orthonormal set, which means that different eigenvectors have zero overlap or a zero dot product, i.e.

$$\boldsymbol{v}_i^{\mathrm{T}} \boldsymbol{v}_j = 0 \quad i \neq j \tag{A2.3}$$

Eigenvectors with different eigenvalues are automatically mutually orthogonal, whereas those with identical eigenvalues can always be chosen to be so.

The orthogonality property of the eigenvectors implies that an orthogonal matrix, $\mathbf{V}$, may be constructed whose columns are the (normalized) eigenvectors.

This matrix may then be employed to diagonalize the original matrix, **A**, as follows:

$$\mathbf{V}^{\mathrm{T}}\mathbf{A}\mathbf{V} = \mathbf{\Lambda} \tag{A2.4}$$

where $\mathbf{\Lambda}$ is a diagonal matrix with the eigenvalues, λ_i, as its diagonal elements.

Transformations of the type in Equation (A2.4) form the basis of most numerical techniques for solving Equation (A2.1). Although a full diagonalization requires a knowledge of a matrix's eigenvectors, it proves relatively straightforward to come up with orthogonal matrices that make the matrix 'more' diagonal. Repeated application of these matrices then permits a complete transformation to diagonal form and the eigenvalues and eigenvectors of the matrix to be found. Matrix diagonalization algorithms of this sort are standard parts of numerical libraries and very efficient implementations exist. Their cost scales as $O(N^3)$ with matrix dimension N.

More complicated versions of Equation (A2.1) exist. An example of one of these generalized eigenvalue equations is

$$\mathbf{A}\boldsymbol{v} = \lambda\mathbf{B}\boldsymbol{v} \tag{A2.5}$$

in which **B**, as well as **A**, is a symmetric matrix. There are a number of strategies for solving problems of this type but one, which is employed by the quantum chemical algorithms of Section 4.5.1, proceeds by generating a matrix, **X**, such that

$$\mathbf{X}^{\mathrm{T}}\mathbf{B}\mathbf{X} = \mathbf{I} \tag{A2.6}$$

Equation (A2.5) may then be rearranged into the simpler form

$$\mathbf{A}'\boldsymbol{v}' = \lambda\boldsymbol{v}' \tag{A2.7}$$

in which

$$\mathbf{A}' = \mathbf{X}^{\mathrm{T}}\mathbf{A}\mathbf{X} \tag{A2.8}$$

$$\boldsymbol{v}' = \mathbf{X}^{-1}\boldsymbol{v} \tag{A2.9}$$

Solution occurs by transforming **A** to $\mathbf{A}'$ using Equation (A2.8), diagonalizing $\mathbf{A}'$ to obtain the eigenvalues and the modified eigenvectors, $\boldsymbol{v}'$, and then *back-transforming* the latter to the desired eigenvectors by multiplying the $\boldsymbol{v}'$ by **X**.

The success of this approach relies on the ability to construct a suitable transformation matrix, **X**, from **B**. This is so in the quantum chemical case because the matrix **B** is *positive-definite*, which means that all of its eigenvalues are positive. There are a number of definitions of **X** in use but one, which is applicable when none of the eigenvalues of **B** are very close to zero, is

$$\mathbf{X} \equiv \mathbf{B}^{-\frac{1}{2}} = \mathbf{V}\mathbf{\Lambda}^{-\frac{1}{2}}\mathbf{V}^{\mathrm{T}} \tag{A2.10}$$

In this equation, $\mathbf{V}$ is the matrix of the eigenvectors of $\mathbf{B}$, not $\mathbf{A}$ (!), and $\mathbf{\Lambda}^{-\frac{1}{2}}$ is a diagonal matrix whose diagonal elements are the inverse square roots of the eigenvalues of $\mathbf{B}$.

A2.2 The method of Lagrange multipliers

A common mathematical problem is the minimization (or maximization) of the value of a function, $\mathcal{F}$, with respect to the function's variables, $\boldsymbol{v}$. That maximization and minimization are equivalent is apparent if $\mathcal{F}(\boldsymbol{v})$ is replaced by $-\mathcal{F}(\boldsymbol{v})$. Often a minimization is required that puts no restrictions on the values of the variables. For these *unconstrained* cases, many standard minimization procedures exist, some of which are described in Chapter 7.

A related problem is the minimization of a function, but this time subject to a series of constraint conditions. These are also functions of the variables, $\boldsymbol{v}$, and may be expressed as

$$\Lambda_k(\boldsymbol{v}) = 0 \;\forall\; k = 1, \ldots, N_\mathrm{c} \tag{A2.11}$$

It should be emphasized that the constraint conditions need to be compatible. It is easy to come up with sets of constraints that cannot be satisfied simultaneously!

Assuming that the constraints are *independent*, their number, N_c, must be less than the number of variables, N_v, which means that the space in which the minimization is to be done has dimension $N_\mathrm{v} - N_\mathrm{c}$. A strategy that is sometimes possible is to devise a new set of $N_\mathrm{v} - N_\mathrm{c}$ independent variables, $\boldsymbol{v}'$, from the original ones that incorporate the constraint behaviour. The function, $\mathcal{F}(\boldsymbol{v}')$, can then be minimized in the usual way. In practice, however, this approach is often not feasible because the forms of the constraint conditions do not permit the required manipulations.

An alternative approach is the method of Lagrange multipliers which proceeds by defining a new $(N_\mathrm{v} + N_\mathrm{c})$-dimensional function, $\mathcal{L}$, that is a function of the original variables, $\boldsymbol{v}$, and of N_c new variables, λ_k:

$$\mathcal{L}\left(\boldsymbol{v}, \lambda_1, \ldots, \lambda_{N_\mathrm{c}}\right) = \mathcal{F}(\boldsymbol{v}) - \sum_{k=1}^{N_\mathrm{c}} \lambda_k \Lambda_k(\boldsymbol{v}) \tag{A2.12}$$

This function is then minimized, in an unconstrained fashion, with respect to the original and new variables. Performing the differentiation with respect to $\boldsymbol{v}$ and the λ_k gives the following set of $(N_\mathrm{v} + N_\mathrm{c})$ equations that must be solved:

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial \boldsymbol{v}} &= \frac{\partial \mathcal{F}}{\partial \boldsymbol{v}} - \sum_{k=1}^{N_\mathrm{c}} \lambda_k \frac{\partial \Lambda_k}{\partial \boldsymbol{v}} \\ &= \mathbf{0} \end{aligned} \tag{A2.13}$$

$$\frac{\partial \mathcal{L}}{\partial \lambda_k} = \Lambda_k$$
$$= 0 \; \forall \; k = 1, \ldots, N_c \tag{A2.14}$$

Equation (A2.14) is clearly a restatement of the original constraint conditions of Equation (A2.11).

Although it may not be obvious, the unconstrained minimization of $\mathcal{L}$ is equivalent to the constrained minimization of $\mathcal{F}$. To get an idea of why this is so, consider a reference point in variable space, $\boldsymbol{v}_0$, and an infinitesimal displacement away from it, $\boldsymbol{\delta}$. The value of the kth constraint condition at the displaced point, $\boldsymbol{v}_0 + \boldsymbol{\delta}$, can be approximated using a Taylor expansion about the reference point as follows:

$$\Lambda_k (\boldsymbol{v}_0 + \boldsymbol{\delta}) \approx \Lambda_k (\boldsymbol{v}_0) + \boldsymbol{\delta}^{\mathrm{T}} \left. \frac{\partial \Lambda_k}{\partial \boldsymbol{v}} \right|_{\boldsymbol{v}_0} \tag{A2.15}$$

If the reference point obeys the constraint condition, the first term on the right-hand side of Equation (A2.15) is zero, and implies that the displaced point will itself only satisfy the constraint when the displacement is orthogonal to the constraint's derivative vector. We can now make use of this result by interpreting Equation (A2.13) which says that, at a stationary point, the components of the derivative of $\mathcal{F}$ with respect to $\boldsymbol{v}$ are zero in the space orthogonal to the derivatives of the constraint conditions. This means that optimization can occur in the part of variable space that does not violate the constraints, i.e. those parts that are orthogonal to the $\partial \Lambda_k / \partial \boldsymbol{v}$, but not in those parts that do, i.e. those parts that are spanned by the $\partial \Lambda_k / \partial \boldsymbol{v}$.

Appendix 3

Solvent boxes and solvated molecules

In the last few chapters of the book, we concentrated on studying systems with periodic boundary conditions. These systems corresponded either to pure solvent or to simple solutes immersed in solvent. In the examples, it was assumed that a partially or fully equilibrated solvent or solvated system was available for the simulations that we performed but we did not describe how they could be obtained in the first place. This omission is rectified here by considering two example programs that can carry out these tasks.

A3.1 Example 27

It should be stated initially that there is no unique way of creating solvent boxes. The method adopted here works well for the examples given in this book but it need not be the most efficient or the most appropriate if, for example, non-cubic boxes are required.

A simple way of building an approximate solvent box that is cubic in shape is to construct a regular cubic lattice and centre solvent molecules on the lattice points. The lattice should contain at least as many points as the number of solvent molecules and the spacing between lattice points should be large enough that neighbouring molecules do not overlap. In pDynamo, this procedure is implemented in the function `BuildCubicSolventBox` whose definition is:

Function `BuildCubicSolventBox`

Build a cubic box of solvent molecules.

Usage:

```
solvent = BuildCubicSolventBox (
              molecule, nmolecules )
```

`molecule` is an instance of `System` containing a single solvent molecule. The operations performed by the function are purely geometric

and so the only information that it requires are the molecule's atom composition and its coordinates.

`nmolecules` is the number of molecules to put in the box. This number will often, although need not, be an exact cube.

`solvent` is an instance of `System` corresponding to the cubic solvent box.

The steps employed by the function are: (i) determine a size, h, for the solvent molecule. This is done by finding the box of smallest dimension that will enclose the molecule taking into account the fact that the molecule's atoms have finite radii; (ii) construct a regular cubic lattice with spacing h that contains, at the minimum, `nmolecules` points; (iii) place molecules on the lattice points. These sites are chosen at random unless `nmolecules` is an exact cube in which case all sites are filled; and (iv) randomly rotate the coordinates of each molecule on the lattice points.

A program that uses this function is:

```
"""Example 27."""

from Definitions import *

# . Define some parameters.
MOLECULENAME =    "water"
NLINEAR      =          6
NMOLECULES   = NLINEAR**3

# . Define the MM and NB models.
mmmodel = MMModelOPLS ( "booksmallexamples" )
nbmodel = NBModelMonteCarlo ( )

# . Define the solvent molecule.
molecule = MOLFile_ToSystem ( \
                os.path.join ( molpath, MOLECULENAME + ".mol" ) )
molecule.Summary ( )

# . Build the cubic system.
solvent = BuildCubicSolventBox ( molecule, NMOLECULES )
solvent.DefineMMModel ( mmmodel )
solvent.DefineNBModel ( nbmodel )
solvent.Summary ( )

# . Do Monte Carlo simulations to equilibrate the system.
```

```
25 MonteCarlo_SystemGeometry ( solvent,           \
                               blocks   =      5, \
                               moves    = 100000, \
                               pressure = 1000.0  )
26 MonteCarlo_SystemGeometry ( solvent,           \
                               blocks   =     10, \
                               moves    = 100000  )
27
28 # . Calculate and print the final density.
29 mass    = sum ( solvent.atoms.GetItemAttributes ( "mass" ) )
30 volume  = solvent.symmetryparameters.volume
31 density = ( mass / volume ) * ( UNITS_MASS_AMU_TO_KG * 1.0e+30 )
32 logfile.Paragraph ( \
                "Solvent density = %.2f kg m^-3." % ( density, ) )
33
34 # . Save the system.
35 XMLPickle ( os.path.join ( xpkpath, \
             MOLECULENAME + 'NMOLECULES' + "_cubicbox_mc.xpk" ), \
                                                    solvent )
```

Lines 6–8 define some parameters that are needed by the program.

Line 15 reads in the definition of the solvent molecule, in this case water.

Lines 19–22 create a water box with 216 ($\equiv 6^3$) molecules and assign to it appropriate MM and NB energy models.

Lines 25–26 perform two Monte Carlo simulations at constant temperature and pressure to equilibrate the system. This is essential to ensure that the regular lattice geometry of the starting structure is disrupted and that cavities due to empty sites are removed. The function `BuildCubicSolventBox` is conservative and so the system it builds will normally have a volume substantially greater than that required. Hence, a simulation is first done at higher pressure to rapidly reduce the box size before equilibration at the desired pressure is carried out.

It is to be noted that the solvent molecules in the system generated by the function `BuildCubicSolventBox` have identical internal geometries and so are suitable for simulation with the Monte Carlo technique. If it is desired to have a water box for molecular dynamics simulation, these statements can be replaced by ones to an appropriate function, such as `LeapFrogDynamics_SystemGeometry`.

Lines 29–32 calculate and print the density of the system resulting from the Monte Carlo calculation in units of kg m^{-3}. This serves as a rapid check on how well the system has been constructed. For water, the experimental value at 300 K is about 996 kg m^{-3}.

Line 35 saves the prepared system as an XPK file.

Programs similar to this one were employed to generate the water boxes simulated in Examples 20, 22 and 26 of Sections 10.6, 11.3 and 12.6, respectively.

A3.2 Example 28

The last section outlined a method for the construction of solvent boxes. A complementary task concerns the production of the coordinates of condensed phase systems containing a mixture of molecules, such as, for example, a solute molecule in solvent. Just as for the problem addressed in the last section, there is no unique way of doing this. A possible approach would be to adapt the `BuildCubicSolventBox` function to handle molecules of different types. Another method, which is the one we adopt here, is to overlay the solvent box onto the solute molecule and then delete solvent molecules that overlap with any of the solute molecule's atoms. In pDynamo a function that can do this is:

Function `SolvateSystemBySuperposition`

Solvate a system by superimposing it upon a solvent box and removing all overlapping solvent molecules.

Usage:
```
solution = SolvateSystemBySuperposition (
                                  solute, solvent )
```

`solute` is an instance of `System` defining the solute.
`solvent` is an instance of `System` containing the solvent box.
`solution` is an instance of `System` with the solvated system.

The operation of this function is purely geometric. It first centres the solute and the solvent box at the origin and then determines which atoms of solvent molecules overlap with those of the solute. This is done by assigning suitable elemental radii to the atoms of each of the systems. Finally, the solvated system is created by merging the solute system with those solvent molecules that have no overlap with the solute.

The example program in this section creates a solvated methane system equivalent to that simulated in Example 25 of Section 12.4. The program uses the water box that was produced by Example 27 and an appropriate definition for methane and is:

```
1 """Example 28."""
2
3 from Definitions import *
```

```
4
5 # . Define the MM and NB models.
6 mmmodel = MMModelOPLS ( "booksmallexamples" )
7 nbmodel = NBModelMonteCarlo ( )
8
9 # . Define the solute molecule.
10 solute = MOLFile_ToSystem ( \
                          os.path.join ( molpath, "methane.mol" ) )
11 solute.Summary ( )
12
13 # . Define the solvent box.
14 solvent = XMLUnpickle ( \
             os.path.join ( xpkpath, "water216_cubicbox_mc.xpk" ) )
15
16 # . Create the solvated system.
17 solution       = SolvateSystemBySuperposition ( solute, solvent )
18 solution.label = "Methane in Water."
19 solution.DefineMMModel ( mmmodel )
20 solution.DefineNBModel ( nbmodel )
21 solution.Summary ( )
22
23 # . Do a Monte Carlo calculation to equilibrate the system.
24 MonteCarlo_SystemGeometry ( solution,       \
25                             blocks =    10, \
26                             moves  = 100000 )
27
28 # . Save the system.
29 XMLPickle ( \
        os.path.join ( xpkpath, "ch4_water215_cubicbox_mc.xpk" ), \
                                                        solution )
```

The structure of this program is similar to that of Example 27 and so it will not be described in detail. *Line 17* is the one on which the solvated methane system is produced. In this case, only a single water molecule overlaps with the methane and so the resulting system comprises methane and 215 water molecules. Likewise, system construction is followed by Monte Carlo (or molecular dynamics) simulation at constant temperature and pressure. As before this is essential so as to remove cavities caused by the removal of solvent molecules and to ensure that the solvent structure fully adapts itself to the solute.

Bibliography

General references

In keeping with the spirit of the book as a practical introduction, no attempt to give a complete bibliography has been made. References to work that is referred to explicitly are, of course, listed here but the remainder represent a highly subjective selection and have been chosen because they give overviews of certain topics or because they complement the discussion of algorithms or applications that occur in the text. Apologies go to those many workers who have contributed to the field of molecular simulation in one way or another but whose work is not acknowledged.

Specific references are given for each chapter. There are, however, a number of general references that readers will probably either need or find useful. Of the many available in each category, the author's recommendations are listed below.

Python

There are many good books that describe Python programming but by far the most comprehensive resource is the Python website whose address is www.python.org. Complete documentation about the language as well as links to tutorials and other references may be found there.

Numerical algorithms

A book that provides a good all round introduction to the subject and is a mine of valuable information is

1.1 W. H. Press, S. A. Teukolsky, W. T. Vetterling and B. P. Flannery. *Numerical Recipes in C: The Art of Scientific Computing*. Second Edition, Cambridge University Press, 1992.

Molecular simulations

A very good book that covers a wide range of simulation methodologies for atomic and small-molecule systems is

1.2 M. P. Allen and D. J. Tildesley. *Computer Simulations of Liquids*. Oxford University Press, 1987.

Other books covering similar topics are

1.3 D. Frenkel and B. Smit. *Understanding Molecular Simulation: From Algorithms to Applications*. Second Edition, Academic Press, 2001.
1.4 D. C. Rapaport. *The Art of Molecular Dynamics Simulation*. Second Edition, Cambridge University Press, 2004.

Two books giving overviews of many molecular modeling techniques are

1.5 C. J. Cramer. *Essentials of Computational Chemistry: Theories and Models*. Second Edition, J. Wiley & Sons, 2004.
1.6 A. R. Leach. *Molecular Modelling: Principles and Applications*. Second Edition, Prentice Hall, 2001.

In addition to these general texts, there are volumes that contain compilations of articles about various aspects of molecular simulations. One good series that is published regularly is

1.7 K. B. Lipkowitz *et al.* (Editors). *Reviews in Computational Chemistry*. J. Wiley & Sons.

A very comprehensive set of articles about various methods in computational chemistry may be found in the volumes of the following work:

1.8 P. v. R. Schleyer (Editor in Chief). *Encyclopedia of Computational Chemistry*. John Wiley & Sons, 1998.

There is a vast and ever-growing literature about molecular simulation techniques and reviews dealing with state of the art methods and applications appear regularly in scholarly journals. Nevertheless, some useful older references that emphasize simulations of biomolecular systems are

1.9 J. A. McCammon and S. Harvey. *Dynamics of Proteins and Nucleic Acids*. Cambridge University Press, 1987.
1.10 C. L. Brooks III, M. Karplus and B. M. Pettitt. 'Proteins: a theoretical perspective of dynamics, structure and thermodynamics'. *Adv. Chem. Phys.* **71**, 1–259, 1988.
1.11 W. van Gunsteren, P. Weiner and A. Wilkinson (Editors). *Computer Simulation of Biomolecular Systems: Theoretical and Experimental Applications*. Volumes 1, 2 and 3, ESCOM, 1989, 1993 and 1997.

Chapter 2

Chemical models

The representation of chemical information, particularly molecular structure, interrogation of this information and transformation between different representations is the subject of the discipline of *chemoinformatics*. A good introductory textbook is

2.1 J. Gasteiger and T. Engel. *Basic Chemoinformatics: A Textbook*. Wiley-VCH, 2003.

Most pDynamo modules concerned with reading and writing molecular representations are in pDynamo's `pBabel` package. This package was named in homage to a very widely used program, called `Babel`, that was designed for converting between different molecular representations. The original `Babel` was developed by P. Walters and M. Stahl at the University of Arizona but this has been superseded by the program `OpenBabel`, details of which may be found at http://openbabel.sourceforge.net. Both `Babel` and `OpenBabel` can handle a great many more molecular formats than pDynamo.

The formats of many molecular representations are described in the literature but good up to date sources for those formats presented in the text may be found on the web. The appropriate links are: CML – http://cml.sourceforge.net; InChI – www.iupac.org/inchi; MOL – www.mol.com; PDB – www.rcsb.org; and SMILES – www.daylight.com.

The myoglobin that is shown in Figure 2.2 is actually from the sperm whale. The structure has the PDB identification code 1CQ2 and is described in

2.2 F. Shu, V. Ramakrishnan and B. P. Schoenborn. 'Enhanced visibility of hydrogen atoms by neutron crystallography on fully deuterated myoglobin'. *Proc. Natl. Acad. Sci. USA* **97**, 3872–7, 2000.

Chapter 3

Internal coordinates

The definition for the dihedral angle in the text can be found in

3.1 H. Bekker, H. J. C. Berendsen and W. F. van Gunsteren. 'Force and virial of torsional-angle-dependent potentials'. *J. Comput. Chem.* **16**, 527–33, 1995.

Miscellaneous transformations

When dealing with some of the more complicated coordinate transformations it is often handy to have a reference book on classical mechanics. A good one is

3.2 H. Goldstein. *Classical Mechanics*. Second Edition, Addison-Wesley, 1980.

Superimposing structures

The method of superposition of two sets of coordinates is described in the following articles by Kabsch and by Kneller:

3.3 W. Kabsch. 'A solution for the best rotation to relate two sets of vectors'. *Acta Cryst.* A**32**, 922–3, 1976.
3.4 W. Kabsch. 'A discussion of the solution for the best rotation to relate two sets of vectors'. *Acta Cryst.* A**34**, 827–8, 1978.
3.5 G. R. Kneller. 'Superposition of molecular structures using quaternions'. *Molec. Simul.* **7**, 113–19, 1991.

Stereochemistry

The Cahn–Ingold–Prelog notation is defined in most textbooks of organic chemistry but a good, comprehensive guide can be found on the IUPAC website whose address is www.iupac.org.

Chapter 4

Quantum mechanics

The quote from Dirac comes from

4.1 P. A. M. Dirac. 'Quantum mechanics of many-electron systems'. *Proc. Roy. Soc. (London) A* **123**, 714–33, 1929.

For those wanting to know more about quantum mechanics, a good comprehensive guide is

4.2 A. S. Davydov. *Quantum Mechanics*. Second Edition, Pergamon, 1976.

Potential energy surfaces

A nice overview of potential energy surfaces is given in the article

4.3 B. T. Sutcliffe. 'The idea of a potential energy surface'. *J. Molec. Struct. (Theochem.)* **341**, 217–35, 1995.

Molecular orbital methods

Ab initio MO methods are reviewed in the following works:

4.4 W. J. Hehre, L. Radom, P. v. R. Schleyer and J. A. Pople. *Ab Initio Molecular Orbital Theory*. John Wiley & Sons, 1986.
4.5 R. McWeeny. *Methods of Molecular Quantum Mechanics*. Second Edition, Academic, 1989.
4.6 A. Szabo and N. S. Ostlund. *Modern Quantum Chemistry: Introduction to Advanced Electronic Structure Theory*. First Revised Edition, McGraw-Hill, 1989.

Articles mentioned in the text that describe the introduction of Gaussian basis functions into quantum chemistry, the derivation of the Roothaan–Hall equations and Mulliken's population analysis are

4.7 S. F. Boys. 'Electronic wavefunctions. I. A general method of calculation for stationary states of any molecular system'. *Proc. Roy. Soc. (London) A* **200**, 542–54, 1950.
4.8 G. G. Hall. 'The molecular orbital theory of chemical valency VIII. A method of calculating ionization potentials'. *Proc. Roy. Soc. (London) A* **205**, 541–52, 1951.
4.9 R. S. Mulliken. 'Electronic population analysis on LCAO–MO molecular wave functions. I'. *J. Chem. Phys.* **23**, 1833–40, 1955.
4.10 C. C. J. Roothaan. 'New developments in molecular orbital theory'. *Rev. Mod. Phys.* **23**, 69–89, 1951.

A nice overview of work into linear-scaling quantum chemical methods is

4.11 S. Goedecker. 'Linear-scaling electronic structure methods'. *Rev. Mod. Phys.* **71**, 1085–123, 1999.

A good introduction to semi-empirical MO theory is

4.12 J. A. Pople and D. L. Beveridge. *Approximate Molecular Orbital Theory*. McGraw-Hill, 1970.

References for various MNDO semi-empirical methods are

4.13 M. J. S. Dewar and W. Thiel. 'Ground states of molecules. 38. The MNDO method: approximations and parameters'. *J. Am. Chem. Soc.* **99**, 4899–907, 1977.
4.14 M. J. S. Dewar, E. G. Zoebisch, E. F. Healy and J. J. P. Stewart. 'AM1: a new general purpose quantum mechanical molecular model'. *J. Am. Chem. Soc.* **107**, 3902–9, 1985.
4.15 J. J. P. Stewart. 'Optimization of parameters for semi-empirical methods I. Method'. *J. Comput. Chem.* **10**, 209–20, 1989.
4.16 J. J. P. Stewart. 'Optimization of parameters for semi-empirical methods II. Applications'. *J. Comput. Chem.* **10**, 221–64, 1989.
4.17 M. P. Repasky, J. Chandrasekhar and W. L. Jorgensen. 'PDDG/PM3 and PDDG/MNDO: improved semi-empirical methods'. *J. Comput. Chem.* **23**, 1601–22, 2002.

Density functional theory

DFT methods are not described extensively in the text but a few references will be given here for those who are interested. Two good books, one with an applied emphasis and the other theoretical, are

4.18 W. Koch and M. C. Holthausen. *A Chemist's Guide to Density Functional Theory*. Wiley-VCH, 2000.
4.19 R. G. Parr and W. Yang. *Density-Functional Theory of Atoms and Molecules*. Oxford University Press, 1989.

The classic, original DFT papers are

4.20 P. Hohenberg and W. Kohn. 'Inhomogeneous electron gas'. *Phys. Rev. B* **136**, 864–71, 1964.
4.21 W. Kohn and L. J. Sham. 'Self-consistent equations including exchange and correlation effects'. *Phys. Rev. A* **140**, 1133–8, 1965.

A review concerning DFT calculations with plane-wave basis sets is

4.22 D. Marx and J. Hutter. 'Ab initio molecular dynamics: theory and implementation'. In *Modern Methods and Algorithms of Quantum Chemistry*, NIC Series, Volume 1, J. Grotendorst (Ed.), 301–449, John von Neumann Institute for Computing, 2000.

Quantum chemical derivatives

A reference to Pulay's original work is

4.23 P. Pulay. 'Ab initio calculation of force constants and equilibrium geometries. I. Theory'. *Mol. Phys.* **17**, 197–204, 1969.

Chapter 5

Molecular mechanics

There is an extensive literature on molecular mechanics energy functions. A useful overview by one of its strongest proponents is

5.1 U. Burkert and N. L. Allinger. *Molecular Mechanics*. ACS Monograph 177, American Chemical Society, 1982.

A shorter reference by another strong proponent of the molecular mechanics method can be found in

5.2 S. Lifson. 'Theoretical foundations for the empirical force field method'. *Gazz. Chim. Ital.* **116**, 687–92, 1986.

The various categories of non-bonding energy terms and their origins are discussed in the following classic reference:

5.3 J. O. Hirschfelder, L. Curtiss and R. B. Bird. *Molecular Theory of Gases and Liquids*. John Wiley & Sons, 1954.

A shorter review is

5.4 A. J. Stone and S. L. Price. 'Some new ideas in the theory of intermolecular forces: anisotropic atom–atom potentials'. *J. Phys. Chem.* **92**, 325–35, 1988.

Atomic dipole polarizability models are described in

5.5 A. Warshel and S. T. Russell. 'Calculations of electrostatic interactions in biological systems and in solutions'. *Q. Rev. Biophys.* **17**, 283–422, 1984.

Other ways of including polarization effects are with *fluctuating charge*, or *charge equilibration*, and *Drude oscillator* models. Examples of implementations in molecular mechanics force fields may be found in

5.6 A. Rappé and W. A. Goddard III. 'Charge equilibration for molecular dynamics simulations'. *J. Phys. Chem.* **95**, 3358–63, 1991.
5.7 V. M. Anisimov, G. Lamoureux, I. V. Vorobyov *et al.* 'Determination of electrostatic parameters for a polarizable force field based on the classical Drude oscillator'. *J. Chem. Theor. Comput.* **1**, 153–68, 2005.

Force fields

The OPLS-AA force field used in the book is fully described in the following paper:

5.8 W. L. Jorgensen, D. S. Maxwell and J. Tirado-Rives. 'Development and testing of the OPLS all-atom force field on conformational energetics and properties of organic liquids'. *J. Am. Chem. Soc.* **118**, 11225–36, 1996.

The reference for the TIP3P model of water is

5.9 W. L. Jorgensen, J. Chandrasekhar, J. D. Madura, R. W. Impey and M. L. Klein. 'Comparison of simple potential functions for simulating liquid water'. *J. Chem. Phys.* **79**, 926–35, 1983.

There are many other force fields. Not all will be referenced here but some of the more common ones are listed below in alphabetical order. The OPLS-AA force field uses parts of the all-atom AMBER force field described in

5.10 W. D. Cornell, P. Cieplak, C. I. Bayly *et al.* in P. Kollman group. 'A second generation force field for the simulation of proteins, nucleic acids, and organic molecules'. *J. Am. Chem. Soc.* **117**, 5179–97, 1995.

The CHARMM force field is also widely used for biomolecular simulations. One of its more recent incarnations is

5.11 A. D. MacKerell, D. Bashford, M. Bellott *et al.* in M. Karplus group. 'All-atom empirical potential for molecular modeling and dynamics studies of proteins'. *J. Phys. Chem. B* **102**, 3586–616, 1998.

Allinger and co-workers have developed a number of force fields over the years that are used extensively, particularly for calculations on smaller molecular systems. The latest one, MM4, was introduced in a consecutive series of articles in an issue of the *Journal of Computational Chemistry*. The first in the series is

5.12 N. L. Allinger, K.-H. Chen and J.-H. Lii. 'An improved force field (MM4) for saturated hydrocarbons'. *J. Comput. Chem.* **17**, 642–68, 1996.

The MMFF94 force field was developed by Halgren and is described in a series of papers in the same issue of the *Journal of Computational Chemistry* as the one in which MM4 first appeared. The leading article is

5.13 T. A. Halgren. 'The Merck molecular force field. I. Basis, form, scope, parameterization and performance of MMFF94'. *J. Comput. Chem.* **17**, 490–519, 1996.

Some force fields have been developed with the aim of covering all elements. Probably the most widely used is UFF (the 'Universal Force Field'):

5.14 A. K. Rappé, C. J. Casewit, K. S. Colwell, W. A. Goddard III and W. M. Skiff. 'UFF, a full periodic table force field for molecular mechanics and molecular dynamics simulations'. *J. Am. Chem. Soc.* **114**, 10024–35, 1992.

A nice overview of force fields for simulations of proteins is

5.15 J. W. Ponder and D. A. Case. 'Force fields for protein simulation'. *Adv. Prot. Chem.* **66**, 27–85, 2003.

A useful reference for readers interested in the modeling of coordination compounds (which is a separate topic in its own right) is

5.16 P. Norrby and P. Brandt. 'Deriving force field parameters for coordination complexes'. *Coord. Chem. Rev.* **212**, 79–109, 2001.

Parametrization

Much information about the parametrization of Gaussian basis sets, semi-empirical QC methods and MM force fields can be found in the books by Hehre *et al.* [4.4], by Pople and Beveridge [4.12] and by Burkert and Allinger [5.1], respectively. Other essential sources are the original papers that describe the development of particular semi-empirical QC methods and MM force fields. For those interested in non-linear least squares methods, mathematical details are given in the book by Press *et al.* [1.1].

The methodology for the fitting of charges from electrostatic potential data that is employed by many force fields is described in

5.17 S. R. Cox and D. E. Williams. 'Representation of the molecular electrostatic potential by a net atomic charge model'. *J. Comput. Chem.* **2**, 304–23, 1981.
5.18 U. C. Singh and P. A. Kollman. 'An approach to computing electrostatic charges for molecules'. *J. Comput. Chem.* **5**, 129–45, 1984.
5.19 C. I. Bayly, P. Cieplak, W. D. Cornell and P. A. Kollman. 'Well behaved electrostatic potential based method using charge restraints for deriving atomic charges: the RESP model'. *J. Phys. Chem.* **97**, 10269–80, 1993.

Chapter 6

Hybrid potentials

The classic paper in which hybrid QC/MM potentials were introduced is

6.1 A. Warshel and M. Levitt. 'Theoretical studies of enzymic reactions: dielectric, electrostatic and steric stabilization of the carbonium ion in the reaction of lysozyme'. *J. Mol. Biol.* **103**, 227–49, 1976.

Two later papers that made important contributions and which helped pave the way to the widespread adoption of hybrid potentials are

6.2 U. C. Singh and P. A. Kollman. 'A combined ab initio quantum mechanical and molecular mechanical method for carrying out simulations on complex molecular systems: applications to the $CH_3Cl + Cl^-$ exchange reaction and gas phase protonation of polyethers'. *J. Comput. Chem.* **7**, 718–30, 1986.
6.3 M. J. Field, P. A. Bash and M. Karplus. 'A combined quantum mechanical and molecular mechanical potential for molecular dynamics simulations'. *J. Comput. Chem.* **11**, 700–33, 1990.

There are many reviews on hybrid potential methods but the following is notable:

6.4 J. Gao. 'Methods and applications of combined quantum mechanical and molecular mechanical potentials'. *Rev. in Comput. Chem.* **7**, 119–85, 1995.

The paper that introduced the ONIOM series of methods is

6.5 F. Maseras and K. Morokuma. 'IMOMM: a new integrated ab initio and molecular mechanics geometry optimization scheme of equilibrium structures and transition states'. *J. Comput. Chem.* **16**, 1170–9, 1995.

Original papers describing hybrid- and localized-orbital methodologies include

6.6 X. Assfeld and J.-L. Rivail. 'Quantum chemical computations on parts of large molecules: the ab initio local self consistent field method'. *Chem. Phys. Lett.* **263**, 100–6, 1996.
6.7 D. M. Philipp and R. A. Friesner. 'Mixed ab initio QM/MM modeling using frozen orbitals and tests with alanine dipeptide and tetrapeptide'. *J. Comput. Chem.* **14**, 1468–94, 1999.

6.8 J. Gao, P. Amara, C. Alhambra and M. J. Field. 'A generalized hybrid orbital (GHO) method for the treatment of boundary atoms in combined QM/MM calculations'. *J. Phys. Chem. A* **102**, 4714–21, 1998.

Further technical details of the hybrid potentials that are employed in pDynamo may be found in

6.9 M. J. Field, M. Albe, C. Bret, F. Proust-De Martin and A. Thomas. 'The Dynamo library for molecular simulations using hybrid quantum mechanical and molecular mechanical potentials'. *J. Comput. Chem.* **21**, 1088–100, 2000.

Since his 1976 paper, Warshel has pursued an *empirical valence bond* methodology for studying enzyme reactions that is distinct from the hybrid potential models that we describe. An exposition of his approach may be found in

6.10 A. Warshel. *Computer Modeling of Chemical Reactions in Enzymes and Solutions*. John Wiley & Sons, 1991.

Chapter 7

Exploring potential energy surfaces

A comprehensive overview of the properties of potential energy surfaces for different classes of systems is given in

7.1 D. J. Wales. *Energy Landscapes: Applications to Clusters, Biomolecules and Glasses*. Cambridge University Press, 2004.

The analytic form of the Müller–Brown model potential used in Figure 7.1 is given in

7.2 K. Müller and L. D. Brown. 'Location of saddle points and minimum energy paths by a constrained simplex optimization procedure'. *Theor. Chim. Acta* **53**, 75–93, 1979.

Table 7.1 was adapted from

7.3 J. Ma, D. Hsu and J. E. Straub. 'Approximate solution to the classical Liouville equation using Gaussian phase packet dynamics: application to enhanced equilibrium averaging and global optimization'. *J. Chem. Phys.* **99**, 4024–35, 1993.

Locating minima

Details of optimization algorithms can be found in *Numerical Recipes* [1.1]. A nice general reference devoted to the mathematical problem of optimization is

7.4 R. Fletcher. *Practical Methods of Optimization*. John Wiley & Sons, 1987.

Two reviews of geometry optimization methods for molecular systems are

7.5 H. B. Schlegel. 'Optimization of equilibrium geometries and transition structures'. *Adv. Chem. Phys.* **67**, 249–86, 1987.
7.6 J. D. Head and M. C. Zerner. 'Newton based optimization methods for obtaining molecular conformation'. *Adv. Quantum Chem.* **20**, 239–90, 1989.

In this book, all geometry optimizations are performed in terms of Cartesian coordinates. For readers interested in geometry optimization methods that employ internal coordinates as variables, the following references will be of value:

7.7 P. Pulay and G. Fogarasi. 'Geometry optimization in redundant internal coordinates'. *J. Chem. Phys.* **96**, 2856–60, 1992.
7.8 C. Peng, P. Y. Ayala, H. B. Schlegel and M. J. Frisch. 'Using redundant internal coordinates to optimize equilibrium geometries and transition states'. *J. Comput. Chem.* **17**, 49–56, 1996.
7.9 J. Baker, A. Kessi and B. Delley. 'The generation and use of delocalized internal coordinates in geometry optimization'. *J. Chem. Phys.* **105**, 192–212, 1996.

Locating saddle points

A nice review of transition state location algorithms is

7.10 S. Bell and J. S. Crighton. 'Locating transition states'. *J. Chem. Phys.* **80**, 2464–75, 1984.

Papers covering the location of transition states using surface walking methods are

7.11 C. J. Cerjan and W. H. Miller. 'On finding transition states'. *J. Chem. Phys.* **81**, 2800–6, 1981.
7.12 J. Simons, P. Jorgensen, H. Taylor and J. Ozment. 'Walking on potential energy surfaces'. *J. Phys. Chem.* **87**, 2745–53, 1983.
7.13 J. Baker. 'An algorithm for the location of transition states'. *J. Comput. Chem.* **4**, 385–95, 1986.

Following reaction paths

The formulation by Fukui of the intrinsic reaction coordinate approach is given in

7.14 K. Fukui. 'A formulation of the reaction coordinate'. *J. Phys. Chem.* **74**, 4161–3, 1970.
7.15 K. Fukui. 'The path of chemical reactions: the IRC approach'. *Acc. Chem. Res.* **14**, 363–8, 1981.

More information on reaction path following may be found in

7.16 K. Ishida, K. Morokuma and A. Komornicki. 'The intrinsic reaction coordinate: an ab initio calculation for HNC $\rightarrow$ HCN and $H^- + CH_4 \rightarrow CH_4 + H^-$'. *J. Chem. Phys.* **66**, 2153–6, 1977.
7.17 M. W. Schmidt, M. S. Gordon and M. Dupuis. 'The intrinsic reaction coordinate and the rotational barrier in silaethylene'. *J. Am. Chem. Soc.* **107**, 2585–9, 1985.
7.18 C. Gonzalez and H. B. Schlegel. 'An improved algorithm for reaction path following'. *J. Chem. Phys.* **90**, 2154–61, 1988.

Determining complete reaction paths

The reaction path methods of Elber and co-workers are described in the following papers:

7.19 R. Elber and M. Karplus. 'A method for determining reaction paths in large molecules: application to myoglobin'. *Chem. Phys. Lett.* **139**, 375–80, 1987.

7.20 R. Czerminski and R. Elber. 'Self-avoiding walk between two fixed points as a tool to calculate reaction paths in large molecular systems'. *Int. J. Quantum Chem.: Quantum Chem. Symp.* **24**, 167–86, 1990.

Some recent algorithms are more efficient than the self-avoiding walk method discussed in the text. A popular one is the *nudged-elastic-band algorithm* which is described in

7.21 G. Henkelman, G. Jóhannesson and H. Jónsson. 'Methods for finding saddle points and minimum energy paths'. In *Progress on Theoretical Chemistry and Physics*, Volume 5, S. D. Schwartz (Ed.), 269–300, Kluwer Academic Publishers, 2000.

Chapter 8

Calculation of the normal modes

The classic text on the normal mode analysis of molecules is the following book:

8.1 E. B. Wilson, Jr, J. C. Decius and P. C. Cross. *Molecular Vibrations: The Theory of Infrared and Raman Vibrational Spectra*. Dover Publications Inc., 1955.

Rotational and translational modes

Useful discussions of the separation of rotational, translational and vibrational motion may be found in the book on classical mechanics by Goldstein [3.2] and in the following paper on reaction path dynamics:

8.2 W. H. Miller, N. C. Handy and J. E. Adams. 'Reaction path Hamiltonian for polyatomic molecules'. *J. Chem. Phys.* **72**, 99–112, 1980.

Calculation of thermodynamical functions

There is a plethora of good books on thermodynamics and statistical thermodynamics. A comprehensive reference is

8.3 D. A. McQuarrie. *Statistical Mechanics*. Harper Collins, 1976.

Two nice introductory texts with very different perspectives are

8.4 E. B. Smith. *Basic Chemical Thermodynamics*. Oxford Chemistry Series 31, Clarendon Press, 1982.
8.5 D. Chandler. *Introduction to Modern Statistical Mechanics*. Oxford University Press, 1987.

Chapter 9

Molecular dynamics

Good introductions to molecular dynamics calculations and methods for the integration of ordinary differential equations can be found in the general books listed at the beginning of the references. Verlet introduced the algorithm that bears his name in

9.1 L. Verlet. 'Computer experiments on classical fluids. I. Thermodynamical properties of Lennard-Jones molecules'. *Phys. Rev.* **159**, 98–103, 1967.

The velocity Verlet algorithm is described in

9.2 W. C. Swope, H. C. Andersen, P. H. Berens and K. R. Wilson. 'A computer simulation method for the calculation of equilibrium constants for the formation of physical clusters of molecules: application to small water clusters'. *J. Chem. Phys.* **76**, 637–49, 1982.

Simulated annealing

The classic reference on the method of simulated annealing is

9.3 S. Kirkpatrick, C. D. Gelatt, Jr and M. P. Vecchi. 'Optimization by simulated annealing'. *Science* **220**, 671–80, 1983.

A nice review of simulated annealing and related methods and their application to the global optimization of complex molecular systems is

9.4 I. Andricioaei and J. E. Straub. 'Finding the needle in the haystack: algorithms for conformational optimization'. *Computers in Physics* **10**, 449–54, 1996.

Another powerful class of techniques for global optimization is the *genetic algorithms*. An accessible review of the principles behind these may be found in

9.5 S. Forrest. 'Genetic algorithms: principles of natural selection applied to computation'. *Science* **261**, 872–8, 1993.

Two examples of applications of genetic algorithms to geometry optimization are

9.6 B. Hartke. 'Global geometry optimization of clusters using genetic algorithms'. *J. Phys. Chem.* **97**, 9973–6, 1993.
9.7 J. Mestres and G. E. Scuseria. 'Genetic algorithms: a robust scheme for geometry optimizations and global minimum structure problems'. *J. Comput. Chem.* **16**, 729–42, 1995.

In the 1980s R. Car and M. Parrinello introduced a novel simulation technique in which a dynamics algorithm was used simultaneously as a tool for simulated-annealing minimization (in this case for finding the optimum orbitals in a plane-wave DFT calculation) and for propagating the dynamics of a system's atoms. *Car–Parrinello* methods are especially widely employed in conjunction with plane-wave DFT calculations but they are generally applicable whenever one has a potential energy function whose evaluation can be formulated as an optimization. The original paper is

9.8 R. Car and M. Parrinello. 'Unified approach for molecular dynamics and density-functional theory'. *Phys. Rev. Lett.* **55**, 2471–4, 1985.

Chapter 10

Cutoff methods for the calculation of non-bonding interactions

Useful accounts of the calculation of non-bonding interactions can be found in the book by Allen and Tildesley [1.2] as well as in *Computer Simulation of Biomolecular Systems*, Volume 2 [1.11]. A specific reference for the atom-based force-switching method is

10.1 P. J. Steinbach and B. R. Brooks. 'New spherical-cutoff methods for long-range forces in macromolecular simulation'. *J. Comput. Chem.* **15**, 667–83, 1994.

Including an environment

The article by Warshel and Russell [5.5] contains a description of a wide variety of models for the representation of the environment. Three brief reviews on the calculation of electrostatic interactions in macromolecules, which also discuss methods based upon the Poisson–Boltzmann equation, can be found in

10.2 K. A. Sharp. 'Electrostatic interactions in macromolecules'. *Curr. Opin. Struct. Biol.* **4**, 234–9, 1994.
10.3 M. K. Gilson. 'Theory of electrostatic interactions in macromolecules'. *Curr. Opin. Struct. Biol.* **5**, 216–23, 1995.
10.4 B. Honig and A. Nicholls. 'Classical electrostatics in biology and chemistry'. *Science* **268**, 1144–9, 1995.

A solvent-accessible-surface-area model was introduced in

10.5 D. Eisenberg and A. D. McLachlan. 'Solvation energy in protein folding and binding'. *Nature* **319**, 199–203, 1986.

The generalized Born/surface-area implicit solvation model of Still and co-workers is described in

10.6 W. C. Still, A. Tempczyk, R. C. Hawley and T. Hendrickson. 'Semianalytical treatment of solvation for molecular mechanics and dynamics'. *J. Am. Chem. Soc.* **112**, 6127–9, 1990.
10.7 D. Qiu, P. S. Shenklin, F. P. Hollinger and W. C. Still. 'The GB/SA continuum model for solvation: a fast analytical method for the calculation of approximate Born radii'. *J. Phys. Chem. A* **101**, 3005–14, 1997.

Ewald summation techniques

A nice general account of the Ewald technique can be found in the book by Allen and Tildesley [1.2]. They also give a physical interpretation of the use of convergence functions in the summation of the electrostatic interactions. The account due to Williams may be found in

10.8 D. E. Williams. 'Accelerated convergence of crystal-lattice potential sums'. *Acta Cryst.* A**27**, 452–5, 1971.

For some original references and details about the derivation consult

10.9 P. Ewald. 'Die Berechnung optischer und elektrostatischer Gitterpotentiale'. *Ann. Phys.* **64**, 253–87, 1921.
10.10 B. R. A. Nijboer and F. W. De Wette. 'On the calculation of lattice sums'. *Physica* **23**, 309–21, 1957.
10.11 S. W. de Leeuw, J. W. Perram and E. R. Smith. 'Simulation of electrostatic systems in periodic boundary conditions. I. Lattice sums and dielectric constants'. *Proc. Roy. Soc. (London) A* **373**, 27–56, 1980.

Fast methods for the evaluation of non-bonding interactions

The fast multipole method of Greengard and Rokhlin is described in

10.12 L. F. Greengard and V. Rokhlin. 'A fast algorithm for particle simulations'. *J. Comput. Phys.* **73**, 325–48, 1987.

Applications of fast multipole methods to molecular systems are described in

10.13 H.-Q. Ding, N. Karasawa and W. A. Goddard III. 'Atomic level simulations on a million particles: the cell multipole method for Coulomb and London nonbond interactions'. *J. Chem. Phys.* **97**, 4309–15, 1992.
10.14 J. A. Board, Jr, J. W. Causey, J. F. Leathrum, Jr, A. Windemuth and K. Schulten. 'Accelerated molecular dynamics simulation with the parallel fast multipole algorithm'. *Chem. Phys. Lett.* **198**, 89–94, 1992.

The particle mesh Ewald methods are discussed in the following articles. The last also provides an analysis of the scaling properties of the original versus the faster Ewald algorithms.

10.15 T. Darden, D. York and L. Pedersen. 'Particle mesh Ewald: an $N \ln(N)$ method for Ewald sums in large systems'. *J. Chem. Phys.* **98**, 10089–92, 1993.
10.16 U. Essmann, L. Perara, M. L. Berkowitz, T. Darden, H. Lee and L. G. Pedersen. 'A smooth particle mesh Ewald method'. *J. Chem. Phys.* **103**, 8577–93, 1995.
10.17 H. G. Petersen. 'Accuracy and efficiency of the particle mesh Ewald method'. *J. Chem. Phys.* **103**, 3668–79, 1995.

A nice synthesis of fast Ewald methods is given in

10.18 Y. Shan, J. L. Klepeis, M. P. Eastwood, R. O. Dror and D. E. Shaw. 'Gaussian split Ewald: a fast Ewald mesh method for molecular simulation'. *J. Chem. Phys.* **122**, 054101, 2005.

Chapter 11

Analysis of molecular dynamics trajectories

Readers are referred to one of the many standard texts on statistical mechanics, such as the book by McQuarrie [8.3], for more details of the types of quantities that can be calculated from simulation data. Another good source of information is the book by Allen and Tildesley [1.2] which has a background chapter on statistical mechanics as well as sections devoted to the calculation of various properties. This book also discusses some statistical methods for assessing the errors in the averages, fluctuations and time correlation functions calculated from a simulation.

A classic reference that discusses the statistical error arising when calculating averages from trajectories is

11.1 R. Zwanzig and N. K. Ailawadi. 'Statistical error due to finite time averaging in computer experiments'. *Phys. Rev.* **182**, 193–6, 1969.

Two other nice discussions of statistical errors can be found in

11.2 R. W. Pastor. 'Techniques and applications of Langevin dynamics simulations'. In *The Molecular Dynamics of Liquid Crystals*, G. R. Luckhurst and C. A. Veracini (Eds.), 85–138, Kluwer Academic Publishers, 1994.

11.3 W. Janke. 'Statistical analysis of simulations: data correlations and error estimation'. In *Quantum Simulations of Complex Many-Body Systems: From Theory to Algorithms*, NIC Series, Volume 10, J. Grotendorst, D. Marx and A. Muramatsu (Eds.), 423–45, John von Neumann Institute for Computing, 2002.

Temperature and pressure control in molecular dynamics simulations

A concise description of constant-pressure and -temperature algorithms is given by Allen and Tildesley [1.2]. The algorithm derived by Berendsen *et al.* that is used in this book may be found in

11.4 H. J. C. Berendsen, J. P. M. Postma, W. F. van Gunsteren, A. Di Nola and J. R. Haak. 'Molecular dynamics with coupling to an external bath'. *J. Chem. Phys.* **81**, 3684–90, 1984.

An early seminal paper on constant-pressure and -temperature algorithms was published by Andersen:

11.5 H. C. Andersen. 'Molecular dynamics simulations at constant pressure and/or temperature'. *J. Chem. Phys.* **72**, 2384–93, 1980.

The extension of this method to allowing the simulation box to change shape as well as size is described in

11.6 M. Parrinello and A. Rahman. 'Crystal structure and pair potentials: a molecular dynamics study'. *Phys. Rev. Lett.* **45**, 1196–9, 1980.

The constant-pressure work of Nosé and Klein and Nosé–Hoover thermostating can be found in

11.7 S. Nosé and M. L. Klein. 'Constant pressure molecular dynamics for molecular systems'. *Molec. Phys.* **50**, 1055–76, 1983.
11.8 S. Nosé. 'A unified formulation of the constant temperature molecular dynamics methods'. *J. Chem. Phys.* **81**, 511–19, 1984.
11.9 W. G. Hoover. 'Canonical dynamics: equilibrium phase-space distributions'. *Phys. Rev. A* **31**, 1695–7, 1985.

Two more recent papers by Klein and co-workers on the extended system methods are

11.10 G. J. Martyna, M. L. Klein and M. E. Tuckerman. 'Nosé–Hoover chains: the canonical ensemble via continuous dynamics'. *J. Chem. Phys.* **97**, 2635–43, 1992.
11.11 G. J. Martyna, D. J. Tobias and M. L. Klein. 'Constant pressure molecular dynamics algorithms'. *J. Chem. Phys.* **101**, 4177–89, 1994.

A novel modification of the Andersen constant-pressure technique may be found in

11.12 S. E. Feller, Y. Zhang, R. W. Pastor and B. R. Brooks. 'Constant pressure molecular dynamics simulation: the Langevin piston method'. *J. Chem. Phys.* **103**, 4613–21, 1995.

The constraint techniques are described in

11.13 D. J. Evans and G. P. Morriss. 'Non-Newtonian molecular dynamics'. *Computer Phys. Rep.* **1**, 297–343, 1984.

Calculating free energies: umbrella sampling

The method of umbrella sampling is detailed in

11.14 G. M. Torrie and J. P. Valleau. 'Nonphysical sampling distributions in Monte Carlo free energy estimation: umbrella sampling'. *J. Comput. Phys.* **23**, 187–99, 1977.
11.15 J. P. Valleau and G. M. Torrie. 'A guide to Monte Carlo for statistical mechanics. 2. Byways'. In *Statistical Mechanics A. Modern Theoretical Chemistry*, Volume 5, B. J. Berne (Ed.), 169–94, Plenum Press, 1977.

Two references detailing the WHAM method of analysing umbrella sampling data are

11.16 A. M. Ferrenberg and R. H. Swendsen. 'Optimized Monte Carlo data analysis'. *Phys. Rev. Lett.* **63**, 1195–8, 1989.
11.17 S. Kumar, D. Bouzida, R. H. Swendsen, P. A. Kollman and J. M. Rosenberg. 'The weighted histogram analysis method for free energy calculations on biomolecules. I. The method'. *J. Comput. Chem.* **13**, 1011–21, 1992.

A nice description of the WHAM methodology and the source for the example used in this book is

11.18 B. Roux. 'The calculation of the potential of mean force using computer simulations'. *Comput. Phys. Commun.* **91**, 275–82, 1995.

Speeding up simulations

References that describe the SHAKE algorithms are

11.19 J. P. Ryckaert, G. Ciccotti and H. J. C. Berendsen. 'Numerical integration of the Cartesian equations of motion of a system with constraints: molecular dynamics of n-alkanes'. *J. Comput. Phys.* **23**, 327–41, 1977.
11.20 J. P. Ryckaert. 'Special geometrical constraints in the molecular dynamics of chain molecules'. *Molec. Phys.* **55**, 549–56, 1985.

Two algorithms similar to SHAKE are *RATTLE* and *LINCS* described in

11.21 H. C. Andersen. 'RATTLE: a velocity version of the SHAKE algorithm for molecular dynamics calculations'. *J. Comput. Phys.* **52**, 24–34, 1983.
11.22 B. Hess, H. Bekker, H. J. C. Berendsen and J. G. E. M. Fraaije. 'LINCS: a linear constraint solver for molecular simulations'. *J. Comput. Chem.* **18**, 1463–72, 1997.

The effects that constraining various degrees of freedom can have on the dynamics of a system are discussed in

11.23 W. F. van Gunsteren and M. Karplus. 'Effects of constraints on the dynamics of macromolecules'. *Macromolecules* **15**, 1528–44, 1982.
11.24 K. Hinsen and G. R. Kneller. 'Influence of constraints on the dynamics of polypeptide chains'. *Phys. Rev. E* **52**, 6868–74, 1995.

Two papers describing multiple timestep integration algorithms are

11.25 M. Tuckerman, B. J. Berne and G. J. Martyna. 'Reversible multiple time scale molecular dynamics'. *J. Chem. Phys.* **97**, 1990–2001, 1992.
11.26 D. D. Humphreys, R. A. Friesner and B. J. Berne. 'A multiple time step molecular dynamics algorithm for macromolecules'. *J. Phys. Chem.* **98**, 6885–92, 1994.

Chapter 12

The Metropolis Monte Carlo method

The book by Allen and Tildesley [1.2] gives a nice discussion of all aspects of the Monte Carlo technique. The original paper introducing the Metropolis algorithm is

12.1 N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller and E. Teller. 'Equation of state calculations by fast computing machines'. *J. Chem. Phys.* **21**, 1087–92, 1953.

For more details on the mathematics behind the Metropolis method, its relation to the theory of Markov chains and other Monte Carlo techniques a good review is

12.2 J. P. Valleau and S. G. Whittington. 'A guide to Monte Carlo for statistical mechanics. 1. Highways'. In *Statistical Mechanics A. Modern Theoretical Chemistry*, Volume 5, B. J. Berne (Ed.), 137–68, Plenum Press, 1977.

The extension of the Monte Carlo method to the NPT ensemble is discussed in

12.3 W. W. Wood. 'Monte Carlo calculations for hard disks in the isothermal–isobaric ensemble'. *J. Chem. Phys.* **48**, 415–34, 1968.
12.4 W. W. Wood. 'NPT ensemble Monte Carlo calculations for the hard disk fluid'. *J. Chem. Phys.* **52**, 729–41, 1970.

Monte Carlo simulations of molecules

Jorgensen and co-workers have done simulation work using the Metropolis Monte Carlo method on a wide variety of systems. References to much of their work can be found in the papers on the development of the OPLS force field [5.8, 5.9] and in the papers listed here. A review giving some idea of their approach is

12.5 W. L. Jorgensen. 'Theoretical studies of medium effects on conformational equilibria'. *J. Phys. Chem.* **87**, 5304–14, 1983.

In an interesting paper Jorgensen and Tirado-Rives discussed the relative efficiencies of the Monte Carlo and molecular dynamics methods for sampling conformational space. They estimated that Monte Carlo methods are approximately twice as efficient for the systems they studied.

12.6 W. L. Jorgensen and J. Tirado-Rives. 'Monte Carlo versus molecular dynamics for conformational sampling'. *J. Phys. Chem.* **100**, 14508–13, 1996.

The method of preferential sampling is described in

12.7 J. C. Owicki and H. A. Scheraga. 'Preferential sampling near solutes in Monte Carlo calculations on dilute solutions'. *Chem. Phys. Lett.* **47**, 600–2, 1977.

More details of generalized coordinates, including how to derive Equation (12.15) by integration over the generalized conjugate momenta, can be found in the books by Goldstein [3.2] and by Wilson, Decius and Cross [8.1].

Example 21

Some early Monte Carlo studies on water and methane were performed by Owicki and Scheraga. These are detailed in

12.8 J. C. Owicki and H. A. Scheraga. 'Monte Carlo calculations in the isothermal–isobaric ensemble. 1. Liquid water'. *J. Am. Chem. Soc.* **99**, 7403–12, 1977.
12.9 J. C. Owicki and H. A. Scheraga. 'Monte Carlo calculations in the isothermal–isobaric ensemble. 2. Dilute aqueous solution of methane'. *J. Am. Chem. Soc.* **99**, 7413–18, 1977.

A paper by the Jorgensen group on the same system is

12.10 W. L. Jorgensen, J. Gao and C. Ravimohan. 'Monte Carlo simulations of alkanes in water: hydration numbers and the hydrophobic effect'. *J. Phys. Chem.* **89**, 3470–3, 1985.

Calculating free energies: statistical perturbation theory

The concepts of thermodynamic integration and perturbation and the introduction of coupling parameters into a Hamiltonian have a long history. Two seminal references are

12.11 R. W. Zwanzig. 'High-temperature equation of state by a perturbation method. I. Nonpolar gases'. *J. Chem. Phys.* **22**, 1420–6, 1954.
12.12 J. G. Kirkwood. 'Statistical mechanics of fluid mixtures'. *J. Chem. Phys.* **3**, 300–13, 1935.

Some articles reviewing the calculation of free energies using molecular dynamics and Monte Carlo simulations are

12.13 W. L. Jorgensen. 'Free energy calculations: a breakthrough for modeling organic chemistry in solution'. *Acc. Chem. Res.* **22**, 184–9, 1989.
12.14 D. L. Beveridge and F. M. Di Capua. 'Free energy via molecular simulation: applications to biomolecular systems'. *Annu. Rev. Biophys. Biophys. Chem.* **18**, 431–92, 1989.
12.15 T. P. Straatsma and J. A. Mc Cammon. 'Computational alchemy'. *Annu. Rev. Phys. Chem.* **43**, 407–35, 1992.
12.16 P. A. Kollman. 'Free energy calculations: applications to chemical and biochemical phenomena'. *Chem. Rev.* **93**, 2395–417, 1993.

An application of Monte Carlo free-energy perturbation calculations to a complex problem is

12.17 J. W. Essex, D. L. Severance, J. Tirado-Rives and W. L. Jorgensen. 'Monte Carlo simulations for proteins: binding affinities for trypsin–benzamidine complexes via free energy perturbations'. *J. Phys. Chem. B* **101**, 9663–9, 1997.

Example 22

A paper detailing free-energy simulations of water, methane and the chloride anion is

12.18 W. L. Jorgensen, J. F. Blake and J. K. Buckner. 'Free energy of TIP4P water and the free energies of hydration of CH_4 and Cl^- from statistical perturbation theory'. *Chem. Phys.* **129**, 193–200, 1989.

Exercise 12.2

The simple overlap sampling method and related techniques for improving the precision of free-energy estimates are described in

12.19 N. Lu, J. K. Singh and D. A. Kofke. 'Appropriate methods to combine forward and reverse free-energy perturbation averages'. *J. Chem. Phys.* **118**, 2977–84, 2003.

An earlier seminal article upon which this later work is based is

12.20 C. H. Bennett. 'Efficient estimation of free energy differences from Monte Carlo data'. *J. Comput. Phys.* **22**, 245–68, 1976.

Author index

Subject index