To print photographs or files of computational results (figures and/or data) obtained using Molecular Simulations software, acknowledge the source in the format:

Computational results obtained using software programs from Molecular Simulations Inc.—dynamics calculations were done with the Discover® program, using the CFF91 forcefield, ab initio calculations were done with the DMol program, and graphical displays were printed out from the Cerius² molecular modeling system.

To reference a Molecular Simulations publication in another publication, no author should be specified and Molecular Simulations Inc. should be considered the publisher. For example:

## Table of Contents

### 1. Introduction 1

- What is the Discover® program? ........................................ 1
  - What Discover can do .................................................. 1
- Comparison of C and FORTRAN versions ............................ 3
  - CDiscove—Cerius², Insight and standalone modes ................ 3
- Accessing CDiscove ...................................................... 5
- How to use the documentation ......................................... 6
  - Additional information sources ...................................... 7
  - Typographical conventions and terminology ..................... 8

### 2. CDiscove within Cerius² 11

- Preparing the model and forcefield ................................ 13
  - Choosing the forcefield ............................................. 14
  - Building or loading models ....................................... 15
  - Assigning atom types and charges ............................... 15
  - Defining charge groups ............................................ 17
  - Setting constraints .................................................. 18
    - Fixed atoms ......................................................... 19
    - Constraints during dynamics .................................... 20
  - Preparing the energy expression ................................ 20
    - Nonbond interactions ............................................ 21
    - Parameter scaling ................................................ 23
    - What to do if some parameters cannot be assigned .......... 23
- Setting up the simulation conditions ................................ 24
  - Single-point energy and gradient calculations ................ 25
  - Minimization .......................................................... 25
    - Specifying the minimization algorithm ....................... 26
    - Setting convergence criteria .................................. 26
    - Maximum number of iterations .................................. 27
  - Dynamics .............................................................. 27
    - Defining the timestep ............................................ 28
    - Defining the thermodynamic ensembles ....................... 29
    - Controlling temperature, pressure, and stress ............. 30
    - Type of dynamics simulation .................................. 33
    - Stages of a dynamics run ....................................... 33
- Setting up and starting the run .................................... 34
  - Specifying output ................................................... 34
  - Specifying how and where to send the job ..................... 35
3. CDiscover within Insight

General procedure .......................... 67
Setting up the ESFF forcefield .................. 70
CDiscover background jobs .................... 71
Using the Background Job pulldown .......... 72
  Setting up a background job ................ 73
  Interactive screen coordinate updates ...... 74
  Examining completion status ............... 75
  Killing a background job .................. 75
  In case of run errors ...................... 76
Tutorial—The Insight environment .......... 76
  Pilot tutorials ............................ 76
  Overview of tutorial lessons .............. 77
  Lesson 5: Generating a phi/psi map ....... 79
Command summary—The Insight environment .. 95
  Setup pulldown ............................ 96
  Strategy pulldown ........................ 96
  Specify pulldown .......................... 97
  Calculate pulldown ........................ 97
  Language_Control pulldown ............... 98
Analyze pulldown ........................................ 98
Pseudo_Atom pulldown ................................. 98
Background_Job pulldown ............................. 99
D_Run pulldown ........................................ 99

4. Standalone CDiscover ................................. 101

Input files required .................................. 101
Overview of Btcl commands ......................... 101
Calculation commands ............................... 101
Commands for calculation setup and analysis .... 102
Input commands ...................................... 102
Output commands .................................... 102
Control commands ................................... 102
Exiting a Btcl session ................................. 103
Defining and using your own procedures .......... 103
Performance and convenience ....................... 103
Databases .............................................. 104
Objects .................................................. 104
Initiating a run ...................................... 105
Direct method ......................................... 105
Using the interactive utility ....................... 106
File redirection method .............................. 106
Running CDiscover in parallel ...................... 106
Scripts and program initialization ................. 107
An example CDiscover run .......................... 107
Input files ............................................ 107
Explanation of command input file ................ 109
Initiating the run .................................... 110
History file support .................................. 110
Editing the ESFF forcefield parameters ........... 111
Tutorial—standalone mode ......................... 111
Overview of tutorial lesson ......................... 111
Lesson 1: Using Btcl and Tcl commands to manipulate the geometry of two helices .......... 113
Command summary—standalone mode .............. 118

APPENDIC ES

A. Btcl Language and Commands—Standalone Mode 123

MSI (Biosym) Tcl language syntax .................... 123
General programming syntax ........................................ 123
Variables and expressions ......................................... 124
Special characters and the expr and eval commands .... 126
Control statements .................................................. 129
Conditional statements: if, else, and elseif .......... 129
Conditional looping statements while, for, and foreach ... 130
Btcl procedures ..................................................... 132
Btcl commands and utilities ....................................... 134
Btcl command parameter types ................................. 137
Global Btcl and Tcl variables and environment variables 138
Exiting a Btcl session ............................................... 139
Conventions in documentation of Btcl commands .... 139
Font conventions for keywords ................................. 139
Assignment and types of values ............................... 140
Punctuation conventions ........................................... 141

A. Btcl Language and Commands—Standalone Mode
   - Continued 143
   Command descriptions ........................................ 143
      analyzeNonbond ............................................. 143
      atomMovability ........................................... 149
      begin ....................................................... 152
      cellParameter .............................................. 153
      database .................................................. 154
      $dbHandle .................................................. 158
      diffraction ............................................... 165
      discoverHistory .......................................... 167
      dynamics ................................................. 168
      energy ...................................................... 177
      energyContribution ....................................... 179
      forcefield .................................................. 182
      geometry ................................................... 190
      help ......................................................... 203
      minimize .................................................... 204
      molGeom ..................................................... 210

A. Btcl Language and Commands—Standalone Mode
   - Continued 217
   Command descriptions - Continued ........................ 217
      object ....................................................... 217
      output ....................................................... 241
      peek ......................................................... 243
      print ......................................................... 246
      pseudoAtom ................................................. 250
rattle ......................................................... 256
readFile ..................................................... 259
reset ......................................................... 262
restraint ..................................................... 263
select ......................................................... 272
subset ......................................................... 278
subStructure ................................................ 284
vdWTailCorrection .......................................... 286
vector ......................................................... 289
vibrationalAnalysis ....................................... 300
writeFile ..................................................... 302

B. Databases and Tables ..................................... 305
   Introduction to databases and tables ..................... 305
      Introduction ............................................. 305
      Illustrations ........................................... 306
      Data types for columns ................................ 308
   System database ........................................... 309
      Introduction ............................................. 309
      Contents of the system database common to all systems
         310
         Atom table ............................................ 310
         Monomer table ....................................... 312
         Molecule table ....................................... 312
         NonbondGroup table .................................. 313
         Bond table ............................................ 313
      Contents of the system database for periodic systems
         314
         Introduction to symmetry and periodicity ............ 314
         Differences from simple systems .................... 315
         Descriptions of new and augmented tables .......... 316
   Contents of the system database for systems with
      symmetry and periodicity ................................ 320
      Differences from simpler systems .................... 320
      Descriptions of new and augmented tables .......... 320
   Subset tables ............................................. 323
   PseudoAtom table ......................................... 324
   Internal energy exclusion ................................ 325
   Nonbond energy exclusion ................................ 328
   Consensus dynamics ...................................... 329
   Energy database .......................................... 329
      Introduction ............................................. 329
      Contents of the energy database ....................... 330
         Values table ......................................... 330
         Atom table ............................................ 331
      Accessing the energy database ......................... 331
   Minimize database ....................................... 335
   Dynamics database ....................................... 338
C. Discover Interprocess Communication (IPC) 347

Purpose ........................................ 347
Background ..................................... 347
IPC components ................................. 348
  Discover Btcl IPC commands ................. 348
  MSI IPC client application program interface 349
  ipc procedure ................................ 349
Common IPC operations ....................... 351
  init ........................................ 351
  info ........................................ 352
  shutdown .................................... 353
Advanced IPC operations ...................... 353
  accept ..................................... 353
  connect .................................... 354
  eval ....................................... 354
  listen ..................................... 355
  poll ........................................ 355
  send ....................................... 355
  server ..................................... 356
Interprocess communication for Discover—discoverIO 356
  discoverIpc ................................ 361
    eval ...................................... 361
    poll ...................................... 361
    connect .................................. 362
    listen .................................... 362
    accept ................................... 362
  MSI IPC C application program interface .... 363
  Background ................................ 363
  C functions for IPC ......................... 364
    BiosymIpc_Init ............................ 364
    BiosymIpc_Shutdown ....................... 364
    BiosymIpc_Read ........................... 364
    BiosymIpc_Write ........................... 364
    BiosymIpc_Print ........................... 365
    BiosymIpc_VarWrite ....................... 365
    BiosymIpc_WriteRawBtcl2 ................. 365
    BiosymIpc_WriteRawBtcl ................... 365
    BiosymIpc_SetChannel ..................... 366
    BiosymIpc_GetError ....................... 366
    BiosymIpc_GetLocalVersion ............... 366
    BiosymIpc_GetRemoteVersion .............. 366
  MSI IPC FORTRAN application program interface .... 367
  Background ................................ 367
  Subroutines ................................. 368
BiosymIpc_Initf ........................................ 368
BiosymIpc_ReadHeaderf ................................. 368
BiosymIpc_ReadString128f ............................ 369
BiosymIpc_WriteIntegerf ............................... 370
BiosymIpc_WriteReal8f ................................. 370
BiosymIpc_WriteStringf ................................. 371
BiosymIpc_WriteString128f ........................... 372
BiosymIpc_WriteBtcl128f ............................... 373
BiosymIpc_WriteRawBtcl128f ......................... 374
BiosymIpc_GetErrorf .................................. 375
BiosymIpc_GetLocalVersionf ......................... 375
BiosymIpc_GetRemoteVersionf ....................... 376
BiosymIpc_Int2Charf .................................. 376
BiosymIpc_Char2Intf .................................. 376
BiosymIpc_ReadIntegerf ............................... 377
BiosymIpc_ReadReal8f ................................ 377
BiosymIpc_ReadStringf ................................. 377
BiosymIpc_ReadBtcl128f ............................... 378
BiosymIpc_ReadRawBtcl128f .......................... 378

MSI IPC C language structures ...................... 378
Background ............................................ 378
IPC frames ............................................. 378

D. Files .................................................. 381

Introduction ........................................... 381
Command input file (.inp) ............................ 382
Sample CDIscoper .inp file ......................... 382
Forcefield parameter file (.frc) ...................... 383
Format of .frc file .................................... 383
   Version and reference numbers .................. 384
   MSI file identification .......................... 386
   Forcefield version ................................ 386
   Atom types ....................................... 386
   Equivalence table ................................ 387
   Hydrogen bond donors and acceptors .......... 388
   Quadratic bond-stretching potential .......... 388
   Quartic bond-stretching potential ............. 389
   Morse bond-stretching potential ............... 389
   Quadratic angle-bending potential ............ 390
   Quartic angle-bending potential ............... 391
   Quadratic bond–bond interaction potential .... 391
   Quadratic bond–angle interaction potential .... 392
   One-term torsion potential ...................... 393
   Three-term torsion potential ................... 394
   Four-term torsion potential ..................... 395
   Angle–torsion interaction potential .......... 395
   Peripheral bond–torsion interaction potential .. 396
   Central bond–torsion interaction potential .... 397
   Three-term angle–torsion interaction potential .. 398
Out-of-plane potential, improper torsion definition . 399
Out-of-plane, Wilson definition .................... 399
Out-of-plane interaction potential using improper
torsion definition ................................. 400
Angle–angle interaction potential .................. 401
6–9 Nonbond (van der Waals) potential ............... 402
6–12 Nonbond (van der Waals) potential .............. 404
10–12 Hydrogen bond potential ..................... 406
Bond increments ..................................... 407
Standard output file (.out) ........................... 407
CDiscvery output files (.arc, .xfrc, .out, .tbl, .xdyn, user-named) 408
Discover dynamics restart files (.xdyn) ............... 408
Automatic potential parameter assignment .......... 409
ESFF structure output file (.esf) ................. 409
ESFF forcefield parameter file (.epa) ............. 410
Torsion information .................................. 411
Dynamics tabular output for energies (.gre) ........ 411
Dynamics tabular output for thermodynamic states (.grp) 411
Dynamics scratch file (.pre) ........................ 412
General Index ........................................ 413
List of Tables

Table 1. Finding information about using Discover within Cerius²  
11
Table 2. Finding information about analyzing results ........ 41
Table 3. Summary of operators allowed in Btcl ............. 128
Table 4. Summary of functions available in Btcl .......... 129
Introduction

This introduction includes information on:

What is the Discover® program?
Accessing CDiscover
How to use the documentation

What is the Discover® program?

What Discover can do

The Discover® program performs energy minimization, template forcing, torsion forcing, and dynamic trajectories and calculates properties such as interaction energies, derivatives, mean square displacements, and vibrational frequencies. It provides tools for performing simulations under various conditions including constant temperature, constant pressure, constant stress, periodic boundaries, and fixed and restrained atoms.

What you can study with Discover

Using appropriate methods and strategies, you can study docking interactions such as enzyme–substrate, polymer–polymer, or receptor–ligand interactions. You can also evaluate the multitude of conformations available to a model, energy refine a model-built structure, and evaluate configurations or chemical perturbations of a system.

Uses of minimization

Typical uses of energy minimization include:

♦ Optimizing initial geometries of models constructed from fragments with the Cerius²® or Insight® II modeling program.

♦ Repairing poor geometries occurring at splice points during homology building of protein structures.
1. Introduction

- Mapping the energy barriers for geometric distortions and conformational transitions using "torsion forcing" to obtain Ramachandran-type contour plots for proteins or RIS statistical weights for polymers.
- Evaluating whether a model can adopt a template conformation consistent with a pharmacophoric or catalytic site model ("template forcing").

Uses of dynamics

Typical uses of molecular dynamics include:

- Searching the conformational space of alternative amino acid sidechains in site-specific mutation studies.
- Identifying likely conformational states for highly flexible models such as polymers or for flexible regions of macromolecules such as protein loops.
- Producing sets of 3D structures consistent with distance and torsion constraints deduced from NMR experiments (simulated annealing).
- Calculating free energies of binding, including solvation and entropy effects.
- Probing the locations, conformations, and motions of fragments on catalyst surfaces.

Other uses of Discover

In addition, Discover can be routinely used for:

- Calculating normal modes of vibration and vibrational frequencies.
- Analyzing intramolecular and intermolecular interactions in terms of residue–residue or molecule–molecule interactions, energy per residue, or interactions within a radius.
- Calculating diffusion coefficients of small models in a polymer matrix.
- Calculating thermal expansion coefficients of amorphous polymers.
- Calculating the radial distribution of liquids and amorphous polymers.
What is the Discover® program?

♦ Performing rigid-body rms comparisons between minimized conformations of the same or similar structures or between simulated and experimentally observed structures.

**Comparison of C and FORTRAN versions**

The C version of the Discover program (“CDiscover”) can perform certain calculations that cannot be undertaken with the older FORTRAN version (“FDiscover”, also known as versions 2.9.x, where \( x = 0–9 \), not available within the Cerius\(^2\) interface, and documented separately).

**CDiscover advantages**

For example, CDiscover can handle bonds-across-boundaries—infinite bonding in periodic systems such as a perfect polyethylene crystal or a silicate structure. CDiscover can minimize the cell shape and dimensions or run constant-stress dynamics on periodic systems. Therefore, if you need any of these features, you must use CDiscover. (For other calculations, such as simple minimizations or dynamics on nonperiodic systems or constant-volume minimization or constant-pressure dynamics on periodic systems, you may use either version of the Discover program.)

CDiscover is self-dimensioning—it fits itself to the size of the problem as long as sufficient swap space is available—and it allows considerably more detailed control of the calculation in the Cerius\(^2\), Insight, and standalone modes of operation than does FDiscover.

**Compatibility with older Discover program**

For the most part, CDiscover and FDiscover are reasonably compatible with each other. They accept the same format for the .car and .mdf model-structure files and produce identical .cor, .arc, and .his output files during minimization. The energy calculated by the two programs is identical, within the numerical accuracy of the machines. The input files differ—the Biosym Tool Command Language (Btcl) used in CDiscover differs in both detail and philosophy from the Discover Simulation Language (DSL) used in FDiscover.

**CDiscover—Cerius\(^2\), Insight and standalone modes**

CDiscover can be run as a standalone program (from a textual interface) or as an application module within MSI’s Cerius\(^2\) or...
1. Introduction

Insight II graphical molecular modeling interface, which are available under separate license.

When purchased with the Cerius² or Insight interface, CDiscover is accessible as one of the application modules.

The functionality in CDiscover is accessed through control panels in Cerius² (or parameter blocks in Insight) that contain options to set up the command input file for a CDiscover job. The graphical interface provides you with many useful defaults as well as preset strategies, all of which can be easily modified, as necessary, for your particular calculation. Most commands in the graphical interface are used to set up the command input file—CDiscover does not actually run until you specifically start it.

You can start the run while remaining in the Cerius² or Insight environment. Then you can remain within the graphical interface, waiting for a short run to finish and/or performing other, independent tasks in the meantime. For long runs, you may prefer to quit the graphical interface and allow CDiscover to continue running on its own.

The Cerius² or Insight interface can also be used to prepare input files (both command input files and model structure files) for running CDiscover in the standalone mode. The command input files can be saved and then edited to develop more sophisticated simulation strategies or for repetitive calculations better suited to the automatic control of batch queues.

When run as a standalone program, CDiscover uses a simple textual interface that helps you create a command input file that instructs CDiscover what to do.

The command language used in the input file, Btcl (“Biosym” Tool Command Language), gives a high level of control over the variables and functions used in a calculation. These functions include logical operators and flow control statements that allow you to control the calculation based on energies, derivatives, distances, angles, forcing potentials, temperature, and user-defined variables. You can also use alternative input files and can print with conditional operators and with formatting to user-defined output files. These functions provide useful decision points for branching from minimization to dynamics, for example, and for archiving and producing results, so that you can develop sophisticated sim-
Accessing CDiscover

ulation strategies to be performed in a single run without your intervention.

Examining the results The results of a CDiscover run can be loaded back into Cerius² or Insight for plotting of data and other graphical and statistical analyses.

Accessing CDiscover

In Cerius² To access CDiscover within Cerius², click the deck selector in the main control panel and choose DISCOVER from the list that appears. The deck of cards menu area should now look like this:

In Insight II CDiscover is accessed within the Insight environment by selecting Discover_3 from the Module pulldown (click the MSI logo and choose Discover_3 from the list that appears). Several new pull-down menus on the lower menu bar.

By itself You can also set up and run CDiscover independent of either graphical interface. The standalone user interface is activated by typing discovery at the operating system command level. A textual interface asks you to specify the files that you want to set up for the CDiscover calculation and also enables you to directly issue Btcl commands.
1. Introduction

**Prerequisites**

Before running CDiscover, you should be familiar with the documentation (next section), as well as with:

♦ How to use the Cerius² or Insight graphical interface and what the various types of controls are called (see *Cerius² Tutorials* and *Cerius² Modeling Environment* or *Insight II*).

♦ The theory and general methodology for performing various types of forcefield-based calculations (see *Forcefield-Based Simulations*).

♦ The windowing software on your workstation.

♦ Basic UNIX commands.

Your workstation should have:

♦ Licensed copies of CDiscover and either Cerius² or Insight.

♦ A directory in which you have write permission.

**How to use the documentation**

*CDiscover* is written mainly for the typical scientist-user of the CDiscover modules within the Cerius² and Insight II graphical interfaces, as well as of the standalone CDiscover program. Although all three modes of accessing CDiscover use reasonable default values for simple calculations, you should read this documentation set, as well as *Forcefield-Based Simulations*, if you want to make efficient use of the program, obtain the best results possible, and understand the results.

This CDiscover documentation

You need not read this entire documentation set before you start using CDiscover:

♦ Everyone should scan the *Introduction* and should familiarize themselves with the introductory material and section headings in *CDiscover within Cerius²*, *CDiscover within Insight*, and/or *Standalone CDiscover* (depending on whether you are using CDiscover primarily in the Cerius², Insight, or standalone modes, respectively).

♦ The extent to which you actually read *CDiscover within Cerius²*, *CDiscover within Insight*, and *Standalone CDiscover* and the
Appendices depend on what tasks you want to perform, how much you need to customize your calculations, and your accumulated level of expertise with CD Discover.

**Outline of this documentation set**

CD Discover documents the use of C Discover in the Cerius² and Insight environments (*CD Discover within Cerius²* and *CD Discover within Insight*, respectively) and as a standalone program (*Standalone CD Discover*). Each chapter includes an outline of how to use CD Discover in the respective mode.

*CD Discover within Insight* and *Standalone CD Discover* also include brief summaries of the main functions of each command in the Insight environment or in raw Btcl.

Several appendices contain information that is intended mainly for expert CD Discover users:

♦ **Btcl Language and Commands—Standalone Mode** documents the language and keywords that are used in the input file to run CD Discover. (These keywords are the same whether the input file is constructed by means of the graphical or standalone interfaces to CD Discover.)

**Remember**

The Btcl language of CD Discover differs from the DSL used by F Discover. Therefore, the input files for CD Discover and F Discover are not interchangeable.

♦ **Databases and Tables** explains the databases and tables that CD Discover uses internally and which you can access and manipulate through Btcl scripts.

♦ **Discover Interprocess Communication (IPC)** documents the procedures for communicating between CD Discover and other programs. It is mainly for advanced users of CD Discover.

♦ The files specific to the CD Discover program are documented in *Files*.

**Additional information sources**

**On-screen help**

On-screen help is accessed within the Cerius² environment by clicking the right mouse button while the cursor is over the item in the interface about which you want information. A brief identifi-
1. Introduction

A brief identification of some items appears when you simply allow the cursor to linger over them. Additional help and some demos are accessed from the Help menu.

In Insight, on-screen help is available via a help icon, which opens a separate window of information. This window normally shows information relevant to whatever parameter block is currently open. A brief identification of some items appears when you simply allow the cursor to linger over them. Additional help and some tutorials are accessed from the Help menu.

MSI’s website

The URL for the customer support and documentation areas of MSI’s website are:

http://www.msi.com/doc/
http://www.msi.com/support/

Information relevant to forcefields, simulation engines, CDiscover, and file formats can be found.

The software CD

Many Insight and standalone tutorials are available on the CD containing your software—see Pilot tutorials and Tutorial—standalone mode for more information.

Typographical conventions and terminology

Please see the glossary in Cerius$^2$ Modeling Environment for definitions of the terms model, fragment, molecule, etc. We have tried to use the correct terms consistently within this documentation.

Unless otherwise noted in the text, CDiscover uses these typographical conventions:

- Terms introduced for the first time are presented in italic type. For example:

  Instructions are given to the software via control panels.

- Keywords in the interface are presented in bold type. In addition, slashes (/) are used to separate a menu item from a sub-item. For example:

  Select the View/Colors… menu item means to click the View menu item, drag the cursor down the pulldown menu that appears, and release the mouse button over the Colors… item.
How to use the documentation

♦ Words you type or enter are presented in **bold** type. For example:

Enter **0.001** in the entry box.

♦ UNIX command dialog and file samples are represented in a **typewriter** font. For example, the following illustrates a line in a .grf file:

```
CERIUS Grapher File
```

♦ Words in *italics* represent variables. For example:

```
> cerius2 -b outputfile scriptfile
```

In this example, when you enter the command, replace *outputfile* with the name of the file to which text output should be directed and replace *scriptfile* with the name of a file containing a command script.

Please see *Btcl Language and Commands—Standalone Mode* for information on conventions used in documentation of standalone commands and keywords.
1. Introduction
This chapter explains

- Preparing the model and forcefield
- Setting up the simulation conditions
- Setting up and starting the run
- Analyzing results

Tip

Cerius² contains reasonable default settings for many of the topics discussed in this chapter. However, you may want or need to override default settings, depending on your model and the purpose of your calculation.

Paragraphs that are flagged with When is this needed? help you decide whether you need to read that section.

Table 1. Finding information about using Discover within Cerius²

<table>
<thead>
<tr>
<th>If you want to know about:</th>
<th>Read:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Forcefield and simulation theory.</td>
<td>Forcefield-Based Simulations (separate book from MSI).</td>
</tr>
<tr>
<td>Available and recommended forcefields.</td>
<td>Forcefield-Based Simulations; Preparing the model and forcefield.</td>
</tr>
<tr>
<td>Loading a forcefield.</td>
<td>Choosing the forcefield.</td>
</tr>
<tr>
<td>General strategy for setting up simulations.</td>
<td>Forcefield-Based Simulations.</td>
</tr>
<tr>
<td>Atom types and atom typing.</td>
<td>Forcefield-Based Simulations, Assigning atom types and charges.</td>
</tr>
<tr>
<td>van der Waals and Coulombic (electrostatic)</td>
<td>Nonbond interactions.</td>
</tr>
<tr>
<td>interactions</td>
<td>Methods, types of models, and recommendations.</td>
</tr>
<tr>
<td>Nonbonds and model size and periodicity.</td>
<td></td>
</tr>
</tbody>
</table>
2. C Discover within Cerius$^2$

Table 1. Finding information about using Discover within Cerius$^2$

<table>
<thead>
<tr>
<th>If you want to know about</th>
<th>Read:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Calculating energy or gradient of current structure.</td>
<td>Single-point energy and gradient calculations.</td>
</tr>
<tr>
<td>Performing minimization.</td>
<td>Minimization.</td>
</tr>
<tr>
<td>Choosing a minimizer.</td>
<td>Specifying the minimization algorithm.</td>
</tr>
<tr>
<td>Run-termination criteria.</td>
<td>Setting convergence criteria.</td>
</tr>
<tr>
<td>Performing dynamics simulation.</td>
<td>Dynamics.</td>
</tr>
<tr>
<td>Controlling kinetic or thermodynamic temperature.</td>
<td>Controlling temperature, pressure, and stress.</td>
</tr>
<tr>
<td>Applying constant stress and/or pressure.</td>
<td>Controlling temperature, pressure, and stress.</td>
</tr>
<tr>
<td>The integration timestep.</td>
<td>Defining the timestep.</td>
</tr>
<tr>
<td>Text output, trajectory files.</td>
<td>Specifying output.</td>
</tr>
<tr>
<td>Starting a minimization or dynamics run.</td>
<td>Starting the run.</td>
</tr>
<tr>
<td>Analysis of results.</td>
<td>Analyzing results.</td>
</tr>
</tbody>
</table>

You should already know…

Forcefield-Based Simulations contains information on the theory and general methodology for performing forcefield-based calculations.

Accessing the tools

Most tools for preparing the forcefield and model for a forcefield-based calculation using Discover within Cerius$^2$ are accessed from one of the decks of cards in the main Visualizer control panel, the DISCOVER card deck.

To access the C$^2$•DISCOVER module, click the deck selector in the main control panel and choose DISCOVER from the list that appears. Then click the title of the DISCOVER card to bring it to
Preparing the model and forcefield

The forcefield forms the basis of many energy calculations performed from within Cerius². The forcefield used within the C²•Discover module is set within that module (that is, not in the Open Force Field module). Setup of the energy expression occurs automatically when the run is started.

Choosing the correct forcefield is vitally important in getting reasonable results from energy calculations. *Forcefield-Based Simulations* contains full documentation of what forcefields are best for which purposes. In summary, the forcefields available through the C²•Discover interface are:

<table>
<thead>
<tr>
<th>Use</th>
<th>Forcefield</th>
</tr>
</thead>
<tbody>
<tr>
<td>General-purpose.</td>
<td>CVFF, CFF91, CFF</td>
</tr>
<tr>
<td>Polymers, metals, and zeolites.</td>
<td>PCFF</td>
</tr>
</tbody>
</table>
2. CDiscover within Cerius²

Finding information

This section includes information on:

Choosing the forcefield
Building or loading models
Assigning atom types and charges
Defining charge groups
Setting constraints
Preparing the energy expression

You should already know…

Forcefield-Based Simulations, Forcefields contains full information on what forcefields are available in MSI products and their recommended uses.

Choosing the forcefield

When is this needed?

A default forcefield is automatically loaded. You need to read this section only if you do not want to use the default forcefield.

You should already know…

Reasons for choosing various forcefields are outlined in Forcefield-Based Simulations, Forcefields.

Accessing the tools

Select the Run menu item to open the Run Discover control panel. Then (if necessary, see below) click the More… pushbutton to the right of the Forcefield popup to open the Select Discover Forcefield control panel.

Alternatively, you can select the Forcefield/Select menu item from the DISCOVER card to open the Select Discover Forcefield control panel.

Which forcefield?

Choose the desired forcefield from the Forcefield popup in the Run Discover control panel (or from the popup near the top of the Select Discover Forcefield control panel).

If the desired forcefield is not listed in these popups (and if you have other forcefields in the .frc file format in your directory system), you may use the file browser controls in the Select Discover
Preparing the model and forcefield

Forcefield control panel to find and **SELECT** the desired forcefield. This forcefield is also added to the forcefield popups in both control panels, where it remains throughout the current Cerius² session.

If you want to permanently add nondefault forcefields to these forcefield popups, use the **File/Save Session**… menu item on the main Visualizer control panel to save the session. Then reload the session when you want easy access to its selection of forcefields.

**Additional information**

Please see the on-screen help for details on the functioning of each control in the Run Discover and Select Discover Forcefield control panels.

---

**Building or loading models**

**You should already know…**

Building models is covered in Cerius² *Modeling Environment, Building Models*, and in documentation for the separate builder modules. Reading in models from files and specifying the current model are detailed in *Cerius² Modeling Environment, Managing Models and Sessions*.

*Forcefield-Based Simulations, Preparing the Energy Expression and the Model* contains full information on the theory and general methodology for preparing the energy expression and the model for a forcefield-based simulation run.

**Finding and enforcing symmetry**

If a model is nearly symmetrical and you want to force it to a conformation with exact symmetry before beginning your calculation, you can use the Symmetry control panel that is accessed by selecting the **Geometry/Symmetry** menu item on any of the **QUANTUM 1 cards** (documented separately).

---

**Assigning atom types and charges**

**When is this needed?**

Atom types must be assigned to each atom in a model before a forcefield-based simulation can be run. Atom typing and charge assignment are done automatically when the run is started. However, you may want to change this default behavior, for example, to assign special atom types to certain atoms or to examine what atom types are assigned automatically before starting the run. If so, you need to read this section.
2. CDiscover within Cerius²

You should already know…

Information on assignment of atom types is contained in *Forcefield-Based Simulations* under *Assigning forcefield atom types and charges*, and lists of atom types that are defined in various forcefields are found in that same documentation set in *Forcefield Terms and Atom Types*.

**Important**

A newly assigned atom type mass or charge replaces any previously assigned or calculated value. Also, if a forcefield has been parameterized without atom-type charges, it replaces any existing charges with zero.

An atom-type charge is simply a fixed value associated with an atom type. Depending on the forcefield, overall neutrality of a model may not be achieved by assigning forcefield atom types.

**Accessing the tools**

Select the **Run** menu item from the **DISCOVER** card to open the Run Discover control panel. Click the **Typing…** pushbutton to open the Discover Atom Typing control panel.

Alternatively, select the **Forcefield/Typing** menu item to open the Discover Atom Typing control panel.

**Assigning atom types**

By default, the **Perform Automatic Typing** check box is checked, which means that atom types are assigned when you start the simulation run. To type atoms *only* when desired (for example, to prevent manually assigned atom types, below, from being overwritten), uncheck this check box. To turn automatic atom typing back on, check this box before starting a calculation.

To calculate and assign the atom types and charges for the current model when desired, click the **Calculate Atom Types** action button. If types cannot be assigned to some atoms, you can assign atom types manually (below).

**Tip**

You can label a model according to forcefield atom types by setting the label popup on the main control panel’s toolbar to **FFTYPE** to see if all atoms in your model are typed as expected. Atoms labeled with question marks have not had atom types assigned.

To manually assign atom types to specific atoms, select the atom(s), enter the desired atom type in the **Assign Atom Type** entry box, and click the **Assign Atom Type** action button. You are,
Preparing the model and forcefield

of course, restricted to atom types that are valid in your current forcefield.

Verifying atom type assignments

You can check the atom typing by labeling your model according to FFFTYPE. You can also label according to MASS, CHARGES, and HYBRID to check the atom attributes. (Atomic charges in particular can change when the atoms are typed.)

Repeat the atom type assignment procedure for selected atoms if you need to change some assignments.

Additional information

Please see the on-screen help for details on the functioning of each control in the Discover Atom Typing control panel.

Defining charge groups

A charge group is a small group of atoms close to each other which have a net charge of zero or almost zero. They are defined so as to avoid creating spurious monopoles (by artificially splitting dipoles), which can lead to errors when nonbond cutoffs are used during a calculation. In many cases, charge groups are identical to common chemical functional groups. Thus, a carbonyl group, methyl group, or carboxylic group would be an approximately neutral charge group.

You should already know...

Information on charge groups is contained in Forcefield-Based Simulations under Charge groups and group-based cutoffs.

Accessing the tools

Select the Run menu item from the DISCOVER card to open the Run Discover control panel. Click the Typing... pushbutton to open the Discover Atom Typing control panel.

Alternatively, select the Forcefield/Typing menu item to open the Discover Atom Typing control panel.

Defining charge groups

By default, the Perform Automatic Grouping check box is checked, which means that grouping is performed when you start the simulation run. To group charged atoms only when desired (for example, to prevent manually assigned groups, see below, from being overwritten), uncheck this check box. To turn automatic grouping back on, check this box before starting a calculation.

To calculate and assign charge groups for the current model on demand, click the Calculate Charge Groups action button.
2. CDiscover within Cerius²

Displaying charge groups

Click the **Color Charge Groups** action button to display the groups. If some groups are not appropriate, you can reassign groups manually (see below).

To restore the default colors, select the **View/Colors...** item from the main Visualizer control panel’s menu bar and then click the **Reset selected object colors** action button in the Color Selected Objects control panel.

Tip

You can label a model according to charges by setting the label popup on the main control panel’s toolbar to **CHARGES** to see if the assigned groups are roughly neutral overall.

To manually define a group, select the atoms to be in that group and click the **Define Charge Group** action button.

This panel also contains controls for removing definitions of charge groups and for setting the tolerance that is used during automatic grouping.

Additional information

Please see the on-screen help for details on the functioning of each control in the Discover Atom Typing control panel.

Setting constraints

Atom constraints enable you to fix the atomic coordinates of selected atoms during simulations. Fixing atoms simplifies the energy expression, since interactions between fixed atoms are constant and can be ignored. This decreases computational expense.

Uses of constraints

Use atom constraints when you want to apply minimization or dynamics to *part* of a model, keeping the remainder of the model fixed and rigid. For example, you can quickly minimize a sorbate in a zeolite by fixing the atom positions of the zeolite and allowing only the sorbate atoms to move.

When is this needed?

Use of constraints is optional. If used, constraints must be set before the energy expression is generated. (Automated generation of the energy expression occurs as the final step before a simulation run starts.)
## Finding information

This section includes information on:

- **Fixed atoms**
- **Constraints during dynamics**

## You should already know...

The difference between restraints and constraints, when to use constraints, and the types of constraints available are covered in *Forcefield-Based Simulations*, under *Applying constraints and restraints*.

## Related information

You can specify that external stress or pressure (which may be viewed as a kind of restraint) be applied to a periodic system when you set up a minimization (see *Minimization*) or dynamics (see *Dynamics*) simulation.

## How constraints work

Forces between fixed atoms are not included in the energy expression, but forces between fixed and mobile atoms are included. This means that the total energy value calculated for a given model depends on whether the model contains fixed atoms.

By default, all atoms are movable.

### Fixed atoms

#### Accessing the tools

To constrain (i.e., fix) atoms, select the **Constraints/Atom** menu item from the DISCOVER card to open the Atom Constraints control panel.

#### Constraining atoms

To set atom constraints, select the atoms whose positions you want to remain fixed during subsequent simulation runs. Then click the **Fix Atomic Position** action button in the Atom Constraints control panel.

To allow previously constrained atoms to move during subsequent simulation runs, select the atoms and click the **Allow Atomic Motion** action button.

#### Tip

This control panel allows you to quickly check which atoms are fixed or movable, by coloring them distinctively.

You can also use the Color Selected Objects control panel to color atoms according to their movability (see *Cerius² Modeling Environment, Coloring atoms by properties*).
2. CDiscover within Cerius²

Additional information Please see the on-screen help for details on the functioning of each control in the Atom Constraints control panel.

Constraints during dynamics

The RATTLE algorithm effectively removes very-high-frequency vibrations from consideration during dynamics simulations. Its use can allow for a larger time step during simulations under NVE and NVT conditions.

Accessing the tools

Select the Run menu item from the DISCOVER card to open the Run Discover control panel. Set the Task popup to Dynamics. Click the More... pushbutton to the right of the Task popup to open the Discover Dynamics control panel. Check the Rattle check box and click the More... pushbutton to the right of the Rattle check box to open the Rattle control panel.

Constraints during dynamics

Check or uncheck the Rattle check box in the Discover Dynamics control panel and the Rattle Bond and Rattle Angle check boxes in the Rattle control panel. The latter boxes control whether inter-atomic distances or angles (respectively) are constrained during a dynamics simulation. Angles should be constrained only if the lengths of the component bonds are also constrained.

Tip

We do not recommend that angles be constrained even though this functionality is available. However, if angle constraints have to be used, you should use a larger tolerance than with bonds. (Tolerance can be set by saving and editing the command input file (see Other file-control issues and rattle) before starting the run.)

Additional information Please see the on-screen help for details on the functioning of each control in the Rattle control panel.

Preparing the energy expression

If you have a large model, you probably do not want to calculate the nonbond interactions between all possible pairs of atoms not bonded to each other, to decrease computational expense. Or you might want to enhance or decrease the effect of other terms in the energy expression.
Preparing the model and forcefield

Finding information
This section includes information on:

- Nonbond interactions
- Parameter scaling
- What to do if some parameters cannot be assigned

You should already know…
Please see Preparing the Energy Expression and the Model of Force-field-Based Simulations for all the relevant theory and general methodology on preparing energy expressions.

Nonbond interactions
How long-range nonbond interactions are treated is an important factor in determining both the accuracy and the computational expense of a forcefield-based simulation.

When is this needed?
Specifying how nonbond interactions are handled is optional. If performed, it must be done before the energy expression is set up.

You should already know…
Forcefield-Based Simulations, under Handling nonbond interactions, contains information on the theory of nonbond interactions in forcefield-based simulations, functional forms of the energy terms, combination rules, the simulated dielectric constant, distance-dependent dielectric “constants”, cutoffs and other methods for handling nonbond interactions, neighbor lists, etc. It also includes full citations of the scientific literature.

Methods, types of models, and recommendations
Cerius²•Discover offers several methods for calculating long-range atom–atom interaction energies:

♦ Switching function—Calculate nonbond interaction energies for all atom pairs whose members are closer than the (Cutoff Distance minus the Spline Width) (or for all atom pairs belonging to groups whose switching atoms are closer than the (Cutoff Distance minus the Spline Width)); gradually attenuate the interaction energy (by a switching function) from its full value to zero for atom–atom distances between the (Cutoff Distance minus the Spline Width) and the Cutoff Distance; set the interaction energy to zero for atom–atom distances greater than the Cutoff Distance (see the curve labelled “Discover 97.0/4.0.0” in Figure 15 in Forcefield-Based Simulations for an illustration of these distances).
2. C Discover within Cerius²

Recommended for nonperiodic models. Use a nonbond list (that is, set an appropriate Buffer Width) for faster calculation with most models.

Not recommended for periodic models, except for van der Waals terms in models having 2D periodicity.

♦ Cell multipole method—Treat short-range interactions with no cutoffs; treat long-range interactions in terms of multipoles (Greengard and Rokhlin 1987, Schmidt and Lee 1991, Ding et al. 1992).

Currently available in C²•Discover only for nonperiodic systems.

♦ Ewald sum—Calculate Coulombic interactions and attractive van der Waals interactions by the Ewald method (Karasawa & Goddard 1989, Ewald 1921); calculate repulsive van der Waals interactions (which fade out quickly with distance) without cutoffs.

Recommended for 3D-periodic models (and preferred over the switching-function method). Convergence is generally faster if the geometric combination rule is used (and any defined off-diagonal parameters are ignored) for van der Waals interactions between nonidentical atom types. Must be used with a nonbond list.

Cannot be used for nonperiodic models.

♦ Cell—Evaluate all nonbond interactions between atoms within a cell and between atoms in the central cell and a specified number of layers of surrounding cells.

Applicable only to periodic models.

Please see Forcefield-Based Simulations under Modeling periodic systems and Handling nonbond interactions for details on these methods.

Accessing the tools
Select the Forcefield/Nonbond menu item from the DISCOVER menu card to open the Discover Non-Bond control panel.

Technical notes
If you want to treat van der Waals and Coulombic interactions differently, check the Treat VDW and Coulomb Separately check box. Otherwise, changes you make in Van der Waals section of this
Preparing the model and forcefield

control panel are automatically made in the Coulomb section (and vice versa).

Setting nonbond parameters
Use the Discover Non-Bond control panel to choose the summation method and then set the parameters required by your chosen method. (The controls that appear in this panel depend what Summation Method is chosen.)

You can also set the dielectric constant and specify using a distance-dependent dielectric “constant” with this control panel.

Additional information
Please see the on-screen help for details on the functioning of each control in the Discover Non-Bond control panel.

Parameter scaling
The contributions of various terms in the potential energy expression to the total energy can be scaled up or down (or completely excluded). This can be useful, for example, in the early stages of minimizing very “bad” structures, where large contributions by certain terms might interfere with convergence.

Accessing the tools
Select the Forcefield/Scaling menu item on the DISCOVER card to open the Discover Scaling control panel.

Setting scale factors
You can set scale factors for all instances of these classes of terms—bond, angle, torsion, out-of-plane, cross term, van der Waals, Coulomb—by entering values in the appropriate entry boxes.

Additional information
Please see the on-screen help for details each control in the Discover Scaling control panel.

What to do if some parameters cannot be assigned
Most forcefields contain parameters for every combination of forcefield atom types for which the forcefield has been designed. Thus, the appropriate energy terms are read from the forcefield file for each atom type in each atom, bond, angle, etc. in the model.

However, if some atom types do not have specific (“explicit”) parameters for some energy terms, you can request that Discover try to find and use generic terms instead. You can also request that the calculation by stopped if terms cannot be assigned for some components of the energy expression.
2. C Discover within Cerius

However, if an atom cannot be typed or has been assigned a type that is not present in the current forcefield, the Discover run will terminate.

Accessing the tools

Select the Forcefield/Parameters menu item on the DISCOVER card to open the Discover Parameters control panel.

Handling unassigned parameters

You can ask Discover to assign (or not) automatic parameters if no explicit parameters are found for some bond, angle, torsion, and/or out-of-plane terms by checking or unchecking the appropriate Auto check box.

You can tell Discover to stop the calculation if energy terms cannot be assigned for any bond, angle, torsion, out-of-plane, or cross terms by checking or unchecking the appropriate Stop check box.

Additional information

Please see the on-screen help for details on the functioning of each control in the Discover Parameters control panel.

Setting up the simulation conditions

Once you have prepared the model and the forcefield (Preparing the model and forcefield), you need to set up the calculation, that is, to specify the calculation conditions and desired output.

Finding information

This section includes information on:

- Single-point energy and gradient calculations
- Minimization
- Dynamics

Prerequisites

Please read Preparing the model and forcefield, for how and when you need to load a forcefield, prepare the model by assigning atom types, constraints, etc., and prepare the forcefield by choosing appropriate terms. See especially the paragraphs that are flagged with When is this needed? The tasks in Preparing the model and forcefield (whether optional or required) must be performed before those discussed in this section.
Setting up the simulation conditions

**Single-point energy and gradient calculations**

The total energy and the forces on each atom can be calculated for the current model by using the single-point energy and gradient tasks.

**Accessing the tools**

Select the Run menu item from the DISCOVER card to open the Run Discover control panel.

**Calculating the current energy**

To calculate the current energy of the model without changing the position of any atoms, set the Task popup to Single Point Energy.

**Evaluating the current forces**

To evaluate the current forces applied to each atom of the model without changing the position of any atoms, set the Task popup to Gradient.

**Viewing the results**

The output from these tasks is sent to the .out file associated with the job, which also appears in a window during the course of an interactive job.

The force vectors can be viewed by selecting the Geometry/Vector Properties... item from the main Visualizer control panel’s menu bar and clicking the SHOW pushbutton in the Vector Properties control panel.

**Additional information**

Please see the on-screen help for details on the functioning of each control in the Run Discover and Vector Properties control panels.

---

**Minimization**

The main purpose of a minimization run is to optimize the conformation of the model so that its conformation is one it is likely to assume at 0 K.

**Finding information**

This section includes information on:

- Specifying the minimization algorithm
- Setting convergence criteria
- Maximum number of iterations

**Accessing the tools**

Select the Run menu item from the DISCOVER card to open the Run Discover control panel.
2. C Discover within Cerius²

**Specifying minimization**
To specify that you want to perform minimization, set the Task popup to Minimization.

**Refining the minimization conditions**
If you want to select a nondefault minimization algorithm or strategy or change the convergence criteria, click the More… pushbutton that appears to the right of the Task popup in the Run Discover control panel (when Task is set to Minimization) to open the Discover Minimize control panel.

**Additional information**
Please see the on-screen help for details on the functioning of each control in the control panels mentioned in this section.

**Specifying the minimization algorithm**
Several minimization algorithms are packaged into a “smart” minimizer, which uses a robust but less accurate algorithm in the first stage of the calculation and a highly accurate but less robust one near the end of the run. You can omit any of the stages in the smart minimizer and can choose alternative minimizers for some stages.

**When is this needed?**
If the default minimization strategy is sufficient to your needs, you do not need to read this section. However, you should read this section if you want to examine and/or change the default settings.

**You should already know…**
Forcefield-Based Simulations, Minimization, contains full information on how the various minimizers work and their advantages and disadvantages, to enable you to choose the best one to use for your model and calculation.

**Controlling the minimizer**
To omit a minimization stage, uncheck the Steepest Descent, Conjugate Gradient, and/or Newton check boxes in the Discover Minimize control panel.

For the Conjugate Gradient stage of minimization, you can choose between the Fletcher–Reeves and Polak–Ribiere minimizers.

Four Newton minimizers are available and can be specified in the Discover Minimize control panel: BFGS (Broyden–Fletcher–Goldfarb–Shanno), DFP (Davidon–Fletcher–Powell), truncated Newton–Raphson, and iterative (pure) Newton–Raphson.

**Setting convergence criteria**
A minimization run ends when either the convergence criteria are met or the maximum number of iterations has been performed.
Setting up the simulation conditions

When is this needed?
If the default convergence criteria are sufficient to your needs, you do not need to read this section. However, you should read this section if you want to examine and/or change the default settings.

You should already know...
Forcefield-Based Simulations, under Convergence criteria, contains full information on convergence criteria and on how strict they need to be, depending on the purpose of your calculation. The significance of the minimum-energy structure is discussed under Significance of minimum-energy structure in that documentation set.

Quick-set of convergence criteria
To quickly specify any of three predefined levels of convergence criteria, set the Convergence Strategy popup in the Discover Minimize control panel (accessed as described under Accessing the tools) to MODERATE, STANDARD, or HIGH.

Detailed specification of convergence criteria
Whether or not you first use the Convergence Strategy popup, you may edit any of the Convergence and Line Search values to set the convergence criteria and specify the accuracy of the line searches for each stage of minimization.

Minimization of 3D-periodic models
If the current model is periodic, an Optimize Cell check box appears in the Discover Minimize control panel. To optimize only the atomic positions (and not also the cell parameters), uncheck this check box.

Maximum number of iterations

When is this needed?
If the default maximum number of iterations is sufficient to your needs, you do not need to read this section. However, you should read this section if you want to examine and/or change the default setting.

Setting the maximum number of iterations
To change the maximum number of iterations before a run terminates in Cerius² Discover, enter the desired number in the Iterations entry box in the Discover Minimize control panel (accessed as described under Accessing the tools).

Dynamics

While minimization (Minimization) computes the forces on the atoms and changes their positions to minimize the interaction energies, dynamics computes forces and moves atoms in response to the forces.
2. C Discover within Cerius²

Finding information

This section includes information on:

- Defining the timestep
- Defining the thermodynamic ensembles
- Controlling temperature, pressure, and stress
- Type of dynamics simulation
- Stages of a dynamics run

Prerequisites

You almost always need to minimize the model (see Minimization) before performing a dynamics simulation. (You may also want to perform minimization after the run, on several conformations that were output from the run.)

Accessing the tools

Select the Run menu item from the DISCOVER card to open the Run Discover control panel.

Specifying dynamics

To specify that you want to perform dynamics, set the Task popup to Dynamics.

Refining the dynamics conditions

If you want to perform a nondefault dynamics run, click the More… pushbutton that appears to the right of the Task popup in the Run Discover control panel (when Task is set to Dynamics) to open the Discover Dynamics control panel.

The exact controls available in this panel depend on whether the current model is periodic or nonperiodic and on the settings of some controls within this panel.

By default, the velocity scale integrator and NVT thermodynamic conditions are used in the equilibration stage of the run. If you prefer to use different conditions, click the More… pushbutton in the Equilibration section of the Discover Dynamics control panel to open the Equilibration Options control panel.

Additional information

Please see the on-screen help for details on the functioning of each control in the control panels mentioned in this section.

Defining the timestep

A key parameter in the integration algorithms is the integration timestep \( \Delta t \). To make the best use of the computer time, a large
Setting up the simulation conditions

timestep should be used. However, too large a timestep causes instability and inaccuracy in the integration process.

When is this needed? You need to read this section only if the default timestep or run length is inadequate to your needs.

You should already know... Forcefield-Based Simulations, Molecular Dynamics, contains full information on the various integration algorithms, choosing an appropriate timestep, and the effect of timestep on integration errors.

Setting the timestep and run duration You can set the Total time (in ps), the number of Steps, and/or the length of the Time step (in fs) for both the equilibration and data-collection stages of the dynamics run in the Discover Dynamics control panel (accessed as described under Accessing the tools). (Setting any two of these values determines the third value.)

Defining the thermodynamic ensembles

Simply integrating Newton’s equations of motion allows you to explore the constant-energy surface (NVE dynamics) of a system. However, most natural phenomena occur under conditions where a system is exposed to external pressure and/or exchanges heat with the environment. These conditions can also be simulated.

When is this needed? If you want the default conditions, you do not need to read this section. However, you should read this section if you want to examine and/or change the default settings.

Related information Control of temperature and of pressure or stress are covered below (Temperature-control parameters).

You should already know... Information on thermodynamics ensembles and equilibrium thermodynamic properties is contained in Forcefield-Based Simulations under Statistical ensembles.

Concepts The Cerius²•Discover module allows several thermodynamics ensembles to be simulated:

♦ Constant-volume/constant-energy dynamics (NVE)—The Newtonian equations of motion, which conserve the total energy, are used. The temperature is allowed to vary, but the energy is not allowed to vary beyond a set value between dynamics steps.
2. CDiscover within Cerius²

- Constant-volume/constant-temperature dynamics (NVT)—The dynamics is modified to allow the system to exchange heat with the environment at a controlled temperature. Several methods of scaling or controlling the temperature are available (see Controlling temperature, pressure, and stress).

- Constant-pressure/constant enthalpy (NPH)—For periodic systems. The size and possibly the shape of the unit cell are allowed to vary, while the temperature is also allowed to vary. The energy is not allowed to vary beyond a set value between dynamics steps. The enthalpy is a constant of the motion.

- Constant-pressure/constant-temperature dynamics (NPT)—For periodic systems. Similar to NPH, except that temperature is controlled. Several methods of scaling or controlling the temperature are available (see Controlling temperature, pressure, and stress).

Setting the constant conditions

Set the Ensemble pushbutton in the Discover Dynamics control panel (accessed as described under Accessing the tools) as desired. What ensembles are available depends on whether the current model is periodic or nonperiodic.

Then set other parameters that are associated with each ensemble (see below).

Controlling temperature, pressure, and stress

You can use the velocity scale method to control the kinetic temperature of a system to quickly bring it to equilibrium at some temperature. Alternatively, you can use the Nosé or Andersen methods to control the thermodynamic temperature to generate the correct statistical ensemble (so that probability of occurrence of a certain configuration obeys the laws of statistical mechanics).

When is this needed?

The required method of temperature control to use depends on the run conditions (what factors are kept constant, Defining the thermodynamic ensembles) and the purpose of the simulation.

Constant-pressure and -stress methods are relevant only for simulations of periodic systems. You must specifically define any stress and/or pressure that you want to apply. Otherwise none is applied.
Setting up the simulation conditions

You should already know...

Information on the Maxwell–Boltzmann relationship between atomic velocities and temperature, theory and application of methods of temperature control, the relaxation time, etc. is contained in *Forcefield-Based Simulations* under *How temperature is controlled*.

Information on pressure and stress and how they are controlled is contained in *Forcefield-Based Simulations* under *Pressure and stress*. This includes information on pressure, stress, the Parrinello-Rahman method of pressure and stress control, sign conventions, the mass-like factor $W$, etc.

How it works

With the velocity scale method, you keep the temperature within a given range of a *target temperature*. Atomic velocities are simply scaled (which is chemically nonrealistic and so not used for true temperature control, see below) when the average temperature drifts outside a specified *temperature window*.

For true temperature control, you keep the thermodynamic (not kinetic) temperature constant by, in effect, allowing the simulated system to exchange energy with a heat bath. Two methods are available: Nosé–Hoover (Hoover 1985) and Andersen (Andersen 1980), referred to in the Cerius$^2$•Discover interface as NOSE and ANDERSEN, respectively.

Technical notes

The velocity scale and Andersen methods lead to discontinuous trajectories and should *not* be used in velocity autocorrelation studies. The Nosé method implemented in CDiscover is the Nosé–Hoover variant, which gives trajectories with smooth velocity changes suitable for autocorrelation studies.

Accessing the tools

The Discover Dynamics and Equilibration Options control panels are accessed as described under *Accessing the tools*. Unless otherwise specified, all the controls mentioned in this section are found in the Discover Dynamics control panel. Many are also found in the similar Equilibration Options control panel (for use if you want conditions to differ in the equilibration and data-collection stages of the run).

To open the Discover Stress Constraints control panel, select the *Constraints/External Stress* menu item in the *DISCOVER* card.

Specifying the temperature control method

Set the *Thermostat* popup in the Discover Dynamics control panel (accessed as described under *Accessing the tools*) to the desired integrator: VELOCITY SCALE, NOSE, or ANDERSEN. Which inte-
2. CDiscover within Cerius

grator is accessible (if any) depends on what Ensemble (above) is chosen.

The controls that appear below the Thermostat popup depend on which integrator and thermostat are chosen.

**Temperature-control parameters**

For information on the controls affecting how temperature is maintained constant, please see *Forcefield-Based Simulations*.

For temperature scaling:

♦ To set the target temperature, enter a value in the Temperature entry box of the Discover Dynamics control panel.

♦ To use temperature scaling, set the Ensemble popup to NVT or NPT and set the Thermostat popup to VELOCITY SCALE.

♦ To change the range within which the scaled temperature is allowed to vary, edit the value in the Temp. Difference entry box.

For true temperature control:

♦ To set the temperature for the simulation, enter a value in the Temperature entry box of the Discover Dynamics control panel.

♦ To choose the method of temperature control, set the Ensemble popup to NVT or NPT and set the Thermostat popup to NOSE (Nosé–Hoover) or ANDERSEN.

♦ You may change the default values for parameters specific to your chosen method.

**Controlling pressure**

You can control the rate of change of the volume/shape matrix in constant-pressure dynamics (NPH or NPT) by specifying a Cell Mass. This indicates how strongly the system is coupled with the pressure or stress “bath”.

To specify an external hydrostatic pressure, enter a value in the Equivalent hydrostatic pressure entry box of the Discover Stress Constraints control panel.

**Controlling stress**

Specify the individual components of the stress in the entry boxes in the External Stress section of the Discover Stress Constraints control panel. These entry boxes indicate the normal (xx, yy, and zz) and shear (xy, xz, yz) stress tensors.
Setting up the simulation conditions

Since pressure is the negative isotropic component of the (normal) stress tensor, values entered in some of these entry boxes may change values that appear in other entry boxes.

To reset the stress and pressure to zero, click the Reset External Stress action button.

**Type of dynamics simulation**

Simulated annealing and other complex simulations can be set up by using the appending and editing facilities found on the Discover Input File control panel (see Saving, editing, and using input files).

**Stages of a dynamics run**

Both equilibration and data-collection stages are automatically incorporated into a single dynamics run. The equilibration stage must run for at least one step, since the assignment of velocities occurs during equilibration. Similarly, the data-collection stage must run for at least one step.

You may also include a preliminary minimization stage in the same run (that is, without needing to start a separate dynamics run from the output of the minimization run).

**You should already know...**

Information on stages in dynamics simulations is contained in Forcefield-Based Simulations under Equilibration stage and Has equilibrium been achieved?, as well as under Production (data-collection) stage and How long should the simulation be?.

**Accessing the tools**

Select the Run menu item from the DISCOVER card to open the Run Discover control panel. Set the Task popup to Dynamics. A More... pushbutton appears. Click the More... pushbutton to the right of the Task popup in the Run Discover control panel to open the Discover Dynamics control panel.

**Preliminary minimization**

To run minimization before dynamics, set up minimization (Minimization) and check the Pre-Minimize check box in the Discover Dynamics control panel. The More... pushbutton to the right of this check box also gives you access to the Discover Minimize control panel (Refining the minimization conditions).
2. CDiscover within Cerius²

Setting up and starting the run

This section includes information on:

- Specifying output
- Specifying how and where to send the job
- Starting the run

Specifying output

When is this needed?

If the default output specifications are sufficient to your needs, you do not need to read this section. However, you should read this section if you want to examine and/or change the default settings.

The desired output is specified before starting the calculation, since they cannot be efficiently calculated after the run is complete.

You should already know…

Information on trajectory files and their uses is contained in Force-field-Based Simulations under Dynamics trajectories.

Accessing the tools

Select the Run menu item on the DISCOVER card to open the Run Discover control panel.

To open the Discover Minimize Output control panel, set the Task popup to Minimization and click the Output... pushbutton in the Run Discover control panel.

To open the Discover Dynamics Output control panel, set the Task popup to Dynamics and click the Output... pushbutton in the Run Discover control panel.

Updating the model display

The model is always updated when a minimization ends successfully. The original structure is automatically saved as an .msi file before the minimization starts.

File output

For minimization runs, you can send output to archive and output files.

For dynamics runs, you can send output to trajectory, output, and table files. (The trajectory file type can be specified as archive or history.) These three filetypes are useful (respectively) for visualizing the consecutive history of the simulation, examining all the
output at each iteration, and analyzing the results with the Cerius²•Graphs module.

Controls in the Discover Minimize Output and Discover Dynamics Output control panels allow you to choose whether to output the different types of files and also to specify the kinds of results to include in an output file. For dynamics, you also specify how the results are averaged across sequential steps of the run.

Additional information Please see the on-screen help for details on the functioning of each control in the control panels mentioned in this section.

Related information Cerius² Modeling Environment contains complete documentation on working with tables in the C²•Tables module, in Working with Tables. In addition, some other modules (which are documented separately) include other tables utilities.

File formats File formats are documented in Files.


ceriuss

Specifying how and where to send the job

When is this needed? If you intend to run Discover on the same machine on which you are running Cerius² and do not care about monitoring the job while it runs or about transferring files, you do not need to read this section and may proceed to Starting the run.

This section includes information on:

Interactive vs. background or NQS run mode
Selecting a machine and base directory
Setting up NQS-mode jobs
Monitoring and controlling running jobs
Transferring files from one machine to another
Other file-control issues

Accessing the tools Select the Job Control menu item in the DISCOVER card to open the Discover Job Control control panel.

Additional information Please see the on-screen help for details on the functioning of each control in the Discover Job Control control panel.
2. CDISCOVER within CERIUS

Interactive vs. background or NQS run mode

Set the Run Mode popup to INTERACTIVE, BACKGROUND, or NQS to start your Discover run in interactive or background mode or via the Network Queueing System (if installed), respectively.

Interactive

In interactive mode, CERIUS displays the Discover output file so you can monitor the progress of the job. However, you cannot do anything else in the CERIUS interface until the job is complete, so this is useful only for jobs that you expect to finish quickly.

You can stop an interactive job by using the CERIUS Interrupt window, which is displayed while the job is running to indicate that CERIUS is busy. If you click the INTERRUPT button and select the Stop current process ASAP option, CERIUS displays an additional dialog box from which you can confirm or cancel your request or send the job into the background.

Background

In background mode, the job runs without communicating with the CERIUS interface, and you can quit CERIUS, allowing the job to run by itself. When you exit CERIUS, a status file is automatically saved so that, if CERIUS is restarted, the job can be selected for monitoring or (if it has finished) the trajectory files can be read in for analysis (see Preparing trajectories).

NQS

In NQS mode, the job is submitted to the Network Queueing System, provided that this software has been installed, on the local or remote host. Once the job is queued, you can quit CERIUS, allowing the job to run by itself. An NQS-mode job can be selected for monitoring or file transfer in a later CERIUS session.

Selecting a machine and base directory

By default, the Discover job runs on the machine on which you are running CERIUS. However, you may, for example, send it to a faster machine.

Other machines

Click the Hosts arrow in the Discover Job Control control panel to obtain a pulldown listing machines at your site to which you can send the Discover job. The list contains all suitable hosts on your network (as defined in applcomm.db—for a description and example file entries, see the CERIUS Installation and Administration Guide).
Setting up and starting the run

Select a host by clicking its name in the list. This also closes the list. You can also close the list by clicking the triangular icon again. The host selected is displayed in the entry box. You can also enter a machine name by typing it in the entry box.

Permission and password

Depending on your site’s setup, you might not have permission to run on all the machines listed in the pulldown, or you may need to specify a user ID and password to access some machines. Click the Options… pushbutton to access the Discover Job Control Options control panel, where you can input a User ID and Password before selecting your host machine.

Tip

If in doubt about any of the options relating to machines and network configuration at your site, please see your system administrator.

Base directory

You also need to specify a working (base) directory on the host machine if the default specification is not correct. For more information on how to do this, as well as whether files need to be transferred between machines in consequence, please see the on-screen help (click the right mouse button while the cursor is over the Base Directory entry box).

Setting up NQS-mode jobs

Setting Run Mode to NQS on the Discover Job Control control panel causes a More… pushbutton to appear. This button provides access to the NQS Control control panel, which allows you to set commonly used NQS flags (such as queue name and time and memory limits) and to set the names of the commands used to submit, monitor, and kill NQS jobs on the chosen host.

Other controls enable you to display your NQS jobs and the supported limits on the chosen host. The More… pushbutton on the NQS Control control panel gives access to the NQS Options control panel, which allows you to change less commonly used NQS job submission flags.

Please see the on-screen help for information about all the controls in these control panels.
2. CDiscover within Cerius²

**Monitoring and controlling running jobs**

The Cerius2 Discover Job Status list box in the Discover Job Control control panel shows the filenames and directories associated with your Discover runs. For each job, the list shows the host name, datafile prefix, status (started, running, or complete for interactive and background jobs, or NQS_SUB, NQS_RUN, or complete for NQS jobs), process or NQS-request ID, and working directory. Click the **UPDATE** pushbutton to update this list. To remove an item from the list (if that job has completed), click the **REMOVE** pushbutton.

You can also monitor or kill a job that is running or import the files from a remote machine to the machine on which you are running Cerius² after your job has completed:

- Selecting a job from the **Job Status** list and clicking the **Monitor Discover output file** action button opens a window showing the contents of the output file for that job. Because Discover jobs can run for long periods of time, the job list is maintained between Cerius² sessions via a status file (called .MSIcdiscover-stat and typically located in your home directory).

- To kill a queued or currently running job, select it in the **Job Status** list and click the **Kill job** action button.

**Important**

Remote hosts on which a Discover job is running must be authorized to make connections to your X-server in order to monitor the logfile output from jobs in this manner. To authorize such access, enter the following from a shell on your local system:

```
> xhost remote_host_name
```

**Transferring files from one machine to another**

If it is necessary to move output files from a remote machine to the run directory on your local machine (in some situations they are automatically returned at the end of a job), select the job in the **Job Status** list in the Discover Job Control control panel and click the **TRANSFER** pushbutton.
Setting up and starting the run

**Note**
The remote file system may actually be the same as your local file system (if, for example, it is NFS mounted the same way on both systems). In this case, no transfer of files is necessary.

**Other file-control issues**

**Accessing the tools**
Select the Run menu item on the DISCOVER card to open the Run Discover control panel.

**Filenames**
To change the default root name for files associated with your run, edit the contents of the File Prefix entry box in the Run Discover control panel. Alternatively, you can select a root name from existing datafiles using controls on the Discover Input File control panel (see below). We refer to this root (or “seed”) name as run_name in this documentation.

**Saving, editing, and using input files**
With the Discover Input File control panel, you can:

- Save your input file.
  
  Click the **Save Strategy to .inp File** action button to write the input file you have built up so far in your Cerius² session to disk. The filename will be run_name.inp.

- Merge two input files.
  
  Select a file by clicking its name in the list box, then click the **Append Strategy** action button to add the current strategy to an existing input file. You can, for example, use this to build up multistage simulations such as simulated annealing.

- Edit an existing (saved) input file.
  
  Select a file by clicking its name in the list box, then click the **Edit .inp File** action button to edit this file. This automatically calls up vi (or EMACS, if it is specified by the $EDITOR environment variable) to edit the file in a separate window. If you prefer some other editor, open your editor in the usual way, then edit and save the input file. Before using this functionality, you should of course ensure that no jobs with the specified file prefix are currently running.

**Setting up sophisticated jobs through Cerius²**
Cerius² allows you much flexibility in performing your studies. You could, for example, build your model and set up most of your input file via the Cerius² interface, save this file, then include extra Btcl statements and keywords that are available in
2. CDiscover within Cerius²

standalone but not through the Cerius² interface (Btcl Language and Commands—Standalone Mode) by editing this file and saving it from the editor, and finally start your run from that edited file via the Cerius² interface (see next paragraph) or by running Discover in standalone mode.

♦ Run an existing input file.

Select a file by clicking its name in the list box, then click one of the Run .inp file action buttons to start a Discover run from a previously saved input file. The job uses the current settings of the Discover Job Control control panel (host machine, run mode, etc., see Specifying how and where to send the job).

Additional information

Please see the on-screen help for details on the functioning of each control in the Run Discover and Discover Input File control panels.

Starting the run

When is this needed?

The minimum action required for performing a Discover simulation is to click the RUN pushbutton in the Run Discover control panel. Of course, any changes to default settings discussed previously in this chapter should be made before starting the run.

Accessing the tools

Select the Run menu item on the DISCOVER card to open the Run Discover control panel.

Starting the Discover run

To start the Discover job, click the RUN pushbutton in the Run Discover control panel.

Additional information

Please see the on-screen help for details on the functioning of each control in the Run Discover control panel.

Analyzing results

The C² Analysis module is used to analyze trajectory files generated by simulations (whether by Discover or some other simulation modules in Cerius²). Several types of analyses can be performed, depending on the run conditions and the type of data generated. These include statistical analysis of properties, calculation of diffusion constants from mean-square displacements, cal-
Analyzing results

calculation of the dipole–dipole autocorrelation function and power spectrum, and analysis of property fluctuations.

Velocity data can be analyzed to obtain the velocity autocorrelation function and power spectrum. Structural analysis can also be performed, where the radial distribution function is calculated and a Fourier transform is applied to obtain the structure factor.

**Tip**
If you need to retrieve the final structure from a Discover simulation, you can select the **Retrieve** menu item from the **DISCOVER** card to open the Retrieve Discover File control panel. Use the file browser controls to retrieve the desired file and its associated model structure.

*This section explains*

This section includes information on:

- Preparing trajectories
- Output from analysis
- Property statistics

### Table 2. Finding information about analyzing results

<table>
<thead>
<tr>
<th>If you want to know about</th>
<th>Read:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Finding the Analysis module.</td>
<td>Accessing the tools.</td>
</tr>
<tr>
<td>Specifying output.</td>
<td>Specifying output.</td>
</tr>
<tr>
<td>Loading a trajectory file.</td>
<td>Loading a trajectory file.</td>
</tr>
<tr>
<td>Format of trajectory file.</td>
<td>Files.</td>
</tr>
<tr>
<td>Analyzing part of a trajectory file.</td>
<td>Selecting frames from a loaded trajectory.</td>
</tr>
<tr>
<td>Graphing various properties.</td>
<td>Basic statistics of various properties.</td>
</tr>
<tr>
<td>Displacement analysis.</td>
<td>Mean-square displacement, Self-diffusion constant.</td>
</tr>
<tr>
<td>Velocity results.</td>
<td>Velocity autocorrelation, Power spectrum.</td>
</tr>
<tr>
<td>Structural distribution.</td>
<td>Radial distribution function, Structure factor.</td>
</tr>
<tr>
<td>Dipole properties and functions.</td>
<td>Dipole–dipole interactions.</td>
</tr>
<tr>
<td>Macroscopic response properties.</td>
<td>Fluctuation analysis.</td>
</tr>
<tr>
<td>Display of graphical and printed output from analyses.</td>
<td>Output as graphs and text.</td>
</tr>
</tbody>
</table>
2. CDiscover within Cerius²

**Table 2. Finding information about analyzing results**

<table>
<thead>
<tr>
<th>If you want to know about:</th>
<th>Read:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Animating models.</td>
<td>Displaying model conformations.</td>
</tr>
<tr>
<td>Displaying individual model conformations.</td>
<td>Displaying model conformations.</td>
</tr>
<tr>
<td>Converting trajectory data to tables.</td>
<td>Output in tables format.</td>
</tr>
</tbody>
</table>

*You should already know…*  
*Forcefield-Based Simulations, Molecular Dynamics*, contains information on the types of information that can be obtained from trajectory files output by dynamics simulations run under various conditions.

*Accessing the tools*  
To conveniently access the C²•Analysis module, click the deck selector in the main control panel and choose **DISCOVER** from the list that appears. Then click the title of the **ANALYSIS** card to bring it to the front. The deck of cards menu area should now look like this:

![Diagram of the deck of cards menu area]
Preparing trajectories

Data from simulations are stored in trajectory files, and these files are used for analyzing the results.

Finding information

This section includes information on:

- Loading or creating a trajectory
- Selecting frames from a loaded trajectory

You should already know...

Information on trajectories is contained in Forcefield-Based Simulations under Dynamics trajectories.

Loading or creating a trajectory

A trajectory file must be loaded before any of the results contained in it can be analyzed. Alternatively, you can use a collection of selected models as one set of configurations.

You should already know...

Using Cerius$^2$ file browser controls is explained in Cerius$^2$ Modeling Environment under Loading model structure files.

Accessing the tools

Select the Input menu item from the ANALYSIS card to access the Analysis Input control panel.

Loading a trajectory file

Use the browser box in the Analysis Input control panel to find a trajectory file.

- Discover trajectory file names end in .his, or .arc.
- To load the desired file, either double-click its name in the file list box or select its name and then click the SELECT pushbutton.

Using a set of models to create a “trajectory”

To use selected models as, in effect, a trajectory, select SELECTED MODELS under Choose Data Source in the Analysis Input control panel. It is possible, for some purposes (noted below where relevant), to analyze only one model; however, more reliable results are obtained when a set of conformations (from selected models or from a trajectory file) is analyzed.

Additional information

Please see the on-screen help for details on the functioning of each control in the Analysis Input control panel.

File formats

File formats are documented under Files.
2. CDiscover within Cerius²

**Selecting frames from a loaded trajectory**

You may want to analyze only part of a loaded trajectory file if, for example:

- The structure was not fully equilibrated for the first few steps saved in the file.
- You have a very long trajectory file and do not need to analyze all of it.
- You want to compare various portions of the trajectory with one another.
- You saved data more frequently than was needed for your present analytical needs.

**How it works**

First you load a trajectory file (which automatically informs you of the number of frames in the file, via the **Last** entry box in the Analysis Input control panel), and then you specify the frame or time values as desired.

**Accessing the tools**

Select the **Input** menu item from the ANALYSIS card to access the Analysis Input control panel.

**Selecting the frames**

First load the desired trajectory file (*Loading or creating a trajectory*). Then, if you want to analyze only some of the frames in the file, select those frames by changing the **First** frame to extract, the **Last** frame to extract, and/or the **Step** interval entry boxes in the Analysis Input control panel. You can change either the **Frame** or the equivalent **Time**, whichever is more convenient.

**Tip**

You can verify how many frames have been specified and their locations within the trajectory by clicking the up and down arrows between the **Current** entry boxes and comparing the contents of these entry boxes with those of the **Last** entry boxes.

For example, if you set **Step** to 10 for a trajectory whose **Last** step was 640, the maximum value appearing in the **Current Frame** box would be 64 and the **Current Time** for **Current Frame** number 32 would be a little less than half the **Time** for the **Last** step.

**Additional information**

Please see the on-screen help for details on the functioning of each control in the Analysis Input control panel.
Output from analysis

You can display a plot of your data, list the data in the text window, or save the data to a file formatted for display in a Cerius² table.

Finding information

This section includes information on:

- Displaying model conformations
- Output as graphs and text
- Output in tables format

You should already know...

Loading trajectories (Loading or creating a trajectory) and selecting frames (Selecting frames from a loaded trajectory) are covered above.

Displaying model conformations

Once a trajectory file is loaded (Loading or creating a trajectory) and a subset of its frames is extracted (if desired, Selecting frames from a loaded trajectory), you can display the model conformations of individual frames in the model window. In addition, you can animate the model, in effect displaying the trajectory frames like a video.

Related information

The C²•Visualizer has a more limited, independent utility for animating models with a trajectory file loaded within that utility. Please see Cerius² Modeling Environment under Animating models.

Accessing the tools

To access the Analysis Show Frames control panel, select the Show Frames menu item from the ANALYSIS card.

To access the Analysis Input control panel, select the Input menu item from the ANALYSIS card.

Displaying single conformations

To display the model in the conformation of a single frame in the loaded trajectory file, you can use the Current controls in either the Analysis Input or Analysis Show Frames control panel.

To display a single frame, enter a frame number in the left entry box or a time in the right entry box.

To step through frames until you get to the desired frame (or conformation), use the up and down arrows between the two entry boxes.
2. CDiscover within Cerius²

To display a conformation by picking a point from a trajectory graph (Output as graphs and text), assure that the **Pick frame from graph** tool in the Analysis Show Frames control panel is highlighted, then click any point on any graph that was output by a Profile calculation (see Calculations).

**Animating the model**

You can also drag the mouse in the graph window to step through conformations equivalent to the plotted points.

To animate the model at a constant speed and proceeding once through the trajectory frames, set the popup in the Analysis Show Frames control panel to **STOP AT END** and click the forward or reverse video-control arrow.

To animate the model continuously, set the popup in the Analysis Show Frames control panel to **REVERSE AT END** or **LOOP AT END** and click the forward or reverse video-control arrow.

To stop a continuous animation, click the square control that is located between the two video-control arrows.

**Information on the current frame**

To obtain information on the energy components, time step, and other properties of the model in an individual frame, select the frame by any of the methods described for displaying a single conformation (above). Click the **Information on current frame** action button in the Analysis Show Frames control panel.

The information is printed in the text window.

**Saving an individual conformation**

To save any single frame (model conformation), select the frame by any of the methods described for displaying a single conformation (above). Select the **File/Save Model...** menu item from the menu bar in the main control panel. Save the model with the Save Model control panel, as described in Cerius² Modeling Environment, under Saving model structure files.

**Additional information**

Please see the on-screen help for details on the functioning of each control in the Analysis Show Frames and Analysis Input control panels.

**Output as graphs and text**

By default, the output of analysis calculations (Property statistics) is displayed as a graph. However, you can send additional output to the text window and/or turn off the production of graphs.
Analyzing results

Related information
*Cerius*² Modeling Environment, *Working with Graphs*, gives information on managing graphs (*Managing graphs*) and controlling aspects of their display (*Displaying and editing graphs*). Printing graphs is discussed under *Printing models and graphs*.

Accessing the tools
Select the **Analyze/Statistics** menu item from the ANALYSIS card to access the Analysis Statistics control panel. Then click its **Output**… pushbutton to access the Analysis Output control panel.

Sending output to graphs and/or the text window
Check or uncheck the **Plot** check box to send output to a graph or not.
Check or uncheck the **Text Port** check box to send additional output to the text window or not. (Some information is always sent to the text window.)

Additional information
Please see the on-screen help for details on the functioning of each control in the Analysis Statistics and Analysis Output control panels.

Output in tables format
You can send the output of analysis statistics calculations to a tables file for later use in the C²•Tables module.

Related information
*Cerius*² Modeling Environment contains complete documentation on working with tables in the C²•Tables module, under *Working with Tables*. In addition, some other modules (which are documented separately) include other tables utilities.

Accessing the tools
Select the **Analyze/Statistics** menu item from the ANALYSIS card to access the Analysis Statistics control panel. Then click its **Output**… pushbutton to access the Analysis Output control panel.

Saving results as table files
To send the output to a file that can be viewed, analyzed, and manipulated with the C²•Tables module, check the **Output to Tables File** check box in the Analysis Output control panel.
Use the file browser controls to specify the directory in which to store the file, as well as the file’s name.
To not send output to a Cerius² table file, uncheck the **Output to Tables File** check box.

Additional information
Please see the on-screen help for details on the functioning of each control in the Analysis Output control panel.
2. CDiscover within Cerius²

Property statistics

The available statistical analysis functions can be used to calculate and display plots of several properties using data from a trajectory file. The properties include total energies, individual energy components, temperature, radius of gyration, stress, pressure, volume, and cell parameters. Plots can be created showing the time-dependent profile of a selected property or the running or block averages. Calculated statistics can be displayed in the text window or written to an output file.

Finding information

This section includes information on:

- Basic statistics of various properties
- Mean-square displacement
- Self-diffusion constant
- Velocity autocorrelation
- Power spectrum
- Radial distribution function
- Structure factor
- Dipole–dipole interactions
- Fluctuation analysis

Related information

Cerius² Modeling Environment, Working with Graphs, gives information on managing graphs (Managing graphs) and controlling aspects of their display (Displaying and editing graphs). Printing graphs is discussed on Printing models and graphs of that book.

Forcefield-Based Simulations, Molecular Dynamics, contains full information on the types of information that can be obtained from trajectory files output by dynamics simulations run under various conditions.

Basic statistics of various properties

After loading a trajectory and (optionally) specifying frames and nondefault output, you can graph various properties that were output from the simulation run in several types of graphs.
Analyzing results

You should already know...

Loading trajectories (Loading or creating a trajectory), selecting frames (Selecting frames from a loaded trajectory), and specifying output (Output from analysis) are covered above.

Technical notes

The exact properties listed in the Select Properties list box in the Analysis Statistics control panel depend on the contents of the loaded trajectory.

Accessing the tools

Select the Analyze/Statistics menu item from the ANALYSIS card to access the Analysis Statistics control panel.

Selecting properties

To select a property to analyze, highlight its name in the Select Properties list box in the Analysis Statistics control panel by clicking it.

To select more than one property at once, <Shift>-click to select several adjacent properties or <Ctrl>-click to select properties whose names are not continuous.

To deselect a highlighted property, <Ctrl>-click it. To deselect all highlighted properties and select another one, simply click the desired property.

Properties listed within angle brackets (e.g., <Dipole Magnitude>) cannot be selected or plotted until they have been calculated.

To select measurements, you need to first create them. To do this, select the Geometry/Measurements... menu item from the menu bar in the main Visualizer control panel to access the Measurements control panel. Create measurements as described in Cerius² Modeling Environment, under Measuring models.

Calculations

For the selected property or properties, you can use the Analysis Statistics control panel to calculate:

♦ A time-dependent profile (i.e., a history) of the property values, by clicking the Profile action button.

The value of the property is plotted vs. time for trajectories that contain time information (such as those output by dynamics simulations) and vs. frame number for trajectories that do not contain time information (such as those output by minimization runs).

♦ A histogram showing the frequency distribution of the property, by clicking the Distribution statistics action button.
2. CDiscover within Cerius

You can change the number of bins into which the histogram is divided or request that output be sent only to the text window (several types of information are always sent to the text window) with the Distribution Preferences control panel. Access this control panel by clicking the Preferences... pushbutton to the right of the Distribution statistics action button in the Analysis Statistics control panel.

- The running or block average of the property, by clicking the Running Average or Block Average action button.

The average is plotted vs. time for trajectories that contain time information (such as those output by dynamics simulations) and vs. frame number for trajectories that do not contain time information (such as those output by minimization runs).

To change the width or interval used in block averaging, click the Preferences... button to the right of the Block Average action button to access the Block Average Preferences control panel.

Searching for values

To search the trajectory file for a number of frames containing the highest or lowest values of a selected property, use the Search for controls in the Analysis Statistics control panel.

The results are printed in the text window

Additional information

Please see the on-screen help for details on the functioning of each control in the Analysis Statistics, Distribution Preferences, and Block Average Preferences control panels.

Mean-square displacement

Cerius² displacement analysis functions can be used to determine the self-diffusion constant of a model.

You should already know...

Loading trajectories (Loading or creating a trajectory), selecting frames (Selecting frames from a loaded trajectory), and specifying output (Output from analysis) are covered above.

Concepts

The mean-square displacement (MSD) is calculated as follows:

\[
\text{msd}(m) = \langle |r(t) - r|^2 \rangle = \frac{1}{n} \sum_{i=1}^{n} |r(m+i) - r(i)|^2
\]  

Eq. 1
Analyzing results

Where:

\[ 0 < m + n = k \]

- \(m\)  Maximum number of points allowed for the MSD calculation.
- \(n\)   Number of data points used for averaging.
- \(i\)   Step counter (increment).
- \(k\)   Total number of snapshots read in.

How it works

The value for \(k\) is set when specifying how many frames to analyze (Selecting frames from a loaded trajectory). The default value for \(m\) is \(k/2\), but this, as well as the step counter \(i\), can be changed using the options on the Mean Squared Displacement control panel (see below).

You can calculate the MSD for all or selected atoms in the model. A plot of the MSD vs. time is displayed and updated as the calculation proceeds.

Accessing the tools

Select the Analyze/MSD menu item from the ANALYSIS card to access the Mean Squared Displacement control panel.

Prerequisites

You need to load a trajectory (Loading or creating a trajectory) and select what frames to use (Selecting frames from a loaded trajectory) in the MSD calculation.

Setting up an MSD calculation

Specify whether the calculation should include all or only selected atoms by setting the Atoms popup in the Mean Squared Displacement control panel.

To specify the values of parameters in Eq. 1, set these controls:

- \(m\)    Max. Time (as number of frames or in picoseconds).
- \(i\)    Origin Step (as number of frames or in picoseconds).

Anisotropic components

To perform the MSD calculations assuming the system is anisotropic, check the All Anisotropic Components check box. Six MSD plots (one for each of the anisotropic components) are calculated and displayed. The isotropic MSD plot is also calculated and displayed.

Graph output

To display a plot that is continuously updated as the MSD calculation proceeds, check the Monitor Calculation check box.

To display the finished plot only at the end of the MSD calculation, uncheck the Monitor Calculation check box.
Performing the MSD calculation

To start the MSD calculation, click the Calculate MSD action button in the Mean Squared Displacement control panel.

Additional information

Please see the on-screen help for details on the functioning of each control in the Mean Squared Displacement control panel.

Self-diffusion constant

The trajectory file data generated from simulations is used to calculate mean-square displacement (MSD) as a function of the position of each diffusing particle.

You should already know...

Loading trajectories (Loading or creating a trajectory), selecting frames (Selecting frames from a loaded trajectory), specifying output (Output from analysis), and calculating the MSD (Mean-square displacement) are covered above.

Concepts

The self-diffusion constant is obtained using the Einstein relation:

\[ D = \frac{1}{6Nt} \langle |r(t) - r|^2 \rangle \]  

Eq. 2

Where:

- \( N \) Number of atoms.
- \( r \) Position of the particle.
- \( t \) Time.

The expression within the braces is the MSD. The self-diffusion constant \( (D) \) is obtained by calculating the slope of MSD vs. time. This relation applies to periodic models only.

How it works

A least-squares linear fit of the MSD can be done, and the self-diffusion constant can be determined from the slope of this line.

Often a better linear fit can be produced by discarding the initial quadratic segment of the MSD curve.

Technical notes

The linear fit is always done using the most recently calculated MSD. When all anisotropic components are calculated and displayed, this corresponds to the last plot displayed.

Accessing the tools

Select the Analyze/MSD menu item from the ANALYSIS card to access the Mean Squared Displacement control panel.
Calculating the self-diffusion constant

After calculating the MSD (Mean-square displacement), examine the graph of MSD vs. time to determine whether the entire curve or a segment of it is to be used for linear fitting.

To specify that only part of the MSD curve by used in the self-diffusion calculation, change the values for the beginning and end of the included range in the MSD fit range entry boxes in the Mean Squared Displacement control panel. Otherwise, the entire MSD-vs.-time plot is used.

Start the calculation by clicking the Calculate Line Fit action button in the control panel.

The resulting diffusion constant \( D \) is displayed in the text window, in units of both \( \text{cm}^2 \text{s}^{-1} \text{atom}^{-1} \) and \( \text{Å}^2 \text{ps}^{-1} \text{atom}^{-1} \).

Additional information

Please see the on-screen help for details on the functioning of each control in the Mean Squared Displacement control panel.

Velocity autocorrelation

The velocity autocorrelation function (VACF) can be determined from trajectory file data.

This type of information is useful in predicting the dynamic and vibrational properties of materials at elevated temperatures and in predicting thermal effects on infrared or Raman spectra.

You should already know...

Loading trajectories (Loading or creating a trajectory), selecting frames (Selecting frames from a loaded trajectory), and specifying output (Output from analysis) are covered above.

Concepts

The autocorrelation function \( C \) is defined as follows:

\[
C(m) = \frac{1}{n} \sum_{i}^{n} V(m+i) \times V(i)
\]

Eq. 3

Where:

\( 0 < m + n = k \)

\( m \) Maximum number of points allowed for the autocorrelation function calculation.

\( n \) Number of data points used for averaging.

\( i \) Step counter (increment).
2. C Discover within Cerius²

Velocity.

$k$  Total number of snapshots read in.

**How it works**

The value for $k$ is set when specifying how many frames to analyze (Selecting frames from a loaded trajectory). The default value for $m$ is $k/2$, but this, as well as the step counter $i$, can be changed using the options on the Velocity Autocorrelation control panel (see below).

You can calculate the VACF for all or selected atoms in the model. A plot of $C$ vs. time is displayed and updated as the calculation proceeds.

**Accessing the tools**

Select the Analyze/VACF menu item from the ANALYSIS card to access the Velocity Autocorrelation control panel.

**Prerequisites**

You need to load a trajectory (Loading or creating a trajectory) and select what frames to use (Selecting frames from a loaded trajectory) in the VACF calculation.

**Setting up a VACF calculation**

Specify whether the calculation should include all or only selected atoms by setting the **Atoms** popup in the Velocity Autocorrelation control panel.

To specify the values of parameters in Eq. 3, set these controls:

$m$  Max. Time (as number of frames or in picoseconds).

$i$  Origin Step (as number of frames or in picoseconds).

**Anisotropic components**

To perform the VACF calculations assuming the system is anisotropic, check the All Anisotropic Components check box. Six VACF plots (one for each of the anisotropic components) are calculated and displayed. The isotropic VACF plot is also calculated and displayed.

**Graph output**

To display a plot that is continuously updated as the VACF calculation proceeds, check the Monitor Calculation check box.

To display the finished plot only at the end of the VACF calculation, uncheck the Monitor Calculation check box.

**Performing the VACF calculation**

To start the VACF calculation, click the Calculate VACF action button in the Velocity Autocorrelation control panel.

**Additional information**

Please see the on-screen help for details on the functioning of each control in the Velocity Autocorrelation control panel.
Power spectrum

A Fourier transform of a VACF (Velocity autocorrelation) can be performed and the power spectrum can be calculated and displayed.

This type of information is useful in predicting the dynamic and vibrational properties of materials at elevated temperatures and in predicting thermal effects on infrared or Raman spectra.

You should already know...

Loading trajectories (Loading or creating a trajectory), selecting frames (Selecting frames from a loaded trajectory), specifying output (Output from analysis), and calculating the VACF (Velocity autocorrelation) are covered above.

Concepts

The frequency resolution of the power spectrum, \( \Delta \omega \), is given by:

\[
\Delta \omega = \frac{\pi}{t_{\text{max}}}
\]

Where:

\( t_{\text{max}} \) Time length of the analyzed data.

How it works

Because the time interval between two successive frames \( \Delta t \) must be fixed for a velocity autocorrelation calculation, the maximum frequency point in the power spectrum curve, \( f_{\text{max}} \), is also fixed, as seen by the following relationship:

\[
f_{\text{max}} = \frac{\pi}{\Delta t}
\]

Thus, decreasing the value of \( \Delta \omega \) makes \( t_{\text{max}} \) larger and effectively increases the resolution of the power spectrum curve.

Decreasing \( t_{\text{max}} \) results in a less detailed curve. Options are available that allow you to specify values for both \( \Delta \omega \) and \( t_{\text{max}} \).

Technical notes

The fast version of the Fourier transform is used.

The Fourier transform is always done using the most recently calculated set of VACF data. When all anisotropic components are calculated and displayed, this corresponds to the last plot displayed.

Accessing the tools

Select the Analyze/VACF menu item from the ANALYSIS card to access the Velocity Autocorrelation control panel.
2. CDiscover within Cerius²

**Calculating the power spectrum**

After calculating the VACF *(Velocity autocorrelation)*, specify the **Frequency Interval** (Δω in Eq. 4) and **Maximum Time** (t_{\text{max}} in Eq. 4) allowed, in the **Calculate Power Spectrum** section of the Velocity Autocorrelation control panel.

Start the calculation by clicking the **Calculate Power Spectrum** action button in the control panel.

A graph of the power spectrum is displayed.

**Additional information**

Please see the on-screen help for details on the functioning of each control in the Velocity Autocorrelation control panel.

**Radial distribution function**

The radial distribution function (RDF) can be determined from trajectory file data. This is the spherically averaged distribution of interatomic vector lengths.

This type of information is useful in revealing overall structural properties such as packing, ordering, compressibility, and phase transitions.

**You should already know...**

Loading trajectories *(Loading or creating a trajectory)*, selecting frames *(Selecting frames from a loaded trajectory)*, and specifying output *(Output from analysis)* are covered above.

**Concepts**

Because the system is assumed to be isotropic, the radial distribution function is centrosymmetric. The following expression is used to calculate the radial distribution function G_{AB}(r) between two selected groups, A and B:

\[
G_{AB}(r) = \frac{g_{AB}(r) \times V}{(N_A N_B - N_{AB}) 4\pi r^2 \Delta r}
\]  

Eq. 6

Where:

- N_A Number of atoms in group A.
- N_B Number of atoms in group B.
- N_{AB} Number of atoms common to both groups A and B.
- V Unit cell volume (for nonperiodic systems V = 1).

Groups A and B are specified by selecting the atoms in the model or by specifying particular elements or forcefield types. Groups A...
and B are the same for all or selected atoms but may be the same or different when elements or forcefield atom types are used.

You can also specify that all atoms be included in the calculation (this is the default). In this case, \( N_A = N_B = N \), where \( N \) is the total number of atoms in the model (nonperiodic systems) or in the unit cell (periodic systems). Eq. 6 then becomes:

\[
G(r) = \frac{g(r)}{N^2 - N} \frac{N}{V} 4\pi r^2 \Delta r
\]

The unnormalized distribution function \( N_{AB}(r) \) is defined as follows:

\[
g_{AB}(r) = \left( \sum \delta(r - |r_{AI} - r_{BJ}|) \right)
\]

Where:

- \( r_{AI} \) Position of the \( i \)th atom in group A.
- \( r_{BJ} \) Position of the \( j \)th atom in group B.

The distance interval \( dr \) from Eq. 6 corresponds to the bin width on the \( G_{AB}(r) \) curve:

\[
\Delta r = \frac{r_{cutoff}}{N_{bins}}
\]

Where:

- \( N_{bins} \) Number of bins.
- \( r_{cutoff} \) Cutoff distance.

Atom pairs beyond this distance are not included in the calculations.

You can specify values for \( dr \) and \( r_{cutoff} \) using the Distance Interval and Cut-Off Distance controls in the RDF control panel. For periodic systems, the computation time is proportional to \( r_{cutoff}^2 \). Thus, increasing the cutoff distance significantly increases the time required to calculate \( G_{AB}(r) \). This does not apply to nonperiodic systems, for which a value large enough to include all possible pair
2. *C*Discover within *C*erius²

Interactions should be chosen to assure accurate calculation of the structure factor.

For a given cutoff distance, decreasing the value of the **Distance Interval** results in a larger number of bins and effectively increases the resolution of the curve. If the value is too small, however, the curve can become noisy.

**Technical notes**

Calculations can be done using trajectory file data, a set of conformations (**Using a set of models to create a “trajectory”**), or the coordinates of the current model. Because the use of trajectory file data allows averaging over multiple frames, the results obtained are statistically more reliable than those obtained from the single frame data of the current model.

**Accessing the tools**

Select the **Analyze/RDF** menu item from the **ANALYSIS** card to access the RDF control panel.

**Prerequisites**

Unless you want to use the coordinates of the current model or a set of models (**Using a set of models to create a “trajectory”**), you need to load a trajectory (**Loading or creating a trajectory**) and select what frames to use (**Selecting frames from a loaded trajectory**) in the RDF calculation.

**Setting up an RDF calculation**

Specify whether the calculation should include all atoms, only selected atoms, or only certain atoms by setting the popup in the RDF control panel.

If you set the popup to **ELEMENTS** or **FF TYPES**, then specify the elements or forcefield atom types to use for groups A and B (Eq. 6), using the two popups that appear.

To specify the values of parameters in Eq. 9, set these controls:

\[
\begin{align*}
    r_{\text{cutoff}} & \quad \text{Cut-Off Distance (Å).} \\
    \Delta r & \quad \text{Distance Interval (Å).}
\end{align*}
\]

**Plot display**

To display a plot that is continuously updated as the RDF calculation proceeds, check the **Monitor Calculation** check box.

To display the finished plot only at the end of the RDF calculation, uncheck the **Monitor Calculation** check box.

**Performing the RDF calculation**

To start the RDF calculation, click the **Calculate g(r)** action button in the RDF control panel.

**Additional information**

Please see the on-screen help for details on the functioning of each control in the RDF control panel.
Structure factor

A Fourier transform of an RDF (*Radial distribution function*) can be performed to obtain the structure factor for a model. This type of information is useful in revealing overall structural properties such as packing, ordering, compressibility, and phase transitions. The structure factor is particularly useful in that it can be directly compared with X-ray diffraction data.

You should already know...

Loading trajectories (*Loading or creating a trajectory*), selecting frames (*Selecting frames from a loaded trajectory*), specifying output (*Output from analysis*), and calculating the RDF (*Radial distribution function*) are covered above.

Concepts

The structure factor, $S_{AB}(k)$ between two selected groups, A and B, is related to the radial distribution function $G_{AB}(r)$ through a 3D Fourier transform:

$$S_{AB}(k) = \frac{N_{AB}}{N} + \frac{(N_A \times N_B) - N_{AB}}{V \times N} \times \int dr G_{AB}(r) e^{ik \cdot r} \tag{Eq. 10}$$

Where:

- $k$ Wave vector.

All the other variables are as defined previously (see Eq. 6 and Eq. 7). When $N_A = N_B$, this equation becomes:

$$S_{AB}(k) = \frac{N_A}{N} + \frac{N_A^2 - N_A}{V \times N} \times \int dr G_{AB}(r) e^{ik \cdot r} \tag{Eq. 11}$$

When including all atoms in the calculations, $N_A = N_B = N$ and the above equation becomes:

$$S_{AB}(k) = 1 + \rho \left(1 - \frac{1}{N}\right) \times \int dr G_{AB}(r) e^{ik \cdot r} \tag{Eq. 12}$$

Where:

- $\rho$ Atomic density, $N/V$.

Eq. 12 is accurate for nonperiodic systems because $G_{AB}(r) = 0$ when $r \to \infty$, and $\rho = N/V = N$. 

---
For periodic systems, however, $G_{AB}(r) = 1$ when $r \to \infty$. Therefore, Eq. 12 should be rewritten as follows:

$$S_{AB}(k) = 1 + \rho \left(1 - \frac{1}{N}\right) \delta(k) + \rho \left(1 - \frac{1}{N}\right) \times \int dr \left[G_{AB}(r) - 1\right] e^{i k \cdot r} \quad \text{Eq. 13}$$

Because the delta function $\delta(k) = 0$ when $k \neq 0$, it can be removed analytically, resulting in the following:

$$S_{AB}(k) = 1 + \rho \left(1 - \frac{1}{N}\right) \times \int dr \left[G_{AB}(r) - 1\right] e^{i k \cdot r} \quad \text{Eq. 15}$$

**k-space interval and maximum distance**

Two controls are available in the RDF control panel that can be used to alter the resolution of the $S_{AB}(k)$ curve: **k-Space Interval** and **Maximum Distance**. The **k-Space Interval** specifies the value for the $k$-space interval $\Delta k$, which corresponds to the bin width in the $S_{AB}(k)$ curve:

$$\Delta k = \frac{\pi}{M_{\text{bins}} \Delta r} \quad \text{Eq. 16}$$

Where:

- $\Delta r$ = Distance interval used in the $G_{AB}(r)$ calculations.
- $M_{\text{bins}}$ = Number of bins in the $S_{AB}(k)$ curve.

For a given distance interval, decreasing the value of $\Delta k$ results in a greater number of bins and effectively increases the resolution of the $S_{AB}(k)$ curve.

The **Maximum Distance** specifies the cutoff distance $r_{\text{max}}$ used in calculating the structure factor:

$$r_{\text{max}} = M_{\text{bins}} \Delta r \quad \text{Eq. 17}$$

For a given distance interval, increasing the maximum distance also results in a greater number of bins, leading to higher resolution of the $S_{AB}(k)$ curve.

The Fourier transform is always done using the most recently calculated set of RDF data.
Accessing the tools

Select the Analyze/RDF menu item from the ANALYSIS card to access the RDF control panel.

Calculating the structure factor

After calculating the RDF (Radial distribution function), specify the k-Space Interval ($dk$ in Eq. 16) and the Maximum Distance ($r_{\text{max}}$ in Eq. 17) allowed, in the RDF control panel.

Start the calculation by clicking the Calculate Structure Factor action button in the control panel.

A graph of structure factor vs. $k$ is displayed.

Additional information

Please see the on-screen help for details on the functioning of each control in the RDF control panel.

Dipole–dipole interactions

The dipole vector and dipole–dipole correlation function can be calculated from trajectories. A Fourier transform of the correlation function can then be done to obtain the dipole–dipole power spectrum.

The power spectrum of the dipole correlation function can be related to the infrared spectrum of the system you are studying.

You should already know...

Loading trajectories (Loading or creating a trajectory), selecting frames (Selecting frames from a loaded trajectory), and specifying output (Output from analysis) are covered above.

Concepts

The dipole moment is defined as the magnitude of the dipole vector. It is origin-independent only if there is no net charge on the system. By convention, the origin used for the dipole moment is halfway between the centers of the positive and negative charge.

For a collection of point charges, the dipole vector is defined as:

$$\text{dipole} = 4.802Q_{\text{min}}(Q_{\text{COC}}^\text{positive} - Q_{\text{COC}}^\text{negative})$$  \hspace{1cm} \text{Eq. 18}

Where:

- 4.802 Factor necessary to convert the value to Debyes from angstroms and atomic changes.
- $Q_{\text{min}}$ The smaller absolute value of the total positive charge and the total negative charge.
- $Q_{\text{COC}}^z$ Center of charge (positive or negative) calculated by:
2. CDiscover within Cerius²

\[ Q^{\pm}_{\text{COC}} = \sum_{i=1}^{N} \frac{(q_i r_{i,a})}{\sum q_i} \]

Eq. 19

Where the sums run over all positive or negative charges as appropriate, and:

- \( q_i \) charge on atom \( i \).
- \( r_{i,a} \) \( x, y, \) or \( z \) component of the coordinates of charge \( i \).

How it works

Calculation of the dipole autocorrelation and power spectrum is analogous to similar calculations using velocities (Velocity autocorrelation and Power spectrum).

Technical notes

Calculations can be done using trajectory file data, a set of models (Using a set of models to create a "trajectory"), or the coordinates of the current model. Because the use of trajectory file data allows averaging over multiple frames, the results obtained are statistically more reliable than those obtained from the single frame data of the current model.

The fast version of the Fourier transform is used for calculating the dipole–dipole power spectrum.

Accessing the tools

Select the Analyze/Dipole menu item from the ANALYSIS card to access the Dipole control panel.

Prerequisites

Unless you want to use a set of models (Using a set of models to create a "trajectory") or the coordinates of the current model, you need to load a trajectory (Loading or creating a trajectory) and select what frames to use (Selecting frames from a loaded trajectory) in the calculation of the dipole vector and moment (magnitude).

Calculating the dipole–dipole correlation function requires that the dipole vector and moment have been calculated.

Calculating a dipole–dipole power spectrum requires that the dipole–dipole correlation function has been calculated.

Dipole vector and dipole moment

Specify whether the calculation should include all or only selected atoms by setting the atoms popup in the Dipole control panel.

Click the Calculate Dipole action button to perform the calculation.
Analyzing results

Once the dipole vector and moment have been calculated, their values can be plotted with the commands in the Analysis Statistics control panel (Basic statistics of various properties).

Dipole–dipole correlation function

First calculate the dipole vector and moment (above), and make sure the trajectory contains constant time steps.

To specify the values of parameters in Eq. 3, set these controls:

\[
\begin{align*}
m & \quad \text{Max. Time (as number of frames or in picoseconds).} \\
i & \quad \text{Origin Step (as number of frames or in picoseconds).}
\end{align*}
\]

To calculate the dipole–dipole correlation function and power spectrum assuming the system is anisotropic, check the All Anisotropic Components check box. Six plots (one for each of the anisotropic components) are calculated and displayed. The isotropic average plot is also calculated and displayed.

To display a plot that is continuously updated as the calculation of the dipole–dipole correlation function proceeds, check the Monitor Calculation check box.

To display the finished plot only at the end of the calculation, uncheck the Monitor Calculation check box.

To start the calculation, click the Calculate Dipole Autocorrelation action button in the Dipole control panel.

Dipole–dipole power spectrum

First calculate the dipole–dipole correlation function (above).

Specify the Frequency Interval (\(\Delta \omega\) in Eq. 4) and Maximum Time (\(t_{\text{max}}\) in Eq. 4) allowed, in the Dipole–Dipole Power Spectrum section of the Dipole control panel.

Start the calculation by clicking the Calculate Power Spectrum action button in the control panel.

A graph of the power spectrum is displayed.

Additional information

Please see the on-screen help for details on the functioning of each control in the Dipole control panel.

Fluctuation analysis

Statistical mechanics allows fluctuations in various properties to be related to macroscopic response properties.
2. CDiscover within Cerius²

**You should already know…**

Loading trajectories (Loading or creating a trajectory), selecting frames (Selecting frames from a loaded trajectory), and specifying output (Output from analysis) are covered above.

**Thermodynamic properties**

You can calculate the fluctuations in the following properties:

- Adiabatic compressibility — Response of system volume to a reversible change in pressure.
- Isothermal compressibility — Response of system volume to changes in pressure at constant temperature.
- Thermal pressure coefficient — Response of pressure to changes in temperature at constant volume.
- Gruneisen parameter — Response of pressure to changes in internal energy at constant volume.
- Isobaric heat capacity — Response of internal energy to changes in temperature at constant volume.
- Isometric heat capacity — Response of internal energy to changes in temperature at constant pressure.
- Volume expansivity — Change in volume, per-unit volume, with temperature at a constant pressure.
- Isoenthalpic Joule–Thomson coefficient — Change in pressure with temperature at constant enthalpy.
- Isothermal Joule–Thomson coefficient — Change in enthalpy with pressure at a constant temperature.
- Velocity of sound — The speed of sound waves in the system.

**Electrostatic properties**

Dielectric constant — How well the system screens the interaction between charged sites within it. Calculated using fluctuations in the dipole moment, which must be calculated first (Dipole–dipole interactions). The calculation uses the Kirkwood–Frohlich equation:

\[
(2e + 1)(e - 1) = \left( \frac{4\pi}{3} \right) \frac{M^2}{3VkT}
\]

Eq. 20

Where:

- \(e\) Dielectric constant.
- \(M\) Dipole moment of sample.
Analyzing results

\[ V \quad \text{Volume of sample.} \\
T \quad \text{Temperature.} \]

The calculation is valid only for 3D-periodic systems.

**Technical notes**

Some trajectory files do not contain enough information to generate all possible fluctuation properties. The software automatically lists only the properties that can be calculated.

**Accessing the tools**

Select the Analyze/Fluctuation Properties menu item from the ANALYSIS card to access the Fluctuations control panel.

**Prerequisites**

You need to load a trajectory (Loading or creating a trajectory) and select what frames to use (Selecting frames from a loaded trajectory) in the fluctuation calculations.

**Selecting properties**

To select a property to analyze, highlight its name in the Select Properties list box in the Fluctuations control panel by clicking it.

To select more than one property at once, <Shift>-click to select several adjacent properties or <Ctrl>-click to select properties whose names are not continuous.

To deselect a highlighted property, <Ctrl> click it. To deselect all highlighted properties and select another one, simply click the desired property.

**Calculations**

For the selected property or properties, you can use the Fluctuations control panel to calculate:

- The value of the property, by clicking the Calculate action button.
  The value of the property is printed in the text window.

- The running or block average of the property, by clicking the Running Average or Block Average action button.
  The average is plotted vs. time.

  To change the width or interval used in block averaging, click the Preferences... button to the right of the Block Average action button to access the Block Average Preferences control panel.

**Additional information**

Please see the on-screen help for details on the functioning of each control in the Fluctuations control panel.
2. C Discover within Cerius$^2$
3 CDiscover within Insight

General procedure

"Stages" in the Discover_3 module

The Discover_3 module is used to set up and then start a CDiscover calculation. Unlike many commands in the Insight interface, nothing actually happens until the Discover job is started, making it possible to set up complex calculations and even edit or change parts of the calculation before the job is started. To help accomplish this, CDiscover uses the concept of stages, where a stage is a distinct portion of the Discover calculation. For example, a calculation might consist of a minimization stage, followed by a dynamics stage designed to equilibrate the system, followed by another dynamics stage designed to collect thermodynamic information. Output and flow-control stages can be interspersed within the sequence of stages. The menus allow control over almost every detail of the calculation, stage by stage. The defaults for parameters in a stage are the values from the previous stage of the appropriate type. For instance, if you change the nonbond cutoffs as the first stage, then all subsequent stages use the modified cutoffs until you explicitly change them to a different value.

Types of pulldowns

The pulldowns in the Discover_3 module can be grouped in three classes:

♦ **Setup** is used to select the model, to delete or list the stages in a calculation, and to specify whether other parameter blocks will be shown in their expert or nonexpert mode.

♦ **Specify, Calculate, Language_Control, Analyze, and Pseudo_Atom** are used repetitively to build up the command input file that specifies the Discover calculation, by defining each stage of the calculation. **Strategy** enables you to use prerecorded calculations.
3. CDiscover within Insight

- **Background_Job** and **D_Run** are used to specify run conditions and start the Discover job, as well as to examine error messages for failed runs.

*Very general strategy of setting up a calculation*

Setting up and starting a calculation requires using at least one pulldown from each of the three classes:

1. The **Setup/System** command to select the object or assembly on which to run the calculation.

2. One of the commands from the **Calculate** or **Strategy** pull-downs to create a stage in the calculation.

3. The **D_Run/Run** command to start the background Discover job.

*More complex calculation*

More complicated calculations can be set up by editing the stage created in the second step, by repeating the second step to create more stages, or by using the commands in the **Language_Control** pulldown to control the flow of the calculation. The first and third steps need to be performed only once in setting up any calculation. Thus, a complete description of the setup of any calculation is:

1. Use the **Setup/System** command to select the object or assembly on which to run the calculation.

2. Repeat the following steps as many times as needed and in the order needed to completely set up the calculation:
   - a. Use the commands of the **Specify** pulldown to modify the details of how the forcefield is used in the calculation.
   - b. Use one of the commands from the **Calculate** pulldown to create the next stage of the calculation.
   - c. Use the **Analyze/Output** command to tailor the output for this stage, if needed.
   - d. Use the commands in the **Language_Control** pulldown to further refine the progress of the run, if needed.
   - e. Repeat Steps 2a–d as needed until the calculation is completely specified (see below for additional details).

3. Use the **Background_Job** pulldown if you want to specify non-default run conditions, then use the **Run** command of the **D_Run** pulldown to start the background job.
General procedure

Editing the calculation-setup

At any time before executing the D_Run/Run command, you can change any detail of the calculation that you are setting up. To do this in the context of the commands in most of the pulldowns, change the Activations parameter from the default of Add, which creates a new stage, to Insert, to insert a new stage between existing stages, or to Modify, which allows you to edit a stage. For Insert, you need to specify where to add a stage in the sequence of stages, by entering the name of the following stage in the Insert Before Stage parameter box. To Modify an existing stage, enter the name of the stage that you want to edit in the Stage Name parameter box. At this point, the parameter block is updated to reflect the values of the parameters for that stage, which may then be changed as desired and accepted by executing the command.

Specifying output

The Analyze/Output command does not set up a stage, but controls output from existing stages. Simply enter the name of the stage in the Stage Name parameter of the Output command and then specify the output you want from that stage. You can use this command at any time after some or all the stages are set up.

Deleting and listing stages

The Setup/Stage command can be used for deleting individual stages or clearing all stages. Reselecting the object or assembly with the Setup/System command deletes all previously defined stages. You can use the Setup/List command to examine your command input file, to be sure that it is what you expected.

Conditional access to parameters

Certain parameters in many commands are conditional; that is, your ability to modify them depends on the values of other parameters or on whether the specified system is periodic. Thus, only parameters that are relevant to the conditions of your calculation can be modified. Depending on whether the Expert level in the Setup/System command or the More button in several of the Discover_3 commands are toggled on, inaccessible parameters may be hidden from view.

Using variables

Some parameters whose values are integers or real numbers can also take variables of the appropriate type. For example, the counter in a loop could be an integer variable.

Analysis of results

Several icons and modules in the Insight interface (for example, the Graph and Contour icons, the Analysis module) facilitate graphical analysis of Discover results. These utilities are documented in the Insight User Guide.
3. C Discover within Insight

Setting up the ESFF forcefield

The ESFF forcefield can be used for molecular modeling of organic, inorganic, and organometallic systems in gas or condensed phases. It covers all the elements in the periodic table up to Rn. In the current release, however, infinitely bonded systems cannot be treated with ESFF, and the maximum number of atoms allowed is 10,000 per unit cell.

1. Like the other forcefields supported in the Discover program, ESFF can be selected within the Insight environment with the Forcefield/Select command.

2. For ESFF to work correctly it is very important that the bond orders in each model be set properly. Be sure to check and, if necessary, modify the bond orders so that there are no open valences on any atom.

3. To assign atom types, the total molecular charge on each model must be set before using the Forcefield/Potentials command. To make this step easier, two BCL macro scripts called initialize_charge.bcl and set_charge.bcl, are available. If you know the total molecular charge, use initialize_charge.bcl. If you do not know the total molecular charge, but do know the formal charge(s) of metal atom(s), use set_charge.bcl.

For setting the (known) total molecular charge, use the File/Source_File command to source the $BIOSYM/data/discover/esff/initialize_charge.bcl file. This adds a Set_Molec_Chg command to the Forcefield pulldown. Choosing the Set_Molec_Chg command prompts you to enter the name of the model. The macro then zeros all partial charges and sums the formal charges in the model. The resulting sum is reported as a suggested total molecular charge. You may accept this or specify a different value. After the macro has run, the model is ready for atom type assignment with the Forcefield/Potentials command.

If you know the formal charges on the metal atoms and therefore choose to use set_charge.bcl you have to first execute the command:
Select Clear_Potentials Clear_Charges esff.frc -Make_Copy.

by entering it on the command line near the bottom of the Insight window. This sets the initial formal and partial charges on all the atoms to 0.

Then use the File/Source_File command to source the $BIO-SYM/data/discover/esff/set_charge.bcl file. This adds a Set_Charge command to the Forcefield pulldown. Choosing the Set_Charge command prompts you for the name of the model. The script loops through all the atoms in the model and prompts you to input the formal charge of metal atoms whenever one is found. After the macro has run, the model is ready for atom type assignment with the Forcefield/Potentials command.

4. Since ESFF calculates partial charges on the fly, the default options in the Forcefield/Potentials command are:

- Potential Action = Fix
- Partial Chg Action = Accept
- Formal Chg Action = Fix

Setting Partial Chg Action to Fix only allows you to review the partial charges calculated with ESFF before submitting the Discover job. It would take quite a while for a model with a large number of atoms to actually correct the partial charges.

5. If fixing the formal charge fails, setting Formal Chg Action to Accept is suggested if the total molecular charge has been set as described in Step 3 and/or the formal charges have been assigned manually.

Much of the computational work of MSI products is performed by background jobs that are run using the Insight user interface. Background jobs run concurrently with the Insight program; this is possible because, once started, they do not require user interaction. If you have access to more than one computer (mainframe or workstation), you may want to run some of the background jobs on a
3. C Discover within Insight

different computer (the remote host) than the one that is running the Insight interface (the local host).

Running the background job on a remote host involves several general requirements:

- The actual background job program must exist as an executable image for that host.
- Files transferred between local and remote hosts must be readable and writable on both hosts.
- You must have an account on the remote host, and sufficient disk space to contain all the input, temporary, and output files produced by that background job.
- The local host must be able to communicate statuses, submit jobs, and copy files to the remote host.

Making an executable image compatible with a remote host involves recompiling and relinking the program; this is done by MSI for host types that the company supports.

Background jobs are often started from the Insight interface in immediate submission mode; that is, the job is run on the selected machine immediately upon submission. However, background jobs can also be submitted to a queue. The queueing mechanism pulls jobs from the queue and starts them on one of the machines available for the queue. Typically, each job is run to completion before the next job is pulled off the queue. Support for the network queueing system (NQS) is now available to allow you to submit background jobs to a queue on a local or remote machine.

It is up to you to acquire and install NQS products and deal with its support issues. The Background Job queueing mechanism assumes a properly configured NQS.

Using the Background Job pulldown

Use of the Background Job pulldown is optional. If it is not used, the default is to run all jobs on the local host in Cont_Insight mode. If you prefer this mode, you do not need to read the remainder of this subsection.
When the **Setup_Bkgd_Job** command is used, the background job list shows only those background jobs that are run from the current module and that can be run on a remote host. If the module contains only one job, the parameter is automatically filled in. The list of hosts shows only those hosts that are associated with that background job in the background_job_hosts file at your site. It is possible for you to specify a remote host that is unavailable (off line, for instance) or for which you have no login account.

Use the **Background_Job/Control_Background_Job** command to coordinate running background jobs by detaching selected jobs from or attaching them to Insight. In addition, you can use this command to specify the interval for invoking a task specific to a particular background job for processing its output.

Every background job submitted via the generic background utility is assigned a job number. This number is displayed in the information area when the job is submitted (e.g., *Starting Discover 3 background job ... as job 1*). You should note the job number when the job is submitted, since it can be used later to check on the job’s completion status and/or kill the job.

### Setting up a background job

The **Setup_Bkgd_Job** command does not actually run the command; it simply records your host and **Execution_Mode** preference. The default host is local. Your selected host and **Execution_Mode** are used for any subsequent background jobs for the duration of the Insight session. When you start up a new session, all background job parameters are again set to their default values.

The **Execution_Mode** parameter allows you to run a background job concurrently (**Cont_Insight**) or interactively (**Wait_For_Job**) or to simply create the necessary command files to submit the job, but not actually execute them (**Cmd_File_Only**).

The **Send_Mail** parameter allows you to have the system send you an electronic mail message upon completion of the background job. This parameter is not active if **Execution_Mode** is set to **Wait_For_Job**. You may find this option useful when running long jobs where you exit the Insight program before the job completes.
3. CDiscover within Insight

The **Save_Cmd_Files** parameter allows you to save the command file used to submit the background job (bkgd_job_run_name#.csh). Otherwise this file is deleted when the job completes. This parameter is not active when **Execution_Mode** is set to **Cmd_File_Only**.

All background jobs return a completion status. The completion status is an integer code that indicates success, failure, and/or reason for failure of the job. The status code is always displayed when you are notified that the job has completed.

If you consistently want to send background jobs to another host, you can modify your personal Insight startup file to invoke **Setup_Bkgd_Job** for each module/background job(s) that you want to automatically assign. Note that you must first change to the module in which the background job’s interface is found before using **Setup_Bkgd_Job** to set a preference for that background job.

The **Completion_Window** parameter can be used to prevent the notification window from appearing when the background job completes. The default value is **on**.

Support for the network queuing system (NQS) is now available in the **Background_Job/Setup_Bkgd_Job** command.

For the user interface to present parameter defaults and a value-aid containing available queues, and to correctly formulate an NQS command, the user’s NQS queue environment information must be provided to the Insight program. The **Background_Job_Hosts** file contains the NQS queue information, or you may enter the required information directly using the **Background_Job/Setup_Bkgd_Job** command.

Based on the parameter values provided in the **Background_Job/Setup_Bkgd_Job** command for **Queued_Submission_Mode**, the Insight background job mechanism formulates a standard NQS command and starts a process to execute it. It is assumed that the NQS command constructed by the Insight program functions with your NQS configuration.

---

**Interactive screen coordinate updates**

CDiscover supports interactive screen updates of an energy/iteration status display and the model coordinates by default.
Examining completion status

The **Completion_Status** command has three modes of operation. The **One_Job** option displays a brief message indicating whether a specific job has completed. The message is displayed in the information area of the screen. Certain background jobs generate a status file containing additional information while they are running. If this additional status information is available, it is displayed in the textport. If **All_Jobs** is chosen, the job number, job name, run name, status code, and job status are displayed in the textport for every job submitted during the current Insight session. The **Look_Up_Status** option is used to find the meaning of a return status code.

The **Report_Mode** parameter is used to indicate what information you would like the command to return: status of one job, status of all jobs, or the meaning of a return status code from a particular job.

The **Job_Number** parameter becomes active when **One_Job** is selected. It is used to specify a specific background job that you want to monitor.

The **Background_Job** and **Status** parameters become active when **Look_Up_Status** is chosen. They are used to specify a status code that you would like to look up.

Killing a background job

The **Kill_Bkgd_Job** command is used to stop execution of a background job by killing the process in which it is running and, optionally, deleting its output files.

The **Job_Number** parameter is used to specify which background job to kill. A value-aid containing a list of all currently running background jobs is provided.

If the **Save_Output** parameter is toggled **on**, then all output files generated by the background job are saved when the job is killed. The default value of this parameter is **off**, in which case all output files are deleted.
3. CDiscover within Insight

**In case of run errors**

You can use the D_Run/Disco_Error_List command to search the output file for error messages.

---

**Tutorial—The Insight environment**

This chapter introduces you to the commands and functionalities of the Discover_3 module. The lessons (most of which are now supplied in electronic format) assume that you already have some familiarity with the Insight interface (please see the Insight User Guide for relevant tutorials).

---

**Pilot tutorials**

Most tutorials are now available for use with the Pilot interface. To access these tutorials for the Discover program, click the Pilot icon in the Insight interface.

Then, from the Open Tutorial window, select Discover_3 tutorials (in Insight with the blue interface) or Molecular Mechanics Tutorials and then Discover_3 Tutorials (in Insight with the grey interface) and choose from the list of available lessons:

- Lessons 1-3 = Introduction to Discover_3 Module
- Lesson 4 = Constant-Stress Dynamics
- Lesson 6 = Using a Precoded Btcl Script for a Complex Simulation
- ESFF Lesson # = (The ESFF lessons differ in the blue and grey Insight releases.)

You can access the Open Tutorial window at any time by clicking the Open File button in the lower left corner of the Pilot window. For a more complete description of Pilot and its use, click the on-screen help button in the Pilot interface or refer to the Insight II guide.
Overview of tutorial lessons

Lessons 1–3 are merged into one Pilot logfile.

In Lesson 1: Minimizing a model, you will minimize a cyclic pentapeptide, cyclo-[gly–pro–gly–ala–arg], with CDiscover, using the default cascade method. This minimization method employs the steepest descent, conjugate gradients, and Newton–Raphson algorithms in sequence.

The topics covered in this lesson are:
- Minimizing a cyclic peptide.
- Comparing the minimized and initial structures.
- Finding out the energy of the minimized structure.

In Lesson 2: Simple Dynamics with Equilibration and Data Collection Stages, you will perform a simple dynamics simulation on the optimized structure using the constant-volume/constant-temperature ensemble. Then you will animate the dynamics trajectories and plot out energy averages and temperatures.

The topics covered in this lesson are:
- Setting up the equilibration stage of molecular dynamics on a minimized structure.
- Setting up the data-collection stage.
- Animating the trajectories.
- Plotting the instantaneous and batch averages of the total energy, potential energy, and temperature.

In Lesson 3: Restarting a Dynamics Calculation, you will continue the molecular dynamics run that was started in Lesson 2, using the dynamics restart file.

You will learn:
- To read in the dynamics restart file through the File_Control command.
- To continue a dynamics run that was stopped earlier.
Lesson 4: Constant-Stress Dynamics is supplied as a Pilot script. In this lesson, you will read in the amorphous cell of a 76-mer of polypropylene, which is a periodic system. Next you will specify the applied stress and run constant-stress dynamics. Then you will submit the run as a background job through the Insight interface. Finally, you will plot the stress and strain components of the system vs. time.

The topics covered in this lesson are:

♦ Performing constant-stress dynamics on a periodic system.
♦ Running a background job.

Lesson 5: Generating a phi/psi map is supplied in this documentation, not in Pilot. In this lesson, you will explore how to set up a repetitive calculation with the Discover_3 module in the Insight interface. Polypropylene—\((\text{CH}_2\text{CH}\text{(CH}_3)\text{)}_n\)—as a bulk polymer prefers a gauche conformation about the C–C bond. The lowest-energy conformation for a C–C bond in a chain is the trans conformation, but the steric interaction of the methyl groups destabilizes it enough in polypropylene that the gauche conformation has the lowest energy. In this exercise you will generate the potential energy surface for a small segment of the polymer chain as a function of the two torsion angles in the chain. The contour map of such a surface is useful both for understanding the possible conformations and flexibility of such a system and also as a test of the accuracy of a forcefield.

The commands in the Discover_3 module support a concept of stages of a calculation. Each stage sets up a distinct step of the calculation. To generate the phi/psi map, you need to obtain the energy at a grid of points. You will accomplish this by setting up a pair of nested loops over the two torsion angles. Since a rigid rotation about the torsions would not allow the model to relax and would result in an artificially high steric energy, you will need to minimize the structure at each grid point, applying a restraint so that the two torsion angles cannot change. Although this may sound complex, it is quite straightforward. The following outline gives the steps needed:
Loop from -180 degrees to +180 degrees for the first torsion
   Set the torsion to the desired angle
   Add a restraint to keep it at the desired angle
Loop from -180 degrees to +180 degrees for the second torsion
   Set the torsion to the desired angle
   Add a restraint to keep it at the desired angle
   Minimize the structure
   Write out the structure and energy for later analysis
End loop over second torsion
End loop over first torsion

First, you need to build the segment of the polymer. You will use 2,4-dimethylpentane as the model in this lesson.

Lesson 6: Using a Precoded Btcl Script for a Complex Simulation is supplied as a Pilot file. In this lesson, you will use a Btcl script to run a diffusion calculation.

The topics covered in this lesson are:

♦ Reading a pre-recorded Btcl input file into the Insight interface and running it.

♦ Copying models and changing their default display characteristics to make the course of a calculation easier to see.

♦ Defining subsets and deleting the definition.

♦ Graphically differentiating between subsets of models whose calculation conditions differ.

ESFF lessons are supplied as Pilot files. They show you how to construct and simulate several types of models that contain metals. Since the number and content of these lessons differ for the Materials Science and Life Sciences releases of MSI’s software, they are not described here.
3. C Discover within Insight

**Lesson 5: Generating a phi/psi map**

1. Invoking the Insight II program

   Issue the command `biosym_tutorial -i` at the system prompt. If the Discover tutorial files are not installed, enter `discover` at the first prompt.

   Press <Enter> to get out of the installation menu, then enter 3 to change to the tutorial directory, followed by 1 to start up the Insight program.

   *It takes a few moments for the Insight program to start up.*

2. Getting a monomer of polypropylene

   Select the File/Import (or Molecule/Get) command. When the parameter block appears, fill in the File_Name parameter box with `pp.car` by choosing it from the value-aid. Scroll the list, if necessary, by using the scrollater on its right side. Select Execute.

   The model provided corresponds to a fragment of isotactic polypropylene, with the same stereochemistry for the methyl side chains. If you want, after completing this tutorial you can modify or rebuild the model to correspond to syndiotactic polypropylene, having the methyl groups on opposite sides of the main chain.

   You can use the mouse buttons to translate and rotate the model to facilitate viewing it.
3. Labelling the model

Select the Molecule/Label command. Pick the model to set Molecule Spec to PP, if necessary, and Execute the command.

Each atom of the model in the display area becomes labelled. This will aid in later definition of the torsion angles.

4. Activating the CDIscover program

Go to the Module pulldown (click the MSI logo) and select Discover_3.

The Discover_3 pulldowns appear on the lower menu bar.

5. Defining the system

Select the Setup/System command from the lower menu bar. Set the Disco Object Name parameter value to PP by picking the model in the display area or choosing its name from the value-aid. Select Execute.

This defines PP as the model that you want to work with.

The Discover_3 commands function a bit differently than most commands in the Insight interface. Since you are setting up a perhaps long, complex calculation, the commands store the information you enter. The calculation does not start until you select the D_Run/Run command. Thus, nothing actually happens when you execute all of the other commands in the Discover_3 module except that the information is stored. If you make a mistake or forget something, you can come back and add, delete, or modify anything in the setup for the job at any time until you finally run the job with the Run command. The last part of this tutorial shows you how to add a step that you forgot.
3. C Discover within Insight

6. Specifying the range of nonbond interactions

Select the **Specify/Nonbonds** command. When the parameter block appears, set **Summation Method** to **No_Cutoffs** (click the box under **Summation Method** and choose **No_Cutoffs** from the list that appears) and select **Execute**.

The Discover program uses the cell multipole method by default for nonbond calculations. This is the method of choice for most models, but since this tutorial is working with such a small model you can ask Discover to simply calculate all the pairwise interactions. This is the fastest and most accurate method for models with fewer than about 500–1000 atoms, depending somewhat on what type of computer you are using.

7. Setting up the loop over the first torsion

Select the **Language_Control/Looping_Control** command. Set the following parameter values:

- **Control Procedure** = **Loop**
- **Loop Variable Name** = **phi** (a $ sign is automatically added to the name)
- **From** = -180
- **To** = 180
- **Every** = 30

Select **Execute** to store this stage.
8. Rotating the first torsion to the current angle

Select the **Calculate/Geometric** command. Set the following parameter values:

- **Geom Type** = Torsion
- **Move Value** = $\phi$
- **Set Atom 1** through **Set Atom 4** = pick the backbone atoms C1 through C4 of the model in the display area, in order.

Select **Execute** to store the stage.

9. Specifying the restraint on the first torsion

Select the **Specify/Restraint** command and set the following parameter values:

- **More** = on
- **Restraint_Target** = Absolute
- **Target Value** = $\phi$
- **Force Constant** = 1000.0
- **Pick Atom 1** through **Pick Atom 4** = pick the backbone atoms C1 through C4 of the model in the display area, in order.

Select **Execute** to store the stage.
3. CDiscover within Insight

10. Setting up the loop over the second torsion

Select the **Language_Control/Looping_Control** command. Set the following parameter values:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop Variable Name</td>
<td>psi</td>
</tr>
<tr>
<td>From</td>
<td>-180</td>
</tr>
<tr>
<td>To</td>
<td>180</td>
</tr>
<tr>
<td>Every</td>
<td>30</td>
</tr>
</tbody>
</table>

Select **Execute** to store this stage.

11. Rotating the second torsion to the current angle

Select the **Calculate/Geometric** command. Set the following parameter values:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Geom Type</td>
<td>Torsion</td>
</tr>
<tr>
<td>Move Value</td>
<td>$\psi$</td>
</tr>
</tbody>
</table>

**Set Atom 1** through **Set Atom 4** = pick the backbone atoms C2 through C5 of the model in the display area, in order.

Select **Execute** to store the stage.
12. Specifying the restraint on the second torsion

Select the **Specify/Restraint** command and set the following parameter values:

- **More = on**
- **Restraint Types = Torsion**
- **Restraint Target = Absolute**
- **Force Constant = 1000.0**
- **Target Value = $psi**

Pick Atom 1 through Pick Atom 4 = pick the backbone atoms C2 through C5 of the model in the display area, in order.

Select **Execute** to store the stage.

13. Setting up to minimize the structure

Choose the **Calculate/Minimize** command. The default parameter values are sufficient, so select **Execute**.

14. Specifying that the minimized structure should be stored in an archive file

Select the **Language_Control/File_Control** command. Set the **File Operation** parameter to **Save**. For **Frame Number**, enter **$count**. Select **Execute** to store this stage.

Notice that the **File_Control** command requires a **Frame Number** to control writing to the archive file. If you enter a constant, such as 1, all the results would be written to the same frame, overwriting all previous results. That is why you were instructed to enter a variable
3. CDiscover within Insight

name ($\text{count}$). However, that variable has not been initialized or incremented for each new structure. Therefore you need to insert a couple of earlier stages into the calculation to take care of this.

15. Inserting a stage to initialize the counter

Select the **Language_Control/Command_Comment** command.

The first stage control parameter, **Activations**, has three choices: **Add**, **Insert**, and **Modify**. Beneath the second parameter, **Stage Name**, is the default name for this stage. When the parameter block first comes up, the default choice for **Activations** is **Add**, and the **Stage Name** parameter is active. You may change the **Stage Name** parameter if you wish, but the default is usually the best choice. The default shown now is **11 comment**. The number is the position of this stage in the list—eleventh in this case. The rest of the name (comment) tells you what this stage does. You may have noticed that you haven’t created a comment stage before now, yet another one (**1 begin**) claims to be the first one (according to the list in the value-aid). The reason for this is that Discover automatically starts every calculation with a comment, so the very first stage in any calculation is a comment. If you use the default scheme for stage names, Discover maintains them for you.

Try setting the **Activation** parameter to **Modify**. The **Parameters** value-aid that appears lists all the stages of the same type (comments in this case). If you needed to revise some of the parameters in one of the stages, you would select the stage’s name in the value-aid. The parameter block would reflect the settings of all the parameters in that stage. After changing anything that you needed to, you would select **Execute** to save the modified stage.

You need to insert a stage, however, so:

Set the **Activations** parameter to **Insert**. The stage name should be **11 comment**. Now click the **Insert Before Stage** parameter box to activate it.
The value-aid lists all the stages that you have created so far. Notice that each nested level of the loops is indented by a pair of periods to help you see the structure of your calculation.

You need to initialize the counter before the first loop:

Choose the 3 loop stage in the value-aid. Set the Comment Type parameter to Command and then type $count = 0 in the Command/Comment parameter box. Note the spaces on either side of the equals sign. Select Execute to create this stage.

Notice in the Parameters value-aid that the third stage is 3 comment and the numbers for subsequent stages have been incremented by one.

16. Inserting a stage to increment the counter

Still in the Command_Comment parameter block, check that the Activations parameter is still set to Insert. Select the Insert Before Stage parameter if it is not already active. From the value-aid, select the entry ....8 geometry. Then change the Command/Comment parameter to $count = $count + 1. Select Execute to save this stage.

17. Ending the loops over the torsion angles

Select the Language_Control/Looping_Control command. Set the Control_Procedure to End. Select Execute to end the inner loop over the psi torsion angle.

Select Execute a second time to end the outer loop over the phi torsion angle.

You have completed the job setup, but before you run the calculation.
3. C Discover within Insight

it would be a good idea to look over the setup to make sure that you haven’t forgotten anything. There are several ways to do this. You could look back through all the stages using the Modify Activations parameter to look at the settings in the menus for each stage. But it is probably easier to look at the input file for the program, so let’s do that:

18. Checking the input file

Select the Setup/List command, select Input File as the List Options parameter, and Execute the command.

The textport pops in front of the Insight window and should contain a listing similar to the following:

```plaintext
#BIOSYM btcl 3
#
# Input File For Discover Generated By Insight Version 95.0
# Date: Wed Jun 14 11:28:40 1995
# User Name: marj
# Host Name: iris90
# Host Type: iris
#
# System Name: PP
#
#Stage Name: 1 begin
# begin
#Stage Name: 2 nonbonds
forcefield nonbond \ 
   -separate_coulomb \ 
     vdw \ 
       summation_method = no_cutoff \ 
       coulomb \ 
       -distance_dependent_dielectric
#
#Stage Name: 3 comment
$count = 0
#Stage Name: 4 loop
for {$phi = -180} {$phi <= 180} {$phi = $phi + 30} { 
#
#Stage Name: ..5 geometry
molGeom set \ 
   torsion $phi \ "PP:1:C1" "PP:1:C2" "PP:1:C3" "PP:1:C4"
#
#Stage Name: ..6 restraint
    restraint create "S__restraint" torsion "PP:1:C1" "PP:1:C2" "PP:1:C3" "PP:1:C4"
    restraint function "S__restraint" cosine "1000.0" "1"
    restraint target "S__restraint" "$phi"
#
#Stage Name: ..7 loop
for {$psi = -180} {$psi <= 180} {$psi = $psi + 30} { 
#
#Stage Name: ....8 command
$\text{count} = \text{count} + 1
#
#Stage Name: ....9 geometry
molGeom set \n    torsion $\text{psi} \n    "PP:1:C2" "PP:1:C3" "PP:1:C4" "PP:1:C5"
#
#Stage Name: ....10 restraint
restraint create "8__restraint" torsion "PP:1:C2" "PP:1:C3" "PP:1:C4" "PP:1:C5"
restraint function "8__restraint" cosine "1000.0" "1"
restraint target "8__restraint" "$\text{psi}"
#
#Stage Name: ....11 minimize
minimize \n    iteration_limit = 300 movement_limit = 0.200 \n    sd \n    convergence = 1000.0 line_search_precision = 0.100 \n    cg \n    convergence = 10.0 method = polak \n    line_search_precision = 0.100 \n    newton \n    convergence = 0.001 method = bfgs \n    line_search_precision = 0.900 max_atoms = 200 \n    final_convergence = 0.001
#
writeFile coordinate filename = .cor
#
#Stage Name: ..12 file control
writeFile archive frame = $\text{count}
#
#Stage Name: ..13 end
#
#Stage Name: 14 end

The atom names in the \text{molGeom} and \text{restraint} commands might be different, if you built the model instead of using the one that was provided or if you defined the torsions different than in the instructions. Other than that, the listing of your input file should be identical to that given above.

Check it carefully, especially noting the use of the $\text{phi}$ and $\text{psi}$ variables. You should notice that each command in the input file corresponds to one of the stages that you set up with the parameter blocks, except that a \text{writeFile coordinate filename = .cor} line was added by the Discover program. The \text{writeFile} command is always added.
3. CDiscover within Insight

*after minimizations to write out the .cor file.*

---

**After comparing your input file with the example, select the Textport off button near the bottom of the Insight screen to put the textport away.**

If your input file does not correspond to the one above, you need to correct the errors. If some of the parameters for a command are not correct, you need to modify that stage, as was outlined at Step 15. If a command is missing completely, you accidentally skipped it or forgot to select Execute when creating the stages. In this case, you need to go back and insert the appropriate stage in the correct place. Follow the same procedure that you used in Step 16 to insert the Command/Comment stage, but of course insert a stage of the correct type.

---

19. Starting the Discover job

**Select D_Run/Run and be sure that pp0 is entered as the Run_Name. Execute the command to start the calculation.**

The calculation will take anything from about 10 minutes on a fast workstation (such as an R4000-based SGI or IBM RS-6000 370) to a few hours on slower machines (such as an SGI 4D-25). If you leave the Insight interface running while the Discover job executes, you will be notified when the job is completed. (If you want to check whether the job has finished, you can choose the Background_Job/Completion_Status command and select Execute. A message on the job’s status appears in the information area near the bottom of the Insight screen.) If you quit the Insight interface, you can use commands such as gr_top on the SGI, smit on the IBM, or ps on either machine to check if the job is still running. After it completes, you are ready to read the results back into the Insight environment and generate a phi/psi map.
Tutorial—The Insight environment

20. Restarting the Insight program (if needed)

If you exited the Insight environment, restart it as in Step 1 and use the File/Import or Molecule/Get command to read in the 2,4-dimethylpentane model you created previously. This model was saved in a file called pp0.car when you started the Discover calculation.

You must load a model before loading its trajectory.

21. Reading in the archive file generated by the calculation

Go to the Analysis module by clicking the MSI logo. Select the Trajectory/Get command from the lower menu bar, choose pp0.arc in the value-aid, set the Trajectory Object to PP, and select Execute to read in the archive file.

In the information area at the bottom of the Insight window, you should see a message that trajectory records for 23 atoms and 169 images were processed. If fewer frames were found, your job was either not completed or terminated early for some reason. You can look in the pp0.out file for error messages.

22. Constructing a graph of energy vs. the two torsion angles

Select the Trajectory/Construct_Graph command. Select Execute.

Another parameter block, titled Axis_Function Trajectory, appears
3. CDiscover within Insight

*for defining the graph.*

Choose **Geometry** as the **Function Type** and select **Dihedral** in the scrolling list that appears. Select **Execute**.

*A Dihedral_Def Trajectory* parameter block appears for defining the dihedral.

Choose **0_Is_Trans** for the **Angle Convention**, then pick the first four carbon atoms in the backbone (C1 through C4, in that order). This fills in the **Atom 1 Spec** through the **Atom 4 Spec** parameters. The command automatically executes when you pick the fourth atom.

When the **Axis Function Trajectory** parameter block reappears, the default **Graph Axis** has changed to **Y** and **Dihedral** is still selected. Select **Execute**.

When the **Dihedral_Def Trajectory** parameter block appears, choose **0_Is_Trans** for **Angle Convention**. Now select the second through fifth carbon atoms in the backbone. Again the command automatically executes when you pick the last atom and fill in the **Atom 4 Spec** parameter.

The **Axis_FunctionTrajectory** parameter block appears again, this time with the **Z** axis chosen. Choose **Energy** for the **Function Type** and then select **Execute**.
A 3D plot appears in the Insight window. Initially you are looking down the Z axis, so it appears 2D. If you want to rotate and move the graph, press the <F10> key (or equivalently select the on-screen button labeled Connect Object in the lower part of the Insight window with the left mouse button). Now pick one of the corners of the graph with the left mouse button. You should see a message in the information window at the lower part of the Insight screen saying that the dials are connected to GRAPH1. You may need to be quite accurate in picking the corner of the graph, so try again if you do not see the message in the information window. After you are connected to the graph, you can rotate it by dragging with the right button held down. You translate it by dragging with the center button down and resize by dragging with both the center and right buttons down. To resize, dragging up on the screen enlarges the graph, down shrinks it.

A Energy_Def Trajectory parameter block appears for defining what type of energy to plot. The default is Total for Energy Type. Check that it is indeed chosen, and then select Execute. (This parameter block appears in Insight with the blue interface, but not in Insight with the grey interface.)

When the Axis_FunctionTrajectory parameter block reappears, choose End_Graph for the Function Mode and select Execute.
23. Constructing a spectrum

You will use a spectrum to color the phi/psi map.

Select the **Spectrum/Edit** command from the top menu bar and click **New Spectrum**. In the **New Spectrum** dialog box, set the following parameter values:

- **Spectrum Name**: Energy
- **Minimum Value**: -23
- **Minimum Color**: blue
- **Maximum Value**: -3
- **Maximum Color**: red

Select OK.

(The above is true for Insight with the blue interface. For Insight with the grey interface, use the **Spectrum/Create** command and also set the **Under Range Color** and **Over Range Color** parameters to blue and red, respectively.)

Cancel the Edit Spectrum Dialog unless you want to play with it.

You can shrink and position the spectrum in a convenient place in the same way that you manipulated the graph in the previous step.
24. Creating the phi/psi map

Select the Graph/Contour command from the lower menu bar. Choose Range_Create from the popup under Contour_Action_2D. Toggle the New_Graph parameter off. The Graph Name should be GRAPH1, and Plot Spec should be GRAPH1:1. If they are not, correct them. Toggle Use_Spectrum to on and set the Spectrum Name to ENERGY, the name you gave the spectrum that you created.

Set the following additional parameter values:

- Pick Atom 4 Value = -23
- Interval Size = 2
- Termination = End_value
- End Value = -3

Select Execute to create the map.

You can connect to the graph, move it, and resize it as was outlined in Step 22. The blue contours are the low-energy points; red are high. If you pick any vertex on the contours, a line such as:

MAP1:p3.0:60 Contour level: 3.00, X=99.28, Y=213.04, Z=0.00

appears in the information window. The Contour Level is the energy of the contour in kcal mol⁻¹. The minimum for this system is at about -23.9 kcal mol⁻¹ at a gauche-trans conformation.

25. Animating the model

Select the Trajectory/Animate command and select Execute.

The displayed model starts to cycle through the archived structures. You can cancel the parameter block to see more of the display area. The model’s structure is updated, and a cursor moves along the graph to indicate the position of that structure in the graph. The <F8> key (or
3. CDiscover within Insight

the button now labeled Anim Frame) toggles between automatic playback and a manual mode. In the manual mode, you click the dialbox labeled Anim Frame in the lower left portion of the Insight window. Selecting the left side of the button moves a frame backward; the right side, forward. The frames cycle in the direction you select for as long as you hold the mouse button down.

You should see a pair of minima in the contour map on either side of the diagonal in the upper left quadrant. One is at -120,0; the other 0,120. These correspond to the two symmetric conformations gauche—trans and trans—gauche. Because we truncated the polypropylene to a very symmetric dimethylpentane, these two structures are in fact identical: they interchange the methyl groups that terminate the backbone with the methyl groups that are the sidechains. If we had used a longer segment of the polymer—five or six monomers, say—rotating and restraining all the phi angles as a group and all the psi angles also as a group, the two minima would have become slightly different. One corresponds to a left-handed helix repeating after three monomers; the other, a right-handed helix. Because of the chirality due to the side chains, which are all R or all S in isotactic polypropylene, one of the minima has a lower energy than the other.

26. Exiting the Insight program

Type quit on in the command line near the bottom of the Insight screen and press <Enter>.

Now enter 5 in the shell window from which you started the Insight program, to exit the biosym_tutorial script.

Command summary—The Insight environment

CDiscover is accessed in Insight by choosing Discover_3 from the Module pulldown. The Discover_3 module contains several pull-downs in addition to the core pulldowns on the top menu bar: Setup, Strategy, Specify, Calculate, Language_Control, Analyze, Pseudo_Atom, Background_Job, and D_Run.
Command summary—The Insight environment

What the commands in these pulldowns do is summarized briefly below. For more detailed explanations of individual commands, please refer to the Insight on-screen help facility (by clicking the help icon, which is the button containing a question mark, near the bottom corner of the main Insight window) and to *Btcl Language and Commands—Standalone Mode*.

**Setup pulldown**

The *Setup/System* command is used to specify the object(s) on which to perform Discover calculations. It also allows you to choose whether the parameter blocks for other commands will be shown in their expert or non-expert mode.

The *Setup/Stage* command is used to delete stage(s) from an existing list of stages.

The *Setup/List* command is used to output information about the calculation stages to the textport or to a specified file.

**Strategy pulldown**

The *Strategy* pulldown gives you access to several pre-recorded specific types of calculations.

The *Strategy/Simple_Minimize* command is used to set up a simple minimization run.

The *Strategy/Simple_Min_Dyn* command is used to set up a simple minimization and dynamics run.

The *Strategy/Phi_Psi_Map* is used for generating a phi/psi map of rotation about two torsions.

The *Strategy/Use_Existing_Input* command is used to run an existing Discover input file.

**Specify pulldown**

The *Specify* pulldown is used to insert commands into the command input file that set conditions that affect all subsequent stages in a calculation.
3. CDiscover within Insight

The **Specify/Nonbonds** command is used to control calculation of the van der Waals and electrostatic interactions.

The **Specify/Forcefield** command is used to control assignment and handling of forcefield parameters. Subtypes of a forcefield (for example, CVFF with or without the inclusion of cross terms in the calculation) can be selected. Note that changing the forcefield itself, for example from CVFF to CFF, is done in the **Builder** or a similar module, since this changes the atom types in the model and assigns charges.

The **Specify/Fix** command is used to set and unset movability characteristics of atoms.

The **Specify/Scaling** command is used to set factors used for scaling the forcefield energy terms.

The **Specify/Restraint** command is used to specify forces, in addition to those of the forcefield, to bias the simulation of all or part of the system.

---

**Calculate pulldown**

The **Calculate** pulldown contains commands that allow you to specify many calculation commands that are used to build up an input file for a CDiscover run.

The **Calculate/Minimize** command is used to set up parameters for refining the structure of a system by regressing its potential energy to a local minimum, with a choice of several minimization methods that can be used in sequential stages.

The **Calculate/Dynamics** command is used to set up parameters to control a dynamics run, allowing control of temperature and pressure (or stress) during the initialization and data-collection stages.

The **Calculate/Geometric** command is used to fix (freeze) the bond length, bond angle, torsion angle, or out-of-plane angle for selected atoms.

The **Calculate/RMS_Comparison** command is used to set up parameters for a rigid-body rms comparison between two systems or subsets of two systems. (Subsets are defined with the **Subset** pulldown on the upper menu bar.)
The Calculate/Vibrational command is used to set the parameters for carrying out vibrational analysis.

Language Control pulldown

The Language Control/File_Control command is used to control reading and writing structures from or to archive files, reading information for continuing an interrupted dynamics calculation, and reading a restraints file.

The Language Control/Looping_Control command is used to insert flow-control statements (loops, if-tests, etc.) into the input command file and to assign variables. It can also be used to insert an existing input file into the new input file, when the run is started.

The Language Control/Command_Comment command is used for direct insertion of commands or comments into the input command file. An inserted command can be continued on the following line by ending it with a backslash (\).

Analyze pulldown

The Analyze/Output command is used to specify which information about a Discover run to print, and where and when to print it.

The Analyze/GeometricSubset command creates subsets based on geometrical considerations and/or returns the lists of items in the subsets.

The Analyze/EnergyAnalysis command performs energy analysis such as calculation of subset-subset interaction energies (blue Insight release only).

Pseudo_Atom pulldown

The Pseudo_Atom/Define command is used to create new pseudoatoms.

The Pseudo_Atom/Rename command is used to change the name of pseudoatoms.
3. CDiscover within Insight

The **Pseudo_Atom/Delete** command deletes a pseudoatom or a group of pseudoatoms from a model.

The **Pseudo_Atom/List** command lists information about the pseudoatoms.

---

**Background Job pulldown**

The **Background_Job** pulldown allows you to set up background jobs to run concurrently or interactively with the Insight program. You can also choose whether to send background jobs to a local or remote host.

The **Background_Job/Setup_Bkgd_Job** command allows you to set up the execution mode and select the host upon which to run a job. This command is also used to control the notification method for background job completion and cleanup options.

The **Background_Job/Control_Bkgd_Job** command allows you to coordinate running background jobs by detaching selected background jobs from or attaching them to the current Insight session. In addition, this command allows you to specify the interval for invoking a task specific to a particular background job for processing its output.

The **Background_Job/Completion_Status** command allows you to monitor and evaluate the completion status of one or all background jobs. In addition, this command can be used to look up the meaning of a return status code.

The **Background_Job/Kill_Bkgd_Job** command is used to terminate execution of a background job that has been submitted during the current Insight session.

---

**D_Run pulldown**

The **D_Run/Run** command is used to write out a command input file and start a CDiscover job.

The **D_Run/Disco_Error_List** command enables you to grep CDiscover output files for error messages.
Input files required

A CDiscover run can be set up in a standalone manner, completely outside the Cerius² or Insight environment. To do this, you need three files:

♦ An input file (run_name.inp, where run_name is your name for the calculation) containing Btcl commands to control the calculation.

♦ Two files describing the model (run_name.mdf and run_name.car).

It is usually easiest to create the model description files with the Cerius² or Insight interface. If you write out the command input files from the C²•Discover or Insight•Discover_3 module, the graphical interface program also writes out an input file reflecting the calculation set up via the menus of the module. This input file can serve as a convenient starting point, which you can modify with a text editor.

Overview of Btcl commands

Calculation commands

The primary Btcl commands are calculation commands: energy, minimize, and dynamics. As the names suggest, energy performs a single energy and energy derivative evaluation, minimize modifies atom coordinates to minimize energy, and dynamics performs a simulation of molecular dynamics.
4. Standalone CDiscover

**Commands for calculation setup and analysis**

Calculation setup is accomplished through the use of a number of auxiliary commands for input (begin, readFile, reset), output (output), print, writeFile), and control (atomMovability, energyContribution, forcefield, restraint, pseudoAtom). Analysis commands (diffraction and vibrationalAnalysis) allow you to perform additional analysis of output results. The peek command can be used to monitor the progress of minimization and dynamics calculations.

**Input commands**

The Btcl commands begin, readFile, and reset all read information into the Discover program from external files. The begin command is generally placed at the start of a run_name.inp file to read in the molecular data files and a forcefield file. The readFile command is used to read in a data file created by a previous CDiscover, Cerius\(^2\), or Insight run (e.g., a coordinate snapshot from an archive file). It also allows you to work with multiple systems. The reset command is used to reread the molecular data files and forcefield. All information associated with the current molecular system (e.g., coordinates, energy expression) is lost with this command. However, the settings of command defaults (e.g., for dynamics) are unchanged.

**Output commands**

The output command specifies amount of output desired. The print command (which must be part of a minimize or dynamics command) generates output of system properties to various file types. The writeFile command is used to write a data file that can be read by subsequent CDiscover, Cerius\(^2\), or Insight runs.

**Control commands**

The atomMovability command allows you to fix the position of an atom or disregard its contribution to total energy throughout a calculation. The energyContribution command allows you to specify energy contributions of your own design. The forcefield command allows you to select a forcefield and tailor the contributions of built-in energy terms. The restraint command allows you to
Overview of Btcl commands

impose distance, torsion, or angle restraints of various forms on selected atoms. The **pseudoAtom** command allows you to create collections of atoms which are to be treated as individual atoms.

**Exiting a Btcl session**

If you want to end a Btcl session without starting a CDiscover run, enter **exit**, **quit**, or **<Ctrl> d** at the BTCL prompt.

**Defining and using your own procedures**

One of the most useful features of Btcl is that you can quickly define and use your own procedures (*Btcl procedures*). Procedures defined in files named **filename** and located in the current directory of the Discover run are automatically accessible. Procedures defined in any other files may be used by including:

```
source filename
```

in the run_name.inp file. Here, **filename** is the full path to the procedure-definition file.

**Performance and convenience**

Interpretive script languages are convenient for user applications. It is generally true, however, that they do not provide high performance. To address this issue, Btcl provides low-level commands, **geometry** and **vector**, that support high-performance, low-level geometric and mathematical manipulations. These commands operate on arbitrary lists of scalars or coordinates with performance comparable to that of compiled code. This allows you to construct your own analysis and energy terms for incorporation into the Discover program. Vector operations currently employ a function-call interface.

If you would like to avoid dealing with geometric and mathematical details, a higher-level command (**molGeom**) allows you to work directly with models, monomers, atoms, and their relationships. **molGeom** employs some of the same performance enhancement techniques as **geometry** and **vector**. There is a bit more overhead involved in the use of **molGeom**, but its performance should be fine for most purposes. You will probably want to work
4. Standalone CDiscover

first with **molGeom**, and employ the lower-level commands only as necessary. The **molGeom** command can be used to get and set geometric properties of models, such as the coordinates of atoms, bond lengths, bond angles, etc.

**Databases**

Information that is read into the Discover program is organized into databases for subsequent use. Each distinct molecular system resides in a separate database with the same name as the molecular system. You can make your own databases and perform various operations on them (or on the system databases) with the **database** command and database handle operations (see $dbHandle). The **database** command supports database creation, deletion, and other high-level operations. (Use of the **database** command to create or delete system databases is not recommended.) The database handle operations are an extension of the **database** command. They are used to access the data within a particular database.

In addition to standard database search operations provided by the database handle operations, there are special data acquisition operations supported by the following commands:

♦ **select** — Return a list of models, monomers, or atoms matching a specification that you supply.

♦ **subStructure** — Return a list of all the atoms connected to a given atom which are not also bonded to a second given atom.

The **subset** command is used to create or access subsets in a database. A subset is a particular type of database table which is designed to be compatible with the mechanism of the **select** command as well as standard database operations.

Please see *Databases and Tables* for additional information on databases.

**Objects**

The **object** command is used to add information to or extract information from a Btcl object. A Btcl object is, in essence, a list of entries of arbitrary type. Btcl objects (of differing types) can be cre-
Initiating a run

Initiating a run

Given that these files exist, the Discover job can be started by several methods, depending on where you want its output to appear and the format of the command input file.

Direct method

Enter the following at the UNIX prompt:

```bash
> discovery run_name
```

where `run_name` is your name for the calculation.

The first line of the command input file `run_name.inp` must contain the line:

```
#BIOSYM dsl 3
```

or (preferably):

```
#BIOSYM btcl 3
```

which identifies it as a Btcl command input file for CDiscover.

The output from the calculations goes to the file `run_name.out`; and `run_name.err` contains messages from any serious errors that may have occurred.

ated with various commands, including object, vector, geometry, molGeom, database, database handle operations ($dbHandle), select, and subStructure. A Btcl object is deleted when the Btcl variable used to store its identity (or handle) is unset. As noted previously, objects are used to avoid processing burdensome amounts of data at the Btcl level. The object name is passed from Btcl to compiled code which deals with the underlying data.

Please see Databases and Tables for additional information on objects.
4. Standalone CDiscover

Using the interactive utility

If no run name is specified after `discovery`, an interactive Btcl utility is started:

```
> discovery
```

The BTCL prompt appears, and you need to enter the following:

```
BTCL > set PROJECT run_name
BTCL > source run_name.inp
```

Output from the calculations goes to standard output.

If you decide not to start a Discover run, you can end the Btcl session by entering `exit`, `quit`, or `<Ctrl>d` at the BTCL prompt.

File redirection method

Alternatively, the Btcl command `set PROJECT run_name` can be the first Btcl command in the file `run_name.inp`. In this case, initiate the run by entering:

```
> discovery < run_name.inp
```

Output from the calculations goes to standard output.

Running CDiscover in parallel

CDiscover can be run in parallel for certain operations on some platforms. At present, parallelism is supported on SGI multi-processors, the IBM SP2 series, and Convex Exemplar computers. To run in parallel in standalone mode, add the following options anywhere on the command line:

```
> discovery -mpi "-np N"
```

where $N$ is the number of processors.

To run CDiscover jobs submitted from the Insight program in parallel, set the environment variable `MPI_ARGS` to `"-mpi "-np N""`
An example CDiscover run

before running CDiscover. This can be done with the Session/Env_var command in the Insight interface or via the UNIX shell before invoking the Insight program:

> setenv MPI_ARGS '-mpi "-np <N>"'
> insightII

The efficiency of parallel computation depends on the system size and the calculation method chosen. At present, moderate numbers of processors work best (e.g., 4 CPUs).

Scripts and program initialization

CDiscover automatically initializes with a default discoverRc script. By modifying this script, the site administrator can make site-specific Btcl libraries available to users.

The script directory has been reorganized to include more example and gift scripts which demonstrate how to build scientific functionality with CDiscover.

An example CDiscover run

Input files

Below is a relatively simple example command input file which could be used to initialize a molecular system, minimize it to improve the molecular structure, and then perform a dynamics simulation on the minimized structure.

A more complicated series of command statements is demonstrated in the file named $BIOSYM/gifts/discover/tcl/acentm.inp. (The file sets up a calculation that compares the results obtained with several different forcefields.) Since the filename in this example is acenm.inp, the molecular data are read from the files acenm.car and acenm.mdf, which must be present in the same directory as the acenm.inp file. These example input files can be also found in the directory $BIOSYM/gifts/discover/tcl. To test them, you need to first move to a directory in which you have
4. Standalone CDiscover

write permission and copy the files by entering at the UNIX prompt:

```
> cp $BIOSYM/gifts/discover/tcl/acentm.* .
```

**A short example command input file**

```
#BIOSYM btcl 3
begin
minimize \
  iteration_limit = 20 \
  newton method = newton_raphson \
  execute \
    frequency = 1 \n    before = 1 \n    after = 1 \n    command = {
      print output energy_summary = 1 \n      internal_energy = 1 nonbond_energy = 1}

dynamics \
  boltzmann = 1 \n  time = 100.000000 \n  timestep = 0.250000 \n  ensemble = nvt \
  execute \
    frequency = {10 * 0.250000} \n    command = {print output energy_summary = 1 state = 1}

dynamics \
  time = 200.000000 \n  timestep = 0.250000 \n  ensemble = nve \
  execute \
    frequency = {10 * 0.250000} \n    command = {print output energy_summary = 1 state = 1}
```

This file can be written entirely by hand, using any text editor, or a file from some previous run could be edited to produce this input file.

Alternatively, the input file can be constructed using the Cerius² or Insight interface to CDiscover.

These Cerius²•Discover control panels would be used to construct the above file:

♦ Run Discover
♦ Discover Minimize
An example CDiscover run

- Discover Minimize Output
- Discover Dynamics
- Discover Dynamics Output
- Finally, write out the input file by clicking the Save Strategy to .inp File action button in the Discover Input File control panel.

The following Insight•Discover_3 pulldowns and commands would be used to construct the above file:

- Setup/System
- Commands/Minimize
- Commands/Dynamics
- Analyze/Output
- Finally, write out the input file by executing the Background_Job/Setup_Bkgd_Job command with the Execution_Mode parameter set to Cmd_File_Only and then executing the D_Run/Run command.

The file is now ready to use in a standalone run.

The sample acenm.car and acenm.mdf files

The model in these files (in the $BIOSYM/gifts/discover/tcl directory) can also be constructed with the Cerius² or Insight interface, by using the Builder facilities, followed by assigning force-field atom types. Alternatively, model description files output by other software can be used, provided they have the appropriate formats, contents, and filename extensions.

Explanation of command input file

Setting up the computational system

The first command to be executed is the begin command. This loads the molecular system having the specified name acenm (i.e., the files acenm.car and acenm.mdf). The begin command also opens and writes some initialization information to the default output file, acenm.out.

The computational steps

The minimize command assumes that minimization should be complete within 20 steps.
4. Standalone CDiscover

The dynamics command is then executed, to perform a simulation of 100 femtoseconds (since the timestep parameter is set to 0.25 fs, this amounts to 400 time steps).

**Output of results**

The dynamics command includes an execute subcommand to print energy information every 2.5 fs, as well as at the beginning and end of the dynamics run. This information is appended to the file acenm.out (which was opened earlier, by the begin command), although it is possible to specify some other filename.

If any errors are encountered during the run, error messages would also appear in the acenm.out file, and the run would be aborted.

**Initiating the run**

The commands contained in the acenm.inp file are executed by simply changing to the directory that contains the acenm.inp, acenm.car, and acenm.mdf files and entering, at the system prompt:

> discovery acenm

**History file support**

CDiscover can read and write history files. These files are in the same format as those of FDiscover and can be read by the Cerius² and Insight programs. The history file is written by using a print command during a minimization or dynamics simulation.

The readFile command may be used to read a particular frame of a history file into the CDiscover program. In this way a history file might be converted into an archive file, for instance, by using the writeFile archive command. The return value of the readFile command, when it is applied to a history or archive file, is the potential energy of that frame. This would allow you to, for instance, construct scripts that sort the frames in an archive or history file based on energy.
Editing the ESFF forcefield parameters

The explicit parameters of the ESFF forcefield may be modified for a particular molecular system. To do this, you first have to run a job for your system in the default mode (with the edit option flag `esffOverrideParameters` set to 0). An explicit parameter file called `run_name.epa` is output when the run finishes. You can then edit this `run_name.epa` file to modify the explicit parameters. Only the explicit parameters, such as reference values, force constants, etc, may be changed. To rerun the job with the modified explicit parameters you need to insert one command:

```
set esffOverrideParameters 1
```

after the command:

```
begin
```

in the input command file `run_name.inp`. With the edit option flag `esffOverrideParameters` now set to 1, only the parameters in the `run_name.epa` file are used. To use the default ESFF parameters to run the same molecular system once again, you need to reset the edit option flag `esffOverrideParameters` back to 0 by deleting the command line:

```
set esffOverrideParameters 1
```

from the input command file `run_name.inp`.

Tutorial—standalone mode

This tutorial is not available as a Pilot logfile, since Pilot functions only within the Insight interface.

Overview of tutorial lesson

In Lesson 1: Using Btcl and Tcl commands to manipulate the geometry of two helices, we have two helices and want to find the relative orientation that has the lowest energy. The two helices are identical.
4. Standalone CDiscover

We will first find their axes, then rotate one of them so that the axes are parallel. Then we will translate one of them so that the line joining the centroids of two axes is perpendicular to both axes at a distance you define. After that, the second helix will be moved along the line joining the centroids. It will be spun around its own axis and then it will also be rotated around the first helix. At each orientation, the energy will be calculated. The data will be written in a table file readable by the Insight program, so we can sort the data using the spreadsheet. All the different conformations are stored in an archive file and can be replayed within the Insight interface using the **Analysis/Trajectory** command.

This lesson assumes you are already familiar with using the Insight program.

In this lesson you will learn specific syntax and commands for:

- Specifying atoms.
- Creating a subset of atoms for a system, e.g., backbone atoms, using the **subset** command.
- Finding the axis of a helix by finding the least-squares fit line for the backbone atoms, using the **molGeom get lsqLine** command.
- Rigidly rotating a model around an axis, using the **molGeom rotate** command.
- Translating a model using the **molGeom translate** command.
- Calculating the energy using the **energy** command.
- Using the **geometry** command to find:
  - The angle between two lines using **geometry ang1 angle line1 line2** (You can examine the input file for the relevant commands after you copy it in Step 1 of the lesson.)
  - A point on a line using **geometry point1 point line1**
  - The direction vector of a line using **geometry vec vector line1**
  - The distance between a point and a line using **geometry dist distance point1 line2**
  - The distance between two lines using **geometry dist distance line1 line2**
Using the vector command to find:
The cross product between two vectors using vector vec3 cross vector1 vector2
The negation of a scalar or vector using vector argout negate argin
The difference (subtraction) between two scalars or vectors using vector output subtract inarg1 inarg2
How to change an angle from radians to degrees using vector angdegree degree angradian (This is necessary because the geometry command works in radians but the molGeom command works in degrees)

Printing data to a file using the writeFile command.
Setting a variable using the set command.
Writing a for loop to do repeated operations.
Opening a file by using the open command.
Writing a procedure using the proc statement.
Writing a conditional if statement.
Writing out data using the puts and echo commands.
Using format statements to print in formatted form.
Using the Insight spreadsheet to sort the energies in ascending order.
Using the Analysis/Trajectory command of the Insight interface to replay various orientations of the helices and their energies.
Lesson 1: Using Btcl and Tcl commands to manipulate the geometry of two helices

1. Obtaining the required files

Copy the files helix.inp, helix.car, and helix.mdf from the directory $BIOSYM/tutorial/discover/ to a directory in which you have write permission.

2. Examining the command input file

Read the command input file to understand it—the commands are associated with explanations in comment lines (starting with #).

A few hints to understanding the script are:
♦ Locate the beginning of the main commands section by finding the begin command.
♦ Understand the logic of the program by reading the comment lines first.
♦ Find the procedures fixorientation and writeheading and see how they are defined.
♦ Identify the Tcl and Btcl commands useful for the following purposes:
  Assigning values to variables.
  Choosing a subset of atoms and the atom specification syntax. (The most general syntax is “model_name:residue_name:atom_name”.)
  Writing and calling a procedure.
  Writing a for loop.
  Using an if statement.
Opening a file and writing data to a file.
Using various `molGeom` and `geometry` statements to do geometry manipulation.
Using various `vector` statements to do vector and scalar operations.

3. Running the CDiscover program in standalone mode

Run CDiscover by entering at the UNIX prompt (indicated as `>`):

```
> discovery helix
```
Wait for the run to end.

4. Finding the output files

List the files in your working directory by entering:

```
> ls -lt
```

*Output files of the run should include helix.out, helix.cor, helix.arc, and helix.tbl*
4. Standalone CDiscover

5. Examining the output files

Examine the helix.out file (for example, with the UNIX **more** command). This is the default output name for the file that contains all the input lines in addition to the relevant output information.

Search for these output lines:

```plaintext
axis1 = line {{11.0926 17.2649 3.27548}} {{-0.821539 0.123481 0.55662}}
axis2 = line {{5.90624 11.5912 -3.12062}} {{-0.821539 0.123481 0.55662}}
line distance = 10
point distance = 10
```

*The lines for axis1 and axis2 show the centroid, (the first { } in each line) and the direction vector (the second { } in each line) of the axes. Note that axis1 and axis2 have the same direction, indicating that they are parallel after execution of the fixorientation procedure. The line distance is the distance between two lines, and the point distance is the distance between two centroids. These two distances are the same, indicating that the centroids are lined up perpendicular to both axes at the predefined distance apart.*

6. Displaying the structures with the Insight modeling program

*The helix.cor file is the coordinate file written out after fixing the orientation. It should contain two helices parallel to each other and 10.0 Å apart.*

*You can display the structures in the helix.cor file with the Insight*
Tutorial—standalone mode

Start up the Insight program from the directory in which you ran the Discover program by entering `insightII` at the UNIX prompt. Read in the helix.cor file with the File/Import or Molecule/Get command, and use Geometrics/Vector command of the DeCipher module to create the axes of two helices by finding the least-squares fit line of the backbone atoms subset specified by model_name:*:ca,n,c.

Before doing this, you might also want to display the models in the helix.car file to show that the helices in the original file were randomly oriented.

7. Displaying all the configurations that were found

Then use the Trajectory pulldown of the Analysis module to display all configurations of the different orientations of the two helices:

- **Trajectory/Get**, to get the trajectory for the assembly called command HELIX
- **Trajectory/Animate**, to display all the conformations in sequence command

The trajectory file is in the helix.arc file.

8. Sorting the data to find the lowest-energy configurations

*Use the Spreadsheet functionality to read in the helix.tbl file and*
4. Standalone CDiscover

then sort the energies in ascending order.

Select the **Spreadsheet** icon, then select the **Open** command from the list that appears. Set the following parameter values:

- **File Type** = Graph
- **File Name** = helix.tbl
- **Object Name** = Sheet

Select **Execute**.

Grab a corner of the spreadsheet window with the cursor and enlarge it to have a clearer view.

Use the **Data/Sort** command to sort the data, by first highlighting the **Total energy** column and then setting the following parameters:

- **Extent Choice** = Column
- **Primary Key** = E0 (type it in or select the **Total energy** column by clicking the E column label)
- **Secondary Key** = None
- **Tertiary Key** = None
- **Descending Order** = Off

Select **Execute**.
Command summary—standalone mode

9. Displaying the lowest-energy conformations

Go back to the animation process and display the configurations that have the lowest energies. You need to load only those conformations, with the Get/Trajectory command. Set Selection Mode to Specified and enter the desired frame numbers in the Frame Spec parameter box, separated by commas.

10. Applying what you have learned

You could try writing your own script to reorient a three-helix system or a four-helix bundle.

Command summary—standalone mode

The Btcl language and its commands are documented in full in Btcl Language and Commands—Standalone Mode, where the commands are listed in alphabetical order. Below is a list of the Btcl commands according to their functions, followed by a brief description and the page on which the complete description starts.

Molecular operations and i/o

begin

Read in the model and choose the forcefield

molGeom

Get and set geometric properties of models

readFile

Read in a data file

reset

Reread the structural data files and forcefield
4. Standalone CDiscover

subStructure
List all atoms connected to an atom that are not also bonded to a second atom

writeFile
Write a data file

Calculations

atomMovability
Set and unset movability characteristics of atoms

cellParameter
Get and set cell parameters

dynamics
Perform molecular dynamics simulations

energy
Perform a single energy evaluation and calculate first and second derivatives

energyContribution
Pass energy and energy gradient contributions from user-defined Tcl routines to the CDiscover program

forcefield
Select a forcefield, change and scale the contributions from various energy terms

minimize
Refine the structure of a molecular system

rattle
Set up constraints in bonds, angles, or water fragments for a dynamics run

restraint
Create, delete, and scale restraints on relationships among atoms
Command summary—standalone mode

<table>
<thead>
<tr>
<th>Analysis and output</th>
<th>analyzeNonbond</th>
</tr>
</thead>
<tbody>
<tr>
<td>Calculate and examine repulsive, dispersive, and electrostatic energy components of individual atoms and total nonbond interaction energies between sets of atoms</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>diffraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Calculate the X-ray, neutron, or electron scattering pattern</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>discoverHistory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Manipulate the history information from a dynamics run</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control the amount of printing to the output file</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>print</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write out information for various system properties</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>vibrationalAnalysis</th>
</tr>
</thead>
<tbody>
<tr>
<td>Perform normal mode analysis</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Database operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>database</td>
</tr>
<tr>
<td>Perform operations on databases</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$dbHandle</th>
</tr>
</thead>
<tbody>
<tr>
<td>Access data within a database</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>object</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extract or print information from a Btcl object variable</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>pseudoAtom</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create, access, and update pseudoatoms</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>select</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create a list of row numbers from a hierachial database</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>subset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create or access subsets of objects in a database</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Geometry and math operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>geometry</td>
</tr>
<tr>
<td>Create, review, and manipulate geometry objects</td>
</tr>
</tbody>
</table>
4. Standalone CDiscover

vector
   Perform vector manipulation

Miscellaneous
help
   Print information about commands

Discover Interprocess Communication (IPC) commands
   Commands for interprocess control

peek
   Control the monitoring of iterative processes in minimization and dynamics calculations
The following sections describe the commands and programming language used in the standalone version of CDiscover:

- *MSI (Biosym) Tcl language syntax* (below) gives an overview of the CDiscover simulation language.
- *Conventions in documentation of Btcl commands* (page 139) explains the conventions and format used in the final section to document the Btcl commands.

The remainder of this chapter (starting on page 143) contains information on all the commands that may appear in an input command file for a CDiscover run (whether these commands are input via the Cerius\(^2\) Discover module or the Insight Discover module or whether the command file is one you directly write or edit). The commands are described in alphabetical order.

---

**MSI (Biosym) Tcl language syntax**

**General programming syntax**

Btcl is based on the public domain package Tcl written by Dr. John Ousterhout of the University of California, Berkeley. It is freely available and comes with an X windowing package called TK.

The general programming syntax for Btcl is essentially that of Tcl and, thus, it performs operations on character strings to produce results in the form of strings. Whenever arithmetic operations are performed, the strings are first converted into numbers. The result is converted back to a string.
A. Btcl Language and Commands—Standalone Mode

The description here is limited mainly to mathematical operations and program flow-control statements. For a full description of the list- and string-manipulation commands in Tcl, as well as its many other powerful features, please see Ousterhout (1994).

Variables and expressions

Assigning a variable value

The following two expressions can be used to assign a value to a variable (here, the variable \(a\)), either in a command input file or interactively:

\[
\text{set } a \ 3.4 \\
\text{$a = 3.4$}
\]

The second example is part of the MSI extension of Tcl. Entering these commands interactively causes Btcl to print the resulting value, 3.4.

To find the value of \(a\) you can use either:

Finding a value

\[
\text{set } a \\
\text{echo $a}
\]

Again, entering these expressions interactively causes the current value of \(a\) to be printed. If \(a\) is undefined, however, an error message is printed. This would happen if, for example, you entered the following at the BTCL prompt:

\[
\text{BTCL > set a "hello"}
\text{echo $a}
\text{BTCL > unset a}
\text{BTCL > set a}
\]

In general, whenever $a is encountered (where \(a\) could be any variable name), the string $a is replaced by its value. The reason echo $a is cited in the above example is because here, $a at the start of a new command line would have been interpreted as the command 3.4—which would have produced an error. This is seen more clearly in the following example script, where the commands are preceded by BTCL >, and the output is not:

\[
\text{BTCL > set a "echo"}
\text{echo}
\]
General programming syntax

BTCL > $a MSI
   MSI

Only the value resulting from the command processed last is printed:

BTCL > set a 3 ; $b = 4 ; set c "last set"
   last set

BTCL > echo $a $b $c
   3 4 last set

Semicolon to enter >1 command on a line

In this example, a semicolon (;) is used to separate distinct commands that are entered on the same line.

A command line is usually ended with a carriage return. However, a command can be continued onto a subsequent line:

Continuing a command over >1 lines

♦ When the carriage return is directly preceded by a backslash ( \ ). (That is, the backslash must be the last character on the line.)

♦ When quotes " " , brackets [ ] , or braces { } have not been fully matched.

For example:

BTCL > set a \n   (Almost
   a
   sentence...
   )

   Almost a sentence...

The last, and most important, difference between set a and $a = is that set can take only one value argument, but $a = can take any number of arguments and evaluate them as an expression. Consider the following example script:

BTCL > set a 3*4
   3*4

BTCL > $a = 3 * 4
   12
A. Btcl Language and Commands—Standalone Mode

It is important to note that the syntax $varName = something is valid only for numeric expressions. For example, $a = hello would give an error, because the expression evaluates to a string constant.

**Special characters and the expr and eval commands**

In the section Variables and expressions above, the special characters dollar ($), semicolon (;) and backslash (\) were introduced. The hash mark (#) is considered special only if it is the first character encountered on a new command line, in which case the remaining characters are ignored up to the next carriage return. The # is mainly useful for commenting scripts stored in files.

**Quoting spaces and other characters**

A space is always used to separate command arguments. If you want to have a space within a command argument, you must enclose the argument within quotes (" " ) or curly braces ( { }) or escape the space with a backslash (\ ). These methods are not all equivalent. The backslash has the highest priority, and only it can be used to quote any special character (even itself). Braces have the second-highest priority.

The difference between braces and quotes is that quotes can quote only spaces (and braces, see below). Consider the following examples, where the braces cause everything within them to be understood literally, but quotes allow the variable within them to be interpreted:

```plaintext
BTCL > set a 50
      50

BTCL > set b {The value of a is $a}
      The value of a is $a

BTCL > set b "The value of a is $a"
      The value of a is 50
```

Braces and quotes can be used to quote each other. When programming complex scripts in Btcl, it is important to remember these rules and that only the outermost quotation characters are significant.

For example:

```plaintext
BTCL > set b "($a"
      "$a"
```
Commands within commands

Square brackets ([ ]) are used to evaluate subcommands within another command. Variable substitution within brackets does not occur until the command script is executed, so that, in the following example, the variable \texttt{a} is set to 5 and not 3, as would occur if the $ were evaluated first:

\begin{verbatim}
BTCL > set b 3 ; set a "[set b 5; echo $b]"
5
\end{verbatim}

Subcommands

An alternative way to execute subscripts is to use the \texttt{eval} command. The \texttt{eval} command takes a single argument, as opposed to brackets, which contain a script. This difference reflects the way in which Btcl handles list elements and is shown in the following example:

\begin{verbatim}
BTCL > set a "set b 3.4"
set b 3.4
BTCL > eval $a
3.4
BTCL > \[$a\]
Error: invalid command name "set b 3.4"
\end{verbatim}

Evaluating mathematical expressions

The \texttt{expr} command is provided to evaluate mathematical expressions. \texttt{Expr} takes one or more arguments. If the expression is contained in one argument and contains spaces, it should be quoted. For example:

\begin{verbatim}
BTCL > echo 3 * $b = [expr "3 * $b"]
3 * 3.4 = 10.2
\end{verbatim}

Parentheses ( ) are special characters only when used in mathematical expressions, where they are used in the usual way to specify operational precedence. In Btcl the following commands are equivalent:

\begin{verbatim}
BTCL > set a [expr (3-1)*4]
8
BTCL > $a = (3-1)*4
8
\end{verbatim}
A. Btcl Language and Commands—Standalone Mode

Table 3 lists the remaining Btcl special characters and character combinations, which are used for logical and arithmetic expressions. Table 4 lists the mathematical functions available in Btcl.

**Table 3. Summary of operators allowed in Btcl**

These operators have the same syntax and behavior as in C. The FORTRAN-style operators (.lt., .or., etc.) that were allowed in DSL version 3.1 are not supported in Btcl.

<table>
<thead>
<tr>
<th>syntax</th>
<th>operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>-a</td>
<td>negative of a</td>
</tr>
<tr>
<td>!a</td>
<td>logical NOT: evaluates to 1 if a is zero, to 0 otherwise</td>
</tr>
<tr>
<td>~a</td>
<td>bit-wise complement</td>
</tr>
<tr>
<td>a+b</td>
<td>add a to b</td>
</tr>
<tr>
<td>a-b</td>
<td>subtract b from a</td>
</tr>
<tr>
<td>a*b</td>
<td>multiply a by b</td>
</tr>
<tr>
<td>a/b</td>
<td>divide a by b</td>
</tr>
<tr>
<td>a%b</td>
<td>remainder after dividing a by b</td>
</tr>
<tr>
<td>a&amp;b</td>
<td>bit-wise AND of a and b</td>
</tr>
<tr>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>a ^ b</td>
<td>bit-wise exclusive-OR of a and b</td>
</tr>
<tr>
<td>a==b</td>
<td>evaluates to 1 if a is equal to b, to 0 otherwise</td>
</tr>
<tr>
<td>a!=b</td>
<td>evaluates to 1 if a is not equal to b, to 0 otherwise</td>
</tr>
<tr>
<td>a&lt;=b</td>
<td>evaluates to 1 if a is less than or equal to b, to 0 otherwise</td>
</tr>
<tr>
<td>a&gt;=b</td>
<td>evaluates to 1 if a is greater than or equal to b, to 0 otherwise</td>
</tr>
<tr>
<td>a&lt;b</td>
<td>evaluates to 1 if a is less than b, to 0 otherwise</td>
</tr>
<tr>
<td>a&gt;b</td>
<td>evaluates to 1 if a is greater than b, to 0 otherwise</td>
</tr>
<tr>
<td>a&amp;&amp;b</td>
<td>logical AND: evaluates to 1 if a and b are nonzero, to 0 otherwise</td>
</tr>
<tr>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>a?b:c</td>
<td>choice: if a is nonzero then b, else c</td>
</tr>
</tbody>
</table>

**Control statements**

**Conditional statements: if, else, and elseif** Below are valid structures for if control structures:

```tcl
if ($i == "help") { echo Sorry, no help is on-line }
```

128 C Discover/September 1997
Table 4. Summary of functions available in Btcl

The functions have the same syntax and behavior as in C. The \(^{(\text{power})}\) operator in DSL version 3.1 is no longer supported and has been replaced by the \(\text{pow}(x,y)\) function.

<table>
<thead>
<tr>
<th>syntax</th>
<th>operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{cos}(x))</td>
<td>returns the cosine of (x) ((x\ \text{in radians}))</td>
</tr>
<tr>
<td>(\text{sin}(x))</td>
<td>returns the sine of (x) ((x\ \text{in radians}))</td>
</tr>
<tr>
<td>(\text{tan}(x))</td>
<td>returns the tangent of (x) ((x\ \text{in radians}))</td>
</tr>
<tr>
<td>(\text{acos}(x))</td>
<td>returns the principal value of arccosine of (x) ((x\ \text{is between 0 and 1 and the result is between 0 and } \pi))</td>
</tr>
<tr>
<td>(\text{asin}(x))</td>
<td>returns the principal value of arcsine of (x) ((x\ \text{is between 0 and 1 and the result is between } -\pi/2 \text{ and } \pi/2))</td>
</tr>
<tr>
<td>(\text{atan}(x))</td>
<td>returns the principal value of arctangent of (x) ((x\ \text{is between 0 and } \pi))</td>
</tr>
<tr>
<td>(\text{atan2}(x,y))</td>
<td>returns the principal value of arctangent (y/x), where the signs of (x) and (y) determine the quadrant of return value ((\text{between } -\pi \text{ and } \pi))</td>
</tr>
<tr>
<td>(\text{cosh}(x))</td>
<td>returns the hyperbolic cosine of (x)</td>
</tr>
<tr>
<td>(\text{sinh}(x))</td>
<td>returns the hyperbolic sine of (x)</td>
</tr>
<tr>
<td>(\text{tanh}(x))</td>
<td>returns the hyperbolic tangent of (x)</td>
</tr>
<tr>
<td>(\text{exp}(x))</td>
<td>returns the exponential function value of (x)</td>
</tr>
<tr>
<td>(\text{log}(x))</td>
<td>returns the natural logarithm of (x) ((x\ \text{is positive}))</td>
</tr>
<tr>
<td>(\text{log10}(x))</td>
<td>returns the base-ten logarithm of (x) ((x\ \text{is positive}))</td>
</tr>
<tr>
<td>(\text{sqrt}(x))</td>
<td>returns the square root of (x) ((x\ \text{is positive}))</td>
</tr>
<tr>
<td>(\text{hypot}(x,y))</td>
<td>returns (\sqrt{x^2+y^2})</td>
</tr>
<tr>
<td>(\text{abs}(x))</td>
<td>returns the absolute value of (x)</td>
</tr>
<tr>
<td>(\text{int}(x))</td>
<td>returns the truncated integer value of (x)</td>
</tr>
<tr>
<td>(\text{double}(x))</td>
<td>returns the floating-point value of (x)</td>
</tr>
<tr>
<td>(\text{ceil}(x))</td>
<td>returns the smallest integer not less than (x)</td>
</tr>
<tr>
<td>(\text{floor}(x))</td>
<td>returns the largest integer not greater than (x)</td>
</tr>
<tr>
<td>(\text{fmod}(x,y))</td>
<td>returns the floating point remainder of (x) divided by (y) ((\text{with the same sign as } x))</td>
</tr>
</tbody>
</table>

if \{\$a > 10\} {
    set a 10
}
A. Btcl Language and Commands—Standalone Mode

```tcl
if {$a > 20} {
    set a 20
} elseif {$a < 0} {
    set a 0
} else {
    $a = $a + $b
}
```

Quoting strings

It is also important to note (as in the examples above) that, when variables are compared with strings, the strings are quoted, since this is how Tcl distinguishes strings from invalid numeric constants. Note also that `elseif` is always one word.

Although Btcl does not generally pay attention to how your script is structured, you should remember that each command line is terminated by a carriage return. In the last example an open brace ({) is placed after the `if` test to escape the carriage return. The following example lacks this brace and would produce an error, since Tcl does not find a second argument for the `if` command:

```tcl
if {$a > 10} {
    set a 10
}
```

Conditional looping statements: while, for, and foreach

The Btcl `while` command takes two arguments, the second argument (the body) being evaluated repeatedly so long as the first argument (the test) evaluates to 1. The following example script prints all even numbers between 0 and 24:

```
$i = 0
while  {($i < 25)} {
    echo $i
    $i = $i+2
}
```

The Btcl `for` command is similar to the `while` command except that it takes two extra arguments: one that is evaluated before the test and body arguments and one that is evaluated after the body and before the test arguments. By using the `for` command, the previous example can be written more concisely:

```tcl
for {$i = 0} {$i < 25} {incr i 2} {
    echo $i
}
```
In this example, the statement \( I = i + 2 \) is replaced by the more efficient Btcl command `incr i 2`, where the `incr` command is used to increment the variable `i` by 2. If the last argument to `incr` is omitted, the increment value is 1.

The Btcl `foreach` command is used to loop through elements of a list. The simplest and most common form of list in Btcl is a string of words separated by spaces.

The following example shows how the `foreach` command functions:

```tcl
BTCL > $a = "first second third" ; foreach i $a {echo $i}
first
second
third
```

**Escaping from loops**

You may want to skip the remainder of the body script or abandon a loop under certain conditions. The subloop commands `continue` and `break` enable this. For example:

```tcl
BTCL > eval {
    set a {4.0 1.25 10000 1e+90 0.0 20.0}
    foreach i $a {
        if ($i > 1e+80) {
            echo reciprocal = 0.0
            continue
        } elseif ($i == 0.0) {
            echo cannot take the reciprocal of zero
            break
        } else {
            echo reciprocal of $i = [expr 1.0/$i]
        }
    }
    echo ok
}
```

```
reciprocal of 4.0 = 0.25
reciprocal of 1.25 = 0.8
reciprocal of 10000 = 0.0001
reciprocal = 0.0
cannot take the reciprocal of zero
ok
```
A. Btcl Language and Commands—Standalone Mode

**Integer arithmetic**

Here, it was necessary to use 1.0/$i$ to generate the reciprocal value. This is because Btcl uses integer arithmetic by default, so that $1/$i would have given the result 0 for 1/10000.

**Btcl procedures**

**Define your own**

One of the most useful features of Btcl is that you can quickly define and use your own procedures. Most Btcl procedures have the form:

```
procedure_name argument_1 argument_2 ... argument_n
```

The arguments here are positional and completely general, being distinguished only by separating spaces. You can create a procedure by using the `proc` command which takes three arguments:

```
proc procedure_name procedure_argument_list
procedure_body
```

A simple procedure to add two numbers could be defined as:

```
BTCL > proc add {a b} {expr $a+$b}
```

**Using your defined procedures**

This procedure can then be used immediately by entering, for example:

```
BTCL > add 3 4
7
```

**Overwriting existing procedures**

If a procedure already exists with the name `add`, then the new procedure replaces it. For example, you could define your own version of the `for` command:

```
BTCL > proc for {init test every body} {
    eval $init
    while $test {eval $body ; eval $every}
}
```

Note that this simulation of the `for` statement is not very good, since the `break` and `continue` statements are not processed, and variables inside the `body` variable are only local to this procedure (which might be desirable).

**Example procedure—simulated annealing**

An example is the procedure `annealed_dynamics`, defined below, which could be used to repeatedly equilibrate a system (set up outside the procedure) over a range of target temperatures:
proc annealed_dynamics {temp_start temp_final iterations num_anneals} {
    if {\$num_anneals <= 0} return
    if {\$num_anneals == 1} {
        \$T_start = \$temp_final
        \$T_scale = 0
    } else {
        \$T_start = \$temp_start
        \$T_scale = (\$temp_final - \$temp_start) / (\$num_anneals - 1)
    }
    for {\$i = 0} {\$i < \$num_anneals} {incr \$i} {
        \$temp = \$T_start + \$i * \$T_scale
        dynamics iteration = \$iterations temperature = \$temp
    }
}

Global and local variables

The arguments to a procedure are read-only, and any variables changed or created (e.g., \$i = 0) are local. However, global variables can be accessed by using the global declarator. You can also use the optional return statement to return a result that was not the last result generated. The following example shows these capabilities:

```
BTCL > set world 0 ; set i 0
0
BTCL > proc example {a b} {
    global world
    set i [expr \$world*$a+$b]
    incr world 1
    return \$i
}

BTCL > example 3 4
4

BTCL > example 3 4
7

BTCL > echo \$i
0

BTCL > echo \$world
2
```

Notice that here, the variables world and i were set at the global level, but only world is affected by the procedure, because of the global declarator.
A. Btcl Language and Commands—Standalone Mode

Learning more about procedures

It is also possible to create Btcl (Tcl) procedures, like the echo command, that take any number of arguments. Since this requires that you know more about Tcl’s list-handling capabilities, please refer to the Tcl documentation (Ousterhout 1994).

Btcl commands and utilities

Btcl commands are used to set up and control CDIscov'e simulation runs. Their use is more complicated than the usual Btcl procedures, in that their arguments may be specified by position or by name—the actual format depending on the particular command. A full list of Btcl commands starts on page 143.

Most Btcl commands have named-argument parameters and the syntax:

\[ \text{command parameter\_path\_name} = \text{name parameter\_path\_name} = \text{value} \]

Here parameter\_path\_name is usually just the name of particular parameter, for example:

analysis method = mean\_square\_displacement subset = "he"

Keyword paths

Some command parameters require that a keyword path be included to access a particular parameter, for example:

minimize cg method = polak cg convergence = 5.0 ad convergence = 400.0

Notice that the parameter name convergence is used twice, but, because of the path tree mechanism, cg convergence and newton convergence are different parameters. A path may consist of more than one keyword, for example:

forcefield nonbond coulomb cutoff = 3.4

Path trees

The full pathname is not always required for parameters in the same path tree. For example, the following two examples are equivalent:

minimize newton method = truncated newton convergence = 0.05 newton line\_s = 0.1
minimize newton method = truncated convergence = 0.05 line\_search\_precision = 0.1

All keywords are treated as path specifiers until an assignment operator = or the end of the command line is encountered. If the next word is not a path branch from (or parameter of) the current
General programming syntax

path, then the path is traversed backwards until the parameter name or path keyword is found. For example, the command:

```bash
forcefield scale nonbond use_cutoffs = True
```

does not produce an error and assigns a value to `nonbond use_cutoffs` even though no assignment was made at the `scale` path level.

Once all parameter setting has been processed from the command line, with no anomalous parameter names or values, the command (e.g., `forcefield`) is executed. Any parameters that were not set directly on the command line (or indirectly as a consequence) assume their default values.

Changing default parameter values

It is possible to change the default parameters using the Btcl utility command `cdlConfig`. For example, the previous command could have been set up and performed by entering:

```bash
BTCL > cdlConfig minimize newton method = truncated \n  convergence = 0.05 line_search_precision = 0.1

0.1

BTCL > minimize
```

Notice that the `cdlConfig` command returns a value—actually the value of the last parameter that was set. Whenever a `minimize` command is invoked subsequently, the default values are the same as those set by the `cdlConfig` utility. Here, `newton method` takes the value `truncated` unless it is respecified for a command, for example:

```bash
minimize ... newton method = bfgs ...
```

or unless another `cdlConfig` command is used.

Positional arguments

Some Btcl commands take positional arguments which are then mapped onto named parameters. The following command:

```bash
BTCL > readFile archive frame = 5
```

reads an archive file into the system for frame number 5. Here, the keyword `archive` is actually an enum value for the `readFile` parameter `filetype`, even though the `filetype` parameter is not specified here by name. However, if you want to set the default for
A. Btcl Language and Commands—Standalone Mode

the filetype, the *filetype* parameter must be referenced by name. Thus, the equivalent *cdlConfig* statement would be:

```
BTCL >   cdlConfig filetype = archive frame = 5
```

```
BTCL >   readFile
```

**Storing default values**

*cdlGather* is a utility command that is complementary to *cdlConfig*. This command collects all the parameter names and their current default values together in one string, for example:

```
BTCL >   cdlGather begin
        begin coord = {} topol = {} forcef = {}
```

Its main use is for preserving the current default values of a command. It is particularly useful in procedures and can also be used to store the current default parameters for a particular parameter path, for example:

```
BTCL >   cdlGather minimize newton
```

```
BTCL >   set Newton [cdlGather minimize newton]
```

```
BTCL >   cdlConfig minimize newton method = bfgs ...
```

```
... ...
```

```
BTCL >   cdlConfig $Newton
```

**Resetting the original defaults**

To reset a command to its original default settings, you can use the Btcl utility *cdlSetDefaults*, for example:

```
cdlSetDefaults minimize
```

**Viewing default values**

To view the current default setting for a particular Btcl command parameter, you can use the Btcl utility *cdlFetch*. Unlike *cdlConfig* and the Btcl commands themselves, *cdlFetch* operates only on a single parameter path and has the general syntax:

```
cdlFetch target parameter_path_name
```

Here, the *target* argument can be one of several keywords, the most useful of which is *value*, which retrieves the current default value, for example:

```
BTCL >   cdlFetch value minimize newton method truncated
```
Btcl command parameter types

The Btcl parameter values themselves depend on the particular parameter to be assigned. There are five types of parameters:

- **real**: takes any numeric value.
- **integer**: takes any integer value (rounded down towards zero).
- **string**: takes any character sequence up to a space or enclosed between a pair of quotes (" " ) or braces ( { } ).
- **enum**: takes a string that may be only one of a list of allowed patterns. Enum values are case-insensitive and some can accept abbreviations.
- **Boolean**: takes a string that maps to the values 1 or 0. Values that map to 1 are TRUE, T, YES, Y, ON, and 1. Values that map to 0 are FALSE, F, NO, N, OFF, and 0. The value-strings read for Boolean types are case-insensitive.

Specifying Boolean parameters

With Boolean-type parameters, an acceptable shorthand for \texttt{b\_param = TRUE} is \texttt{+b\_param} and for \texttt{b\_param = FALSE} is \texttt{-b\_param}. Note that the mere presence of a Boolean parameter is not enough to set it to TRUE.

Arrays

Some parameters can be fixed or dynamic-length arrays of these types. For arrays, a list of values is required, for example, the \texttt{stress} parameter of the \texttt{dynamics} command, which can be used to set the stress parameters \texttt{sxx}, \texttt{syy}, etc.:

\begin{verbatim}
dynamics temperature = 300 time = 500 stress = \{0.5 0.5 0.5 0.0 0.0 0.0\}
\end{verbatim}

Subcommands

Some Btcl commands allow you to specify subcommands that are executed at specified instances from within a Btcl command. For example, assuming that a model has already been set up:

\begin{verbatim}
BTCL > minimize iteration = 50 method = cascade newton method = bfgs \execute +before command = "echo hello from minimizer" \execute every = 2 command = "print output +energy_summary +nonbond_energy"execute every = 5 command = "print archive +velocity"execute +after command = "echo good-bye from minimizer"
\end{verbatim}
A. Btcl Language and Commands—Standalone Mode

This command would set up a minimization run using the cascade method with BFGS Newton minimization over a maximum of 50 iterations. In addition, “hello from minimizer” and good-bye from minimizer” would be printed at the start and end of the minimization, respectively, with instantaneous energy summary and non-bond energy values being printed every other step and velocities written to an archive file (run_name.arc) every 5 steps.

Global Btcl and Tcl variables and environment variables

Global Btcl variables

Btcl maintains some global variables that are used mainly as defaults for Btcl commands. The two most important are:

The run name

♦ PROJECT—Used to store the current project name (also indicated here as run_name), which is used for filename completion. You may want to set this in interactive mode to test source files, for example:

```
BTCL > set PROJECT my_project1 ; source my_project1
BTCL > set PROJECT my_project2 ; source my_project2
```

In this example, the Btcl source codes are saved in files without the .inp filename extension.

The random seed

♦ randomSeed—This special variable is used to set the random seed for the random-number generator used by the Discover program. Setting this global variable actually causes the random number generator to be reset to this value and causes its value to be echoed, for example:

```
BTCL > set randomSeed 35
randomSeed is set to 35
BTCL > set randomSeed
35
```

```
BTCL > set a [echo $randomSeed]
35
```

When the Discover program starts, the value of randomSeed is set randomly (according to the time of day), and its value is printed. This is so that the same random seed can be used again in later Discover runs, if desired. Setting the randomSeed variable is equiva-
lent to using the control command of version 3.1 of DSL, a command that is no longer supported.

**Global Tcl variables: the precision**

For commands that return the result of a numerical evaluation, the precision is controlled by the `tcl_precision` variable. The maximum value is 17. This value also has the special characteristic that any value converted from an IEEE floating-point value to a string and back is guaranteed to have the same binary representation before and after the conversion.

**Environment variables**

The Discover program makes use of several environment variables. These are easily accessed by Btcl by using the `$env()` array.

**The current forcefield**

For example, the `$FORCEFIELD` environment variable can be accessed by:

```
BTCL > echo $env(FORCEFIELD)
```

```
cvff
```

**Exiting a Btcl session**

If you want to end a Btcl session without starting a Discover run, enter `exit`, `quit`, or `<Ctrl>` d at the Btcl prompt.

---

# Conventions in documentation of Btcl commands

## Font conventions for keywords

Several type style conventions are used to distinguish different kinds of keywords. For example:

```
command keyword = value ...
```

**Case-sensitive keywords**

Words (or letters) in **bold** indicate the names of commands and must be typed exactly as shown. Note that all commands and parameter names are case-sensitive and must be typed as shown.

Words in *italics* indicate options that must be replaced with appropriate text. These may be items that must be replaced by specific text (for example, `path`, `keyword`, `filetype`, `operation`), items to be
A. Tcl Language and Commands—Standalone Mode

replaced by values of the appropriate type (for example, value, arg),
or user-supplied variable names (for example, varName).

*Italics* are also used to indicate the names of internal Discover data-
bases and their named parts.

### Assignment and types of values

The syntax *keyword = value* is used for many command options,
where *value* is used to change the default value of an option.

**Case-sensitive keywords**
The lists of *keywords* in the Syntax sections explain the character
sequences that may be used as keywords and their values for each
command, including occasional synonymous keywords that may
be used interchangeably. The keywords are *case-sensitive* and must
be typed exactly as written.

**Truncated keywords**
Some of the listed keywords contain an asterisk (*) within the
word. This indicates that the letters to the left of the asterisk must
be typed exactly as written and that the word can be truncated at
any point to the right of the asterisk. However, any characters that
are typed must be exactly as written in the table. (Note that the
asterisk itself is used only for documentation and must not be
typed as part of the keyword.)

**Types of values and case sensitivity**
The *values* appropriate to each keyword are also specified in the
table. Their type is indicated if it is real, integer, string, or Boolean;
for enumerated constants, the allowed values are specified.

♦ *Real* numbers can be represented in floating-point or exponential
form. Both *real* and *integer* numbers are actually input as
strings. The Discover program converts the string to the appro-
priate numerical form, depending on the context. Numbers are
used for model numbers, force constants, etc.

♦ *Strings* need to be enclosed in quotation marks or braces only if
they contain spaces. These are *case-sensitive*. Character strings
are typically used for file names, residue names, atom names,
etc. *Residue names must have the same case (or mixture of cases) as
they have in the residue library or Cartesian coordinate file.*

♦ *Boolean* values are used to select items that can be present or
absent; they can act as toggles to activate and deactivate some
options; and they are used for setting some variables to *true* or
false. Boolean values are case-insensitive (e.g., ON, on, and On are equivalent).

* Enumerated parameters (enums) take specific string constants (listed in the keyword tables) and are used for selecting mutually exclusive options. For example, in the minimize command, the method can be steepest, conjugate, newton, or cascade, and nothing else. Enum values are case-insensitive.

Where a value is expected, an appropriate variable or expression may also be used.

**Punctuation conventions**

Optional items in the syntax line are enclosed between question marks (?) . Ellipses (...) indicate that the preceding item can be repeated. A vertical line ( | ) separates alternative keywords. These symbols themselves are used for syntax-documentation only and should not be included in the real command.
A. Btcl Language and Commands—Standalone Mode
Command descriptions

**analyzeNonbond**

**Purpose**

The `analyzeNonbond` command allows the calculation and examination of repulsive, dispersive, and electrostatic energy components of individual atoms and the total nonbond interaction energies and forces between sets of atoms.

**Syntax**

```
analyzeNonbond operation ?arg1 arg2? ...
```

<table>
<thead>
<tr>
<th>Operation</th>
<th>Arguments</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>create</code></td>
<td><code>setName1</code> <code>setName2</code> <code>-name</code> <code>tableSuffix</code>?</td>
<td>Create the tables <code>NonbondInteractionEnergy</code>- and <code>NonbondInteractionForce</code>- in the energy database. If <code>tableSuffix</code> is given for this pair of sets, it is appended to the above table names. <code>tableSuffix</code> is required if there are more than one pair of sets, i.e., the <code>create</code> command is used more than once. The two sets may be the same or different. Note that an atom can appear only once in a set.</td>
</tr>
<tr>
<td><code>delete</code></td>
<td><code>-name tableSuffix</code>?</td>
<td>Delete the tables <code>NonbondInteractionEnergy</code>- and <code>NonbondInteractionForce</code>- and any other related information from the energy database and the system database. <code>tableSuffix</code> is required only if it is used in the <code>analyzeNonbond create</code> command.</td>
</tr>
</tbody>
</table>
A. Btcl Language and Commands—Standalone Mode - Continued

<table>
<thead>
<tr>
<th>Operation</th>
<th>Arguments</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>on</strong></td>
<td></td>
<td>Set the proper flags in the Energy database to turn on nonbond energy analyses. After an analyzeNonbond on command, nonbond energy analyses are done for every energy evaluation until an analyzeNonbond off command is issued. Note that analyzeNonbond create must have been used prior to this operation.</td>
</tr>
<tr>
<td><strong>off</strong></td>
<td></td>
<td>Turn off nonbond energy analyses.</td>
</tr>
<tr>
<td><strong>-row</strong></td>
<td>rowIndex varName, -rowIndex varName ?-name tableSuffix?</td>
<td>Send a row in the table NonbondInteractionEnergy in the energy database to an object array varName. analyzeNonbond create and analyzeNonbond on must have been used prior to this operation. If tableSuffix is not used in the create operation, it should not be used here. The row indices range from 1 to the number of groups in set1. Additionally, for the extracted results to make sense, at least one energy evaluation must have been performed.</td>
</tr>
<tr>
<td><strong>-column</strong></td>
<td>columnIndex varName, -columnIndex varName ?-name tableSuffix?</td>
<td>Similar to the analyzeNonbond -row command, but sends a column in the table NonbondInteractionEnergy in the energy database to an object array varName. The column indices range from 1 to the number of groups in set2 plus one. The last column contains the interaction energies between groups of set1 and all atoms that are not in set2.</td>
</tr>
<tr>
<td><strong>-element</strong></td>
<td>rowIndex columnIndex varName, -element rowIndex columnIndex varName ?-name tableSuffix?</td>
<td>Similar to the analyzeNonbond -row command, but sends an element in the table NonbondInteractionEnergy in the energy database to an object array varName.</td>
</tr>
</tbody>
</table>

**Description**

**Background**

The per-atom nonbond energy is the nonbond interaction energy between an atom and all the other atoms in the system. It is decomposed into repulsive, dispersive, and electrostatic components. The sum of the per-atom energies over all atoms gives the total nonbond energy.

The nonbond interaction energies are the energies between groups of atoms in two separate sets. For example, if there are M groups of atoms in set1 and N groups of atoms in set2, there are (M x (N + 1)) nonbond interaction energy terms. The interactions are organized as the rows (groups of set1) and columns (groups of set2) of a matrix. The extra column represents the interaction of each of the groups in set1 with a special group which is the complement of set2. That is, all the atoms not in set2 are represented in the last column of the matrix. A group in a set may be a model, a monomer, or an
arbitrary group of atoms in the system. Note that the per-atom nonbond energies may also be obtained as a special case of the nonbond interaction energy calculation. The nonbond interaction energy calculation does not separate repulsive, dispersive, and electrostatic components, as does the per-atom nonbond energy calculation.

The nonbond interaction forces between the same groups of atoms are calculated similar to the nonbond interaction energies.

Due to the nature of the Ewald sum and cell multipole methods, nonbond interaction energies cannot be calculated. When these methods are used, the per-atom energy is calculated by using the Ewald sum or cell multipole method, and the nonbond interaction energy is calculated using the group-based method. You can specify cutoffs for the group-based method of nonbond analysis. A large cutoff in the group-based method may give reasonably accurate energies compared with the Ewald sum and the cell multipole methods.

Notes

The setName arguments following the create operation represent object arrays containing grouped rids (row identifiers, see object keyword, page 217) which reference atoms in the MainCell/Atom table of the System database. Sets may be created by using the select or $dbHandle select commands.

Each create operation creates two columns of flags indicating the grouping of the selected atoms in the MainCell/Atom table of the System database. The flags are integers ranging from 0 to the number of groups. In the column corresponding to set1, 0 indicates the atoms that are not selected. In the column corresponding to set2, the largest number (i.e., the number of groups) indicates the atoms that are not in set2.

The create operation then creates the table NonbondInteractionTable in the System database. NonbondInteractionTable contains the name of the pair, the name of each set in the pair, the corresponding interaction energy and force table names, and the size of the energy and force tables. Each row in NonbondInteractionTable defines a collection of nonbond interactions. The names of the sets are used to locate the flag columns in the MainCell/Atom table of the System database.

The create operation also creates the interaction energy and force tables based on the optional tableSuffix given above. These tables
generally contain rectangular matrices. The rows are labeled by the groups in the first set; columns are labeled by the groups in the second set. Additionally, the last column contains nonbond contributions from any atoms not in set2. The table entries are initialized to zero.

The on operation creates a row (if it does not already exist) in the Global table of the System database and sets the entry name to AnalyzeNonbondOn and the corresponding value to true. The off operation sets its value to false. The Discover program’s energy evaluation code looks at this flag and decides whether or not to do nonbond energy analyses. The nonbond energy analyses are done at every energy evaluation after the on operation, until the off operation is encountered.

Performance is slowed when nonbond analysis is requested. It is a good idea to do nonbond analyses only when necessary, e.g., every certain number of steps of dynamics or minimization.

For your convenience, -row, -column, and -element operations are also supplied. These operations send a row or a column or an element in the nonbond interaction energy table to an object array, which can be printed out by using the object print command. Computations on the vector of energies can be performed using the vector command.

**Example 1**

```
BTCL > database handle dbSystem System.
BTCL > database handle dbEnergy Energy.
BTCL > $dbSystem select -regexp {^C ^H} Atom.Name set1
BTCL > select set2 "*:*:H*
BTCL > analyzeNonbond create $set1 $set2
BTCL > database print $dbSystem
BTCL > database print $dbEnergy
BTCL > analyzeNonbond on
BTCL > energy
BTCL > analyzeNonbond off
```
In this example, two database handles are created first. Then two sets are created by using the `select` command. There are two groups in the first set and only one group in the second. The two groups of set1 consist of all atoms with names beginning with “C” and all atoms with names beginning with “H”, respectively. Set2 contains all atoms beginning with “H”. The `analyzeNonbond` command creates two columns of flags in the `MainCell/Atom` table of the `System` database as well as nonbond interaction tables in the `Energy` database. Since no `-name` flag was used, the name of the nonbond interaction energy table will be `NonbondInteraction`. These columns and tables are printed out by the `database print` command. The nonbond analysis flag is turned on and off before and after an energy calculation. The per-atom energies for the repulsive, dispersive, and electrostatic components and the total nonbond interaction energies are printed by the last `database print` command.

Example 2

```plaintext
BTCL >  database handle dbSystem System.
BTCL >  database handle dbEnergy Energy.
BTCL >  $dbSystem select -regexp {^C ^H} Atom.Name set1
BTCL >  select set2 "*::*:H*"
BTCL >  select set3 "*::*:N*"
BTCL >  $dbSystem select -regexp {.*} Atom.Name set4
BTCL >  analyzeNonbond create $set1 $set1 -name i
BTCL >  analyzeNonbond create $set1 $set2 -name j
BTCL >  analyzeNonbond create $set2 $set3 -name k
BTCL >  analyzeNonbond create $set3 $set4 -name l
BTCL >  database print $dbSystem
BTCL >  database print $dbEnergy
BTCL >  analyzeNonbond on
BTCL >  energy
```
Example 3

BTCL > database handle dbSystem System.
BTCL > database handle dbEnergy Energy.
BTCL > set bindingList (57,99,189,190,195)
BTCL > select binding "*:(bindingList:Atom;*)"
BTCL > select bulk "*:!(bindingList:Atom;*)"
BTCL > analyzeNonbond create $binding $bulk
BTCL > analyzeNonbond on
BTCL > energy print energies = 1
BTCL > database print $dbEnergy

This example first creates a list of monomer numbers in the system and then creates two sets, called binding and bulk. The binding set contains atoms that belong to the monomers in the list, and the bulk set contains atoms that do not belong to the monomers in the list. The nonbond interaction energy between the binding group and the bulk group will be calculated.
Example 4

BTCL > database handle dbEnergy Energy.
BTCL > $dbEnergy get repulsive_energy Atom.Repulsive
BTCL > object repulsive_energy print

In this example, an energy database handle called dbEnergy is created first. Then the `$dbEnergy get` command is used to get the per-atom nonbond repulsive energy from the `Atom` table of the energy database into an object array called `repulsive_energy`. Finally, the `object print` command is used to print out the object array called `repulsive_energy`.

atomMovability

Purpose

The `atomMovability` command is used to set and unset movability characteristics of atoms.

Syntax

atomMovability operation arg1 ?arg2 ...?

Command operations:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Arguments</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>set</td>
<td><code>type name</code></td>
<td>Set movability characteristic <code>type</code> for the atom set specified by <code>spec</code>. The values of <code>type</code> and <code>spec</code> are associated with <code>name</code> for later reference.</td>
</tr>
<tr>
<td>unset</td>
<td><code>name</code></td>
<td>Unset movability characteristic for the atom set associated with <code>name</code> or unset all movability characteristics if value of <code>name</code> is <code>-all</code>. When movability characteristic is unset, it reverts to the default value (<code>movable</code>).</td>
</tr>
</tbody>
</table>
A. Btcl Language and Commands—Standalone Mode - Continued

Command arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>set</td>
<td>Legal values are <strong>movable</strong>, <strong>fixed</strong>, and <strong>excluded</strong>. Movable atoms participate fully in system motions and interactions. Fixed atoms exert forces on other atoms, but do not move. Excluded atoms have no effect on other atoms and do not move.</td>
</tr>
<tr>
<td>name</td>
<td>set/unset</td>
<td>Arbitrary string selected by user.</td>
</tr>
<tr>
<td>spec</td>
<td>set</td>
<td>General atom or subset spec.</td>
</tr>
</tbody>
</table>

**Description**

Atom movability characteristics specified with the **atomMovability** command are applied in the order in which they are specified. Thus, an initial movability specification for an atom can be overridden by a subsequent command. Unsetting the subsequent specification restores the effect of the initial command.

The **atomMovability** command is actually a Tcl procedure (defined in $BIOSYM/data/discover/utility/atomMovability.tcl) that employs the set command to store the specification data into Tcl associative arrays.

**Examples**

These examples show the differences between atom movability exclusion and energy calculation exclusion in calculations. If an atom is excluded using the atomMovability command, all energies (bond, angle, nonbond) involving that atom are excluded. You can run these examples by copying them into a command input file and adding these lines before the example lines in the input file:

```tcl
#BIOSYM btcl
set PROJECT acenm
begin
forcefield nonbond vdw summation_method = group_based
```

You need to copy the acenm.car and acenm.mdf files from $BIOSYM/gifts/discover/tcl into the directory that contains your command input file.
Example 1—Allow all atoms to move

```plaintext
energy print energies = 1
database handle ener_h Energy.
$ener_h select "Internal" Values.Name e1; $ener_h get int1 Values.Value $e1
$ener_h select "Bond" Values.Name e1; $ener_h get b1 Values.Value $e1
$ener_h select "Angle" Values.Name e1; $ener_h get a1 Values.Value $e1
$ener_h select "Nonbond" Values.Name e1; $ener_h get nb1 Values.Value $e1
```

Example 2—Prevent one atom from moving and ignore its energy contribution

```plaintext
atomMovability set excluded ex1 "ACENM:ACE_1:HA1"
energy print energies = 1
database handle ener_h Energy.
$ener_h select "Internal" Values.Name e1; $ener_h get int2 Values.Value $e1
$ener_h select "Bond" Values.Name e1; $ener_h get b2 Values.Value $e1
$ener_h select "Angle" Values.Name e1; $ener_h get a2 Values.Value $e1
$ener_h select "Nonbond" Values.Name e1; $ener_h get nb2 Values.Value $e1
atomMovability unset ex1
```

If you run this example, note that both the nonbond and internal energies are different.

Example 3—Prevent one atom from moving but include its energy contribution

```plaintext
atomMovability set fixed ex1 "ACENM:ACE_1:HA1"
energy print energies = 1
database handle ener_h Energy.
$ener_h select "Internal" Values.Name e1; $ener_h get int2 Values.Value $e1
$ener_h select "Bond" Values.Name e1; $ener_h get b2 Values.Value $e1
$ener_h select "Angle" Values.Name e1; $ener_h get a2 Values.Value $e1
$ener_h select "Nonbond" Values.Name e1; $ener_h get nb2 Values.Value $e1
atomMovability unset ex1
```

If you run this example, the energies should be the same as for Example 1.
begin

Purpose

The `begin` command is placed at the start of the main commands section of the command input file. It reads in the structural data files (.car and .mdf) and the forcefield.

Syntax

```
begin ?keyword = value?...
```

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Values</th>
<th>Default</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>coordinate</td>
<td>string</td>
<td><code>run_name.car</code></td>
<td>Name of coordinate data file. If none given, defaults to <code>run_name.car</code>.</td>
</tr>
<tr>
<td>or car</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>topology</td>
<td>string</td>
<td><code>run_name.mdf</code></td>
<td>Name of structural data file that defines connections, etc. If none given, defaults to <code>run_name.mdf</code>.</td>
</tr>
<tr>
<td>or mdf</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>forcefield</td>
<td>string</td>
<td><code>$FORCEFIELD</code></td>
<td>Name of forcefield file. If none given, defaults to value of the <code>$FORCEFIELD</code> environment variable.</td>
</tr>
</tbody>
</table>

Description

The `begin` command is the first Discover command to be executed in the input file. It reads in the model and the forcefield parameters. It does not set up the energy expression or the nonbond list. This is done the first time an energy calculation is requested (e.g., by a `minimize` command).

The typical use of the `begin` command does not specify any filenames. The structural data are then picked up from the .car and .mdf files corresponding to the current run, and the forcefield is specified by the environmental variable `$FORCEFIELD`.

Example 1

```
begin
```

This example of the `begin` command uses the default structural data and forcefield file assignments.
Example 2

```
begin forcefield = cvff
```

This example of the `begin` command uses the default structural data files but explicitly reads the CVFF forcefield.

---

**cellParameter**

**Purpose**

The `cellParameter` command is a Tcl function to get and set the cell parameters of a periodic model.

**Syntax**

Getting cell parameters:

```
cellParameter
```

Setting cell parameters:

```
cellParameter -option? a b c alpha beta gamma
```

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Values</th>
<th>Default</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-option</code></td>
<td><code>-atomAffine</code> <code>-atomAffine</code></td>
<td>The fractional coordinates of all atoms in the cell are unchanged.</td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>-moleculeAffine</code></td>
<td>The fractional coordinates of the center of mass of each molecule in the cell are unchanged.</td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>-noAffine</code></td>
<td>The Cartesian coordinates of all atoms before and after the cell distortion remain the same.</td>
<td></td>
</tr>
<tr>
<td><code>a</code></td>
<td>real</td>
<td>The length in angstroms of the <code>a</code> coordinate of the cell.</td>
<td></td>
</tr>
<tr>
<td><code>b</code></td>
<td>real</td>
<td>The length in angstroms of the <code>b</code> coordinate of the cell.</td>
<td></td>
</tr>
<tr>
<td><code>c</code></td>
<td>real</td>
<td>The length in angstroms of the <code>c</code> coordinate of the cell.</td>
<td></td>
</tr>
<tr>
<td><code>alpha</code></td>
<td>real</td>
<td>The angle between the <code>b</code> and <code>c</code> axes of the cell.</td>
<td></td>
</tr>
<tr>
<td><code>beta</code></td>
<td>real</td>
<td>The angle between the <code>a</code> and <code>c</code> axes of the cell.</td>
<td></td>
</tr>
<tr>
<td><code>gamma</code></td>
<td>real</td>
<td>The angle between the <code>a</code> and <code>b</code> axes of the cell.</td>
<td></td>
</tr>
</tbody>
</table>
A. Btcl Language and Commands—Standalone Mode - Continued

Description

The `cellParameter` keyword, used alone, gets the cell parameters. That is, it returns a Tcl list (using the current `tcl_precision`) of the *a*, *b*, *c*, *alpha*, *beta*, *gamma* parameters.

If you supply the *a*, *b*, *c*, *alpha*, *beta*, *gamma* parameters in the command line, the `cellParameter` command sets the system parameters to the values supplied. **Note:** if fewer than 6 parameters are given, then the existing (default, or previously set) values are used for the remaining parameters. If more than 6 parameters are supplied, an error is generated. The optional `-option` values and their effects are listed in the above table.

Example 1

cellParameter

This example of the `cellParameter` command returns the present values of the cell parameters: the *a*, *b*, and *c* lengths and the *alpha*, *beta*, and *gamma* angles.

Example 2

cellParameter 5.0 5.0 5.0

This example of the `cellParameter` command sets the *a*, *b*, and *c* lengths to 5 Å without changing the values of the cell angles.

database

Purpose

The `database` command performs operations on a database in the Discover environment.
database

Syntax

```
database operation arg1 ?arg2 ...?
```

<table>
<thead>
<tr>
<th>Operation</th>
<th>Arguments</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>varName ?dbHandle? ?obName?</td>
<td>Assign to the Tcl variable varName a string which is a command name used to address the database specified by dbHandle. The optional argument obName is used to return an object that contains the row in the Database table for the database handle just created. Use a value of &quot;&quot; for dbHandle if you want obName to relate to the current database.</td>
</tr>
<tr>
<td>print</td>
<td>?dbHandle? ?detail?</td>
<td>Print out the contents of the database specified by dbHandle. The optional detail argument allows you to control the amount of information printed.</td>
</tr>
<tr>
<td>default</td>
<td>?dbHandle?</td>
<td>Set the default database name for any database type to that specified by dbHandle. The default database type is always called System.</td>
</tr>
<tr>
<td>delete</td>
<td>?dbHandle?</td>
<td>Delete a database specified by dbHandle. Any objects referencing this database are no longer valid.</td>
</tr>
<tr>
<td>rename</td>
<td>?-force? ?Type.?newName * ?dbHandle?</td>
<td>Rename a database specified by dbHandle to newName. If Type is given, then this must be the same as the database type specified by dbHandle; if the -force option is included, no error is issued when a database matching newName exists, and this old database is deleted.</td>
</tr>
<tr>
<td>create</td>
<td>?-force? varName Type.?Name?</td>
<td>Create a new database with a type and name specified by Type.Name. The handle to the new database is set to varName; if the -force option is given, no error is issued when a database matching Type.Name exists, and this old database is deleted. If dbHandle is not given, then the current System database is used by default.</td>
</tr>
</tbody>
</table>

Database Specification

For most of the database command operations, the first argument after the operation keyword is the database specification (dbHandle). A database can be specified by a number associated with that database, by a string denoting the database's type and name, from an object handle, or from a previously defined database handle.

A numeric database specification has the form #Number, e.g., #2.

A string database specification has the form Type.Name, where both the Type and Name parts are optional. If the Type is omitted,
A. Btcl Language and Commands—Standalone Mode - Continued

the default database type becomes System. If the Name is omitted, the default database name is used for the specified database type. The specification string for database types and names is case-insensitive. The following table shows valid string database specifications:

<table>
<thead>
<tr>
<th>dbHandle</th>
<th>Database type</th>
<th>Database name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type.Name</td>
<td>Type</td>
<td>Name</td>
</tr>
<tr>
<td>.Name or Name</td>
<td>System</td>
<td>Name</td>
</tr>
<tr>
<td>Type.</td>
<td>Type</td>
<td>current Type database name</td>
</tr>
<tr>
<td>&quot; or &quot;.&quot;</td>
<td>System</td>
<td>current System database name</td>
</tr>
</tbody>
</table>

A database specification from an object or database handle usually has the value of the variable, e.g., $object or $db_handle.

**Description**

The **database** command has two main uses. First, it allows you to access the Discover databases, for printing, deleting, etc. Second, it allows you to create a handle to a database using the **handle** operation. Further manipulation of the data stored in that database is performed using this handle. (See database handle operations, examples, below, and documentation of databases in the Discover program.)

**Example 1**

**BTCL** > readFile molecular_system file = protein system = prot
**BTCL** > readFile molecular_system file = rna system = rna
**BTCL** > database print system.rna
**BTCL** > database default prot
**BTCL** > database print

The two **readFile** statements in this example are used to read in two new systems into the Discover system database with system names prot and rna. Note that, if a system name was not defined, these systems would have been given the default system name
database

($PROJECT), and the previous system database with this name would have been deleted. The first database print command would then print out the rna system database (no output is shown). The database default command then sets the prot system database as the default, so that the next database print command causes that database to be printed.

Example 2

```
BTCL > database handle rnaDb rna
BTCL > $rnaDb get rnaMonomers Monomer.Type
BTCL > object rnaMonomers
BTCL > database delete $rnaDb
```

In this example, the database handle command associates a database handle (rnaDb) to the rna system database. This is then used to get the all the monomer types of the fragment(s) in the rna system. The subsequent object command is used to print those names (no output is shown here—refer to the object command). The last database delete command is used to remove the rna system database using the database handle as a reference ($rnaMonomers could also have been used).

Example 3

```
BTCL > database handle dbh_DB #0
BTCL > $dbh_DB print Database
"Database"
  { Row Type Name Methods
--- -------- -------- ----------
  0) Database Database 0x00000000
  1) System acenm 0x00000000
  2) Energy () 0x00000000
  }
```

This example shows how to obtain a database handle to the Database database, database number 0, which contains information about all other databases currently defined in the Discover pro-
A. BTCL Language and Commands—Standalone Mode - Continued

gram. For example, the types and names of the databases are stored in the Database table of the Database database.

Example 4

```
BTCL > database handle dbh #1 db_n

BTCL > object db_n print
   type:     relation
   items:    1
   elements: 1
   table:    Database
   database: 1
   contents:
      103

BTCL > $dbh_one get dbName Name $db_n

BTCL > object dbName
   acenm

BTCL > $dbh_one get dbType Type $db_n

BTCL > object dbType
   System
```

This example shows how to use the database handle command with the optional last argument to retrieve the object db_n. A handle is created (called dbh) to database number 1 (#1), which happens to be the acenm system database. Note that the object db_n contains a rid number, 103, which is the code for the row in the Database table containing information about database number 1. If this object were cast to an integer, using the object cast command, this value would become 1 (a row index value). The following database handle get operations show how the new object db_n can be used to retrieve information about a specified database, e.g., its name and type.

---

$dbHandle

Purpose

Database handle operations are an extension of the database command. They are used to access the data within a particular data-

---
base for which a database handle ($\texttt{dbHandle}$) has been created with the database handle command.

**Syntax**

$\texttt{dbHandle} \text{ operation ?-option? arg1?arg2?arg3?...}$

The database command operations:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Arguments</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>get</td>
<td>varName</td>
<td>Get the information from a particular table column to create an object handle varName. If spec is provided, the table specification is not required, since it is specified implicitly by spec.</td>
</tr>
<tr>
<td>set</td>
<td>valueList</td>
<td>Set the values in a particular table column to those specified in valueList. If spec is provided, the table specification is not required, since it is specified implicitly by spec. valueList must be a value specification of the correct type and must specify one or as many items as there are rows in the specified table.</td>
</tr>
<tr>
<td>unset</td>
<td></td>
<td>Set the values in a particular table column to their unset or undefined values.</td>
</tr>
<tr>
<td>exists</td>
<td>Table?.Column?</td>
<td>Return 1 or 0 (true or false) if the specified table or table column exists or does not, respectively.</td>
</tr>
<tr>
<td>rows</td>
<td>Table.?varName?</td>
<td>Return the number of rows in the specified table. If the optional argument varName is supplied, a rid (row-ID) Btcl object of this name is created, which contains rid numbers corresponding to these rows.</td>
</tr>
<tr>
<td>select</td>
<td>valueList</td>
<td>Return a rid object varName containing rid numbers for rows in table table for which values in object specification valueList match those in column Column. If spec is provided and specifies a subset of rows in the table, then the select is restricted to just these rows. This operation is identical to the objectselect operation, except that the select space is a table row rather than another object.</td>
</tr>
<tr>
<td>sort</td>
<td>varName?</td>
<td>Sort the values in the specified table column into ascending order and return the rid numbers of the table rows in this order to an object varName. If spec is provided and specifies a subset of rows in the table, then the sort is restricted to these rows (and the table does not need to be specified).</td>
</tr>
<tr>
<td>createTable</td>
<td>Table</td>
<td>Create a new table in the database with the name Table.</td>
</tr>
</tbody>
</table>
A. Btcl Language and Commands—Standalone Mode - Continued

<table>
<thead>
<tr>
<th>Operation</th>
<th>Arguments</th>
<th>Description</th>
</tr>
</thead>
</table>
| createColumn  | Table?.Column?  
|               | (Type ?elements?)  
|               | ?Default          
|               | | Ref?            | Create a new column in the table Table of name Column and type Type. The number of elements per item in the new column is optionally given by elements (default = 1). The default value for the column is optionally given by Default (default = 0). If Type is reference, a reference table name (Ref) may be specified (the default is Column). The acceptable values for Type are all the valid Btcl object types (reference being synonymous with rid). |
| create        | num Table ?varName?         | Create num new rows at the end of table Table. If varName is provided, a rid object is created of this name, which contains the row-identification numbers for the new table rows. |
| alias         | alias Table?.Column         | Create an alias name, alias, for an existing table Table or an existing column Table.Column. A column may also be referenced as table.alias. |
| delete        | spec                         | Delete part of a database base specified by spec, which may be Table, Table.Column, or an object handle. These specify the deletion of a table, column, or list of rids (table rows), respectively. |
| print         | Table ?Detail?               | Print out information about table Table with an optional detail specification Detail. The Detail options are the same as for the database print command. |
| getIndex      | varName                    
|               | Table.Index                 | Get a list of rows from a table index, specified by Table.Index, and return it as a Btcl object varName. |
| setIndex      | spec ?Table.?Index           | Set a new index of rid numbers, specified by spec, to a table Table and with a name Index. The table part of the index specification, Table, is optional but must agree with that inferred by spec, if it is provided. |

System object specification

A specification argument (spec in the table above) is usually an object containing a list of rid numbers (see object keyword, page 217) that was created previously, e.g., via a select command or select $dbHandle operation.

The specification spec may also be a valid specification string that could be accepted by the select command, e.g., *:(1..3):H*, although it is illegal in $dbHandle operations to re-specify the database, as in rna:*:(1..3):H*, for example. (Refer to the select command.) If you supply a string specification for spec, then this must specify a non-empty object, or an error will be reported. For example, rna would cause an error if the system database did not
contain a model named rna. A specification is also only valid if $dbHandle$ is a handle to a system database.

**Value lists**

A value list specification (valueList in the table above) may be an object handle to some previously defined object containing a set of values.

The value list specification string may be a single string, e.g., "Zn-ion" or a list of strings, e.g., [string1 string2 string3], such as might be used, for example, to set a list of strings to a Name column of a table.

The value list specification string may also be a vector command string type specification, e.g., {{1 2 3} {4 5 6}} (two 3 x 1 vectors—refer to the vector command). Note that if the table column is of type reference, e.g., Atom.Monomer, the value list is taken to be a set of integers which will be converted to rid numbers.

**$dbHandle options:**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>get</td>
<td>-grouping</td>
<td>The default action for a get operation is to create an object from a database table column with no grouping information. If the -grouping keyword is used, grouping information is transferred from the spec arg.</td>
</tr>
<tr>
<td>set</td>
<td>-nooverwrite</td>
<td>Prevent the $dbHandle set operation from writing over existing data. The values in a database column can be set with this option only if that column was just created.</td>
</tr>
<tr>
<td>select</td>
<td>-bygroup -itempergroup -nosubdivide -regexp -cas-sensitive != &gt;= &lt;= &gt; &lt; -anyof -noneof</td>
<td>The $dbHandle select operation behaves like the object select operation, except that the select space is the specified database table column, restricted to the rows specified by the optional spec argument (if provided). The grouping on spec is used only when the -bygroup or -itempergroup option is specified. The -grouping option specifies that the select operation is performed with respect to grouping on valueList and insists that the spec argument not be provided. If no grouping options are provided, the grouping on the result, varName, is determined by the inherent grouping on the specified column.</td>
</tr>
<tr>
<td>sort</td>
<td>-nosubdivide -reverse -cas-sensitive</td>
<td>The $dbHandle sort operation behaves identically to the object sort operation, the difference being that the sort space is a database table column rather than another Btcl object and the options -value and -bygroup are not supported.</td>
</tr>
</tbody>
</table>
A. Btcl Language and Commands—Standalone Mode - Continued

**Example 1**

```
BTCL > database handle molecule_h
BTCL > $molecule_h createTable Trajectory
BTCL > $molecule_h createTable Trajectory.Frame int
BTCL > $molecule_h createColumn Trajectory.DipoleMoment {double 3} {0 0 0}
```

This example shows how a new table is created in the current system database. A database handle molecule_h is first created to address the current system database using the `database` command. This handle is then used to perform database handle operations. The first of these is the `createTable` operation, which creates a new table with the name `Trajectory`. A `createColumn` operation is then performed to create a column `Frame` in the `Trajectory` table, which contains integers (no default value for the column is supplied). Another `createColumn` operation is then performed to create a table `DipoleMoment` in the table `Trajectory`, which contains triplets of doubles (element size = 3) which have the default value of [0.0 0.0 0.0].
Example 2

```
BTCL > select atoms mol:*:*
BTCL > $molecule_h get coords Coord $atoms
BTCL > $molecule_h get charges Charge $atoms
BTCL > vector dipole multiply $coords $charges
BTCL > vector dipole average $dipole
```

This example shows how the dipole moment of a model named mol is calculated using data from the Atom table in the current database. The `select` command is used to obtain a list of all the atoms in the current model. These are returned to an object called atoms, which actually contains a list of rows in the Atom table. Two `$dbHandle get` operations are then used to obtain objects called coords and charges, which contain lists of values from the Coord and Charge columns of the Atom table, respectively, for the rows specified by the object called atoms. Two `vector` commands are then performed to calculate the dipole moment vector to the object called dipole.

Example 3

```
BTCL > $molecule_h create 1 Trajectory trj_n
BTCL > $molecule_h set $frame Frame $trj_n
BTCL > $molecule_h set $dipole DipoleMoment $trj_n
```

This example continues from the last two examples. The `$dbHandle create` operation is used to create one new row in the Trajectory table, the new row being returned to an object called trj_n. The Frame column value for this new row, specified by $trj_n, is then set to the value $frame (an integer) by the first `$dbHandle set` operation. The DipoleMoment column value for this row is then set in the same manner to the calculated value $dipole. The sequence of commands in this and the previous example might be quite useful in a procedure or a loop that could, for example, be used to create a table of dipole moments from a set of coordinates in an archive file.
A. Btcl Language and Commands—Standalone Mode - Continued

Example 4

BTCL > $molecule_h select options regexp "N.*" Atom.Name nitros
BTCL > $molecule_h get N-crds Coord $nitros

This example shows how the database tables may be directly accessed to obtain, for example, coordinates of the nitrogens atoms in the current system. First a select operation is performed with a regexp option to select all the rows from the Atom table which have a name beginning with N in the Name column. Using the rows obtained (in an object variable called nitros), the coordinates can now be retrieved from the Coord column of the Atom table, by using a get operation. Note that the table does not have to be re-specified, since this information is stored in the object variable nitros. The coordinates are returned as a (vector) object called N-crds.

Example 5

BTCL > $db_h select -1 Atom.P1Atom platoms
BTCL > $db_h select -1 Atom.AsymmetricAtom asymAtoms

It is often useful to have unset values in a column so that they may not be selected in the usual way. This feature is used frequently in the System databases. The unset values in a reference column (a column in a table which refers to rows in another table) may be selected using a value of -1. In the example above, the database handle $dbh refers to a periodic system which also has symmetry. Unset values in reference columns Atom.P1Atom and Atom.AsymmetricAtom indicate that the corresponding atoms (rows) are part of the P1 and asymmetric subunits defining the system. Using -1 allows a selection to be made on columns for unset values.

If just the P1 atoms of a system are required, then these may be obtained simply by using the following command:

BTCL > select platoms "*:*:Atom;*

This is superior to the command above, since it does not fail if the system is not periodic. However, a more efficient way, taking advantage of the way System databases use unset values to denote periodic images, would be the following:

BTCL > $db_h select >= 0 Atom.Monomer platoms
The `diffraction` command is used to calculate the X-ray, neutron, or electron scattering pattern for a model. The model may comprise one or several isolated fragments or may be of the periodic type, as used to model bulk liquids and amorphous solids. The structure used may be either the current model or a series of structures obtained from an .arc or .his file. Output is directed to a graph file in the standard .tbl format.

### Syntax

```
diffraction ?keyword = value?
```

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Values</th>
<th>Default</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>radiation_type</td>
<td>xray</td>
<td>xray</td>
<td>Perform calculation for X-ray radiation.</td>
</tr>
<tr>
<td></td>
<td>electron</td>
<td></td>
<td>Perform calculation for electron beam.</td>
</tr>
<tr>
<td></td>
<td>neutron</td>
<td></td>
<td>Perform calculation for neutron beam.</td>
</tr>
<tr>
<td>limit_type</td>
<td>q</td>
<td>q</td>
<td>Use $q = 4 \pi \sin \theta / \lambda$ for calculation (Å$^{-1}$).</td>
</tr>
<tr>
<td></td>
<td>s</td>
<td></td>
<td>Use $s = 2 \sin \theta / \lambda$ (Å$^{-1}$).</td>
</tr>
<tr>
<td></td>
<td>theta</td>
<td></td>
<td>Use scattering angle = $2 \theta$ (degrees).</td>
</tr>
<tr>
<td>wavelength</td>
<td>real</td>
<td>0</td>
<td>Radiation wavelength—required only for <code>limit_type = theta</code>.</td>
</tr>
<tr>
<td>limit_min/imum</td>
<td>real</td>
<td>0</td>
<td>Minimum limit for q, s, or scattering angle, as appropriate.</td>
</tr>
<tr>
<td>limit_max/imum</td>
<td>real</td>
<td>0</td>
<td>Maximum limit for q, s, or scattering angle, as appropriate.</td>
</tr>
<tr>
<td>frame/inition</td>
<td>string</td>
<td>—</td>
<td>Name of file containing a specification of a Discover .arc or .his file and frames to be used.</td>
</tr>
<tr>
<td>subset</td>
<td>string</td>
<td>—</td>
<td>Name of the atom set to be used in the analysis. The atom set must be present in the .mdf file; if not, the analysis is performed on the entire model.</td>
</tr>
<tr>
<td>table</td>
<td>Boolean</td>
<td>false</td>
<td>Write output to a table file.</td>
</tr>
<tr>
<td>file/name</td>
<td>string</td>
<td>run_name.tbl</td>
<td>Name of .tbl file to receive output.</td>
</tr>
</tbody>
</table>
A. Btcl Language and Commands—Standalone Mode - Continued

**Description**

The `diffraction` command is used to perform analysis on the current model or on structures that were previously saved in .arc or .his files.

In general, it is used to perform analysis on all the atoms composing the system, and this is indicated by omitting the `subset` keyword and name. However, if desired, a subset of the atoms can be indicated. This is useful for reducing the computation time when the contributions of certain atoms are negligible (e.g., H atoms for X-ray scattering).

Intensities are given as $I / I_0$, where $I_0$ denotes the intensity at zero angle. Limits may be given in units of $s = 2 \sin \theta / \lambda$, $q = 4 \pi \sin \theta / \lambda$, or as the scattering angle $2 \theta$ (in which case specification of the wavelength, in angstroms, is mandatory).

The following restriction applies to the use of this command:

- For periodic systems, the cell must be orthorhombic.

All output is written to standard .tbl files, in the form $I / I_0$ vs. $S$, $q$, or $2 \theta$, as appropriate.

For details on theory and application relevant to this command, please consult the generic **Diffraction** pulldown documentation. This can be found in the *Polymer User Guide* and the *Catalysis User Guide*.

**Example 1**

```
diffraction frame_definition = argon.fdef
radiation_type = xray limit_type = q
limit_minimum = 0.1 limit_maximum = 5.0
+table filename = argon_xray.tbl
```

This example of the `diffraction` command calculates the scattering pattern for a modeled argon system, using configurations described in the file argon.fdef. The actual configurations are stored in a .his or .arc file, as indicated in the frame definition file.
Example 2

diffraction radiation_type = electron
  wavelength = 0.1 limit_type = theta
  limit_minimum = 10 limit_maximum = 60
+table filename = cc14_electron.tbl

This example of the diffraction command calculates the electron diffraction curve for the current system, a carbon tetrachloride model. The results are given as a function of the scattering angle.

discoverHistory

Purpose

The discoverHistory command is for manipulating the history information generated from a Discover dynamics run.

Syntax


<table>
<thead>
<tr>
<th>Operation</th>
<th>Option</th>
<th>Default</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>convert</td>
<td>-xdr2his -xdr2his</td>
<td></td>
<td>Convert between platform-independent (XDR) representation of the history file and machine-dependent (FORTRAN) history file. (If no option is given, the default is used.)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Option</th>
<th>Value</th>
<th>Default</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>-xdr2his</td>
<td>xdr_file run_name.his</td>
<td>xdr_file Use the xdr_file file as the .xdr file. Output the history information to the .his_file.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>his_file run_name.his</td>
<td>his_file</td>
<td></td>
</tr>
</tbody>
</table>

Background

Currently, the discoverHistory command has one operation and one option. The purpose is to allow conversion of machine-independent history files into machine-dependent history files. One use of this would be to read a history file generated on a non-IEEE-compliant platform and generate a FORTRAN history file that is
A. Btcl Language and Commands—Standalone Mode - Continued

readable in an IEEE format (so that the Cerius² and Insight programs can read the file).

On non-IEEE-compliant platforms, the Discover program outputs only XDR (external data representation) format history files. On IEEE platforms the Discover program outputs FORTRAN history files by default; however, if the environment variable DISCOVER_HISTORY_XDR is defined, XDR history files are written. The default file name for XDR history files is \textit{run\_name\.his\.xdr}.

\textbf{Description}

The \texttt{discoverHistory} command takes an \textit{operation} and \textit{options} to those operations. At present, the only supported operation is \texttt{convert}, which takes an optional \texttt{-keyword}. The only \texttt{-keyword} supported right now is \texttt{-xdr2his}, which is also the default if no \texttt{-keyword} is specified.

Filenames for the .xdr and .his files can be specified as optional arguments.

\textbf{Example 1}

\begin{verbatim}
BTCL > discoverHistory convert
\end{verbatim}

In this example, the \texttt{discoverHistory convert} command looks for a file named \textit{run\_name\.his\.xdr} and generates a file named \textit{run\_name\.his}, which is in the correct format for a Discover history file.

\textbf{Example 2}

\begin{verbatim}
BTCL > discoverHistory convert -xdr2his acenm.xx acenm.hh
\end{verbatim}

In this example, the \texttt{discoverHistory convert} command looks for a file named \textit{acenm.xx} and generates a file named \textit{acenm.hh}, which is in the correct format for a Discover history file.

\textbf{dynamics}

\textbf{Purpose}

The \texttt{dynamics} command performs molecular dynamics simulations on a model.
It can be used to bring a system to equilibrium at the desired temperature and pressure (or stress) (like the `initialize dynamics` command in FDiscover).

It can also be used to collect data and statistics (like the `resume dynamics` command in FDiscover) or to continue an interrupted dynamics run (like the `restart dynamics` command in FDiscover).

**Syntax**


<table>
<thead>
<tr>
<th>Path/keyword</th>
<th>Values</th>
<th>Default</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>boltzmann</td>
<td>Boolean</td>
<td>false</td>
<td>Randomize velocities to Maxwell-Boltzmann distribution.</td>
</tr>
<tr>
<td>initial_temperature</td>
<td>real</td>
<td></td>
<td>Initial temperature (in K) at which random velocities are generated.</td>
</tr>
<tr>
<td>temperature</td>
<td>real</td>
<td>298.0</td>
<td>Target temperature (in K).</td>
</tr>
<tr>
<td>time or duration</td>
<td>real</td>
<td>5000.0</td>
<td>Time for which dynamics is run (in fs).</td>
</tr>
<tr>
<td>timestep</td>
<td>real</td>
<td>1.0</td>
<td>Time for each dynamics step (in fs).</td>
</tr>
<tr>
<td>integration_method</td>
<td>Velocity_verlet Abm4 Rung_e_Kutta</td>
<td></td>
<td>Integrator to be used during calculation.</td>
</tr>
<tr>
<td>continue_average</td>
<td>Boolean</td>
<td>false</td>
<td>If true, keep the running sum from the previous dynamics run.</td>
</tr>
<tr>
<td>ensemble</td>
<td>NVT NVE NPT NPH</td>
<td></td>
<td>Thermodynamics ensemble desired.</td>
</tr>
<tr>
<td>temperature_control_</td>
<td>Velocity_scaling Nose Andersen Berendsen</td>
<td></td>
<td>Method for controlling temperature.</td>
</tr>
<tr>
<td>method</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>collision_ratio</td>
<td>real</td>
<td>1.0</td>
<td>Factor to multiply the collision frequency by when using the Andersen temperature control method.</td>
</tr>
<tr>
<td>decay_constant</td>
<td>real</td>
<td>0.1</td>
<td>Half-life (ps) for decay to target temperature for the Berendsen temperature-control method.</td>
</tr>
<tr>
<td>pressure_control_</td>
<td>Parinello Andersen_cp Berendsen_pc</td>
<td></td>
<td>Method for controlling pressure.</td>
</tr>
<tr>
<td>method</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>press_choice</td>
<td>pressure stress</td>
<td></td>
<td>Indicate whether target pressure or stress is desired.</td>
</tr>
</tbody>
</table>
### A. Btcl Language and Commands—Standalone Mode - Continued

<table>
<thead>
<tr>
<th>Path/keyword</th>
<th>Values</th>
<th>Default</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>press_unit</td>
<td>GPa MPa bar</td>
<td>GPa</td>
<td>Specify units for pressure.</td>
</tr>
<tr>
<td>pressure</td>
<td>real 0.</td>
<td>0.</td>
<td>Target pressure (used only if press_choice = pressure).</td>
</tr>
<tr>
<td>press_decay_constant</td>
<td>real 0.1</td>
<td>0.1</td>
<td>Half-life (ps) for decay to target pressure for the Berendsen pressure-control method.</td>
</tr>
<tr>
<td>compressibility</td>
<td>real 0.5</td>
<td>0.5</td>
<td>System compressibility γ (in GPa⁻¹) (See Forcefield-Based Simulations) for the Berendsen pressure-control method.</td>
</tr>
<tr>
<td>stress</td>
<td>list of six reals</td>
<td>(0.0 0.0 0.0 0.0 0.0 0.0)</td>
<td>Short form for setting sx, sy, sz, syz, sxz, and sxy (respectively).</td>
</tr>
<tr>
<td>sx, sy and szz</td>
<td>real 0.0</td>
<td>0.0</td>
<td>Target tensile stress.</td>
</tr>
<tr>
<td>syz, sxz and sxy</td>
<td>real 0.0</td>
<td>0.0</td>
<td>Target shear stress.</td>
</tr>
<tr>
<td>temperature_window</td>
<td>real 10.0</td>
<td>10.0</td>
<td>Temperature difference between actual and target temperatures (K) allowed during temperature scaling (used only when temperature_control_method = Velocity_scaling or Berendsen).</td>
</tr>
<tr>
<td>deviation</td>
<td>real 5000.0</td>
<td>5000.0</td>
<td>Energy difference allowed between successive steps (kcal mol⁻¹).</td>
</tr>
<tr>
<td>q_ratio</td>
<td>real 1.0</td>
<td>1.0</td>
<td>Determine Q, the fictitious mass for Nosé; dynamics; Q = q_ratio x program-calculated Q₀.</td>
</tr>
<tr>
<td>cell_mass</td>
<td>real 20.0</td>
<td>20.0</td>
<td>Fictitious mass of unit cell, used in Parrinello method of constant-stress dynamics (in units of atomic mass).</td>
</tr>
<tr>
<td>peek</td>
<td>string peek</td>
<td>peek</td>
<td>Btcl command called after every iteration to write out .ipc and/or .pcar files to monitor the progress of a dynamics simulation. Value may be &quot;&quot; to turn off the peek mechanism.</td>
</tr>
<tr>
<td>cell_parm</td>
<td>Pathname</td>
<td></td>
<td>Pathname to cell parameters.</td>
</tr>
<tr>
<td>save_initial</td>
<td>Boolean false</td>
<td>false</td>
<td>Save initial cell parameters before the run starts.</td>
</tr>
<tr>
<td>save_final</td>
<td>Boolean false</td>
<td>false</td>
<td>Save final cell parameters after the run finishes.</td>
</tr>
<tr>
<td>restore_reference</td>
<td>Boolean false</td>
<td>false</td>
<td>Restore saved cell parameters as the reference values for calculating strain.</td>
</tr>
<tr>
<td>execute</td>
<td>Pathname</td>
<td></td>
<td>Pathname to execute structures (a separate execute structure is passed to the dynamics command for each occurrence of execute).</td>
</tr>
<tr>
<td>command</td>
<td>string &quot;echo hello&quot;</td>
<td>&quot;echo hello&quot;</td>
<td>A full Btcl command, enclosed in quotes (&quot;&quot;&quot;) or braces ({ }), to be executed during dynamics (e.g., print).</td>
</tr>
<tr>
<td>before</td>
<td>Boolean false</td>
<td>false</td>
<td>Execute command before dynamics run begins.</td>
</tr>
</tbody>
</table>
The dynamics command is used to start or to continue a dynamics run.

To start a new dynamics run, you can assign random velocities to atoms in the model according to the Maxwell–Boltzmann distribution (by including the keyword \texttt{boltzmann}) at the initial temperature specified by \texttt{initial\_temperature}. If the initial temperature is not specified, the target temperature is used as the initial temperature.

Once the system has been equilibrated at the desired temperature and pressure (or stress), you may want to start collecting data and statistics.

To clear the sums and averages of properties calculated during the equilibration phase, you issue a new \texttt{dynamics} command. If the \texttt{boltzmann} keyword is not used and if dynamics was run previously, the program uses the final coordinates and velocities of the previous run, which were written out in the dynamics restart data block, as the starting structure and velocities for the new dynamics run.

It is important to remember that, if the structure has been changed after a dynamics run (by minimization, for example), it is essential to randomize the velocities again, since the old velocities do not apply to the new coordinates. The program checks whether the current system structure is the same as in the dynamics restart data written out by the previous dynamics run. If differences are found,
the current structure is used and the initial velocities are randomized.

When different dynamics stages are used in the same job, the accumulating sum for calculating running averages is zeroed out at the start of each stage. However, if you want to carry the sum from one dynamics stage to the next, you can use the `continue_average` keyword.

To continue a dynamics run from a previous job, the dynamics restart file (`run_name.xdyn`) must be read. This can be done by putting the `readFile dynamics_restart` command right before the `dynamics` command.

CDiscover allows the choice of four different ensembles, namely:

- **Constant volume–constant temperature**, sometimes known as the canonical ensemble (NVT)
- **Constant volume–constant energy**, sometimes known as the microcanonical ensemble (NVE)
- **Constant pressure–constant temperature** (NPT)
- **Constant pressure–constant enthalpy** (NPH)

The ensemble is chosen with the `ensemble` keyword. A note of caution for choosing the ensemble—the volume, and hence the pressure, are undefined in nonperiodic systems; thus, constant-pressure dynamics (NPT or NPH) is meaningful only for periodic systems. (If constant-pressure dynamics is requested for a nonperiodic system, program execution stops.) If you use nonbond cutoffs with constant-pressure dynamics, you should also use the `vdWTailCorrection` command (page 287).

The method of temperature control is selected by setting the temperature_control_method to `Velocity_scaling`, `Nose`, `Andersen`, or `Berendsen`. For the Andersen method, the collision frequency parameter is calculated automatically to a suitable value for the system. However, this can be scaled using the `collision_ratio` parameter by setting it to a value other than 1.0.

Q, the fictitious mass used in the Nosé method, can be controlled by the user-defined parameter `q_ratio`. The default for `q_ratio` is 1.0, indicating that the Q0 calculated by the program will be used. If `q_ratio` is any value other than 1.0, the actual Q used is `q_ratio` x Q0.
The decay constant $\tau$ in the Berendsen method of temperature control is specified by the `decay_constant` parameter. The `decay_constant` value is the time for decay to half the difference between the initial and target temperatures and is related to $\tau$ by $\tau = \frac{\text{decay\_constant}}{\ln 2}$.

When the current temperature (calculated from the kinetic energy) and the target temperature (specified by `temperature`) differ by more than the amount specified by `temperature_window`, the velocities are rescaled so that the current and target temperatures are the same. Fluctuations in temperature are greater at larger values of `temperature_window`; however, the trajectory is disturbed less.

Control of pressure or stress is selected by setting the `press_choice` parameter to `pressure` or `stress`. If `press_choice = pressure`, then the `pressure` parameter is used to specify the target pressure for the system during the run.

When `press_choice = stress`, CDiscover uses the Parrinello method of constant-stress dynamics. This method allows the cell to move by an amount that depends on the fictitious mass, defined by `cell_mass`. A heavier mass dictates a smoother change to the cell, but the run requires more time to attain the target stress.

The target stress is defined by the parameters `sxx, syy, szz, syz, sxz, and sxy`, which can be assigned individually or via the `stress` parameter, which takes a list of six values.

Nosé dynamics can also be used with the Parrinello method to produce constant-temperature/constant-pressure dynamics and the NPT ensemble. To do so, set `ensemble` to `NPT` and `temperature\_control\_method` to `Nose`. The methodology for applying the Nosé method to constant-stress dynamics is taken from Ray and Rahman (1985).

The decay constant $\tau$ in the Berendsen method of pressure control is specified by the `press\_decay\_constant` parameter. The `press\_decay\_constant` value is the time for decay to half the difference between the initial and target temperatures and is related to $\tau$ by $\tau = \frac{\text{press\_decay\_constant}}{\ln 2}$. If the system compressibility is known experimentally, then it may be specified in GPa$^{-1}$ with the `compressibility` keyword. If it is not known, values other than the default of 0.5 GPa$^{-1}$ should be used with caution, since it is the ratio of $\gamma$ to $\tau$ that controls the Berendsen pressure coupling.
The `cell_parm save_initial` and `cell_parm save_final` keywords allow you to save the initial and/or final cell parameters of a periodic system internally, before or after a dynamics run. The cell parameters can then be restored as the reference values for calculating strain by using the keywords `cell_parm restore_reference`.

This is useful for calculating the stress–strain relationship—although the stresses may be increased during various dynamics stages, you may want to use only the original non-stressed state as the reference value for calculating the strain. Whether or not these keywords are used, the cell parameters for calculating energy and other quantities will still be the current cell parameters in the system or from the dynamics restart file.

To control for abrupt changes in the energy of the system, the energy difference between successive steps is checked at every step. If this difference is greater than the maximum energy deviation allowed, defined by the `deviation` keyword, program execution stops.

Note that, in contrast to early versions of the Discover program, the length of the dynamics run is input as the actual time in femtoseconds, rather than the number of steps. The run length is specified by either of the synonymous keywords `time` or `duration`. Combined with the duration of each timestep, specified by the `timestep` keyword, these keywords determine the number of dynamics steps.

The `execute pathname` keyword and its associated parameters are used to specify subcommands to be performed during a dynamics simulation and when they should be performed.

The execute structure typically contains the `print` command, to write out the instantaneous and average values that are calculated during a run (see the `print` command and the examples below). The frequency, beginning, and end of the subcommand execution are specified in femtoseconds and are automatically rounded off to the nearest multiple of the timestep parameter.

If `execute print` is not included on the `dynamics` command line, the only output is a summary table of the energy and thermodynamic state.

For dynamics, use of `execute print` structures enables the printing of up to 12 tables containing the total energies, energy compo-
nents, thermodynamic state, cell geometry, stress and strain. This information can also be written to the table file that the Insight program uses for constructing graphs. In addition, coordinate, velocities, and energy derivatives (the negative of the forces) can be archived in archive files.

**Example 1**

```plaintext
dynamics
```

The command `dynamics` with no arguments uses the default values or values previously set by the user with the `cdlConfig` utility.

**Example 2**

```plaintext
dynamics +boltzmann initial_temperature = 300.0 ensemble = NPT \ 
temperature = 300.0 press_choice = stress \ 
Sxx = 0.01 cell_mass = 40.0 deviation = 4000.0 \ 
temperature_window = 5.0
```

This example of the `dynamics` command indicates that dynamics is to be done under conditions of constant stress and temperature (NPT), with initial randomization of velocities at a temperature of 300.0 K.

The target temperature is 300.0 K, and the target tensile stress ($s_{xx}$) is 0.01 GPa. The fictitious cell mass (Parrinello method) is set to 40.0 atomic mass units, the maximum energy deviation between successive steps to 4000.0 kcal mol$^{-1}$, and the temperature window for the direct velocity scaling method to 5.0 K.

**Example 3**

```plaintext
cdlConfig dynamics execute +before +after frequency = 10 ...
dynamics \ 
execute command = "print output +energy_summary +state +cell" \ 
execute command = "print archive coordinates = on" \ 
execute command = "print table +nonbond_energy" -before
```

In this example, a `cdlConfig` command is used to set default values for the `dynamics execute` parameters `before`, `after`, and `frequency`, so that subsequent `dynamics execute` commands are performed before, after, and every 10 fs during the dynamics run by default.
The subsequent `dynamics` command in the example includes three execute structures, to print properties to output, archive, and table files, respectively. The default parameter values are used in all three instances, except for the last `print table` command, where output is not written before (`-before` sets `before` to false) the dynamics run.

**Example 4**

```c
readFile dynamics_restart file = crn.xdyn
...
cdlConfig dynamics temperature = 390.0
...
dynamics
```

This example shows how the `dynamics` command is used to set up a dynamics run based on a previous run.

First, the dynamics restart file `crn.xdyn` is read. This causes the default values of dynamics parameters to be set to those used in the previous dynamics run.

Then the dynamics temperature is changed to 390 K for the new dynamics run. The order in which these two commands appear is important—the `readFile dynamics_restart` command overwrites any previous default parameter values.

**Example 5**

```c
dynamics peek = {peek iteration_step = 4 filename = look +carfile} \ execute command = "print output +energy_summary +state +cell"
```

This command line performs the same dynamics simulation as in Example 3, while only printing to the output file. The `peek` parameter is used in this example to cause `look.pek` and `look.pcar` to be generated every 4 iterations. These files contain the energy and coordinates, respectively, of the system at that point in the simulation. (See the `peek` command for more information.

**Example 6—Consensus dynamics**

To do consensus dynamics, you need to create a column, `Consensus`, in the `MainCell/Atom` table of the system database. This column must contain integers that indicate which group the atoms belong to. An atom in the group labeled 0 will interact with all
other atoms. The atoms in groups labeled other than 0 will, however, interact only with atoms in the same group and with atoms in group 0. Note that the interactions between group 0 and other groups are scaled by a factor of 1/(the number of other groups). Consensus dynamics is for nonperiodic systems only, and the no-cutoffs method is automatically used.

For example:

database handle sys_dbh System.
$sys_dbh createColumn Atom.Consensus int 0
$sys_dbh print Atom

for {set i 2} {$i <= 3} {incr i} {
  select set "*:i:atom;*"
  $sys_dbh set $i Atom.Consensus $set
}

$sys_dbh print Atom

#This would be followed by a normal dynamics run, e.g.:

... 
dynamics time = 1000

In this example, the system database handle is assigned to sys_dbh. A column, Consensus, is then created in the MainCell/Atom table of the system database and initialized to 0. The for loop resets the numbers for atoms in monomers 2 and 3 to their corresponding monomer numbers. Thus, the atoms in monomers 2 and 3 will interact with atoms in the same monomer and with all the atoms labeled 0, i.e., there will be no interactions between monomers 2 and 3. The interactions between monomer 2 (or monomer 3) and the atoms that do not belong to these two monomers are scaled by 0.5.

---

energy

Purpose

The energy command performs a single energy evaluation. It can be used to return the value of the current energy to the Btcl script. It also calculates first and second derivatives by finite difference, which is used internally by MSI for validation. This feature can
A. Btcl Language and Commands—Standalone Mode - Continued

also be used by users who add their own energy and derivative calculations.

**Syntax**

<table>
<thead>
<tr>
<th>Path/keyword</th>
<th>Values</th>
<th>Default</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>print energies</td>
<td>Boolean</td>
<td>false</td>
<td>Print to the output file the total energy and all of its components.</td>
</tr>
<tr>
<td>first_derivative</td>
<td>Boolean</td>
<td>false</td>
<td>Print to the output file the first derivatives of the energy with respect to the coordinates (and cell parameters if it is periodic). The calculation uses fractional coordinates for periodic systems.</td>
</tr>
<tr>
<td>second_derivative</td>
<td>Boolean</td>
<td>false</td>
<td>Print to the output file the second derivatives of the energy with respect to the coordinates (and cell parameters if it is periodic). The calculation uses fractional coordinates for periodic systems.</td>
</tr>
<tr>
<td>hessian</td>
<td>Boolean</td>
<td>false</td>
<td>Print out a Hessian file (default is $PROJECT.hessian). This file contains a list of all the atoms in the model together with their elements, followed by a Hessian (second derivative) matrix. This Hessian may, for instance, be used as input to a minimization using DMol.</td>
</tr>
<tr>
<td>finite_difference first_derivative</td>
<td>Boolean</td>
<td>false</td>
<td>Print to the output file the first derivatives of the energy with respect to the coordinates (and cell parameters if it is periodic) calculated by two-sided finite differencing. In addition to the derivatives, this command prints the rms difference from the analytic derivatives.</td>
</tr>
<tr>
<td>finite_difference second_derivative</td>
<td>Boolean</td>
<td>false</td>
<td>Print to the output file the second derivatives of the energy with respect to the coordinates (and cell parameters if it is periodic) calculated by two-sided finite differencing. In addition to the second derivatives, this command prints the rms difference from the analytic second derivatives.</td>
</tr>
<tr>
<td>value</td>
<td>double</td>
<td>2.5e-6</td>
<td>Step size for finite difference.</td>
</tr>
<tr>
<td>filename</td>
<td>string</td>
<td>&quot;&quot;</td>
<td>Used to override the default name of the Hessian file ($PROJECT.hessian).</td>
</tr>
</tbody>
</table>

**Description**

The **energy** command is used to perform a single energy evaluation. The return value from the command is the value of the energy. This can be used in Btcl scripts as in Example 1. The finite-
difference capabilities of this command are used internally at MSI to test new energy codes. They may also be used by users who add their own energy terms.

Example 1

```tcl
for {$phi = -180} {$phi <= 180} {incr phi 20} {
    molGeom set torsion $phi $tors
    set energyValue [energy]
    puts $file "$tors       $energyValue"
}
```

This example is in the middle of a script that performs a systematic conformational search on a torsion, $tors. There is some initial setup that defines the torsion (see the subset define command) and opens the output file, $file. Then at each iteration, the torsion is rotated to $phi degrees and the energy is determined and stored in the variable energyValue. The torsion angle and energy are then printed to the file $file.

To control the precision of the energy value, the Tcl variable tcl_precision may be used. For instance:

```tcl
set tcl_precision 8
```

ensures that the energy is returned with 8 significant figures.

Example 2

```tcl
energy print +energies +hessian filename = ".hess"
```

This example calculates the energy, prints the energy total and components to the output file, and writes a Hessian file with the name $PROJECT.hess.

---

energyContribution

Purpose

The energyContribution command is used to pass energy, energy gradient, and virial contributions from user-defined Tcl routines or external executables to the Discover program.
A. Btcl Language and Commands—Standalone Mode - Continued

**Syntax**

```
energyContribution type arg1 ?arg2 ...?
```

The `energyContribution` command types:

<table>
<thead>
<tr>
<th>Type</th>
<th>Arguments</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>atom</td>
<td>energy</td>
<td>Cause Discover program to add <em>energy</em> to its application energy sum and also add gradient contributions to the corresponding values for the atoms of <code>atomList</code> plus an optional contribution to the <em>virial</em>.</td>
</tr>
<tr>
<td></td>
<td>gradient</td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>atomList</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>?virial</code></td>
<td></td>
</tr>
<tr>
<td>distance</td>
<td>energy</td>
<td>Causes Discover program to add <em>energy</em> to its distance restraint energy sum and also increment the energy gradient based on <code>deriv</code> and the atom lists. The atom lists are used to determine distances, and must be of equal length.</td>
</tr>
<tr>
<td></td>
<td><code>deriv</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>atomList1</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>atomList2</code></td>
<td></td>
</tr>
<tr>
<td>torsion</td>
<td>energy</td>
<td>Cause Discover program to add <em>energy</em> to its torsion restraint energy sum and also increment the energy gradient based on <code>deriv</code> and the atom lists. The atom lists are used to determine torsions, and must be of equal length.</td>
</tr>
<tr>
<td></td>
<td><code>deriv</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>atomList1</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>atomList2</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>atomList3</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>atomList4</code></td>
<td></td>
</tr>
<tr>
<td>angle</td>
<td>energy</td>
<td>Cause Discover program to add <em>energy</em> to its angle restraint energy sum and also increment the energy gradient based on <code>deriv</code> and the atom lists. The atom lists must correspond to the associated entries of <code>angles</code> and must be of equal length.</td>
</tr>
<tr>
<td></td>
<td><code>deriv</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>atomList1</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>atomList2</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>atomList3</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>angles</code></td>
<td></td>
</tr>
<tr>
<td>tether</td>
<td>energy</td>
<td>Cause Discover program to add <em>energy</em> to its tether restraint energy sum and also increment the energy gradient based on <code>deriv</code> and the atom list and coordinate list, which must be of equal lengths.</td>
</tr>
<tr>
<td></td>
<td><code>deriv</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>atomList</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>coordList</code></td>
<td></td>
</tr>
<tr>
<td>out_of_plane</td>
<td>energy</td>
<td>Cause Discover program to add <em>energy</em> to its out-of-plane restraint energy sum and also increment the energy gradient based on <code>deriv</code> and the atom lists, which must be of equal lengths.</td>
</tr>
<tr>
<td></td>
<td><code>deriv</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>atomList1</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>atomList2</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>atomList3</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>atomList4</code></td>
<td></td>
</tr>
</tbody>
</table>
The energyContribution command arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>atomList#</td>
<td>atom distance torsion angle</td>
<td>A list of atoms produced from an atom spec.</td>
</tr>
<tr>
<td>energy</td>
<td>atom distance torsion angle</td>
<td>A Btcl object containing an energy increment.</td>
</tr>
<tr>
<td>gradient</td>
<td>atom</td>
<td>A Btcl object containing an energy gradient increment.</td>
</tr>
<tr>
<td>deriv</td>
<td>distance torsion angle</td>
<td>Energy derivatives with respect to distance, torsion, or angle.</td>
</tr>
<tr>
<td>virial</td>
<td>atom</td>
<td>A Btcl object containing the virial increment as a $3 \times 3$ matrix.</td>
</tr>
<tr>
<td>angles</td>
<td>angle</td>
<td>A list of angles.</td>
</tr>
<tr>
<td>coordList</td>
<td>tether</td>
<td>A list of coordinates</td>
</tr>
</tbody>
</table>

**Description**

The energyContribution command is intended for use in a user-defined Btcl procedure that calculates energy and gradient increments for a btcl restraint.

The atom type energy contribution allows you total flexibility. The other types are more rigid, but still allow you to use your own energy function. Remember that the units of energy are kcal mol$^{-1}$, gradient units are kcal Å$^{-1}$, and the virial is in kcal mol$^{-1}$.

Example 5 in the description of the restraint command illustrates the use of the energyContribution command for the atom type.

**forcefield**

**Purpose**

The forcefield command is used to select a particular forcefield from the current forcefield file, to control setup of the energy expression when parameter sets are incomplete, and to change and scale the contributions from various energy terms used in Discover calculations. The parameters modified by this command are
A. Btcl Language and Commands—Standalone Mode - Continued

in effect for all subsequent energy calculations, until modified by another forcefield command.

**Syntax**

forcefield ?path? ?keyword = value?...

<table>
<thead>
<tr>
<th>Path/keyword</th>
<th>Values</th>
<th>Default</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>select</td>
<td>string</td>
<td>$FORCEFIELD</td>
<td>Name of forcefield contained in forcefield file.</td>
</tr>
<tr>
<td>parameters</td>
<td>Boolean</td>
<td>true</td>
<td>Pathname to forcefield-selection parameters.</td>
</tr>
<tr>
<td>bond_automatic</td>
<td>Boolean</td>
<td>true</td>
<td>Use automatic bond parameters if no explicit ones are found.</td>
</tr>
<tr>
<td>angle_automatic</td>
<td>Boolean</td>
<td>true</td>
<td>Use automatic angle parameters if no explicit ones are found.</td>
</tr>
<tr>
<td>torsion_automatic</td>
<td>Boolean</td>
<td>true</td>
<td>Use automatic torsion parameters if no explicit ones are found.</td>
</tr>
<tr>
<td>oop_automatic</td>
<td>Boolean</td>
<td>true</td>
<td>Use automatic out-of-plane parameters if no explicit ones are found.</td>
</tr>
<tr>
<td>bond_stop</td>
<td>Boolean</td>
<td>true</td>
<td>End calculation with error if no parameters are found for some bond. If bond_automatic is also true, end only if no explicit or automatic parameters exist.</td>
</tr>
<tr>
<td>angle_stop</td>
<td>Boolean</td>
<td>true</td>
<td>End calculation with error if no parameters are found for some angle. If angle_automatic is also true, end only if no explicit or automatic parameters exist.</td>
</tr>
<tr>
<td>torsion_stop</td>
<td>Boolean</td>
<td>true</td>
<td>End calculation with error if no parameters are found for some torsion. If torsion_automatic is also true, end only if no explicit or automatic parameters exist.</td>
</tr>
<tr>
<td>oop_stop</td>
<td>Boolean</td>
<td>true</td>
<td>End calculation with error if no parameters are found for some out-of-plane. If oop_automatic is also true, end only if no explicit or automatic parameters exist.</td>
</tr>
<tr>
<td>cross_stop</td>
<td>Boolean</td>
<td>false</td>
<td>End calculation with error if no parameters are found for some cross term. If current forcefield has no cross terms this flag has no effect; it applies only when the forcefield has cross terms but lacks parameters for some cross term.</td>
</tr>
<tr>
<td>print_automatic</td>
<td>Boolean</td>
<td>true</td>
<td>List automatic and unfound parameters in the file run_name.prm.</td>
</tr>
</tbody>
</table>
### Nonbond parameters:

<table>
<thead>
<tr>
<th>Path/keyword</th>
<th>Values</th>
<th>Default</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>nonbond</td>
<td></td>
<td></td>
<td>Pathname to nonbond parameters.</td>
</tr>
<tr>
<td>separate_coulomb</td>
<td>Boolean</td>
<td>false</td>
<td>Normally, both van der Waals and Coulombic interactions are calculated with same nonbond list. If this flag =true, 2 lists are used: 1 for van der Waals and 1 for Coulombic interactions.</td>
</tr>
</tbody>
</table>

### van der Waals parameters:

<table>
<thead>
<tr>
<th>Summation method</th>
<th>Values</th>
<th>Default</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>atom_based</td>
<td></td>
<td></td>
<td>Method by which energy terms are summed in the calculation when generating the nonbond list. (see note below)</td>
</tr>
<tr>
<td>cell_based</td>
<td></td>
<td></td>
<td>Pathname to van der Waals parameters.</td>
</tr>
<tr>
<td>ewald_cell_multipole</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cell_based</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>no_cutoff</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| Cutoff                | real     | 9.5     | Distance (angstroms) at which to exclude interactions from the nonbond list for atom_based and group_based methods (used only if no_cutoff is not used). |
| Spline width          | real     | 1.0     | Size (angstroms) of region within which nonbond interactions are splined from their full value to zero for atom_based and group_based methods (used only if no_cutoff is not used). |
| Buffer width          | real     | 0.5     | Distance (angstroms) beyond cutoff at which nonbond interactions are zero (but the atoms are included in the nonbond list) for atom_based and group_based methods (used only if no_cutoff is not used). |
| Update width          | real     | 1.0     | Distance (angstroms) an atom may move before nonbond list is regenerated, for ewald, cell_multipole, and cell_based methods (always applies to both van der Waals and Coulomb nonbond lists). (see note below) |
| Cmm level             | second   |         |                                                                                                                                         |
| Number of cells       | integer  | 1       | Number of layers of cells to include in nonbond list, counting from the base cell, for cell_multipole and cell_based summation methods (e.g., 1 means all the cells one unit cell away—the 26 immediate neighbor cells). |
| Ewald accuracy        | real     | 0.0025  | Accuracy (kcal mol$^{-1}$) used in Ewald calculations. (see note below)                                                                |
| Ewald dipole          | Boolean  | false   | If true, then contributions from dipole moment are taken into account in Ewald calculations.                                          |
### A. Btcl Language and Commands—Standalone Mode - Continued

<table>
<thead>
<tr>
<th>Path/keyword</th>
<th>Values</th>
<th>Default</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Coulomb parameters:</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><em>coulomb</em></td>
<td></td>
<td></td>
<td>Pathname to Coulomb parameters.</td>
</tr>
<tr>
<td><strong>summation method</strong></td>
<td><em>group_based</em></td>
<td><em>group_based</em></td>
<td>Method by which energy terms are summed in the calculation when generating the nonbond list. (see note below)</td>
</tr>
<tr>
<td></td>
<td><em>atom_based</em></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><em>ewald</em></td>
<td><em>multipole</em></td>
<td></td>
</tr>
<tr>
<td></td>
<td><em>cell_based</em></td>
<td><em>no_cutoff</em></td>
<td></td>
</tr>
<tr>
<td><strong>cutoff</strong></td>
<td><em>real</em></td>
<td>9.5</td>
<td>Distance (angstroms) at which to exclude interactions from the nonbond list for <em>atom_based</em> and <em>group_based</em> methods (used only if <em>no_cutoff</em> is not used).</td>
</tr>
<tr>
<td><strong>spline_width</strong></td>
<td><em>real</em></td>
<td>1.0</td>
<td>Size (angstroms) of region within which nonbond interactions are splined from their full value to zero for <em>atom_based</em> and <em>group_based</em> methods (used only if <em>no_cutoff</em> is not used).</td>
</tr>
<tr>
<td><strong>buffer_width</strong></td>
<td><em>real</em></td>
<td>0.5</td>
<td>Distance (angstroms) beyond cutoff at which nonbond interactions are zero (but the atoms are included in the nonbond list) for <em>atom_based</em> and <em>group_based</em> methods (used only if <em>no_cutoff</em> is not used).</td>
</tr>
<tr>
<td><strong>update_width</strong></td>
<td><em>real</em></td>
<td>1.0</td>
<td>Distance (angstroms) an atom may move before nonbond list is regenerated, for <em>ewald</em>, <em>cell_multipole</em>, and <em>cell_based</em> summation methods (always applies to both van der Waals and Coulomb nonbond lists). (see note below)</td>
</tr>
<tr>
<td><strong>cmm_level</strong></td>
<td><em>second</em></td>
<td><em>second</em></td>
<td>Highest-order multipole and highest-order Taylor series expansion coefficient used in cell multipole nonbond calculation.</td>
</tr>
<tr>
<td></td>
<td><em>third</em></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><em>fourth</em></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>number_of_cells</strong></td>
<td><em>integer</em></td>
<td>1</td>
<td>Number of layers of cells to include in nonbond list, counting from the base cell, for <em>cell_multipole</em> and <em>cell_based</em> summation methods (e.g., 1 means all the cells one unit cell away—the 26 immediate neighbor cells).</td>
</tr>
<tr>
<td><strong>ewald_accuracy</strong></td>
<td><em>real</em></td>
<td>0.0025</td>
<td>Accuracy (kcal mol(^{-1})) used in Ewald calculations.</td>
</tr>
<tr>
<td><strong>ewald_dipole</strong></td>
<td><em>Boolean</em></td>
<td>false</td>
<td>If true, then contributions from dipole moment are taken into account in Ewald calculations.</td>
</tr>
<tr>
<td><strong>distance_dependent_dielectric</strong></td>
<td><em>Boolean</em></td>
<td>false</td>
<td>Whether to use constant or distance-dependent dielectric for calculating Coulombic interactions.</td>
</tr>
<tr>
<td><strong>dielectric_value</strong></td>
<td><em>real</em></td>
<td>1.0</td>
<td>Magnitude of dielectric constant relative to (\varepsilon_0).</td>
</tr>
</tbody>
</table>
The **forcefield** command may be used to select one of the forcefields contained in a forcefield file. If this command is not specified, the default is to use the forcefield with the same name as the file. This command is currently available only with the CVFF forcefield file. This forcefield file contains the following forcefields:

- cvff
  - default—cross terms and Morse bonds
- cvff_nocross
  - Morse bonds but no cross terms
- cvff_nomorse
  - cross terms but quadratic bonds
- cvff_nocross_nomorse
  - no cross terms and quadratic bonds

In addition, the **forcefield** command is used to specify the Discover program's behavior for forcefields that contain automatic parameters. The default is to use automatic parameters for any

---

<table>
<thead>
<tr>
<th>Path/keyword</th>
<th>Values</th>
<th>Default</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>scale</td>
<td>real</td>
<td>1.0</td>
<td>Pathname to scale parameters.</td>
</tr>
<tr>
<td>bond</td>
<td>real</td>
<td>1.0</td>
<td>Scale the bond energy terms.</td>
</tr>
<tr>
<td>angle</td>
<td>real</td>
<td>1.0</td>
<td>Scale the angle energy terms.</td>
</tr>
<tr>
<td>torsion</td>
<td>real</td>
<td>1.0</td>
<td>Scale the torsion energy terms.</td>
</tr>
<tr>
<td>oop</td>
<td>real</td>
<td>1.0</td>
<td>Scale the out-of-plane energy terms.</td>
</tr>
<tr>
<td>vdw</td>
<td>real</td>
<td>1.0</td>
<td>Scale the van der Waals energy terms.</td>
</tr>
<tr>
<td>coulomb</td>
<td>real</td>
<td>1.0</td>
<td>Scale the Coulombic (charge) energy terms.</td>
</tr>
<tr>
<td>cross</td>
<td>real</td>
<td>1.0</td>
<td>Scale all the cross terms.</td>
</tr>
<tr>
<td>hbond</td>
<td>real</td>
<td>1.0</td>
<td>Scale hydrogen-bond energy terms.</td>
</tr>
<tr>
<td>vdw_1_4</td>
<td>real</td>
<td>0.5</td>
<td>Scale the 1-4 van der Waals interactions for the Amber forcefield. No effect for other forcefields.</td>
</tr>
<tr>
<td>coulomb_1_4</td>
<td>real</td>
<td>0.5</td>
<td>Scale the 1-4 Coulombic interactions for the Amber forcefield. No effect for other forcefields.</td>
</tr>
</tbody>
</table>
interactions for which there are no explicit parameters. This can be
disabled on a per-term basis. For instance, setting `oop_automatic`
to false disables automatic parameter assignment for the out-of-
plane interactions.

The `_stop` parameters are used to control whether the Discover
program exits with an error when parameters are not found for
any interaction. The default behavior is that the Discover program
stops if any bond, angle, torsion, or out-of-plane term is not found
but continues for unfound cross terms, just setting the parameters
for any missing cross terms to zero.

The `forcefield` command is also used to control the methods used
in calculating nonbond interactions. Some of the methods avail-
able for nonbonds are:

- Calculate all the nonbond interactions in the system (`summa-
tion method = no_cutoff`).
- Atom-based cutoffs—generate a list of nonbond interactions
  based on all the atoms within the `(cutoff + buffer_width)` dis-
tance from an atom.
- Group-based cutoffs—generate a list of nonbond interactions
  based on the charge groups within the `(cutoff + buffer_width)`
distance from a charge group. The advantage of this method
  over atom-based cutoffs is that dipoles are not broken at the
cutoff (since charge groups are neutral), but it does require the
  specification of the charge groups.
- Use the Ewald, cell-multipole, or cell-based methods.

It is possible to have two different lists—one for van der Waals
interactions (`vdw`) and one for electrostatic interactions (`cou-
lomb`). This allows, for instance, use of a large cutoff for electro-
statics and a smaller cutoff for van der Waals (since these
interactions fall off more rapidly).

The lists are automatically updated whenever any atom moves
more than `buffer_width/2`. This prevents any discontinuities
from arising due to atoms moving from outside the nonbond list
directly into the spline region.

The `forcefield scale` command offers control over the individual
types of energy terms in the forcefield.
Note that the `forcefield` command differs from other Btcl commands in that the parameter values specified are retained, i.e., the parameter default values are set when the command is performed. For example:

```
cdlConfig forcefield select = cvff_nocross parameters
   bond_automatic = FALSE  scale angle = 1.1
forcefield scale bond = 1.1
```

would be identical in functionality and operation to:

```
forcefield select = cvff_nocross parameters
   bond_automatic = FALSE  scale angle = 1.1
forcefield scale bond = 1.1
```

Whether to scale 1–4 interactions is determined by the keyword `amber` in the name of the forcefield file—if the word `amber` is not part of the name of the forcefield used, 1–4 scaling is not done. 1–4 van der Waals and 1–4 Coulombic interactions are scaled separately (as required by the new AMBER forcefield). The default scale factors are 0.5 for both. These two scale factors (`vdw_1_4` and `coulomb_1_4`) just like any other scale factors. However, they have no effect on the energies if `amber` is not part of the name of the forcefield used.

**Note on update_width** keyword: For the Ewald and cell multipole methods, the update frequency of neighbor lists or Taylor coefficients is determined by `update_width`. If `update_width` is small (e.g. 0.0000001 Å), the neighbor lists or Taylor coefficients may be updated at every energy evaluation. By default (`update_width = 1.0 Å`), the neighbor lists or Taylor coefficients are updated whenever an atom moves more than `update_width` from its original position. Thus, the number of steps of dynamics or minimization that occur between updates increases. In minimization runs, an energy discontinuity may result. This is because the neighbor lists or Taylor coefficients are created at one geometry and used for energy evaluations at that geometry as well as the slightly different geometries of subsequent steps. The discontinuity becomes apparent if the final geometry is evaluated later (in a subsequent energy command), since new neighbor lists and Taylor coefficients are generated. An appropriate value of `update_width` depends on the application. More frequent updates of the Taylor coefficients
A. Btcl Language and Commands—Standalone Mode - Continued

slow down the cell multipole method, but make it more accurate. The discontinuity is insignificant for the Ewald method.

**Note** on Ewald accuracy: Although the default Ewald accuracy is acceptable for most single-point energy calculations, a tighter accuracy may be required for some minimization and dynamics runs, to insure acceptable gradient accuracy. In particular, using 0.00025 may be preferable if your minimization run fails to converge or your dynamics run misbehaves.

**Note** on cell multipole method: Currently, the cell multipole method does not calculate second derivatives. For nonperiodic systems with fewer than 200 atoms, the cell multipole method is switched to the no_cutoff method if second derivatives are requested.

Also note that the cell_multipole method does not calculate virials (cell derivatives). Thus, cell minimization and constant-pressure dynamics cannot be carried out using this nonbond method.

If use_quartic is set to true, the cutoff parameter is not used. The nonbond repulsive energy is calculated using the quartic potential formula and is included only when the distance between the pair of atoms is less than the scaled van de Waals radius. The nonbond dispersive and Coulombic parts are automatically turned off. Also the atom_based nonbond summation method is automatically used. The use_quartic keyword should be set to false explicitly if a normal calculation is desired after the use of the quartic potential.

The Buckingham (exponential-6) nonbond interaction potential is also supported. To use this potential form, a section containing the appropriate forcefield parameters should be inserted into the forcefield file, for example:

```
#nonbond(exp-6) cvff
> E = Aij * exp(-r/rho) - Cij/r^6
```

<table>
<thead>
<tr>
<th>!Ver</th>
<th>Ref</th>
<th>I</th>
<th>J</th>
<th>A</th>
<th>rho</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>14</td>
<td>LR</td>
<td>O2</td>
<td>6740.43800</td>
<td>0.34720</td>
<td>0.00000</td>
</tr>
<tr>
<td>1.0</td>
<td>14</td>
<td>LS</td>
<td>OS</td>
<td>6740.43800</td>
<td>0.34720</td>
<td>0.00000</td>
</tr>
</tbody>
</table>

188  C Discover/September 1997
**Example 1**

```plaintext
deforcefield select = cvff_nocross
```

This example of the `forcefield` command selects the `cvff_nocross` forcefield from the current forcefield file. Use of this command causes the energy expression to be recreated with the new forcefield parameters the next time an energy evaluation is performed.

**Example 2**

```plaintext
deforcefield parameters bond_automatic = FALSE \ angle_automatic = FALSE \ torsion_automatic = FALSE \ oop_automatic = FALSE
```

This example of the `forcefield` command disables all use of automatic parameters.

**Example 3**

```plaintext
deforcefield nonbond vdw summation_method = atom_based
```

This example of the `forcefield` command specifies that subsequent calculations use the atom-based method to generate the nonbond summation list. All other `forcefield` parameters are set to their current defaults.

**Example 4**

```plaintext
deforcefield scale angle = 1.5
```

This example of the `forcefield` command indicates that the contributions of the angle energy terms are to be multiplied by 1.5. All other `forcefield` parameters are set to their current defaults.

**Example 5**

```plaintext
deforcefield nonbond +separate_coulomb_cutoffs \ coulomb_cutoff = 12 spline_width = 1.5 \ buffer_width = 0.5
```

In this example, the defaults specify the use of separate nonbond lists for the van der Waals and Coulombic energy terms and set the Coulomb list cutoff to 12 Å, the Coulomb spline width to 1.5 Å, and the Coulomb buffer width to 0.5 Å. This corresponds to a
spline region going from the full value at 10.0 Å down to zero at 11.5 Å (followed by a 0.5-Å buffer region before the cutoff at 12.0 Å).

**Example 6**

```bash
forcefield scale \n  vdw_1_4 = 0.4 \n  coulomb_1_4 = 0.4
```

This Btcl statement is used to scale both 1–4 van der Waals and 1–4 Coulombic interactions by a factor of 0.4 for the AMBER force-field. The statement has no effect if the forcefield used is not AMBER.

---

**geometry**

**Purpose**

The `geometry` command is a general-purpose command to create, review, analyze, and manipulate geometry objects.

**Syntax**

*Direct review and assignment of geometry objects:*

```
gamey varName ?geometry_object?
```

*Get-type operations:*

```
gamey varName get_operation arguments
```

where `arguments = geometry_object ?(geometry_object | keyword value) ...?`

*Set-type operations:*

```
gamey matName matrix ?relative? set_operation arguments
gamey varName transform ?relative? set_operation arguments
gamey varName transform matrix
```

where `arguments = ?relative? target geometry_object geometry_object ?(geometry_object | keyword value) ...?`

*Geometry operations:*
<table>
<thead>
<tr>
<th>Operation</th>
<th>Arguments</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>point</strong></td>
<td>arg</td>
<td>Create a point object which is the point part of the argument arg. The argument may be a geometry point, line, or plane object.</td>
</tr>
<tr>
<td><strong>line</strong></td>
<td>arg1 arg2</td>
<td>Create a line object which is the line between the points of 2 geometric objects, arg1 and arg2. These arguments may be 2 points, 2 lines, a point and a line, or a point and a plane.</td>
</tr>
<tr>
<td><strong>plane</strong></td>
<td>arg1 arg2 arg3</td>
<td>Create a plane object which is the plane which passes through the three specified geometry point objects arg1, arg2, and arg3, which must not be colinear.</td>
</tr>
<tr>
<td><strong>vector</strong></td>
<td>arg1 arg2</td>
<td>Create a point object (a vector) defined between 2 geometric objects, arg1 and arg2. These arguments may be 2 points, 2 lines, a point and a line, or a point and a plane. When the first argument is a line, the vector is perpendicular to that line, with direction and magnitude given by the second argument. If the second argument is also a line, then the vector is perpendicular to both lines. If either argument is a plane, then the vector is normal to that plane. When only one argument is provided, it must be a plane or a line, in which case the vector is the vector part of that plane (the normal) or line.</td>
</tr>
<tr>
<td><strong>get</strong></td>
<td>nItems</td>
<td>Return either the number of items or the type of an existing geometry object.</td>
</tr>
<tr>
<td><strong>get Item</strong></td>
<td>ItemNum geoObject</td>
<td>Create a geometry object which is the ItemNum’th item of an existing geometry object geoObject.</td>
</tr>
<tr>
<td><strong>concat</strong></td>
<td>geoObject1 geoObject2</td>
<td>Create a geometry object which is the concatenation of two existing geometry objects geoObject1 and geoObject2. These geometry objects must be of the same type unless either is empty or the first is an array type. If both are of type array, the second is appended to the end of the contents of first directly, i.e., the contents of the second are not appended to the contents of the first. This allows arrays of geometry arrays to be composed.</td>
</tr>
<tr>
<td><strong>distance</strong></td>
<td>?target? arg1 arg2</td>
<td>Determine the distances between 2 geometric objects arg1 and arg2 or transform a geometric object by the matrix that would move arg2 to set the distances along the vectors to arg2 to values target. The arguments may be 2 points, a point and a line, a point and a plane, or 2 lines.</td>
</tr>
<tr>
<td><strong>angle</strong></td>
<td>?target? arg1 arg2 arg3 arg2</td>
<td>Determine the angle between 2 or 3 geometric objects arg1, arg2, and arg3? or transform a geometric object by the matrix that would move the last of the geometric object defining the angles to set them to values target. The arguments may be 3 points, 2 points and a line, 2 points and a plane, 2 lines, a line and a plane, or 2 planes. The set operation only supports the input of 3 points.</td>
</tr>
<tr>
<td><strong>torsion</strong></td>
<td>?target? arg1 arg2 arg3 arg4</td>
<td>Determine the torsion angles between 4 geometric point objects arg1, arg2, arg3, and arg4 or transform a set of geometric points by the matrix that would move the last of the points (arg4) so that the torsion angles would be set to values target.</td>
</tr>
</tbody>
</table>
A. Btcl Language and Commands—Standalone Mode - Continued

<table>
<thead>
<tr>
<th>Operation</th>
<th>Arguments</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>outOfPlane</td>
<td>?target? arg1 arg2 arg3 arg4</td>
<td>Determine the Wilson out-of-plane angles between 4 geometric point objects arg1, arg2, arg3, and arg4 or transform a set of geometric points by the matrix that would move the last of the points (arg4) so that the out-of-plane angles would be set to values target.</td>
</tr>
<tr>
<td>superimpose</td>
<td>?target? arg1 arg2</td>
<td>Transform a set of target points by the matrix used to produce the maximum overlap of the point objects arg1 and arg2, such that points arg2 would be superimposed on points arg1. Keywords rms and weight may be used to obtain the rms residue from the overlap of arg1 and arg2 and to weight the points for the maximum overlap or rms calculation.</td>
</tr>
<tr>
<td>rms</td>
<td>arg1 arg2</td>
<td>Return the root-mean-square residual for the maximum overlap of 2 sets of points defined by geometric point objects arg1 and arg2 to varName. Using the weight keyword, it is possible to set weights on points for the maximum overlap or rms calculation.</td>
</tr>
<tr>
<td>principalAxes</td>
<td>arg</td>
<td>Create a geometric line object that is the centroid and principal axes through the set of points in the geometric point object arg. This operation also takes optional keywords.</td>
</tr>
<tr>
<td>lsqLine</td>
<td>arg</td>
<td>Create a geometric line object which is the least-squares fit line through the set of points in the geometric point object arg. This operation also takes optional keywords.</td>
</tr>
<tr>
<td>lsqPlane</td>
<td>arg</td>
<td>Create a geometric plane object which is the least-squares fit plane through the set of points in the geometric point object arg. This operation also takes optional keywords.</td>
</tr>
<tr>
<td>translate</td>
<td>distance vector</td>
<td>Translate a set of points along the vectors defined by the point geometry object vector by the distances (magnitudes) specified by values distance.</td>
</tr>
<tr>
<td>rotate</td>
<td>angle axis</td>
<td>Rotate a set of points about the lines defined by the line geometry object axis by the angles specified by values angle.</td>
</tr>
<tr>
<td>orientAxes</td>
<td>arg1 ?arg2? ?arg3?</td>
<td>Transform a set of points by the matrix which would move the geometric line objects arg1, arg2, and arg3 onto the external frame axes, i.e., the Cartesian z-, y-, and x-axes, respectively. The first set of lines arg1 is always moved exactly onto the z-axis at the origin. If the second set of lines, arg2, is provided, then the vectors must be normal to those of arg1, and the transformation matrix rotates about the z-axis to align these vectors with the y-axis. If the third set of lines, arg3, is provided, then a further check is made that the transformed vectors of these lines lie parallel with the x-axis in the positive direction.</td>
</tr>
</tbody>
</table>
Geometry objects

The arguments to a `geometry` command are always geometry objects, i.e., a geometry object definition string or a handle to a geometry object that already exists. There are actually four types of geometry objects:

1. Point objects
   
   ```
   ?point? vector
   ```
   
   A point geometric object is represented by the Cartesian coordinates of a set of points. A point geometric object is equivalent to a vector object (see the `vector` command).

2. Line objects
   
   ```
   line vector vector
   ```
   
   A line geometric object is represented by the Cartesian coordinates of a set of points on the set of lines and a set of unit vectors along the directions of a set of lines.

3. Plane objects
   
   ```
   plane vector vector
   ```
   
   A plane geometric object is represented by the Cartesian coordinates of a set of points on the set of planes and a set of unit vectors perpendicular to a set of planes.

4. Array objects
   
   ```
   array geometry_object ?geometry_object ...?
   ```
   
   An array is a collection of point, line, plane, and/or array geometry objects. Currently, array geometry objects may not be used as arguments to `geometry` commands, but they may be used in geometry set-type operations as the target set to be transformed. For further explanation of the definition of geometry objects and the use of geometry array objects, refer to the Background section and examples below.

Geometry operations

The most basic use of the `geometry` command is to define or examine (convert to a Tcl list) geometry objects. However, most of the power of the `geometry` command comes from the ability to ana-
A. Bcl Language and Commands—Standalone Mode - Continued

lyze and manipulate geometry objects. This is achieved by adding extra operation arguments to the geometry command, as detailed in the Syntax section and operations table. There are two distinct types of geometry operation:

1. Get-type geometry operations

The syntax for a get-type operation requires that the second argument to the geometry command, after the variable name of the geometry object to be output (i.e., got), is a keyword defining the operation. These are listed in the Geometry Operations table above.

2. Set-type geometry operations

The syntax for a get-type operation requires that the second argument to the geometry command, after the variable name of the geometry object which is to be transformed or created (set), is the keyword transform or matrix. Set-type geometry operations can be performed in two ways:

The first way is to directly use the transform keyword followed by an operation keyword to perform the desired transformation, i.e.:

```
geometry geoMoveSet transform operation target arg1 ...
```

where geoMoveSet would be a geometry object to be transformed.

The second way is to perform the same operation in two steps by first creating the transformation matrix using the matrix keyword and then transforming using the transform keyword followed by a geometry (matrix) object rather than an operation keyword, i.e.:

```
geometry geoMat matrix operation target arg1 ...
geometry geoMoveSet transform $geoMat
```

For some geometry operations, e.g., distance, the transform or matrix keywords may be followed by the relative keyword (and then the operation, etc.). This connotes that value(s) specified by the target argument are to be added to the current value of the property, e.g., distance, to specify the actual target for the transformation.
Other geometry get-type and set-type operations allow specific keywords to appear in the argument list, followed by a geometry object or variable name to specify extra, optional, input or output arguments, e.g., \texttt{rms} for the \texttt{lsqLine} operation.

A summary of the get-type and set-type operations, and lists of optionally allowed keywords, are tabulated below.

All angles are specified in radians. This is the same as for the \texttt{vector} command.

\begin{center}
\textit{Summary of geometry operations and optional argument keywords}
\end{center}

\begin{table}[h]
\begin{tabular}{|l|c|c|c|}
\hline
Get-type operations & Optional keywords & Set-type operations & Optional keywords \\
\hline
\texttt{point} & — & — & — \\
\texttt{line} & — & — & — \\
\texttt{vector} & — & — & — \\
\texttt{distance} & — & \texttt{distance} & \texttt{relative} \\
\texttt{angle} & — & \texttt{angle} & \texttt{relative} \\
\texttt{torsion} & — & \texttt{torsion} & \texttt{relative} \\
\texttt{outOfPlane} & — & \texttt{outOfPlane} & \texttt{relative} \\
\hline
\texttt{rms} & \texttt{weight} & — & — \\
\texttt{principalAxes} & \texttt{weight} & — & — \\
& \texttt{moment} & — & — \\
\texttt{lsqLine} & \texttt{weight} & — & — \\
& \texttt{ms} & — & — \\
\texttt{lsqPlane} & \texttt{weight} & — & — \\
& \texttt{ms} & — & — \\
& — & \texttt{translate} & — \\
& — & \texttt{rotate} & — \\
& — & \texttt{orientAxes} & — \\
\hline
\end{tabular}
\end{table}
Optional keyword arguments to geometry operations

**weight vector**

The point geometry object vector is used as a set of scaling values to weight the points in the principal axes, least-squares line, least-squares plane, or root-mean-squares calculations. Typically, the **weight** argument vector would be, for example, particle masses.

**rms varName**

The rms residual from a least-squares line, least-squares planes, or root-mean-squares calculation is returned to a geometry object with name varName.

**moment varName**

The principal moments with respect to the principal axes, generated in the **principalAxes** operation, are returned to a geometry object with name varName.

### Background

#### Specifying geometry objects

The basic data type in the Discover geometry math package is the geometry object. A geometry object consists of a number of items which are handled in the same manner in an operation involving the object. One single item of a geometry object can be a point, line, plane, or an array of them.

For example, a **point** geometry object can be specified by their Cartesian coordinates and represented as a string:

"{1 2 3} {4 5 6}" or "point {{1 2 3} {4 5 6}}".

The geometry operations currently supported are described above. Geometry objects can be input as character strings. To specify if an input string is to be converted to a geometry object of points, lines, or planes, the keyword **line** or **plane** are needed for the latter two cases. If the keyword **point** is specified or if there is no keyword specified, the input string is converted to a **point** geometry object of points.

The string representations of geometry objects are illustrated in the examples below.

#### Definitions of types of angles

The angle formed by three points is defined by:
A torsion or dihedral angle formed by four points is defined by:

\[ \theta \equiv \cos^{-1} \left( e_{21} \cdot e_{23} \right) ; \quad 0 \leq \theta \leq \pi \]

An out-of-plane angle formed by four points is defined by:

\[ \theta \equiv \text{sign} \cos^{-1} \left( -[(e_{12} \times e_{23}) \cdot (e_{43} \times e_{32})] \right) ; \quad -\pi \leq \theta \leq \pi \]

where \( \text{sign} \) is the sign of:

\[ [(e_{12} \times e_{23}) \times (e_{43} \times e_{32})] \cdot e_{23} \]

An out-of-plane angle formed by four points is defined by:

\[ \theta \equiv \sin^{-1} \left( (e_{12} \times e_{13}) \cdot e_{14} \right) ; \quad -\frac{\pi}{2} \leq \theta \leq \frac{\pi}{2} \]

**Operations on geometry objects**

A geometry object can be created only with the `geometry` command but can be deleted using the Btcl `unset` command or by reassigning the object name using a `set` or another `geometry` command. When this occurs, the data associated with that object are also deleted.

```
BTCL > geometry g {{1 2 3}}
```

or
A. Btcl Language and Commands—Standalone Mode - Continued

```
BTCL > geometry g "(1 2 3)"

Here, `g` is set to a point (1,2,3). To find out what a geometry object (`g` here) holds, the syntax would be:

```
BTCL > geometry g
   (1 2 3)
```

The Btcl `geometry` command also takes a number of operations (listed in the table above), which themselves take arguments that are either string definitions or geometry variables of compatible dimensions. Refer to the examples below.

There are two types of geometry operations. The `get` operations evaluate and return the configuration formed by the input geometric objects. The `set` operations are used to determine the transformation matrix needed to achieve a certain configuration, given by the `target`, by moving one of the input geometric objects. The geometric object to be transformed is specified, by convention, to be the last one in the input. See Examples 2 and 3 below.

**Example 1: Assignment of geometric objects**

```
BTCL > geometry p0 "(0 0 0)" ; geometry p1 "(1 1 1)" ; geometry p2 "(1 0 0)"

Assign the point (0,0,0) to be geometric object p0, (1,1,1) to be geometric object p1, and (1,0,0) to be geometric object p2.

BTCL > geometry p3 "(0 1 0) (0 0 1)" ; geometry p3
   {0 1 0} {0 0 1}

Assign two points, (0,1,0) and (0,0,1), to be geometric object p3. Note that p3 is a point geometric object and is, in general, a set of points.

BTCL > geometry l1 "line $p0 {{1 0 1} {-1 0 0}}" ; geometry l1
   line {{1 0 1}} {{0.707107 0 0.707107} {-1 0 0}}
```

Assigns two lines to be geometric object l1. The first line has direction (1,0,1) and passes through the point (0,0,0), and the second line has direction (-1,0,0) and passes through the point (0,0,0). Note that the vectors along the directions of the lines are normalized.
Example 2: Get-type geometry operations

BTCL > geometry l2 line $p2 $p3; geometry l2
   line {{1 0 0}} {{-0.707107 0.707107 0} {-0.707107 0 0.707107}}

Get the line formed from p2 to p3. Note that, since both lines given
in l2 pass through the point (1,0,0), the point is given only once in
l2. The vectors parallel to the lines, (-0.707107 0.707107 0) and
(-0.707107 0 0.707107), are normalized.

BTCL > geometry d1 distance $p0 $l2; geometry d1
   0.707107 0.707107

Get the distances between p0 and l2.

BTCL > geometry a1 torsion $p2 $p0 $p3 $p1; geometry a1
   -0.785398 0.785398

Get the dihedral angle formed by p2, p0, p3 and p1.

BTCL > geometry inp "(1 1 -1) (-1 1 1) (1.5 0 -1) (-1 1 1) (1 -1 1) (1.5 0 1)"
BTCL > geometry ref "(1 1 -1) (-1 1 1) (1.5 0 -1) (-1 1 1) (1 -1 1) (1.3 0 1)"
BTCL > geometry wt "2 1 1 1 1 1"

BTCL > geometry l3 lsqLine $inp weight $wt rms res1
BTCL > geometry l3
   line {{0.571429 0.428571 0.142857}} {{-0.758033 0.163668 0.631346}}
BTCL > geometry res1
   1.56542

Get the least-squares line passing through the points given in $inp,
weighted by the weights given in $wt, and the root-mean-square
residual.

BTCL > geometry res2 rms $inp $ref weight $wt; geometry res2
   0.0669741

Get the minimum root-mean-square residual from comparing the
set of points given in $ref and the set of points given in $inp after
they are transformed (to give the minimum residual).
**Example 3: Set-type geometry operations**

```
BTCL > geometry a "(1 2 3)"; geometry b "(1 4 3)"

BTCL > geometry c "array {{0 0 0}} {{1 1 1}}"; geometry c
   array {{0 0 0}} {{1 1 1}}

BTCL > geometry c transform distance 5 $a $b; geometry c
   array {{0 3 0}} {{1 4 1}}

BTCL > geometry c transform relative distance 4 $a $b; geometry c
   array {{0 7 0}} {{1 8 1}}
```

In the above sequence, we assigned \( c \) to be a geometry array object that is an array of two geometric point objects. We apply to \( c \) the transformation that would move geometric point object \( b \) along the vector formed by geometric point objects \( a \) and \( b \) to be at a distance 5 from geometric point object \( a \). A further move was made to the transformed \( c \) with a transformation that would move \( b \) by a relative distance of 4 along the \( a-b \) vector, i.e., the magnitude of the \( a-b \) vector would increase by 4 units.

```
BTCL > geometry p1 "(-2 -3 1) (-2 -2 0) (-2 -2 0) (-2 -2 0)"

BTCL > geometry p2 "(-1 -1 1) (-1 0 0) (-1 0 0) (-1 0 0)"

BTCL > geometry p3 "(1 -1 1) (1 0 0) (1 0 0) (1 0 0)"

BTCL > geometry p4 "(2 -3 1) (2 -2 2) (2 0 2) (2 2 2)"

BTCL > geometry tor1 torsion $p1 $p2 $p3 $p4; geometry tor1
   0 -0.785398 -1.5708 -2.35619

BTCL > geometry target "(-1) (-2) (-4) (-6)"

BTCL > geometry mat1 matrix torsion $target $p1 $p2 $p3 $p4

BTCL > geometry p4 transform $mat1; geometry p4
   (2 -2.0806 2.68294) (2 1.17704 2.57188) (2 1.30729 -1.5136) (2 -2.71577
   -0.790306)

BTCL > geometry tor2 torsion $p1 $p2 $p3 $p4; geometry tor2
   -1 -2 2.28319 0.283185

BTCL > geometry p4 transform torsion $target $p1 $p2 $p3 $p4
```
In the above sequence, we first determine the dihedral angle formed by points p1, p2, p3, and p4. Then we find the transformation mat1 needed to take p4 to a new position such that the new dihedral angle is as given in the target $target. We test our transformation by actually carrying out the transformation on p4 and finding the new dihedral angle explicitly.

Note that the two steps of finding the transformation matrix and performing the transformation can be implicitly done with the command:

```
BTCL > geometry p4 transform torsion $target $p1 $p2 $p3 $p4
```

Here, we generate the transformation matrix implicitly and hence do not have access to it.

```
BTCL > geometry inp transform superimpose $inp $ref weight $wt rms res3
```

Here, we determine the transformation matrix implicitly and apply it to the set of test points $inp. We also determine the root-mean-square residual and return its value in res3.

### Example 4: Geometry object operations

```
BTCL > geometry a "(1 2 3)"; geometry b "(1 4 3)"

BTCL > geometry c concat $a $b

BTCL > geometry c concat $c {{0 0 0}}
```
A. Btcl Language and Commands—Standalone Mode - Continued

**BTCL >** echo [geometry c]
{1 2 3} {1 4 3} {0 0 0}

The example above shows the use of the geometry `concat` operation. First two point geometry objects are defined and concatenated to form another geometry object `c`. The `concat` operation is then used again to add another point (0,0,0) to `c`.

**BTCL >** proc sum_dists {varName geo1 geo2} {
upvar 1 $varName dists
$ng = [geometry b get nItems]
vector dists fill $ng 0
$ng = [geometry a get nItems]
for {$i = 0} {$i < $ng} {incr i} {
    geometry c getItem $i $a
    geometry d distance $c $b
    vector dists add $dists $d
}
}

**BTCL >** geometry a "(1 2 3) (4 5 6) (7 8 9)"
**BTCL >** geometry b "line {(0 0 0) (1 0 0)} {(0 1 0) (0 0 1)}"
**BTCL >** sum_dists distance_sum $a $b
**BTCL >** echo [geometry distance_sum]
21.7751 17.831

The example above shows the use of the geometry `get` and `getItem` operations from inside a procedure named `sum_dists`. This procedure is designed to take two geometry objects and calculate the summed distances from all the items in the first geometry object to all the items in the second geometry object. Given two geometry objects, one containing \(N\) items, it is possible to perform geometry operations, such as `distance`, only if the other object contains exactly 1 or \(N\) items. The procedure `sum_dists` allows \(M\) to \(N\) operations by performing \(M\) (1 to \(N\)) operations, using the `getItem` operation to access individual items in geometry objects.

In this example, this procedure is used to find the summed distances between three points \(a\) and two lines \(b\).
help

Purpose

The **help** command prints information in table form about the commands available in the Discover program.

Syntax

```
```

Description

The **help** command can be used to print out information about a command or a single parameter of a command.

For a main parameter (keyword), the information concerns the parameter path, the parameter aliases, the parameter type, and its current default value. A list of parameters (for a command) or sub-parameters (for a main parameter) is printed if they exist.

Extra information about a particular command or parameter may also be printed. If the last argument to help is an asterisk (*), information about all parameters that branch off the specified parameter path is printed.

If the first argument to **help** is not a Btcl command name, **help** tries to give information as to the type of object the argument is, for example, if it is a known variable or procedure name.

Example 1

```
help
```

The command **help** with no arguments produces a list of all the Btcl commands defined in CDiscover.

Example 2

```
help *
```

This example of the **help** command prints information about all the Btcl commands and their parameters.
A. Btcl Language and Commands—Standalone Mode - Continued

Example 3

```
help minimize final_convergence
```

This example of the `help` command prints information concerning the `minimize final_convergence` parameter.

Example 4

```
help minimize newton *
```

This example of the `help` command prints information concerning the `minimize newton` parameter node and all its subparameters (method, convergence, and line_search_precision).

---

**minimize**

**Purpose**

The purpose of the `minimize` command is to refine the structure of a model by regressing its potential energy to a local minimum.

**Syntax**

```
```

<table>
<thead>
<tr>
<th>Path/keyword</th>
<th>Values</th>
<th>Default</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>method</td>
<td>cascade steepest conjugate newton</td>
<td>cascade</td>
<td>Select the minimization scheme.</td>
</tr>
<tr>
<td>final_convergence</td>
<td>real</td>
<td>0.001</td>
<td>Stopping criterion for maximum derivative.</td>
</tr>
<tr>
<td>iteration_limit</td>
<td>integer</td>
<td>300</td>
<td>Maximum number of iterations to perform.</td>
</tr>
<tr>
<td>sd</td>
<td>Pathname to parameters for steepest-descents method.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>convergence</td>
<td>real</td>
<td>1000.0</td>
<td>Maximum derivative to achieve with steepest-descents method.</td>
</tr>
<tr>
<td>line_search_precision</td>
<td>real</td>
<td>0.1</td>
<td>Precision of the line minimization at each iteration of steepest descents.</td>
</tr>
<tr>
<td>peek</td>
<td>string</td>
<td>peek</td>
<td>Btcl command called after every iteration to write out .ipc and/or .pcar files to monitor the progress of a minimization. Value may be &quot;&quot; to turn off the peek mechanism.</td>
</tr>
<tr>
<td><strong>Path/keyword</strong></td>
<td><strong>Values</strong></td>
<td><strong>Default</strong></td>
<td><strong>Meaning</strong></td>
</tr>
<tr>
<td>-----------------</td>
<td>------------</td>
<td>-------------</td>
<td>-------------</td>
</tr>
<tr>
<td>cg</td>
<td></td>
<td></td>
<td><strong>Pathname to parameters for conjugate-gradients method.</strong></td>
</tr>
<tr>
<td>method</td>
<td>polak fletcher</td>
<td>polak</td>
<td><strong>Select a particular conjugate-gradients algorithm.</strong></td>
</tr>
<tr>
<td>convergence</td>
<td>real</td>
<td>10.0</td>
<td><strong>Maximum derivative to achieve with conjugate-gradients method.</strong></td>
</tr>
<tr>
<td>line_search_precision</td>
<td>real</td>
<td>0.1</td>
<td><strong>Precision of the line minimization at each iteration of conjugate gradients.</strong></td>
</tr>
<tr>
<td>newton</td>
<td></td>
<td></td>
<td><strong>Pathname to parameters for Newton methods.</strong></td>
</tr>
<tr>
<td>method</td>
<td>bfgs dfp newton_raphson_truncated</td>
<td>bfgs</td>
<td><strong>Select a particular Newton algorithm.</strong></td>
</tr>
<tr>
<td>convergence</td>
<td>real</td>
<td>0.001</td>
<td><strong>Maximum derivative to achieve with any Newton method.</strong></td>
</tr>
<tr>
<td>line_search_precision</td>
<td>real</td>
<td>0.9</td>
<td><strong>Precision of the line minimization at each iteration of the chosen Newton method.</strong></td>
</tr>
<tr>
<td>movement_limit</td>
<td>real</td>
<td>0.2</td>
<td><strong>Limit how far atoms can move in one step.</strong></td>
</tr>
<tr>
<td>cell</td>
<td>Boolean</td>
<td>false</td>
<td><strong>For periodic systems, whether to include cell parameters as minimization variables.</strong></td>
</tr>
<tr>
<td>execute</td>
<td></td>
<td></td>
<td><strong>Pathname to execute structures (a separate execute structure is passed to the minimize command for each occurrence of execute).</strong></td>
</tr>
<tr>
<td>command</td>
<td>string</td>
<td>&quot;echo hello&quot;</td>
<td><strong>A full Btcl command, enclosed in quotes (&quot; &quot;) or braces ({}), to be executed during minimization (e.g., print).</strong></td>
</tr>
<tr>
<td>before</td>
<td>Boolean</td>
<td>false</td>
<td><strong>Execute command before minimization run begins.</strong></td>
</tr>
<tr>
<td>after</td>
<td>Boolean</td>
<td>false</td>
<td><strong>Execute command after minimization run ends (ignored if the command was already executed in the last minimization step).</strong></td>
</tr>
<tr>
<td>frequency</td>
<td>real</td>
<td>0.0</td>
<td><strong>Frequency (in fs) with which to execute command.</strong></td>
</tr>
<tr>
<td>first_step</td>
<td>real</td>
<td>0.0</td>
<td><strong>Start executing the command after this period of simulated time (fs).</strong></td>
</tr>
<tr>
<td>last_step</td>
<td>real</td>
<td>0.0</td>
<td><strong>Stop executing the command after this period of simulated time (0.0 means to keep executing the command until the run is complete).</strong></td>
</tr>
</tbody>
</table>
Description

The `minimize` command does not alter any forcefield parameters. It simply minimizes the energy function that has already been set up.

Three minimization methods are available: steepest descents, conjugate gradients, and Newton methods. For highly strained structures, the steepest-descents method is recommended. After the initial strain is removed, the more rapid (linearly convergent) conjugate gradients should be used. If a highly accurate energy minimum is desired, one or more of the Newton methods can be used, since these are quadratically convergent.

The minimizer always starts with the steepest-descents method to make sure any severe strain is removed. After `sd convergence` is achieved, it stops if the method is set to `steepest`; otherwise, it proceeds to the conjugate-gradients method.

Conjugate-gradients minimization is done with the method selected by `cg method` until `cg convergence` is achieved.

If the chosen method is `newton` (or `cascade`), the minimization proceeds with `newton method` until `newton convergence` is reached. However, if convergence is achieved or if `iteration_limit` is exceeded before reaching the end of the sequence of methods, minimization terminates at that point.

There are two slightly different algorithms for conjugate gradients: Polak–Ribiere and Fletcher–Reeves, which can be selected by the `cg method` keyword.

Of the various Newton methods, the Davidon–Fletcher–Powell (dfp) and Broyden–Fletcher–Goldfarb–Shanno (bfgs) methods build a second-derivative matrix iteratively so that the matrix is equivalent to the true second-derivative matrix upon convergence.

Newton–Raphson and truncated Newton–Raphson methods compute the analytic second derivatives at each iteration. Thus, the quasi-Newton methods require less time per iteration but more iterations to achieve convergence than Newton–Raphson methods.

For larger systems, each iteration of the truncated method is faster than the corresponding iteration of the regular Newton–Raphson. However, the latter is expected to be more robust. All the Newton–
Raphson methods require storing the matrix of the second derivatives, which can take prohibitively large amounts of memory for systems having more than a few hundred atoms.

**movement_limit** insures against the possibility of obtaining unrealistic structures by moving a particular atom by a large distance in one step.

**sd line_search_precision**, **cg line_search_precision**, and **newton line_search_precision** indicate how accurately a line minimization is to be done at each iteration of the respective method. Each is expressed as the ratio of the derivative along the line search direction to the initial derivative. The value should be between (but not including) zero and one and may have some minor effect on the overall speed of minimization.

When a periodic system is used, by default, the cell dimensions and shape are fixed. If the **cell** keyword appears in the command, then the cell parameters are modified to find the cell dimensions and angles that optimize the energy.

The **execute** pathname keyword and its associated parameters are used to specify subcommands to be performed during a minimization run and when they should be performed.

The execute structure typically contains the **print** command, to write out the instantaneous and average values that are calculated during a run (see the **print** command and the examples below). The frequency, beginning, and end of the subcommand execution are specified as step number (unlike in the **dynamics** command, where they are specified in femtoseconds).

**Example 1**

```
minimize
```

This command uses default values either internally set in the program or previously set by the user for all the parameters listed above. No printing is done, except for a summary when minimization is completed.

By default, the minimizer starts with steepest descents, then cascades into conjugate gradients as the derivative becomes smaller, and finally refines the structure by the BFGS method.
A. Btcl Language and Commands—Standalone Mode - Continued

Example 2

minimize method = conjugate \ 
  cg convergence = 0.01 sd convergence = 10.0

Here the user has chosen not to use any Newton method. Instead, the convergence criterion for steepest descents and the stopping criterion are tightened.

Example 3

minimize iteration_limit = 500, newton method = truncated

Here, an accurate minimum is sought with the truncated Newton–Raphson method by increasing the iteration limit to 500 from the default of 300.

Example 4

minimize execute frequency = 10 \ 
  command = {print output +energy_summary}

This command line indicates to perform minimization with the default options, printing an energy summary every 10 steps during the minimization. (See the dynamics Example 3 for additional information on how to use the execute structure.)

Example 5

minimize peek = {peek cpu_time_step = 2.0 filename = look +carfile} \ 
  execute frequency = 10 command = {print output +energy_summary}

This command line performs the same minimization as in the previous example, except every 2 seconds (CPU time) the files look.pek and look.pcar are generated. These contain the energy and coordinates, respectively, of the system at that point in the minimization process. (See the peek command for more information.)
molGeom

Purpose

The molGeom command is used to get and set geometric properties of models, such as the coordinates of atoms, bond lengths, bond angles, etc. NOTE: Unlike the geometry and vector commands, this command takes angle values in degrees.

Syntax

molGeom operation attribute ?relative?

The molGeom command operations:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Arguments</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>get</td>
<td>attribute varName arg1 ?arg2 ...?</td>
<td>Get the geometrical attribute attribute to an object varName according to the atom sets specified by the arguments arg2 (etc.).</td>
</tr>
<tr>
<td>set</td>
<td>attribute ?relative? target arg1 ?arg2 ...?</td>
<td>Set the geometrical attribute attribute to a target list of values target according to the atom sets specified by the arguments. If the optional keyword relative is given after the attribute argument, then target is added to the existing values.</td>
</tr>
<tr>
<td>rotate</td>
<td>axis target arg1</td>
<td>Rotate the set of atoms arg1 by target degrees about the axis given by the line object axis.</td>
</tr>
<tr>
<td>translate</td>
<td>vector target arg1</td>
<td>Translate the set of atoms arg1 by target angstroms along the vector vector.</td>
</tr>
</tbody>
</table>

The molGeom command geometry attributes:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Operations</th>
<th>Arguments</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>coords</td>
<td>get/set</td>
<td>1</td>
<td>Store in varName or set to target the coordinates of the atoms specified by the argument.</td>
</tr>
<tr>
<td>line</td>
<td>get</td>
<td>2</td>
<td>Create a geometry line object varName, which are the lines between the coordinates of the atom pairs specified by the arguments.</td>
</tr>
</tbody>
</table>
A. Btcl Language and Commands—Standalone Mode - Continued

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Operations</th>
<th>Arguments</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>plane</td>
<td>get</td>
<td>3</td>
<td>Create a geometry plane object varName, which are the planes between the coordinates of the atom triplets specified by the arguments.</td>
</tr>
<tr>
<td>rms</td>
<td>get</td>
<td>2</td>
<td>The arguments specify two sets of atoms for which an rms comparison is made. The rms residual is calculated from the maximum overlap of the atom coordinates and is returned in varName.</td>
</tr>
<tr>
<td>angle</td>
<td>get/set</td>
<td>3</td>
<td>Same as for the distance attribute, but for angles specified by the arguments.</td>
</tr>
<tr>
<td>torsion</td>
<td>get/set</td>
<td>4</td>
<td>Same as for the distance attribute, but for torsion angles specified by the arguments.</td>
</tr>
<tr>
<td>outOfPlane</td>
<td>get/set</td>
<td>4</td>
<td>Same as for the distance attribute, but for out-of-plane angles specified by the arguments.</td>
</tr>
<tr>
<td>superimpose</td>
<td>set</td>
<td>2</td>
<td>The arguments specify two sets of atoms which are used to generate a matrix, which would transform the second set of atoms to produce the maximum overlap (minimum rms) with the first set of atoms. This matrix is then used to transform the set of atoms specified by target.</td>
</tr>
<tr>
<td>distance</td>
<td>get/set</td>
<td>2</td>
<td>Store in varName or set to target the distances between the atom sets defined by the arguments. For the set operation the set of atoms specified by the last argument, and sets of atoms attached to them, are moved to set the target distance. This operation is valid only if each member of a pair of atoms specified by the args is directly bonded and is not in a ring structure, or if the atoms are in separate fragments.</td>
</tr>
<tr>
<td>principalAxes</td>
<td>get</td>
<td>1</td>
<td>Store in varName the principal axes through the list of atoms specified by the argument. The object returned is a geometric line object composed of a point centroid and three unit principal axis vectors.</td>
</tr>
<tr>
<td>lsqLine</td>
<td>get</td>
<td>1</td>
<td>Store in varName the least-squares-fit line through the list of atoms specified by the argument. The object returned is a geometric line object.</td>
</tr>
<tr>
<td>lsqPlane</td>
<td>get</td>
<td>1</td>
<td>Store in varName the least-squares-fit plane through the list of atoms specified by the argument. The object returned is a geometric plane object.</td>
</tr>
</tbody>
</table>

Description

The molGeom command is used to perform basic geometric analysis and manipulation on models. The molGeom command is actually a Tcl procedure (located in $BIOSYM/data/utility/molGeom.tcl) that employs the $dbHandle get and set operations and
the subset, subStructure, and geometry commands to take atom and subset specification arguments and manipulate the coordinates they refer to.

The arguments specifying the attribute to get or set may be defined either by a single database subset specification or by a number of arguments (given in the Arguments column of the table), which each specify a single list or subset of atoms. This is shown more clearly in Example 1.

**Definitions of types of angles**

The angle formed by three points is defined by:

\[ \theta = \cos^{-1} (e_{21} \cdot e_{23}) ; \quad 0 \leq \theta \leq \pi \]

\[ \begin{array}{c}
\text{1} \\
\text{2} \\
\text{3}
\end{array} \quad \begin{array}{c}
e_{21} \quad e_{23}
\end{array} \]

A torsion or dihedral angle formed by four points is defined by:

\[ \theta = \text{sign} \cos^{-1} (-[(e_{12} \times e_{23}) \cdot (e_{43} \times e_{32})]) ; \quad -\pi \leq \theta \leq \pi \]

where sign is the sign of:

\[ [- (e_{12} \times e_{23}) \times (e_{43} \times e_{32})] \cdot e_{23} \]

\[ \begin{array}{c}
\text{1} \\
\text{2} \\
\text{3} \\
\text{4}
\end{array} \quad \begin{array}{c}
e_{12} \quad e_{32} \\
e_{23} \quad e_{43}
\end{array} \]

An out-of-plane angle formed by four points is defined by:

\[ \theta = \sin^{-1} [(e_{12} \times e_{13}) \cdot e_{14}] ; \quad \frac{-\pi}{2} \leq \theta \leq \frac{\pi}{2} \]
A. Btcl Language and Commands—Standalone Mode - Continued

Notes

The molGeom set torsion command works only for nonperiodic models.

Example 1

```
BTCL > subset define Subset "*:nitros" "GLY_*:N"
BTCL > subset define Angle "glyAngles" "{*:GLY_*:C} {*}:GLY_*:CA} {*}:GLY_*:N"
BTCL > molGeom get Angle GLYang "glyAngles"
BTCL > object GLYang
   115.190873 116.304130 114.314820 115.217229
BTCL > molGeom set Angle 51.6   "*:GLY_*:C" "*:GLY_*:CA" "*:GLY_*:N"
BTCL > molGeom get Angle GLYang "*:GLY_*:C" "*:GLY_*:CA" "*:nitros"
BTCL > object GLYang
   51.6 51.6 51.6 51.6
```

This example shows the use of the molGeom get and molGeom set commands for angle attributes. The first two commands are subset definitions which create subsets of all the GLY monomer nitrogen atoms in the monomer context and one subset of angles between the GLY monomer carbon, alpha carbon, and nitrogen atoms in the system context for the current system. The first molGeom get command is then performed using the glyAngles subset to retrieve all the values for the specified angles to the object GLYang. This object is then printed using the object get command to show that there are four such angles, which all have a value of approximately 115 degrees. The next command is a molGeom set command, which is used to set the same angles all to a value of 51.6 degrees. This time, three arguments are given to specify the angles. Note that they are the same atom specifications as used to define the glyAngles subset. The final molGeom command is used to get back the angle values.
just set, to confirm that they are as they should be. Note that this
time, there are again three arguments given to specify the angles
referenced, but that the last argument is the nitros subsets specifi-
cation. In this example, the arguments to each of the three `mol-
Geom` commands specifying the angles set are entirely equivalent.

Example 2

```
BTCL > molGeom get coords Ncrds "*:nitros"

 BTCL > object Ncrds
  {30.898131 -10.527911 0.140841} {32.294803 -4.643929 0.279658} {31.138713 -
  2.255716 0.373504} {32.405810 10.807364 0.140980}

BTCL > vector Ncrds add $Ncrds "(0.1 0.1 0.1)"

BTCL > molGeom set coords $Ncrds "*:nitros"
```

This example continues from the previous one. The first `molGeom
get` command is used to get the coordinates of all the GLY mono-
mer nitrogen atoms (as specified by the nitros subsets) to the object
Ncrds, which is then displayed using the `object` command. A `vec-
tor` command is then used to add a small displacement vector of
[0.1 0.1 0.1] to these coordinates. The new vector is then used to
reset the specified nitrogen atom coordinates using a `molGeom
set` command.

Example 3

```
BTCL > molGeom set distance relative 0.2 "*:GLY_*:CA" "*:GLY_*:N"

```

This example of the `molGeom set` command is used to add 0.2 to
all the nitrogen–alpha carbon bond lengths in all the GLY mono-
mers.

Example 4

```
BTCL > molGeom get rms RMS "*:GLY_3:Atom;*" "*:GLY_6:Atom;*"

BTCL > object RMS
  0.076795

```

This example shows how the `molGeom` command is used to per-
form a root-mean-square comparison of two sets of atom coordi-
nates; here, two GLY monomers. The value is then printed using the `object` command and is seen to be quite low, as would be expected for two of the same monomers. The value is not zero, since the overall structure of the model affects individual monomer geometries. Note that the “Atom;” parts of the atom specification arguments are only to ensure that no subsets could be included if there were any defined at the atom level of the system database hierarchy.
A. Btcl Language and Commands—Standalone Mode - Continued
Command descriptions - Continued

**object**

**Purpose**

The `object` command is used to create and manipulate Btcl 'object' types of data.

**Syntax**

```
```

(For most command operations, you can click the operation keyword to go directly to additional descriptive information on that keyword.)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Arguments</th>
<th>Options</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-a</td>
<td>-</td>
<td>-</td>
<td>With no arguments following <code>varName</code>, the contents of the object <code>varName</code> are returned as a Tcl list.</td>
</tr>
<tr>
<td><code>append1</code></td>
<td><code>spec1 ?spec2 ...?</code></td>
<td><code>-bygroup -byelement elementVec -copies copyVec</code></td>
<td>Append a list of objects, <code>spec1 ...</code>, to the object <code>varName</code>. The appended objects must all be compatible with the <code>-objects object varName</code> type. (See <code>object</code> Command Options.)</td>
</tr>
</tbody>
</table>
### A. Btcl Language and Commands—Standalone Mode - Continued

<table>
<thead>
<tr>
<th>Operation</th>
<th>Arguments</th>
<th>Options</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>create</strong></td>
<td>(type</td>
<td>$obj)</td>
<td>-database database# -elements nElements -matrix (nRows</td>
</tr>
<tr>
<td><strong>cast</strong></td>
<td>type ?table? -database dbHandle -matrix (nRows</td>
<td>symmetric) nColumns</td>
<td>Recast the object varName as a new type type (see Btcl object Types). For changing an object of type integer to type rid, the table argument is required. This operation may also be used to change the tensor type of the items. (See object Command Options.)</td>
</tr>
<tr>
<td><strong>filter</strong></td>
<td>pattern -bygroup</td>
<td>Filter object varName, using a filter specified by the object spec pattern. pattern must specify an object of type boolean or integer. Each item of varName is duplicated by the corresponding item value of pattern number of times (being nonnegative or 0 or 1 for a boolean type object). Note: the items in the returned object varName are not grouped. (See object Command Options.)</td>
<td></td>
</tr>
<tr>
<td><strong>get</strong></td>
<td>$obj ?index? -bygroup -elements elementVec -objects</td>
<td>Create a new object varName from items indexed by object spec index from object $obj. If index is not given, then $obj is copied. Note: all object indices run from 0 to (number of items – 1). The grouping on varName is taken from the index, if it is provided. (See object Command Options.)</td>
<td></td>
</tr>
<tr>
<td><strong>group</strong></td>
<td>?groupName? -unset -set groupSpec -size -close</td>
<td>Return the number of groups in object varName. If supplied, an object groupName (type integer) is created and contains the grouping specification of object varName. (See object Command Options and more information on grouping.)</td>
<td></td>
</tr>
<tr>
<td><strong>info</strong></td>
<td>keyword ?key-word ...?</td>
<td>Return a Tcl-formatted list containing information stored in object varName, as requested by the specified keywords. (See object info Operation Keywords.)</td>
<td></td>
</tr>
<tr>
<td><strong>print</strong></td>
<td>—</td>
<td>—</td>
<td>Return a tabulated string that contains all information stored in object varName.</td>
</tr>
</tbody>
</table>
Create an object `varName` from a range of items in object `$obj`. This range is specified as indices `first` to `last`, with step `step`. `step` must be positive and defaults to 1 if not specified. If `last` is not specified, only one value at index `first` is specified. If `last` is less than `first`, the items in `varName` will be in reverse order relative to those in `$obj`. `first` or `last` may be the keyword `end`, which means the last item of `$obj` (at [number of items - 1]). `first` and `last` may also be out of range, i.e., less than 0 or greater than `end`. In this case the result is the same as if the boundary values were used if the items in `$obj` fall within range, i.e., `object var range mg $obj -4 1000` would equate to `object var range mg $obj 0 end` (if number of items in `$obj` is < 1000), but `object var range mg $obj -2 -1` returns an empty object called `var`. (See `object Command Options`.)

Read an object from a file or interprocess communication (IPC, Appendix C) channel into `varName`. If `msgVar` is not provided, return the message associated with the IPC header of the incoming object. If end-of-file is detected on the input stream before any part of an object (IPC frame) is read, return an empty string and do not set `varName` (if it was previously set, it is unset). If `msgVar` is provided, then the incoming message from the IPC frame is placed in the Tcl variable `msgVar`, and the command returns the number of items in the object just read. If end-of-file is detected, `msgVar` is set to an empty string, and the command returns -1. Also `varName` is unset (if it previously existed).

Compare items in object `spec` with items in object `$obj` and return an object of indices for the matches to `varName` relative to `$obj`. The types of the objects `$obj` and `spec` must match. The default value comparison is equality (==), but other comparison tests may be specified using the command options. (See `object Command Options`).

<table>
<thead>
<tr>
<th>Operation</th>
<th>Arguments</th>
<th>Options</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>range</td>
<td>?group? $obj -bygroup -pergroup &lt;last? ?step? first?</td>
<td>Create an object <code>varName</code> from a range of items in object <code>$obj</code>. This range is specified as indices <code>first</code> to <code>last</code>, with step <code>step</code>. <code>step</code> must be positive and defaults to 1 if not specified. If <code>last</code> is not specified, only one value at index <code>first</code> is specified. If <code>last</code> is less than <code>first</code>, the items in <code>varName</code> will be in reverse order relative to those in <code>$obj</code>. <code>first</code> or <code>last</code> may be the keyword <code>end</code>, which means the last item of <code>$obj</code> (at [number of items - 1]). <code>first</code> and <code>last</code> may also be out of range, i.e., less than 0 or greater than <code>end</code>. In this case the result is the same as if the boundary values were used if the items in <code>$obj</code> fall within range, i.e., <code>object var range mg $obj -4 1000</code> would equate to <code>object var range mg $obj 0 end</code> (if number of items in <code>$obj</code> is &lt; 1000), but <code>object var range mg $obj -2 -1</code> returns an empty object called <code>var</code>. (See <code>object Command Options</code>).</td>
<td></td>
</tr>
<tr>
<td>read</td>
<td>channel? msg- Var?</td>
<td>Read an object from a file or interprocess communication (IPC, Appendix C) channel into <code>varName</code>. If <code>msgVar</code> is not provided, return the message associated with the IPC header of the incoming object. If end-of-file is detected on the input stream before any part of an object (IPC frame) is read, return an empty string and do not set <code>varName</code> (if it was previously set, it is unset). If <code>msgVar</code> is provided, then the incoming message from the IPC frame is placed in the Tcl variable <code>msgVar</code>, and the command returns the number of items in the object just read. If end-of-file is detected, <code>msgVar</code> is set to an empty string, and the command returns -1. Also <code>varName</code> is unset (if it previously existed).</td>
<td></td>
</tr>
<tr>
<td>select</td>
<td>$obj spec -bygroup -nosubdivide -regexp -casesensitive != &lt;= &lt; -anyof -noneof</td>
<td>Compare items in object <code>spec</code> with items in object <code>$obj</code> and return an object of indices for the matches to <code>varName</code> relative to <code>$obj</code>. The types of the objects <code>$obj</code> and <code>spec</code> must match. The default value comparison is equality (==), but other comparison tests may be specified using the command options. (See <code>object Command Options</code>).</td>
<td></td>
</tr>
</tbody>
</table>
**A. Btcl Language and Commands—Standalone Mode - Continued**

---

**Operation** | **Arguments** | **Options** | **Description**
---|---|---|---
**set** | $obj ?index? | **-bygroup** -**elements** elementVec **-objects** | Set the values in object varName at indices specified by object spec index to values taken sequentially from object $obj. Grouping in object varName is not affected. Unlike the **object get** operation, index is required unless the **-elements** option is specified. (See **object Command Options**.)

**sort** | $obj | **-bygroup** **-nosubdivide** **-casesensitive** **-reverse** **-value** | Sort the items in object $obj by ascending value and return an object of indices to varName such that getting the values from $obj using this result would return the items in sorted order. If the **-value** option is used, the actual sorted values, rather than indices, are returned to varName. (See **object Command Options**.)

**write** | channel ?msg-String? | **-noflush** | Write the contents of object varName to a file or interprocess communication (IPC) channel channel. channel must be an initialized IPC channel or Tcl file handle opened for writing. The IPC message string written is the optional argument msgString if it is provided; otherwise the message with the IPC frame defaults to varName. The **-noflush** option may be used to defer flushing the output stream after the object is written. Using **-noflush** can improve performance if a number of objects are to be written in rapid succession.

---

[a] **object** command operation requires the varName be the name of an existing Btcl object variable.

[b] The **object** read/write commands do not support all object types and attributes. In general, only integer types, floating-point types, and strings are supported. Also, the only attributes maintained by an object, if it is written and read, are the number of items and the number of elements. Grouping and other attributes are lost.

**Background — Btcl object datatypes**

Tcl variables are essentially strings of characters which are used by different commands to represent strings, lists, or numbers. However, since these all require translation by the Tcl parser and since Tcl arrays are associative, performing many mathematical operations using Tcl variables is relatively slow. Btcl objects, often referred to simply as objects, allow data to be stored directly in a binary format and to be manipulated in a vectorized manner.

---

C Discover/September 1997
An object can be thought of as an array of data items of a particular type. Simple object types are numeric types such as integers, doubles, Booleans, etc. Special object types include strings, objects, and rids (row identifiers). (See Btcl Object Types table.) In addition to lists of numbers (scalars), the items in a Btcl object may be tensors, e.g., 3 x 1 coordinate vectors, 4 x 4 transformation matrices, 2 x 2 symmetric (compressed) matrices, etc. Btcl objects may also contain grouping information, which specifies that certain ranges of items are banded together.

In an analogous way to how Tcl distinguishes a string as a number, list, etc., Btcl objects are recognized as such only by specific commands. When a Btcl object is created, a coded string is assigned to the object variable, which is used as a reference to the actual object data (stored in binary). Hence, unlike Tcl variables, the contents of a Btcl object may not be accessed directly. For example, given a Btcl object variable called obj:

```
BTCL > echo $obj
object#8

BTCL > echo [set obj]
object#8

BTCL > echo [object obj]
1.23 3.14159
```

The coded string that a Btcl object contains (object#8 in the example above) is often referred to as the object handle. It is important to remember that the content of an object variable is the object handle and not its actual value, for example:

```
BTCL > set a [expr $obj * 3]
Error: syntax error in expression "object#8 * 3"

BTCL > vector a multiply $obj 3

BTCL > echo [object a]
3.69 9.42477
```

Although an object handle is just a string and may be copied to other variables, it is important to note that only the original object variable is attached to the object data. When the object variable is set or unset, the object data are deleted, and the original string
value of this variable is no longer a valid object handle. For example:

```
BTCL > set a $obj
    BTCL > echo $a: [object a]
          object#8: 1.23 3.14159
    BTCL > unset obj
    BTCL > echo $a: [object a]
          Error: object: requires "a" to be the name of an existing object variable
```

**Btcl object types**

<table>
<thead>
<tr>
<th>Type</th>
<th>C-type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>—</td>
<td>Integer values 0 (false) or 1 (true).</td>
</tr>
<tr>
<td>integer</td>
<td>int</td>
<td>Integer values in the range -2147483648 to 2147483647.</td>
</tr>
<tr>
<td>double</td>
<td>double</td>
<td>Real values in the range -xxxxxxxxxx to yyyyyyyyy.</td>
</tr>
<tr>
<td>string</td>
<td>—</td>
<td>String values, e.g., &quot;hello&quot;.</td>
</tr>
<tr>
<td>rid</td>
<td>—</td>
<td>Short for row-id: integers representing objects in a specific database.</td>
</tr>
<tr>
<td>object</td>
<td>—</td>
<td>Pointers to sub-objects. Btcl objects may recursively contain lists of other object structures (see also object-type objects).</td>
</tr>
<tr>
<td>char</td>
<td>char</td>
<td>Integer values in range 0 to 255.</td>
</tr>
<tr>
<td>byte</td>
<td>char</td>
<td>Integer values in range 0 to 255.</td>
</tr>
<tr>
<td>short</td>
<td>short</td>
<td>Integer values in range -32768 to 32767.</td>
</tr>
<tr>
<td>long</td>
<td>long</td>
<td>Integer values. Range is machine dependent.</td>
</tr>
<tr>
<td>ushort</td>
<td>unsigned short</td>
<td>Integer values in range 0 to 65535.</td>
</tr>
<tr>
<td>unsigned</td>
<td>unsigned</td>
<td>Integer values in range 0 to 4294967296.</td>
</tr>
<tr>
<td>float</td>
<td>float</td>
<td>Real values in the range -xxxxxxxxxx to yyyyyyyyy.</td>
</tr>
</tbody>
</table>

^Ranges quoted here are based on 4-byte integers and floats and 8-byte doubles. On some machines, e.g., Cray, these ranges may be larger. On most machines char is synonymous with unsigned char. On many machines long is synonymous with int.
The **object** command is used to examine or manipulate Btcl objects. The object variable used (first argument) must either exist or be the result of an operation on an existing object. The one exception to this statement is the **object create** operation, which creates a new empty object of a specified type.

Unlike the **vector** command, which creates objects of a particular type (Boolean, integer, or double) from values and implicit tensor information in a Tcl list, the **object** command affects only the number, order, or grouping of an object’s contents and never the actual values of the contents.

The basic operation of the **object** command (i.e., with no extra arguments) is analogous to that of the Tcl **set** and Btcl **vector** commands, in that it returns a Tcl list-formatted string of the contents of the existing object variable specified (first argument). This operation is identical to the **object info contents** operation.

The **object print** operation is used to view all the relevant information about an object. It also returns a formatted string to Tcl. This operation takes no further arguments, and the output is tabulated for display. If you want more specific information, the **object info** operation can be used to return any information that can be specified by one or more keyword arguments (see the **object info** Operation Keywords table below). The result string is formatted as a Tcl list with information in the same order as the keyword arguments specified.

The **object create** command is one of the most basic object operations. It is used to create an empty object of a specified type. The argument may be a recognized type keyword, as listed in the object types table above, or an object handle, e.g., $obj. If the latter is true, the result object has the same type and item tensor information as $obj but contains no items or grouping information. If object $obj is of type rid, then database and table information is also cloned. If the argument is a type keyword then the resulting object, *varName*, has scalar items, i.e., only one element per item. The defaults for database, number of elements, and item tensor information can be superceded using the **create** operation options. (See **object** Command Options and information on addressing element items.)
The `object append` command is used to append items to an existing object. This command may take any number of arguments that are object specifications. A valid object specification is either an object handle to an existing Btcl object, e.g., `$obj`, or a Tcl list that defines values that match the type required for the object operation. If the type is integer, Boolean, or real, then valid vector specification strings are accepted. If the type is rid, then a list of scalar integers is accepted. If the type is string, any Tcl list is accepted, such that each list element becomes a new string item, for example:

```plaintext
BTCL > object obj create string
BTCL > object obj append "{first 1st} {second 2nd} {third 3rd}" last
BTCL > echo [object obj print]
  type: string
  items: 4
  elements: 1
  groups (2):
    "first 1st"
    "second 2nd"
    "third 3rd"
    "last"
```

The `append` operation may take the `-copies` option followed by an integer object specification containing nonnegative scalar values. (See `object` Command Options.) The `-copies` arguments must have as many items as there are arguments to the `append` operation. For example:

```plaintext
BTCL > object o create string
BTCL > object o append -copies {1 1 0 2} xxx $obj yyy zzz
BTCL > echo [object o print]
  type: string
  items: 7
  elements: 1
  groups (5):
    "xxx"
    "first 1st"
```
Notice in these examples that the append operation arguments become new groups in the resulting object. The results of most object commands have grouping information, and the append operation can address item elements (see below).

The object get command is used to take items from an existing object to create a new object variable. The first argument must be an object handle to an existing object, e.g., $obj. The second (optional) argument is an object specification specifying a list of nonnegative scalar indices, running from 0. This argument specifies which items are to be taken from object obj to create the result and the grouping that the result will have. If this argument is not provided, then the get results in an exact copy of object obj. The get operation can address item elements.

The object set command is used to replace items in an existing object. The first argument is an object specification, and the second is an index object specification. The latter argument specifies which items in object varName are to be set to items taken sequentially from the first argument. For example:

```
BTCL > object obj set {MIDDLE END} {1 3}
BTCL > echo [object obj print]
    type:    string
    items:   4
    elements: 1
    groups (2):
      "first 1st"
      "MIDDLE"
      "third 3rd"
      "END"
```

The object set command handles grouping information and can address item elements.

The object select command compares items in the first argument, which is an object specification, with items in the second argument, which must be an object handle, e.g., $obj. The result, var-
Name, is an object containing indices corresponding to items in object obj, which are equal in value to items in the first argument. The comparison test used can be changed from the default equality by using one of the options: !\text{=} \text{, } \text{>=} \text{, -anyof}, \text{ etc. (see object Command Options). By default, the comparison test is performed for each item of the first argument to all items in the second argument, and the resulting object is grouped accordingly. For example:}

\text{BTCL > object ind select } \text{!= (MIDDLE END) $obj}
\text{BTCL > object obj2 get $obj $ind}
\text{BTCL > echo [object obj2 print]}
\text{type: string}
\text{items: 6}
\text{elements: 1}
\text{groups (2):}
\text{"first 1st"}
\text{"third 3rd"}
\text{"END"}
\text{"first 1st"}
\text{"MIDDLE"}
\text{"third 3rd"}

Notice in this example that the grouping on object obj2 is transferred from the index argument, $\text{ind}$, which was the result of the \text{select} operation. The \text{select} operation may be performed for all object types except ‘object’. For \text{select} operations on string type objects, as in this last example, -\text{casesensitive} or -\text{regexp} options may be specified to allow the comparisons to be made in a case-sensitive manner or to use regular-expression strings (in the first argument object specification).

The \text{object sort} command is used to produce a sorted list of the items in the first argument, which must be an object handle, e.g., $\text{obj}$. The default result, \text{varName}, is an object containing indices corresponding to the values in object obj taken in ascending-value order. The -\text{value} option may be used to return the sorted values back to the object \text{varName}, rather than the indices, and the -\text{reverse} option may be used to change the sorting operation to produce a result in descending-value order. The -\text{casesensitive} option may be used for string type objects to make the sorting procedure case-sensitive. (See object Command Options.) The \text{object sort} command can handle grouping information.
The **object range** command is used to extract a range of items from an object specified by the first argument, $obj. If only one extra argument is provided, then this is an index specifying one item to extract from object obj, to produce the result object $varName. If two arguments are provided, the second is an index, which, together with the first index, specifies a range of items to extract. The third argument, if provided, is a positive integer specifying a step value for the range. For example:

```
BTCL > object obj3 range $obj2 1 end 2
```

```
BTCL > echo [object obj3 info groups]
{(third 3rd)} {(first 1st) (third 3rd)}
```

Notice in this last example that the keyword `end` has been used to specify the last item index. The `range` operation also allows the range index values to be out of range. If the first value is negative, this is equivalent to specifying 0. If the first value is greater than the effective value of `end`, then this is equivalent to specifying `end`. If the first index value is larger than the second, then the resulting object will have the items in reverse order. (See **object** Command Operations table above.) Also note in the last example, the grouping in the result (denoted by parentheses). This is determined by the grouping of the original items that the range spans.

The **object filter** command is used to alter the number of copies of each item in an object, $varName. The argument, `pattern`, is an integer type object specification or a handle to a object of type boolean. The values of the items in `pattern` must be nonnegative and specify how many copies to create of the corresponding item in $varName. Hence, there must be as many items in `pattern` as there are in object $varName. The result from a `filter` operation does not have grouping information. For example:

```
BTCL > object obj filter {1 2 0 1}
```
However, you can perform filtering on a by-group basis.

The object cast command is used to recast the type of an object varName. The argument may be a recognized type keyword, as listed in the object types table above, or an object handle, e.g., $obj. Casting may be performed only on numeric object types, although rid type objects may be cast to type integer and vice versa. In this case, the integer type object is considered to contain row numbers, i.e., indices to a particular database table. For conversions from type rid to type integer, all items in the object varName must belong to a single database table and must still exist. For conversions from type integer to type rid, an extra argument is required, which must be the name of an existing table in the default System database. The default database may be superceded by using the -database option. The object cast command may also be used to alter the tensor type of the items in object varName, using the -matrix option, but only if the total elements per item stays the same, e.g., a object whose items are 3 x 2 matrices may be converted to an object whose items are 2 x 3 or 6 x 1 matrices. (See object Command Options and information on elements.)

The object read and object write commands are used to read and write an object varName to a file or interprocess communication (IPC) channel (page 347). The first argument, channel, must be an initialized IPC channel or Tcl file handle opened for reading or writing, respectively. The second argument, msgVar for read, msgString for write, is optional. For read, this is a variable name into which the IPC header message string is returned. For write, this is an IPC header message string to write with the data to the output stream. write also accepts a -noflush option, which specifies that the output stream should not be flushed after the object has been written. Delaying flushes can improve performance if several objects are to be written in succession. The system I/O buffer can
be flushed later by an object write command without the -noflush option or with the standard Tcl flush command.

**object info operation key-words:**

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>Request type information for the specified object.</td>
</tr>
<tr>
<td>nItems</td>
<td>Request the number of items for the specified object.</td>
</tr>
<tr>
<td>nGroups</td>
<td>Request the number of groups for the specified object.</td>
</tr>
<tr>
<td>nElements</td>
<td>Request the number of elements per each item for the specified object.</td>
</tr>
<tr>
<td>rank</td>
<td>Request the rank of the specified object's items: 0 means scalar, 1 means vector, and 2 means matrix.</td>
</tr>
<tr>
<td>columns</td>
<td>Request the number of columns in the specified object's items: 1 means all objects whose items are not matrices.</td>
</tr>
<tr>
<td>rows</td>
<td>Request the number of rows in the specified object's items: equals nElements for vector type objects.</td>
</tr>
<tr>
<td>symmetry</td>
<td>Request the symmetry flag of the specified object: use 1 if symmetric (compacted) matrix, 0 otherwise.</td>
</tr>
<tr>
<td>database</td>
<td>Request the database number for the specified object: equals 0 for non-rid type objects.</td>
</tr>
<tr>
<td>table</td>
<td>Request the table name(s) that the items in the specified object originated from: equals &quot;&lt;NULL &gt;&quot; for non-rid type objects.</td>
</tr>
<tr>
<td>contents</td>
<td>Request a list of items contained within the specified object.</td>
</tr>
<tr>
<td>groups</td>
<td>Request a list of groups contained within the specified object.</td>
</tr>
</tbody>
</table>

**object command options:**

<table>
<thead>
<tr>
<th>Options</th>
<th>Arguments</th>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-set</td>
<td>groupVec</td>
<td>group</td>
<td>Specify that the group operation is to set the grouping on an object to groupVec.</td>
</tr>
<tr>
<td>-unset</td>
<td>—</td>
<td>group</td>
<td>Specify that the group operation is to remove the grouping information from an object.</td>
</tr>
<tr>
<td>-close</td>
<td>—</td>
<td>group-set</td>
<td>Allow groupVec to not group all the items in an object. In this case the remaining items form the last group.</td>
</tr>
<tr>
<td>-size</td>
<td>—</td>
<td>group</td>
<td>Specify that the group operation is to refer to the grouping of an object by group sizes rather than by index bounds.</td>
</tr>
</tbody>
</table>
A. Tcl Language and Commands—Standalone Mode - Continued

<table>
<thead>
<tr>
<th>Options</th>
<th>Arguments</th>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-bygroup</td>
<td>—</td>
<td>append</td>
<td>For <code>select</code> and <code>sort</code> operation is performed for each group individually rather than on all an object's items collectively. For <code>filter</code>, <code>get</code>, <code>range</code>, <code>select</code>, and <code>set</code> operations, addressing arguments refer to an object's groups rather than items.</td>
</tr>
<tr>
<td>-nosubdivide</td>
<td>—</td>
<td><code>select</code></td>
<td>For <code>select</code>, the resulting groups are not subdivided by the number of items selected for the first argument. For <code>sort</code>, the resulting groups are not subdivided for equivalent values.</td>
</tr>
<tr>
<td>-pergroup</td>
<td>—</td>
<td><code>range</code></td>
<td>The range arguments refer to items within each group, rather than to all the items collectively.</td>
</tr>
<tr>
<td>-elements</td>
<td><code>elementIndex</code></td>
<td><code>create</code></td>
<td>For <code>create</code> operations, one value specifying the number of elements per item in the object created. For <code>get</code> and <code>set</code> operations, <code>elementIndex</code> is a list of element indices within the items of an object which are to be addressed.</td>
</tr>
<tr>
<td>-matrix</td>
<td>`(symmetric</td>
<td>nRows)</td>
<td><code>create</code></td>
</tr>
<tr>
<td>-byelement</td>
<td><code>elementIndex</code></td>
<td><code>append</code></td>
<td>Specify the mapping of the argument item elements to the resulting object's item elements. If <code>elementIndex</code> is non-empty, then each element index of the existing object <code>varName</code> must be specified once only. If <code>elementIndex</code> is empty, then elements are addressed in sequence, i.e., &quot;0 1 2 ...&quot;.</td>
</tr>
<tr>
<td>-objects</td>
<td>—</td>
<td><code>append</code></td>
<td>Specify that the arguments to the object operation on an object of type <code>object</code> are to be considered single items rather than objects containing items.</td>
</tr>
<tr>
<td>-copies</td>
<td><code>copyVec</code></td>
<td><code>append</code></td>
<td>Integer object <code>copyVec</code> specifies the numbers of copies of each argument object used for the <code>object append</code> command. <code>copyVec</code> may contain only one value if the <code>-byelement</code> option is also specified.</td>
</tr>
<tr>
<td>-database</td>
<td><code>dbHandle</code></td>
<td><code>create</code></td>
<td>Specify the database from a database handle <code>dbHandle</code> to override the default <code>System</code> database.</td>
</tr>
<tr>
<td>-casesensitive</td>
<td>—</td>
<td><code>select</code></td>
<td>Specify that string comparisons are to be performed in a case-sensitive manner. The default operation is case-insensitive.</td>
</tr>
</tbody>
</table>
Btcl objects may have grouping information indicating that two or more items in an object have something in common. Most object commands set some type of grouping on the resulting object and have options that change their operation with respect to grouping. Grouping is a very powerful feature of Btcl objects which allows for complex object manipulation.

The object group command is used to get, set, or unset the grouping information on an existing object. The basic operation of this command is to return the number of groups in an object as a Tcl string. This string is always an integer value of at least 1. A value of 1 indicates that all the items are essentially a single group, and often such an object is referred to as having 'no' grouping. An object may have more or fewer groups than the number of its items, since it is possible to have groups that do not contain any items. If an argument is provided, then this is used as a variable name to which an integer type object is returned and which contains the grouping information. For example:

<table>
<thead>
<tr>
<th>Options</th>
<th>Arguments</th>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-regexp</td>
<td>—</td>
<td>select</td>
<td>Specify that the first object's items are regular expression strings and that string matching should be performed accordingly.</td>
</tr>
<tr>
<td>!= &lt; &lt;= &gt; &gt;=</td>
<td>—</td>
<td>select</td>
<td>These options override the default == comparison test for the select operation.</td>
</tr>
<tr>
<td>-anyof</td>
<td>—</td>
<td>select</td>
<td>Override the default == comparison test for the select operation. The result is an index object corresponding to each item value in the 2nd argument that matches (==) any value in the 1st argument. Group subdivision does not occur regardless of existing grouping or other options.</td>
</tr>
<tr>
<td>-noneof</td>
<td>—</td>
<td>select</td>
<td>Similar to select-anyof except that the value matching uses a not equal to (!=) comparison of the values.</td>
</tr>
<tr>
<td>-value</td>
<td>—</td>
<td>sort</td>
<td>Specify that the sort operation is to return sorted values rather than sorted indices to those values.</td>
</tr>
<tr>
<td>-reverse</td>
<td>—</td>
<td>sort</td>
<td>Specify that the sort operation is to sort items in descending-value order rather than ascending-value order.</td>
</tr>
<tr>
<td>-noflush</td>
<td>—</td>
<td>write</td>
<td>Specify that the write operation should not flush the system I/O buffer.</td>
</tr>
</tbody>
</table>
Notice here that, although there are only two groups, there are three items in the grouping object, grp. The values are indices to the start of each group and to the outside limit of the previous group. Hence, the first value is always 0, and the last is always equal to the number of items in the object.

The -set option of the group operation allows you to set the grouping of an object explicitly. The -unset option allows you to remove any grouping from an object. For example:

```
BTCL > vector v "1 2 3 4 5 6 7 8"
BTCL > echo [object v group -set "0 3 3 5 8"]
   4
BTCL > echo [object v info groups]
   (1.0 2.0 3.0) () (4.0 5.0) (6.0 7.0 8.0)
BTCL > echo [object v group -unset]
   1
BTCL > echo [object v info groups]
   (1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0)
```

Notice in this example that the number of groups returned is that of the resulting grouping rather than that of the original grouping. Also notice that an empty group was created by specifying the same index twice. The optional argument groupName is not allowed when -set or -unset is specified. It is also illegal for the new grouping object specification to have items in decreasing-value order. The first value must be 0, and the last must be the number of items in the object varName. It is possible, however, to have the last index be less than the number of items in varName if the -close option is specified. In this case the operation proceeds as if the number of items in the object were appended to the end of groupName, for example:

```
BTCL > object v group -close -set "0 3 3 5"
```
The `-size` option specifies that the `object group` command is to refer to grouping by group sizes rather than by bounding indices. This applies to both the basic and the `-set` operations. For example:

```
BTCL > object v group -size grp
BTCL > echo [object grp]
3 0 2 3
BTCL > object v group -size -set "3 0 4 1"
BTCL > echo [object v info groups]
(1.0 2.0 3.0) () () (4.0 5.0 6.0 7.0) (8.0)
```

When setting the grouping to be by sizes, the specification object must be composed of nonnegative integers, the values of which must sum to the number of items in the object `varName`. The values may sum to less than this number if the `-close` option is also specified.

All other `object` commands, except for `object cast`, affect the grouping of the resulting object `varName`. Most `object` commands allow the `-bygroup` option (see `object` Command Options), which changes the default operation from working on a by-item basis to working on a by-group basis. For example, the `filter -bygroup` operation uses the `pattern` argument to duplicate whole groups:

```
BTCL > object v filter -bygroup "2 0 1 1 0"
BTCL > echo [object v info groups]
(1.0 2.0 3.0) (1.0 2.0 3.0) () (4.0 5.0 6.0 7.0)
```

Notice in this last example that the `filter` operation returns the object with grouping information, since the filtering was performed on a by-group basis. This `-bygroup` behavior applies to all the `object` commands that would usually use an indexing argument to address items, namely `filter`, `range`, `get`, and `set`. In these cases the argument must now have a number of items matching the number of `groups` in an object rather than the number of items. For the `set -bygroup` operation, items are taken sequentially from the first argument object and used to set the corresponding values.
of the items in the groups specified by the index argument, for example:

```b TCL
> vector v2 "3.0 2.0 1.0 4.1 5.1 6.1 7.1 8.1"
> object v set -bygroup $v2 "1 3"
> echo [object v info groups]
  (1.0 2.0 3.0) (3.0 2.0 1.0) () (4.1 5.1 6.1 7.1)
```

The `append -bygroup` operation is used to append new items or groups to the end of existing `groups` in an object rather than appending to the end of that object. For this operation, all the argument objects must contain the same number of either items or groups as object `varName` has groups. For example:

```b TCL
> vector v2 "int 1 2 3 4 5 6 7 8"
> object v2 group -size -set "3 2 3"
> echo [object v2 info groups]
  (1 2 3) (4 5) (6 7 8)
> object v2 append -bygroup $v2
> echo [object v2 info groups]
  (1 2 3 1 2 3) (4 5 4 5) (6 7 8 6 7 8)
> object v2 append -bygroup "10 20 30"
> echo [object v2 info groups]
  (1 2 3 1 2 3 10) (4 5 4 5 20) (6 7 8 6 7 8 30)
```

The `select -bygroup` operation selects items from the first argument with respect to groups in the second argument and returns the result with grouping, with empty groups where no matches were found. For example:

```b TCL
> object ind select -bygroup 2.0 $v
> echo [object ind info groups]
  (1) (4) () ()
> object ind select -bygroup "2.0 3.0 2.1 5.1" $v
```
Notice in this example that the first `select` operation, with only one item in the first argument, produces 4 groups in the result, corresponding to matches in the 4 groups of the second argument. The second `select` operation, however, returns 16 groups, with the items matched in the order in which they appear in the first argument. This is because, for each item in the first argument, a `select` is performed against each group of the second argument. When groups are split into additional groups in this manner, the process is known as group subdivision. The `select` operation has the option `-nosubdivide`, to prevent this process from occurring. For example:

```
BTCL > object ind select -bygroup -nosubdivide "2.0 3.0 2.1 5.1" $v
BTCL > object v2 get $v $ind
BTCL > echo [object v2 info groups]
(2.0) (3.0) (2.0) (3.0) (5.1)
```

The `object sort` operation also has the options `-bygroup` and `-nosubdivide`, which work in an entirely analogous way to how they work for the `select` operation. A `sort` operation groups the results so that items having the same values are grouped together. If the `-bygroup` and `-nosubdivide` options are used, then the groups are sorted individually and this grouping is retained in the result. Without the `-nosubdivide` option, the resulting groups are subdivided by items having the same value. For example:

```
BTCL > object v append "10 20 10"
BTCL > object v2 sort -value -bygroup -nosubdivide $v
BTCL > echo [object v2 info groups]
(1.0 2.0 3.0) (1.0 2.0 3.0) (4.1 5.1 6.1 7.1) (10.0 10.0 20.0)
BTCL > object v2 sort -value -bygroup $v
```
Group subdivision can be very useful when doing multiple select and sort operations.

The object range -bygroup operation simply returns ranges of groups rather than ranges of items from the first object argument. The range operation also has a unique option, -pergroup. This allows the range specified to apply to items in each group. For example:

\[
\text{BTCL} \rightarrow \text{echo [object v info groups]}
\]
\[
(1.0) (2.0) (3.0) (1.0) (2.0) (3.0) () (4.1) (5.1) (6.1) (7.1) (10.0)
\]
\[
(10.0)
\]

\[
(20.0)
\]

Notice in this last example that, because of the out-of-range behavior of the range operation, all groups are replaced by their second and fourth items where those items exist. This operation is particularly useful after a sort operation to remove duplicated item values.

As mentioned above, Btcl object items can be composed of more than one element, in which case the object is considered a vector or matrix object, although in fact the object actually contains a list of vector or matrix items. An item that is a 2 x 2 matrix has the same number of elements as a vector containing 4 elements, and object commands that manipulate elements within items do not distinguish between them. Matrix information in an object is an extra subdivision of the elements and is considered only by the vector command.

The object create operation is the only object operation that can create a new object with an explicit element size. If the -elements option is used, followed by an integer object specification containing one positive value, an empty vector object is created whose items have the specified elements. The -matrix option performs a similar function, but the element size is calculated from the matrix.
dimension, and the result is an empty matrix object. (See object Command Options.)

Both the object get and object set commands also have the -elements option. In this case the argument that follows the option is a scalar object of type integer specification containing indices representing elements. The elementIndex object may not be empty and may not contain item values less than 0 nor greater than the number of elements in an object (less one). For example, the y component of a coordinate vector would have the index 1. Matrix items are divided by the row dimension. Consider the example below that extracts the (row-2, column-3) element from a 4(rows) x 3(columns) matrix:

```
BTCL > vector mat "int {{1 2 3} {4 5 6} {7 8 9} {0 0 0}}"

BTCL > echo [object mat print]
    type:           integer
    items:          1
    elements:       12
    contents:       1 2 3 4 5 6 7 8 9 0 0 0
    rank:           2
    columns:        3
    rows:           4

BTCL > $i = [object mat info columns] * (2 - 1) + (3 - 1); echo $i
    5

BTCL > object e get -elements $i $mat

BTCL > echo [object e]
    6
```

A slightly more complicated example is setting the diagonal on a 3 x 3 matrix:

```
BTCL > vector mat "int {{1 0 0} {0 1 0} {0 0 1}}"

BTCL > object mat set -elements "0 4 8" "(2 3 4)" 0

BTCL > echo [object mat]
    {{2 0 0} {0 3 0} {0 0 4}}
```

Notice in this last example that the first argument ("[2 3 4]") is a vector specification. This is because with the -elements option this
argument must have at least as many elements as items in elementIndex. The last argument (0) is always required for the object set operation, even if object varName contains only one item.

The -byelement option for the object append command is very similar to the -elements option. (See object Command Options.) Unlike with the get and set operations, however, the following object specification, elementIndex, must index all the elements in an item once and once only, but it may also be empty. If elementIndex is empty, it is taken to infer "0 1 2 ... nElements-1". The append -byelement operation is used to append to existing vector or matrix objects using argument objects that may have different numbers of elements per item. For example, this operation may be used to join x and y scalar objects, or an xy vector object, to a z scalar object to form one xyz vector object:

```
BTCL > vector x "1.1 2.1 3.1"
BTCL > vector y "1.2 2.2 3.2"
BTCL > vector z "1.3 2.3 3.3"
BTCL > object xyz create -elements 3 double
BTCL > object xyz append -byelement {} $x $y $z
BTCL > echo [object xyz]
   {1.1 1.2 1.3} {2.1 2.2 2.3} {3.1 3.2 3.3}
```

Unlike most object append operations, where one object append command with two arguments may be equivalent to two commands each with one argument, the arguments following a -byelement option are taken to specify one collective argument. Consequently, if the -copies option is used, the copyVec argument may contain only one item. An example where a non-empty ele-
mentIndex object is used would be appending a y scalar object to a xz vector object to create a xyz vector object:

```btcl
BTCL > object xyz create -elements 3 double
BTCL > object xz create -elements 2 double
BTCL > object xz append -byelement {} $x $z
```
```
BTCL > echo [object xz]
   {1.1 1.3} {2.1 2.3} {3.1 3.3}
```
```
BTCL > object xyz append -byelement {1 0 2} $y $xz
```
```
BTCL > echo [object xyz]
   {1.1 1.2 1.3} {2.1 2.2 2.3} {3.1 3.2 3.3}
```

Finally, the object cast command, as mentioned above, may be used to recast a vector or matrix object as a matrix object, provided that the number of elements per item does not change. One use of this is to consider data read in as a vector object to be a symmetric matrix object (symmetric meaning to Btcl compacted symmetric), for example:

```btcl
BTCL > vector sym "int {1 1 0 1 0 0}"
```
```
BTCL > object sym cast -matrix symmetric 3 integer
```
```
BTCL > echo [object sym]
   {{1} {1 0} {1 0 0}}
```

There is no cast option that allows a matrix object to be converted into a vector object (mainly due to syntax problems). However, this can be easily achieved using an object append -bygroup command, for example:

```btcl
BTCL > object unsym create -elements 6 integer
```
```
BTCL > object unsym append -byelement "$sym
```
```
BTCL > echo [object unsym]
   {1 1 0 1 0 0}
```

Certain object commands allow for the construction or destruction of objects of type ‘object’, i.e., a Btcl object whose items are themselves Btcl objects. These object types allow you to have Btcl
A. Btcl Language and Commands—Standalone Mode - Continued

objects that contain mixed data types (similar to having dynamic structures). Since the item in an object of type object may also be an object of type object, it is possible to construct complex trees of object data. However, this is not advised, since a Btcl object variable could only be linked to the top of one of these object trees, and each time the variable is manipulated, all the underlying data are copied (as opposed to manipulating pointers to the data). In spite of this limitation (which affects performance), objects of type object can be very useful, particularly when constructing or using complex databases.

With the notable exception of the select and sort operations, all object commands treat object-type objects like any other Btcl object types, for example, the append operation is used to append the items in an object of type object to an existing object of this type. However, this is different from appending an object as an item to an existing object of type object.

The -objects option is supported by the object command operations append, get, and set to allow objects to be appended, extracted, or placed in objects of type object. For example:

```
BTCL > vector v1 "{1 2 3} {4 5 6}"
BTCL > vector v2 "int 10 11 12"
BTCL > object oa create object
BTCL > object oa append -copies "1 2" -objects $v1 $v2
```

```
BTCL > echo [object oa]
{double {{1.0 2.0 3.0} {4.0 5.0 6.0}}} {integer {10 11 12}} {integer {10 11 12}}
```

```
BTCL > echo [object oa print]
  type:   object
  items:  3
  elements: 1
  groups (3):
    {  
      type:   double
      items:  2
      elements: 3
      contents:
        1.0 2.0 3.0
        4.0 5.0 6.0
  ```
Notice in this example that echoed outputs are necessarily different for objects of type object, and that each object appended to object oa is appended as a new group. The arguments to an `append -objects` operation must be object handles rather than general object specifications. This is because the arguments may be of any object type, and so the command would not know how to determine the type of the specification strings. However, the contents of the argument objects are copied so that, for example, if object v1 or v2 were subsequently unset, object oa would not be affected. These principles are also true for the `object get` and `object set` operations with the `-objects` option, as shown in the following example:

```
BTCL > object oa set -objects $v1 2

BTCL > echo [object oa]
  {double {{1.0 2.0 3.0} {4.0 5.0 6.0}}} {integer {10 11 12}} {double {{1.0 2.0 3.0} {4.0 5.0 6.0}}}

BTCL > object v3 get -objects $oa 1

BTCL > echo [object v3]
  fs10 11 12
```
A. Btcl Language and Commands—Standalone Mode - Continued

In this last example, since the item to be set or got is a single object, the last argument (the index object) must contain exactly one value.

output

Purpose

The output command is used to control the amount of printing to the Discover output file.

Syntax

output ?keyword = value?...

<table>
<thead>
<tr>
<th>Keywords</th>
<th>Value</th>
<th>Default</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>buffering</td>
<td>by_command</td>
<td>by_command</td>
<td>Control the frequency with which output is flushed to the output file(s).</td>
</tr>
<tr>
<td></td>
<td>by_line</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>buffered</td>
<td></td>
<td></td>
</tr>
<tr>
<td>level</td>
<td>normal</td>
<td>normal</td>
<td>Control the amount of output produced by Btcl commands.</td>
</tr>
<tr>
<td></td>
<td>terse</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>verbose</td>
<td></td>
<td></td>
</tr>
<tr>
<td>warning_level</td>
<td>normal</td>
<td>normal</td>
<td>Control the number of warning messages that are output by Btcl every</td>
</tr>
<tr>
<td></td>
<td>every</td>
<td></td>
<td>every commands.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Description

The output command controls the frequency and amount of output and the number of warning messages that are produced by Btcl commands.

The buffering keyword can be set to buffered, which is the usual UNIX block buffering (typically 8 KB). by_command causes the output to be flushed at the end of execution of each Btcl command. This is the default method and is relatively efficient in terms of number of disk accesses, while keeping the output file relatively current. by_line causes output to be flushed after every line.

The output level can be set to none, which means that no command (except for print commands) produces any output; terse,
which results in very brief output (e.g., one line for the `minimize` command; `normal`, which is the usual (default) amount of output; or `verbose`, which is typically used for debugging.

The `warning_level` can be set to `none`, which suppresses the output of any warnings.

`normal` and `every` control output from commands that produce multiple similar warnings (e.g., iterated reports that nonbond groups are too large for the current cutoff). `normal` causes the first message to be output, together with a summary message at the end indicating the total number of warnings encountered. `every` causes all warning messages to be output.

Note that the `output` command differs from other Btcl commands in that the defaults for the output parameters are automatically set when the command is performed. For example:

```
cdlConfig output warning_level = none
```

`output` is identical in functionality and operation to:

```
output warning_level = none
```

**Example**

```
output warning_level = every
```

This example of the `output` command causes every Btcl command that prints warnings to display all such warnings in the output file.

---

**peek**

**Purpose**

The `peek` command is used to control the monitoring of iterative processes in minimization and dynamics calculations. The `peek` command causes .pek and .pcar files to be written at intervals or stages during the calculation so that you can follow their progress.

**Syntax**

```
peek ?keyword = value? ...
```
The **peek** command is useful for monitoring the progress of long Discover calculations, either by CPU time intervals or by number of iteration steps. It is used in a similar way to the **print** command with **minimize** and **dynamics execute** commands and is specified by the **peek** parameter of these commands. The default action for the **peek** command is to produce a file, `run_name.pek`, every minute of CPU time used during an iterative Btcl command. This file contains information about the stage of the calculation and important system statistics, such as the energy contributions and convergence (for minimizations). If the **carfile** parameter is set to **true**, a `run_name.pcar` file is also written, which contains the coordinates of the system so that you can visualize the system as it changes during the run. By using the **iteration_step** parameter you can follow the calculation on a step-by-step basis, which is particularly useful if you want to observe some particular change of state that may occur at some point before the end of the calculation. With iteration-based peeks, the .pek and .pcar files are always written for the first iteration, so that you can tell that a new minimization or dynamics calculation has started.

The **peek** command may also be controlled from outside the Discover program by means of .ipc files. An .ipc file, `run_name.ipc`, contains a simple one-line command, such as **stop**, **detach**, **attach**, **pause**, or **resume**, which is used by the Insight program to control background Discover jobs.

### Description

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Values</th>
<th>Default</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>cpu_time_step</td>
<td>real</td>
<td>60.0</td>
<td>The time, in seconds, between successive writes of the .pek and .pcar files. A value of 0.0 indicates that the peeking is not dependent on CPU time intervals.</td>
</tr>
<tr>
<td>iteration_step</td>
<td>integer</td>
<td>0</td>
<td>The number of iteration steps between successive writes of .pek and .pcar files. A value of 0 indicates that the peeking is not dependent on iteration steps.</td>
</tr>
<tr>
<td>pek_file</td>
<td>Boolean</td>
<td>true</td>
<td>Specify whether a .pek file is to be written.</td>
</tr>
<tr>
<td>car_file</td>
<td>Boolean</td>
<td>false</td>
<td>Specify whether a .pcar file is to be written.</td>
</tr>
<tr>
<td>file_name</td>
<td>string</td>
<td>&quot;&quot;</td>
<td>Specify the .pek and .pcar filename prefix. A value of &quot;&quot; means to use run_name.pek and run_name.pcar.</td>
</tr>
</tbody>
</table>
Example 1

```
BTCL > minimize peek = "peek cpu = 0.0 iteration = 10 -pek +car"
```

This example of the `peek` command shows its use with a standard minimization run (all parameters at their default values). The `peek` parameter of the `minimize` command specifies that a `peek` command should be executed, which causes the system coordinates file `run_name.pcar` to be written out every 10 iteration steps during the minimization.

Example 2

```
BTCL > dynamics peek = ""
```

In this example, a standard dynamics simulation is run with the `peek` parameter set to "". This means that no `peek` command is called, and the `peek` mechanism is disabled. This is useful for long simulations on large systems, since the polling and execution of `peek` commands slows some calculations significantly, especially if the `peek` command is set to generate .pek and .pcar files frequently.

Example 3

```
BTCL > cdlConfig peek cpu = 20.0 file = lookin

BTCL > minimize

BTCL > dynamics
```

In this example a `cdlConfig` command is used to specify the default values for the `peek` command to generate lookin.pek files every 20 seconds of CPU time. Since the `peek` parameters for subsequent `minimize` and `dynamics` commands are not specified, this is the default `peek` action for these commands, i.e., both commands write to lookin.pek every 20 CPU seconds during their execution.
A. Btcl Language and Commands—Standalone Mode - Continued

print

Purpose

The `print` command is used to write out information for various system properties, for example, the energy, structure, and thermodynamic state.

The information chosen can be printed to an archive, output, table, or history file. When it appears as part of the `minimize` or `dynamics` command (in an execute structure), the requested information is printed as the minimization or dynamics run proceeds.

Currently, the `print` command must be part of a `minimize` or `dynamics` command—a `print` command issued directly from the BTCL prompt or in a source file does not do anything.

Syntax

```
print filetype ?path? ?keyword = value?...
```

<table>
<thead>
<tr>
<th>Path/keyword</th>
<th>Values</th>
<th>Default</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>controlling where to print:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>filetype</code></td>
<td><code>output</code></td>
<td>output</td>
<td>Specify type of file to receive the requested information. One of these values must be the first argument after <code>print</code>, since the <code>filetype</code> is specified by a positional argument. Subsequent parameters have defaults that may depend on the <code>filetype</code> specified.</td>
</tr>
<tr>
<td><code>file*name</code></td>
<td><code>string</code></td>
<td><code>run_name.ext</code></td>
<td>User-defined name of archive, output, table, or history file. The default is the current run name with the appropriate extension—<code>.arc</code> for archive, <code>.out</code> for output, <code>.tbl</code> for table, or <code>.his</code> for history files, respectively.</td>
</tr>
<tr>
<td>controlling type of printing:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>average</code></td>
<td>Boolean</td>
<td>false</td>
<td>Print running average of the property.</td>
</tr>
<tr>
<td><code>running_sd</code></td>
<td>Boolean</td>
<td>false</td>
<td>Print running standard deviation of the property (available only for table file).</td>
</tr>
<tr>
<td><code>batch_average</code></td>
<td>Boolean</td>
<td>false</td>
<td>Print batch average of the property.</td>
</tr>
<tr>
<td><code>batch_size</code></td>
<td>Integer</td>
<td>10</td>
<td>Size of the window used in batch averaging.</td>
</tr>
<tr>
<td><code>batch_sd</code></td>
<td>Boolean</td>
<td>false</td>
<td>Print batch standard deviation of the property (available only for table file).</td>
</tr>
<tr>
<td><code>instantaneous</code></td>
<td>Boolean</td>
<td>false</td>
<td>Print instantaneous value of the property.</td>
</tr>
</tbody>
</table>
### Controlling what properties to print:

<table>
<thead>
<tr>
<th>Path/keyword</th>
<th>Values</th>
<th>Default</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>archive</td>
<td>—</td>
<td>—</td>
<td>Pathname to parameters that may be printed only to archive files.</td>
</tr>
<tr>
<td>coordinates</td>
<td>Boolean</td>
<td>false</td>
<td>Cartesian coordinates representing the structure (default is true if filetype = archive).</td>
</tr>
<tr>
<td>derivative</td>
<td>Boolean</td>
<td>false</td>
<td>First derivatives of total potential energy with respect to Cartesian coordinates (only to archive file).</td>
</tr>
<tr>
<td>velocity</td>
<td>Boolean</td>
<td>false</td>
<td>Velocities of the atoms (only to archive file and in dynamics only).</td>
</tr>
<tr>
<td>energy_summary</td>
<td>Boolean</td>
<td>false</td>
<td>Equivalent to specifying total_energy, kinetic_energy, and potential_energy.</td>
</tr>
<tr>
<td>total_energy</td>
<td>Boolean</td>
<td>false</td>
<td>Sum of kinetic and potential energy.</td>
</tr>
<tr>
<td>kinetic_energy</td>
<td>Boolean</td>
<td>false</td>
<td>Kinetic energy, meaningful only in dynamics.</td>
</tr>
<tr>
<td>potential_energy</td>
<td>Boolean</td>
<td>false</td>
<td>Potential energy.</td>
</tr>
<tr>
<td>internal_energy</td>
<td>Boolean</td>
<td>false</td>
<td>Equivalent to specifying total_internal_energy, bond_energy, angle_energy, torsion_energy, oop_energy, and cross_term_energy.</td>
</tr>
<tr>
<td>total_internal_energy</td>
<td>Boolean</td>
<td>false</td>
<td>Total internal energy, including both diagonal (bond, angle, torsion, oop) and non-diagonal (cross) terms.</td>
</tr>
<tr>
<td>bond_energy</td>
<td>Boolean</td>
<td>false</td>
<td>Energy from all bond terms.</td>
</tr>
<tr>
<td>angle_energy</td>
<td>Boolean</td>
<td>false</td>
<td>Energy from all bond angle terms.</td>
</tr>
<tr>
<td>torsion_energy</td>
<td>Boolean</td>
<td>false</td>
<td>Energy from all dihedral angle terms.</td>
</tr>
<tr>
<td>oop_energy</td>
<td>Boolean</td>
<td>false</td>
<td>Energy from all out-of-plane angle terms.</td>
</tr>
<tr>
<td>cross_term_energy</td>
<td>Boolean</td>
<td>false</td>
<td>Energy from all cross terms.</td>
</tr>
<tr>
<td>nonbond_energy</td>
<td>Boolean</td>
<td>false</td>
<td>Equivalent to specifying total_nonbond_energy, vdw_energy, dispersive_vdw_energy, repulsive_vdw_energy, coulomb_energy, and hbond_energy.</td>
</tr>
<tr>
<td>total_nonbond_energy</td>
<td>Boolean</td>
<td>false</td>
<td>Energy of all interactions through space (within the cutoff distance, if any).</td>
</tr>
<tr>
<td>vdw_energy</td>
<td>Boolean</td>
<td>false</td>
<td>Energy of all van der Waals interactions (within the cutoff distance, if any).</td>
</tr>
<tr>
<td>dispersive_vdw_energy</td>
<td>Boolean</td>
<td>false</td>
<td>Dispersive van der Waals energy.</td>
</tr>
<tr>
<td>repulsive_vdw_energy</td>
<td>Boolean</td>
<td>false</td>
<td>Repulsive van der Waals energy.</td>
</tr>
<tr>
<td>coulomb_energy</td>
<td>Boolean</td>
<td>false</td>
<td>Electrostatic energy (within the cutoff distance, if any).</td>
</tr>
</tbody>
</table>
A. Btcl Language and Commands—Standalone Mode - Continued

<table>
<thead>
<tr>
<th>Path/keyword</th>
<th>Values</th>
<th>Default</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>cell</td>
<td>Boolean</td>
<td>false</td>
<td>Equivalent to specifying cell_a, cell_alpha, cell_b, cell_beta, cell_c, and cell_gamma. All these are meaningful only with periodic boundary conditions and only for dynamics.</td>
</tr>
<tr>
<td>cell_a</td>
<td>Boolean</td>
<td>false</td>
<td>Length of side a of unit cell.</td>
</tr>
<tr>
<td>cell_b</td>
<td>Boolean</td>
<td>false</td>
<td>Length of side b of unit cell.</td>
</tr>
<tr>
<td>cell_c</td>
<td>Boolean</td>
<td>false</td>
<td>Length of side c of unit cell.</td>
</tr>
<tr>
<td>cell_alpha</td>
<td>Boolean</td>
<td>false</td>
<td>Angle alpha (between b and c) of the cell.</td>
</tr>
<tr>
<td>cell_beta</td>
<td>Boolean</td>
<td>false</td>
<td>Angle beta (between a and c) of the cell.</td>
</tr>
<tr>
<td>cell_gamma</td>
<td>Boolean</td>
<td>false</td>
<td>Angle gamma (between a and b) of the cell.</td>
</tr>
<tr>
<td>state</td>
<td>Boolean</td>
<td>false</td>
<td>Equivalent to specifying temperature, pressure, volume, and density. Temperature is meaningful only in dynamics; pressure, volume, and density are meaningful only in periodic systems and only in dynamics.</td>
</tr>
<tr>
<td>temperature</td>
<td>Boolean</td>
<td>false</td>
<td>Temperature.</td>
</tr>
<tr>
<td>pressure</td>
<td>Boolean</td>
<td>false</td>
<td>Pressure.</td>
</tr>
<tr>
<td>volume</td>
<td>Boolean</td>
<td>false</td>
<td>Volume.</td>
</tr>
<tr>
<td>density</td>
<td>Boolean</td>
<td>false</td>
<td>Density.</td>
</tr>
<tr>
<td>stress</td>
<td>Boolean</td>
<td>false</td>
<td>Equivalent to specifying sxx, syy, szz, syz, sxz, and sxy. Diagonal terms (sxx, syy, szz) of stress tensor are also called tensile stress; off-diagonal terms (syz, sxz, sxy) of stress tensor are also called shear stress. Meaningful only with periodic systems and only in dynamics.</td>
</tr>
<tr>
<td>sxx</td>
<td>Boolean</td>
<td>false</td>
<td>xx component of stress tensor.</td>
</tr>
<tr>
<td>syy</td>
<td>Boolean</td>
<td>false</td>
<td>yy component of stress tensor.</td>
</tr>
<tr>
<td>szz</td>
<td>Boolean</td>
<td>false</td>
<td>zz component of stress tensor.</td>
</tr>
<tr>
<td>syz</td>
<td>Boolean</td>
<td>false</td>
<td>yz component of stress tensor.</td>
</tr>
<tr>
<td>sxz</td>
<td>Boolean</td>
<td>false</td>
<td>xz component of stress tensor.</td>
</tr>
<tr>
<td>sxy</td>
<td>Boolean</td>
<td>false</td>
<td>xy component of stress tensor.</td>
</tr>
<tr>
<td>strain</td>
<td>Boolean</td>
<td>false</td>
<td>Equivalent to specifying exx, eyy, ezx, eyx, eyz, and ezz. Diagonal terms (exx, eyy, ezz) of strain tensor are also called tensile strain; off-diagonal terms (eyz, exz, eyx) of strain tensor are also called shear strain. Meaningful only with periodic systems and only in dynamics.</td>
</tr>
<tr>
<td>exx</td>
<td>Boolean</td>
<td>false</td>
<td>xx component of strain tensor.</td>
</tr>
<tr>
<td>eyy</td>
<td>Boolean</td>
<td>false</td>
<td>yy component of strain tensor.</td>
</tr>
<tr>
<td>ezz</td>
<td>Boolean</td>
<td>false</td>
<td>zz component of strain tensor.</td>
</tr>
<tr>
<td>eyz</td>
<td>Boolean</td>
<td>false</td>
<td>yz component of strain tensor.</td>
</tr>
<tr>
<td>exz</td>
<td>Boolean</td>
<td>false</td>
<td>xz component of strain tensor.</td>
</tr>
<tr>
<td>exy</td>
<td>Boolean</td>
<td>false</td>
<td>xy component of strain tensor.</td>
</tr>
</tbody>
</table>
Description

Minimization and dynamics usually print only a summary at the beginning and after completion, to the standard output file. The print command is used to obtain additional information and to place it in the archive, output or table files.

A single print command can print to only one of these three places. However, a dynamics or minimize command may have multiple print commands. If the filename keyword is specified, that filename is used; otherwise, output filenames are defaulted to:

- **archive**
  
  run_name.arc

- **output**
  
  run_name.out

- **table**
  
  run_name.tbl

- **history**
  
  run_name.his

where run_name is your name for the calculation. Coordinates, velocities and derivatives can be printed out only in archive files. Other properties given can be specified only for an output or table file. If velocities and derivatives are printed out, the print command should give file extensions of .vel and .der, to prevent the .arc file from being overwritten.

Only instantaneous values can be printed out for coordinates, velocities, and derivatives. For other properties, instantaneous values, running averages, and batch averages can be printed out during dynamics in both the standard output file and the table file. Standard deviations for both the running sum and the batch sum can be printed out only in the table file.

If you do not specify the type of averages to be printed in the standard output and table file, the default is to print both running and batch averages in the standard output and instantaneous values in the table file.

The running average includes every value computed thus far, not just values encountered at the specified frequency; i.e., the fre-
quency (keyword for the execute statement) is the printout frequency, not the sampling frequency. For batch averages, batch size is the number of samples to be used. If the batch size is \( n \), the window consists of the last \( n \) samples (or all samples if there are fewer than \( n \)).

When the filetype selected is history, only the filename parameter is noticed by the print command. History files are similar to binary archive files and contain a fixed set of information which can be read by the Cerius² and Insight II programs.

Example 1

minimize execute +before +after frequency = 10 \
   command = {print output +energy_summary}

This example of the print command means to run the minimizer with its default settings and to print the energy summary to the output file before minimization starts, every 10 steps during the minimization, and after the end of the run.

Example 2

dynamics time = 300 execute frequency = 10 \
   first_step = 20 last_step = 200 \
   command = {print table file = summary.tbl \
      +energy_summary +state +average +average_sd} \
   execute frequency = 20 command = \
   {print archive +coordinates}

This example of the print command means to run dynamics for 300 fs and to print the energy summary and the state information to the table file summary.tbl, starting at step 30 (20 + 10) and continuing every 10 steps until step 200; also, write the coordinates to the archive file run_name.arc every 20 steps.

pseudoAtom

Purpose

The pseudoAtom command is used to create, access, and update pseudoatoms whose coordinates are defined by the position of atoms existing in the Discover environment.
**Syntax**

pseudoAtom operation arg1 ?arg2 ...

<table>
<thead>
<tr>
<th>Operation</th>
<th>Arguments</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>define</strong></td>
<td>path:name (spec ?weight (weight-Vec</td>
<td>column)?)</td>
</tr>
<tr>
<td><strong>update</strong></td>
<td>—</td>
<td>Update the coordinates of all pseudoatoms in the current system database from their definition subsets.</td>
</tr>
<tr>
<td><strong>getDefinition</strong></td>
<td>varName spec</td>
<td>Get the list of atoms defining pseudoatoms specified by spec as an object varName. An error is produced if spec specifies anything other than existing pseudoatoms. If a variable name, objName, is specified, then this is set to weight the object used to define the pseudoatom. This is either the set of scalar weights for user-weighted pseudoatoms, a string object containing the column name for column-weighted pseudoatoms, the xyz coordinates for fixed pseudoatoms (the object returned in varName will be empty), or empty for non-weighted pseudoatoms.</td>
</tr>
</tbody>
</table>

**Description**

Pseudoatoms are very useful in molecular modelling for, e.g., accessing key features of model geometry or simplifying energy calculations. A pseudoatom may have the same properties as a real atom, and so pseudoatoms are stored in the Atom table of the Discover system database in which they are defined. The pseudoA-
A. Btcl Language and Commands—Standalone Mode - Continued

**tom** command is similar to the **subset** command and the definition atoms (**spec** in the above table) are recorded as a subset. (These subsets are only available via the **PseudoAtom** table, which links the pseudoatoms in the **Atom** table to their definition subsets.)

Once a pseudoatom is created using the **pseudoAtom define** command, they are specified in an entirely analogous way to subsets (see the **subset** command). For example, *:psu specifies all the pseudoatoms with the name psu which are accessible at the monomer level of the current system database (assuming that no monomers also have this name). Similarly, *:*:* would specify all the atoms and pseudoatoms (and subsets if they exist) at the atom level of the current system database, whereas *:*:*:**PseudoAtom;** would specify just the pseudoatoms at this level.

The **pseudoAtom update** command causes the coordinates of all the pseudoatoms in the current system database to be recalculated from the coordinates of the objects in their definition subsets. This command should always be performed after any operation (or series of operations) that alter the geometry of the model(s) in the current system database.

The **pseudoAtom getDefinition** command is used to retrieve all the objects (usually atoms) that define a pseudoatom. The **spec** argument may specify only valid pseudoatoms, and the object returned as **varName** will contain grouping information so that each group corresponds to each pseudoatom specified. (If an original pseudoatom definition involved subsets, these are expanded into their component objects prior to creating the pseudoatom subsets, and therefore the **pseudoAtom getDefinition** command returns the expanded list.)

**How pseudoatoms are used**

During a simulation (single-point energy, minimization, or dynamics), the energy due to the pseudoatom and the derivatives of the energy with respect to the pseudoatom coordinates are computed. The particular functional form used to describe the partial dependence of the energy on the pseudoatom coordinates is determined by what type of restraint(s) is imposed on the pseudoatom. The pseudoatom energy contribution appears in the Restraint portion of the energy hierarchy. Once the pseudoatom derivatives have been computed, the contributions are mapped back onto the energy derivatives with respect to real atom coordinates. The mapping depends on the type of pseudoatom, as follows:
**pseudoAtom**

F

fixed—no mapping (derivative is zero)

A

geometric centroid—average pseudoatom gradient values are added to each of the constituent atoms, i.e.:

\[ \text{grad(constituent atom)} + \frac{\text{grad(pseudoatom)}}{N} \]

where \( N \) is the total number of constituent atoms.

C

mass weight—weighted average of pseudoatom gradient values are added to each of the constituent atoms. The weights, \( W_i \), are obtained from the *Pseudoatom Table* column, *PseudoAtom.Weight* of type OA_OA, which contains the explicit weights for each pseudoatom or the name of a column in the *Atom* table where weights can be obtained. For each constituent atom we have:

\[ \text{grad(constituent atom)}_i + \frac{W_i \times \text{grad(pseudoatom)}}{\text{SUM}(W_i)} \]

where \( \text{SUM}(W_i) \) is the sum of all weighting factors.

For this release, no processing is implemented for second derivatives.

**Example 1**

```
BTCL > pseudoAtom define mol:(6,10):psu "CA,N,HN"

BTCL > pseudoAtom define top_psu "mol:(6,10):(CA,N,HN)"

BTCL > database handle dbh ; $dbh get coords Coord "top_psu, *:*:psu"

BTCL > object coords 
{24.383842 8.953845 0.135386} {17.498986 6.778482 0.076494} {31.268697 11.129208 0.194278}
```

This example shows two *pseudoAtom define* commands which define three pseudoatoms, followed by a *database handle* operation to retrieve their coordinates. In the first definition, two pseudoatoms were created with the *path:name* specifications mol:6:psu and mol:10:psu. Both pseudoatoms have the centroids of three atoms: CA, N, and HN, in the corresponding monomers, i.e., monomers 6 and 10 of the model called mol. In the second definition, a pseudoatom is defined with the name top_psu and is
A. Btcl Language and Commands—Standalone Mode - Continued

accessed at the model hierarchy level. Its definition uses the same atoms specified as the other two pseudoatoms but its coordinates are the centroid of all six of these atoms. Note that the coordinates for the top_psu pseudoatom is the average of the coordinates for the two psu pseudoatoms.

Example 2

```
BTCL > pseudoAtom define top_psu2 "*:*:psu"
BTCL > subset define Subset top_sub "mol:6:(CA,N,HN)"
BTCL > pseudoAtom define top_psu3 top_sub
BTCL > $dbh get coords Coord "top_psu2,top_psu3"
```

This example continues from the previous one and shows the definition of two more pseudoatoms, top_psu2 and top_psu3, which are both accessed at the model hierarchy level. The first `pseudoAtom define` command shows top_psu2 being defined from the two psu pseudoatoms. The second shows top_psu3 being defined from a subset that actually contains the same atoms used to define the mol:6:psu pseudoatom. Naturally, the coordinates for top_psu2 and top_psu3 are the same as those for top_psu and mol:6:psu, respectively.

Example 3

```
BTCL > pseudoAtom getDefinition psu_bases "mol:6:psu, top_psu3, top_psu, top_psu2"
BTCL > $dbh get options grouping psu_base_names Name $psu_bases
BTCL > object psu_base_names get groups
```

This example shows the use of the `pseudoAtom getDefinition` command. It is used here to obtain a list of the atoms which defined the pseudoatoms mol:6:psu, top_psu3, top_psu, and top_psu2. The following database handle operation is used to get the atom names from the corresponding atom objects in psu_bases. Note the use of the `grouping` option on the `get` operation to pre-
serve the grouping information from object variable psu_bases to object variable psu_base_names. From the results of the object command, it is seen that, although top_psu3 was defined from a subset (top_sub), this subset was expanded to its component atoms so that the definition atoms are the same as for mol:6:psu. However, although top_psu2 was defined from the two psu pseudoatoms, which were themselves defined from the same atoms that defined top_psu, these were not expanded. This is because, once a pseudoatom has been defined, it is treated almost like an ordinary atom and not a subset.

**Example 4**

```
BTCL > molGeom set distance relative 0.5 "mol:6:CA" "mol:6:N"
BTCL > pseudoAtom update
BTCL > $dbh get new_coords Coord "*:*:psu"
BTCL > object new_coords
{(17.266647 6.540354 0.055878) (31.268697 11.129208 0.194278)}
```

This example shows the use of the `pseudoAtom update` command. The `molGeom` command is first used to increase the bond distance between atoms CA and N of monomer 6 of model mol. The `pseudoAtom update` command is then used to update the coordinates of all the pseudoatoms in the default system database. These are then displayed for the psu pseudoatoms. It is seen that the coordinates for mol:6:psu have changed slightly from their original values (see Example 1) as a result of two of the definition atoms having moved after the `molGeom` command. (The coordinates for mol:10:psu are unchanged, as expected.)

**Example 5**

```
BTCL > pseudoAtom define gly_1_com "*:GLY_3:Atom;*" weight Mass
```

This example shows the definition of a pseudoatom using mass-weighted atom coordinates; specifically, a pseudoatom on the third GLY residue of a model.

**Example 6**

```
BTCL > pseudoAtom define gly_2_usr "*:GLY_3:Atom;*" weight {1 1 1 3 1 1 1}
```
A. Btcl Language and Commands—Standalone Mode - Continued

This example shows the definition of a pseudoatom using user-weighted atom coordinates; specifically, a pseudoatom on the third GLY residue of a model for which the fourth atom coordinate (of seven atoms) is given 3 times as much weighting.

Example 7

**BTCL** > `pseudoAtom define gly_3_fix coord {{1 1.5 -0.3}}`

This example shows the definition of a fixed pseudoatom. Here, no atom specification is needed, and one pseudoatom is created at the fixed coordinates (1.0 1.5 -0.3) (relative to the current system).

Note that if the *path* specifies more than one context for the pseudoatom to be defined in (e.g., *::*:FIX), then the last argument must specify as many coordinate vectors as there are valid contexts. For example:

**BTCL** > `pseudoAtom define "*::GLY_*::fixed" coord {{0 0 0} {1 1 1} {2 2 2} {3 3 3}}`

creates pseudoatoms at the monomer level with the name, fixed, for each GLY-type residue (assuming that there are exactly four such residues).

---

**rattle**

**Purpose**

The *rattle* command is used before the *dynamics* command to set up constraints in bonds, angles, or water fragments in a molecular dynamics simulation. It can be used to constrain bonds or any atom pairs to user-defined distances. It can be used to constrain angles spanned by two constrained bonds. In addition, it can be used to fix the geometry of water fragments so that the fixed-geometry water models SPC and TIP3P can be used in a simulation.

**Syntax and description**

The syntax differs slightly, depending on whether bonds, angles, water, or nothing is being constrained:
Constraining bonds or atom pairs

**rattle bonds** `-bondLength bondlength_value?` `-tolerance tolerance_value?` `atom1_list atom2_list?`

By default, every bond in the system is constrained to its current length if the **rattle bonds** command is issued without the optional keywords and values.

- **-bondLength bondlength_value** specifies the interatomic distance to which the atom pairs are constrained.

- **-tolerance tolerance_value** specifies the tolerance with which the constraints are enforced.

`atom1_list atom2_list` are two sets of atom specifications which identify the atom pairs to be constrained.

Constraining angles

**rattle angles** `-angleValue angle_value?` `-tolerance tolerance_value?` `atom1_list atom2_list atom3_list?`

The roles of the optional keywords in the **rattle angles** command are analogous to those of `**-bondLength**`, `-tolerance`, and the atom lists in the **rattle bonds** command.

By default, every angle in the system is constrained to its current value.

**Note**, however, that the **rattle angles** command actually constrains the distance between the 1,3 atoms of an angle. Therefore, you need to issue the **rattle bonds** command before the **rattle angles** command to constrain the bond distances so that the angle itself can be constant.

Constraining water geometry

**rattle water** `-waterType SPC|TIP3P|CURRENT?` `-tolerance tolerance_value?`

By default, water fragments are constrained to their current positions, using the **CURRENT** water model. However, when the **SPC** or **TIP3P** models are specified, the water fragments are constrained to their equilibrium geometry according to that model. The nonbond parameters are also changed to the values used in the specified water model.
Turning rattle off

This command clears all constraints set up by previous rattle commands. Notice that this command is also needed when you want to change the rattle parameters from those used in previous dynamics stages (for example, to rattle angles in addition to bonds). You have to turn rattle off first and then issue new rattle commands to change these parameters.

Example 1

rattle bonds -bondLength 1.0000 -tolerance 1e-4 "*:WTR_**:H11" "*:WTR_**:O1"

All the bonds between H11 and O1 in residues WTR of all fragments in a box of water fragments are constrained to bond lengths of 1.0 Å during dynamics, with a tolerance of 1e-4 Å.

Example 2

rattle angles -angleValue 109.0 \\ -tolerance 1e-4 "*:WTR_**:H11" "*:WTR_**:O1" "*:WTR_**:H12"

All the angles formed by H11, O1, and H12 in residues WTR of all fragments are constrained to an angle of 109° during dynamics, with a tolerance of 1e-4°.

Example 3

rattle water -waterType SPC -tolerance 1e-4

All the water fragments are maintained at the fixed geometry of the SPC model during dynamics, with a tolerance of 1e-4. Furthermore, the nonbond parameters used for water are changed to those of the SPC model.

This command also makes use of the molGeom command to fix the geometry of the water fragments.

Example 4

rattle water -waterType CURRENT -tolerance 1e-4

All water fragments are maintained at the current geometry during dynamics. The current forcefield is used for water.
Example 5

rattle bonds

Constrain all bonds in models to the current bond lengths. The tolerance is set at the default of 1e-5 Å.

readFile

Purpose

The readFile command is used to read in a data file created by previous Discover, Cerius², or Insight runs (e.g., a coordinate snapshot from an archive file). It also allows you to give your system a name, so that multiple systems can be calculated and compared.

Syntax

readFile filetype ?keyword = value?...

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Values</th>
<th>Default</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>filetype</td>
<td>molec<em>ular_system arc</em>hive coord*inate or car dynamics_restart forcefield history</td>
<td>molecular_system</td>
<td>Specify the type of file(s) to read data from. One of these values must be the first argument after readFile, since the filetype is specified by a positional argument.</td>
</tr>
<tr>
<td>binary</td>
<td>Boolean</td>
<td>false</td>
<td>Read the binary form of the file. Currently available only for the forcefield filetype (which must be binary).</td>
</tr>
<tr>
<td>file*name</td>
<td>string</td>
<td>run_name.extension</td>
<td>User-defined name of file. The default is the current run name with the appropriate extension. See below.</td>
</tr>
<tr>
<td>frame</td>
<td>integer</td>
<td>1</td>
<td>For archive and history files, specifies which frame to read.</td>
</tr>
<tr>
<td>system_name</td>
<td>string</td>
<td>—</td>
<td>If unspecified, the most-recently read system becomes the current system. If specified, the file is read into a virtual system (with this name), which can later be chosen with the database default command.</td>
</tr>
</tbody>
</table>
Description

The **readFile** command reads data from a particular file. If the **filename** parameter is not specified, the filenames used depend on the **filetype** parameter and are as follows:

- **archive**
  - `run_name.arc`
- **coordinate**
  - `run_name.car`
- **molecular_system**
  - `run_name.car` and `run_name.mdf`
- **dynamics_restart**
  - `run_name.xdyn`
- **history**
  - `run_name.his`

(In the above file types, `run_name` is your name for the calculation.)

**forcefield**

`$FORCEFIELD.xfrc` (with **binary** option), where `$FORCEFIELD` is the value of the `$FORCEFIELD` environment variable.

Note—If the **filename** parameter is given without an extension, the appropriate extension is added.

The **dynamics_restart** file contains data needed to restart a dynamics run.

For the **archive** and **history** files, the **readFile** command returns the value of the energy specified for this frame. This is the total energy of the system when it was written to the file (potential energy for frames generated by minimizations and potential + kinetic energy for frames generated by dynamics). If an attempt is made to read beyond the last frame in the file, an empty string is returned. This can be used to indicate when the end of the file has been reached, as in Example 1. To control the precision of the energy value, the Tcl variable **tcl_precision** may be used.
Example 1

for {$i = 1} {[set energy [readFile archive frame = $i]] != ""} {incr i} {
    puts $file "$i $energy"
    ...
}

Construct a loop that reads each frame in turn from the archive file run_name.arc. Print the frame number and energy to $file. The loop ends when the readFile command returns an empty string for the energy. This occurs upon reaching the end of the file. So it is not necessary to know a priori how many frames are in the archive file.

Example 2

readFile history frame = 2

Read the second frame from the history file run_name.his.

Example 3

readFile molecular_system filename = test

Read the coordinate and structural data files test.mdf and test.car.

Example 4

readFile forcefield +binary filename = cff91

Read the binary forcefield file cff91.xfrc.

Example 5

readFile car file = test1

Read the coordinate file test1.car.

Example 6

readFile dynamics_restart filename = crn.xdyn
cdlConfig dynamics temperature = 390.0
dynamics

Read the dynamics restart file crn.xdyn. The default values of dynamics parameters are set to those used by the dynamics run that wrote the crn.xdyn file. Then change the dynamics temperatures to 390 K for the new dynamics run. The order in which these two commands appear is important—the readFile dynamics_restart command overwrites any previous default parameter values.
The `reset` command is used to reread the structural data files and forcefield. Its purpose is to permit a single Discover run to operate on different models. All information associated with the current model (e.g., coordinates, energy expression) is lost with this command. However, the settings of command defaults (e.g., for dynamics) are unchanged.

**Syntax**

```
reset ?keyword = value?...
```

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Values</th>
<th>Default</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>coordinate</td>
<td>string</td>
<td><code>run_name.car</code></td>
<td>Name of coordinate data file. If none given, defaults to <code>run_name.car</code>.</td>
</tr>
<tr>
<td>topol*ogy</td>
<td>string</td>
<td><code>run_name.mdf</code></td>
<td>Name of structural data file that defines connections, etc. If none given, defaults to <code>run_name.mdf</code>.</td>
</tr>
<tr>
<td>forcefield</td>
<td>string</td>
<td><code>$FORCEFIELD</code></td>
<td>Name of forcefield file. If none given, defaults to value of the <code>$FORCEFIELD</code> environment variable.</td>
</tr>
</tbody>
</table>

**Example 1**

```
reset topology = polyeth coordinate = polyeth
```

This example of the `reset` command resets the structural data structures and reads the contents of `polyeth.mdf` and `polyeth.car`. The current forcefield is retained.

**Example 2**

```
reset topology = acenm coordinate = acenm1 \ 
forcefield = cvff
```

This example of the `reset` command resets the structural data structures, reads the contents of `acenm.mdf` and `acenm1.car`, and specifies to use the CVFF forcefield.
restraint

Purpose

The restraint command is used both to create new restraints and to modify or examine existing ones. A restraint is essentially an extra energy term which is added to Discover energy calculations for a specified set of atoms.

Syntax

restraint operation varName ?arg1 ...?

restraint command operations:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Arguments</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>create</td>
<td>varName</td>
<td>Create a new restraint object varName of restraint type restraintType. The number of arguments depends on restraintType (see Restraint Types). For restraint types other than btcl, one n-tuple subset or n atom specifications are required as arguments.</td>
</tr>
<tr>
<td>function</td>
<td>varName function param1 ?param2 ...?</td>
<td>Set the function and parameters used by restraint varName to calculate the restraint energy contribution. The number and types of parameter arguments required depend on the value of function (see Restraint Functions).</td>
</tr>
<tr>
<td>target</td>
<td>varName ?relative? ?value1 ...?</td>
<td>Set the target values used by restraint varName to calculate the restraint energy contribution. The number of value arguments required is usually zero or one, depending on the restraint type (see Restraint Types). Two arguments are required if the function for restraint varName is flatBottomed, and two, four, six, or eight arguments are required if the function is j3dihedral (see also Restraint Functions).</td>
</tr>
<tr>
<td>scale</td>
<td>(varName</td>
<td>-all</td>
</tr>
<tr>
<td>clear</td>
<td>—</td>
<td>Delete all restraint objects from the current system database. To delete a particular restraint object use unset varName.</td>
</tr>
</tbody>
</table>
A. Btcl Language and Commands—Standalone Mode - Continued

### Restraint types:

<table>
<thead>
<tr>
<th>Restraint type</th>
<th>Number of arguments</th>
<th>Default function and parameters</th>
<th>Target value description</th>
</tr>
</thead>
<tbody>
<tr>
<td>chiral</td>
<td>1</td>
<td>flatBottomed 100.0 1000.0</td>
<td>A string object whose items value must be one of A (as is), I (invert), R, or S.</td>
</tr>
<tr>
<td>tether</td>
<td>1</td>
<td>quadratic 100.0 1000.0</td>
<td>Either a coordinate vector or an atom spec. (Tethering to a target is called template forcing.)</td>
</tr>
<tr>
<td>distance</td>
<td>1 or 2</td>
<td>quadratic 100.0 1000.0</td>
<td>A vector of distances in angstroms.</td>
</tr>
<tr>
<td>angle</td>
<td>1 or 3</td>
<td>cosine 100.0 1</td>
<td>A vector of angles in degrees.</td>
</tr>
<tr>
<td>torsion</td>
<td>1 or 4</td>
<td>cosine 100.0 1</td>
<td>A vector of angles in degrees.</td>
</tr>
<tr>
<td>outOfPlane</td>
<td>1 or 4</td>
<td>cosine 100.0 1</td>
<td>A vector of angles in degrees.</td>
</tr>
<tr>
<td>btcl</td>
<td>1 or more</td>
<td>arg1 ? arg2 ...?</td>
<td>You may not use restraint target/function commands. These parameters are implicit to the user btcl function, arg1, which has arguments arg2, arg3, etc.</td>
</tr>
</tbody>
</table>

---

**Description**

The **restraint create** command is used to create a new restraint object, *varName*. This restraint object becomes active immediately to add an extra function (a restraint) to the Discover energy calculation routine for the current (default) system database. You may define as many restraints for as many databases as you want.

A restraint object is similar to a Btcl object in that the value of $varName$ is a restraint object handle. Hence the restraint is deleted if variable *varName* is reassigned or unset and is only recognized as a restraint object by **restraint** commands.
When a restraint object is first created, all the restraint qualities, such as target values, energy function, function parameters, etc., are given default values such that the restraint varName restrains the target set of atoms to their current geometries. (See Restraint Types.) For example:

**BTCL > restraint create rst1 angle myang**

**BTCL > restraint create rst2 chiral *:(3,5):CA**

In this example two restraint objects, called rst1 and rst2, are created in the default system database. Restraint rst1 is an angle type restraint for the existing angle subset called myang. These angles are now restrained to their current values using a cosine function with default parameters (see Restraint Types). Restraint rst2 is a chiral restraint on the chiral (or prochiral) alpha carbon atoms of monomers 3 and 5 of the fragment(s) in the current system database. The chiralities at these atoms are now restrained to their current values (either R or S).

Once a restraint object has been created, other restraint commands may be used to change that restraint object’s qualities. (See restraint Command Operations.) The most usual quality that you would want to change is the restraint target value(s) so that later minimization or dynamics runs can be used to change the geometry of a model. This is achieved using the restraint target command, for example:

**BTCL > restraint target rst1 60**

**BTCL > restraint target rst2 "R S"**

In this example, all the angles specified by restraint rst1 (the subset myang) are to be restrained to 60°, and the two chiralities specified by restraint rst2 are restrained to the R and S configurations, respectively. When the restraint target uses real numbers, the restraint target command may include the keyword relative, which indicates that the target value is to be set to the current value (the restraint measure) plus the argument value(s), for example:

**BTCL > restraint target rst1 relative 10**

In this example, the angles specified by restraint rst1 are set to whatever their current values are plus 10°. Note that this is not rel-
native to the current target value (60°). For restraints of type tether, you may set the target value to a set of coordinates defined by a vector object specification or another set of atoms. For example:

```bash
BTCL > restraint create rst3 tether "*:3:Atom;*
BTCL > restraint target rst3 $atoms
BTCL > restraint target rst3 $atomCoords
BTCL > restraint target rst3 relative "(1 0 0)"
```

In this example, tethering restraint rst3 is first created for all the atoms in monomer 3 of the fragment(s). This would tether these atoms to their current coordinates. The restraint target is then changed to new coordinates taken from the atoms defined by $atoms, which must specify the same number of atoms. These atoms could originate from a different database. (A tethering restraint which has a (nondefault) target is often called a template forcing restraint.) The restraint target is then changed to a set of coordinates defined by $atomCoords, which must have the same number of coordinates as there are atoms specified by restraint rst3. The restraint target is then changed again to finally set the target coordinates to be +1 in the x direction from their current values.

You may also choose to change the function or function parameters used by a restraint using the `restraint function` command. For example:

```bash
BTCL > restraint function rst1 quadratic 100 500
BTCL > restraint function rst2 flatBottomed 5 5 100
```

Notice in this example that, when a `restraint function` command is used, the function and all its parameters must be specified (see Restraint Functions). For restraint rst1, the function is changed to quadratic with a force constant value of 100 and a maximum derivative value of 500. For restraint types tether and chiral, it is illegal to use a restraint function other than the default, but the default may be re-specified in order to change the function parameters, as with restraint rst2 in this last example. It is important to note here that the `restraint function` command may change the target value(s) of a restraint. For example, conversion of a flatBot-
tomed function to a quadratic function will set the quadratic target to the mean of the upper and lower flat-bottomed targets. In many cases, you may want to follow a restraint function command with a restraint target command reflecting your own target choice. For a conversion from a quadratic function to a flat-bottomed function, for example, the upper and lower targets are both set to the quadratic target value.

Although the force of a particular restraint (in, e.g., a minimization) may be changed by changing the function parameters, it is usually easier to scale the restraint function using the restraint scale command (see restraint Command Operations). This sets the scale parameter of a restraint object, which has the default value of 1. If the new scale value is 0, then the restraint specified no longer contributes to the energy calculation. In fact, the energy function is not even called, which is important to note (for performance reasons) when the restraint type is btcl. The scaling may be applied to all restraints defined in a system database by using the keyword -all where varName would usually appear. This is useful for disabling all restraints, resetting all the scaling factors back to 1, or as in the example below, where all restraints are disabled except for one:

```
BTCL > restraint scale -all 0

BTCL > restraint scale rstl 1
```

You can also scale all restraints of a given type with a command like the following:

```
BTCL > restraint scale -type chiral 3.0
```

Note that the scale set by a restraint scale command is absolute, not relative to the previous scaling.

To remove all the current restraints on a system, rather than disabling them, use the restraint clear command.

You may want to interrogate an existing restraint object using the restraint get command. All data associated with a restraint may be accessed through this command by specifying the appropriate keyword and supplying variable names to which this information is returned (see restraint get Keywords). For example:
A. Btcl Language and Commands—Standalone Mode - Continued

BTCL > echo [restraint get rst1 type]
angle

BTCL > echo [restraint get rst1 function]
quadratic

BTCL > restraint get rst1 parameters p1 p2

BTCL > echo [object p1] [object p2]
100.0 500.0

In particular, the *restraint get* command is useful for returning the sum and component energies of a restraint, which are calculated independently of the Discover *energy* command, and for returning the current values associated with a restraint, i.e., the *measure*. The latter operation requires the *measure* keyword and can be used to retrieve quantities such as torsion angles, interatomic distances, etc. and, in particular, the chirality (or prochirality) of a chiral atom, which is not currently available through other Btcl commands.

The *restraint get* command is restricted as to what information can be returned for a restraint object of type *btcl* (see *restraint get* Keywords). This is because this type of restraint allows you to specify your own energy function in the form of a Btcl procedure. The first argument after a *restraint create btcl* command must be the procedure name. However, the existence of this procedure is not checked, so that you may define, or redefine, this procedure after the restraint has been set up. The following arguments are passed to the Btcl procedure when that procedure is called as the restraint is applied, i.e., during a Discover system energy calculation. It should be noted that these arguments are fixed but could be, for example, atom specifications or variable names.

The procedure that the restraint object of type *btcl* calls is linked to the energy code via an *energyContribution* command (see Example 5). Having multiple calls to *energyContribution* in the procedure is not recommended, since only the first of these is scaled by the *scale* factor.
Restraint functions:

<table>
<thead>
<tr>
<th>Restraint function</th>
<th>Arguments</th>
<th>Energy expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>quadratic</td>
<td>$k mD$</td>
<td>$E = \text{scale} \cdot k \cdot (V - V_0)^2$</td>
</tr>
<tr>
<td>cosine</td>
<td>$k \text{ sym}$</td>
<td>$E = \text{scale} \cdot k/2 \cdot (1 - \cos(\text{sym} \cdot (V - V_0)))$</td>
</tr>
</tbody>
</table>
| flatBottomed       | $k_1 k_2 mD$ | $E(V \leq V_0) = \text{scale} \cdot k \cdot (V - V_0)^2$
|                   |           | $E(V_0 < V < V_1) = 0.0$
|                   |           | $E(V \geq V_1) = \text{scale} \cdot k \cdot (V - V_1)^2$ |
| cis                | $k$      | $E = \text{scale} \cdot k/2 \cdot (1 - \cos(V))$ |
| trans              | $k$      | $E = \text{scale} \cdot k/2 \cdot (1 + \cos(V))$ |
| cis/trans           | $k$      | $E = \text{scale} \cdot k/2 \cdot (1 - \cos(2V))$ |
| j3dihedral         | $k_1 k_2 mD$ | Like flatBottomed function, except $V_0$ and $V_1$ are selected from the 2, 4, 6, or 8 target values specified. Which pair of values to select is decided the first time that the restraint is applied, according to which flat-bottomed potential (using these values as $V_0$ and $V_1$) is closest to the initial value of $V$. |

$a, k_1, k_2$ are force constants. The default value is 100. $mD$ is the maximum derivative. The energy expression becomes linear from the points where the derivative would be $>= mD$. The default value is 1000. $\text{sym}$ is a symmetry number, i.e., the periodicity of the cosine function.

$b$ All parameters in these expressions, except for $\text{scale}$, are components of the corresponding vector quantities. $\text{scale}$ is the restraint object scale factor; $V$ is the current measurement of the restraint object, e.g., the current torsional angle; $V_0$ is the target value for the restraint (with $V_1$ where appropriate).

**restraint get** keywords:

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>Return the restraint type as a Tcl string.</td>
</tr>
<tr>
<td>function</td>
<td>Return the restraint function as a Tcl string. Not available for restraints of type btcl.</td>
</tr>
<tr>
<td>parameters</td>
<td>Requires 1, 2, or 3 extra arguments. These are variable names to Btcl objects that are created, containing the function parameters. The number of arguments depends on the restraint function. (See Restraint Functions.) Not available for restraints of type btcl.</td>
</tr>
<tr>
<td>target</td>
<td>Usually requires 1 extra argument, which is the variable for a Btcl object returned containing the restraint's target values. If the restraint function is flatBottomed or j3dihedral, this operation requires 2 arguments to return both sets of target values (the post-initial set for j3dihedral—see Restraint Functions). Not available for restraints of types btcl, cis, trans, or cis/trans.</td>
</tr>
</tbody>
</table>
A. Btcl Language and Commands—Standalone Mode - Continued

### Example 1

```tcl
BTCL > restraint create rst1 distance *:1:CA *:2:CA
BTCL > restraint function rst1 quadratic 100.0 500.0
BTCL > restraint target rst1 4.0
```

These commands create a distance restraint named rst1 between CA atoms in monomers 1 and 2. The form of the restraint is quadratic and the restraint parameters are:

- $k = 100.0$
- $mD = 500.0$
- $V_0 = 4.0$

### Example 2

```tcl
BTCL > restraint create rst2 distance *:1:CA *:2:CA
BTCL > restraint function rst2 quadratic 100.0 500.0
BTCL > restraint target rst1 relative 4.0
```

Keyword Operation

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>measure</td>
<td>Requires 1 extra argument. A Btcl object is created with this name and containing the current values for the restraint, e.g., for a restraint object of type angle, this returns the angle values for the restraint’s atom sets in their current geometry. Not available for restraints of type btcl.</td>
</tr>
<tr>
<td>scale</td>
<td>Return the scale factor of a restraint as a Tcl string.</td>
</tr>
<tr>
<td>database</td>
<td>Requires 1 extra argument. A Btcl object is created with this name and contains the subset of atoms or angles, etc., used to define the restraint. For restraints of type chiral, the initial specification of chiral atoms is augmented by lists of atoms that these are connected to. Not available for restraints of type btcl.</td>
</tr>
<tr>
<td>subset</td>
<td>Return the total energy that a restraint contributes to the energy of the system. If an extra argument is provided, a Btcl object is returned with this variable name, which contains the individual energy components for each atom, distance, etc., that the restraint is defined for. Not available for restraints of type btcl.</td>
</tr>
<tr>
<td>energies</td>
<td>Requires 1 extra argument. A Btcl object is created with this name and contains individual energy components for each atom, distance, etc., for which the restraint is defined. Not available for restraints of type btcl.</td>
</tr>
<tr>
<td>derivatives</td>
<td>Requires 1 extra argument. A Btcl object is created with this name and contains individual derivative components for each atom, angle, etc., for which the restraint is defined. Not available for restraints of type btcl.</td>
</tr>
</tbody>
</table>
These commands create a restraint rst2 which is like restraint rst1, except that the \textit{relative} keyword has been used. If each monomer has one CA atom, and they are separated by 1.5 Å when rst2 is created, then the restraint parameters are:

\begin{align*}
  k &= 100.0 \\
  mD &= 500.0 \\
  V0 &= 5.5
\end{align*}

\textbf{Example 3}

\begin{itemize}
  \item BTCL > \texttt{restraint create rst3 angle *:1:CA *:2:CA *:2:O}
  \item BTCL > \texttt{restraint function rst3 quadratic 100.0 500.0}
  \item BTCL > \texttt{restraint target rst3 60}
  \item BTCL > \texttt{restraint create rst4 angle *:1:CA *:1:O *:2:CA}
  \item BTCL > \texttt{restraint function rst4 cosine 100.0 1.0}
  \item BTCL > \texttt{restraint target rst4 60}
\end{itemize}

These commands create two angle restraints, rst3 and rst4. Restraint rst3 is \texttt{quadratic}, and rst4 is a \texttt{cosine} restraint with parameters:

\begin{align*}
  k &= 100.0 \\
  sym &= 1.0 \\
  V0 &= 60.0
\end{align*}

\textbf{Example 4}

\begin{itemize}
  \item BTCL > \texttt{vector v "10.0 20.0"}
  \item BTCL > \texttt{restraint create rst5 torsion *:*:N *:*:CA *:*:C *:*:O}
  \item BTCL > \texttt{restraint function rst5 cis $v}$
\end{itemize}

This example assumes that the atom specifications in the \texttt{restraint create} command yield atom lists of length 2 corresponding to 2 torsions. The \texttt{cis} form requires only one parameter, \textit{k}, and use of the vector \textit{$v$} in the parameter list causes the use of \textit{k} = 10.0 for the first torsion and \textit{k} = 20.0 for the second torsion. Note that use of a target command is not permissible with the \texttt{cis} form.
A. Btcl Language and Commands—Standalone Mode - Continued

The same restraint could also be created with the following commands:

```
BTCL > subset define Torsion tor "{{*:*:N} {*:*:CA} {*:*:C} {*:*:O}}
BTCL > restraint create rst5 torsion tor
BTCL > restraint function rst5 cis $v
```

**Example 5**

```
BTCL > restraint create rst_user btcl myEner *:1:C
```

This command creates a user-defined restraint which is based on a user-defined Btcl procedure named myEner. The Btcl procedure might look like:

```btcl
proc myEner {spec} {
    subset get obj $spec
    database handle system_h System. db_n
    $system_h get coord Coord $obj
    vector r euclidean_norm $coord
    vector eterms multiply $r $r
    vector esum sum $eterms
    vector egrad multiply 2.0 $coord
    energyContribution atom $esum $egrad $obj
}
```

This procedure computes energy and gradient increments directly in Btcl and feeds them back into the Discover program with the `energyContribution` command.

**select**

**Purpose**

The `select` command is used to create a list of relation numbers from a hierarchial database according to a string syntax. Currently, only the system database type is set up with an inherent hierarchy, which is:

```
system::molecule:monomer:atom
```
select

**Syntax**

```
select varName spec
```

**Atom specifications**

Simple atom specifications, more complicated atom specifications, and specification operator precedences are tabulated separately in this section.

An “atom specification” (spec in the command syntax above) is a general term for a string that specifies a target or list of targets in a hierarchical system database. For the system database the general hierarchy is:

```
system::molecule::monomer::atom
```

The syntax of the specification is made up of strings specifying the target at each level. A specification may contain special characters, such as wildcard characters.

A specification may also be a Btcl object handle, e.g.:

```
select AL2 $AL1
```

in which case, all the `select` command does is to make a copy of an object that was previously defined (AL1) and to assign it to a new object AL2.

**Simple atom specification syntax:**

<table>
<thead>
<tr>
<th>Character sequence</th>
<th>Description/examples</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>target</code></td>
<td>Specify a string of characters and wildcards for name matching at the current level. The <code>*</code> wildcard matches any number of characters (including none). The <code>?</code> wildcard matches any single character. The case-insensitive keywords <code>first</code> and <code>last</code> specify the first and last items in the list of items at the current level.</td>
</tr>
<tr>
<td><code>target:</code></td>
<td>Specify the system database target. This sequence may occur only once in a specification and must be at the start, e.g., <code>md::*:*C</code>.</td>
</tr>
<tr>
<td><code>:</code></td>
<td>Move one level down the system hierarchy. For example, <code>*1</code> causes a list of molecules (rows) to be generated whose names match the string <code>*</code> (i.e., all of them); the <code>:</code> then changes the level to the monomer level for those monomers that are in the specified molecules; the target 1 is then used to match the items at the new level (monomers).</td>
</tr>
</tbody>
</table>
A. Btcl Language and Commands—Standalone Mode - Continued

<table>
<thead>
<tr>
<th>Character-sequence</th>
<th>Description/examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>,</td>
<td>Concatenate specifications; for example, <em>:1,</em>:3 produces a list from all the monomers of names 1 and 3 from all the molecules in the current system. Concatenated lists do not have to be in the same level.</td>
</tr>
<tr>
<td>( )</td>
<td>Group lists at a particular level. Usually used in conjunction with the “,” character; for example, *:(1,3) produces exactly the same list as <em>:1,</em>:3, but the path * is not re-specified. Note that white space in the specification is ignored.</td>
</tr>
<tr>
<td>table;</td>
<td>The semicolon is used to specify a table name to be considered at the current level rather than the default table at this level in the hierarchy. Currently only Molecule, Monomer, and Atom tables are connected by the system database hierarchy, but at each level it is possible to define a subset. For example, <em>:Monomer</em> specifies all monomers only, and <em>:</em> specifies all monomers, subsets, and pseudoatoms defined at the monomer level.</td>
</tr>
<tr>
<td>type_target</td>
<td>Used at the monomer level to specify the monomer type. MSI monomer names usually have the form type_number, e.g., GLY_3. Thus <em>:ALA_</em> might be used to specify all the ALA type monomers in the system.</td>
</tr>
<tr>
<td>target..target</td>
<td>The range operator .. produces a list of rows at the current level from the first to second targets, for example, *:2..5 specifies a list of monomers in order from 2 to 5, and *:5..2 specifies the same list in reverse order. When the range operator is specified on two lists, e.g., *:(1,3)..(5,4), then the lists are matched from paired elements in each list, i.e., the result here would be the same as for *:(1..5, 3..4). The matching of elements from two lists may be complicated by the lists containing different numbers of items, originating from different tables (levels) or columns, or having different groupings. The parser tries to make the most sensible match it can or it returns an error.</td>
</tr>
<tr>
<td>!target</td>
<td>The complement operator ! is used to generate a list by excluding all items in the following specification, e.g., mol:1:!H specification all heavy atoms (nonhydrogens) in monomer 1 of molecule mol. The complement operator is restricted to the table (level) and column (attribute) specified, e.g., <em>:!(Subset;p</em>) specifies all subsets at the atom level whose names do not start with p and does not refer to any atoms or pseudoatoms defined at this level.</td>
</tr>
<tr>
<td>\ char</td>
<td>The backslash is used to quote operators and other special characters. \ \ is used to quote a single \ character.</td>
</tr>
</tbody>
</table>
More atom specification syntax:

<table>
<thead>
<tr>
<th>Character sequence</th>
<th>Description/examples</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;column&gt;</code> target</td>
<td>Specify a column (attribute) that is used to match <code>target</code> in the hierarchy level at which the last list was specified, for example, <code>:*:&lt;Type&gt;GLY</code> and <code>:*::&lt;Type&gt;GLY</code> both specify all the monomers in the system of type GLY, i.e., all rows in the <code>Monomer</code> table for which the <code>Type</code> column value matches the string GLY. The latter of these examples does not work at the atom level, despite the extra <code>::</code>, since no atom target is specified, and this operation is performed as a filter on the last list specified. This feature is useful only when the path to this level is independently specified, as with the subset command.</td>
</tr>
<tr>
<td><code>@step</code></td>
<td>Used after a range operator to specify a step, for example, <code>mol:3..LAST@2</code> specifies every other monomer in molecule mol between 3 and the last one listed. The step value <code>step</code> is always positive, even for ranges that run in reverse order.</td>
</tr>
<tr>
<td><code>+shift</code></td>
<td>Specify objects that are <code>shift</code> rows along in the same table from those in last list specified, e.g., <code>:*:C+2</code> specifies all atoms in the current system that occur 2 after each C atom in the <code>Atom</code> table. The shift is always made in the context of the individual items in the specified list, e.g., <code>:(mol:1,mol:*:C)+1</code> specifies a list containing the monomer listed after 1 and all atoms listed after C atoms in the database for molecule mol. If a shift causes the object specification to go out of range, then that specification is ignored, e.g., <code>mol:LAST+1</code> is equivalent to <code>mol:2</code>, since mol:LAST produces an empty list. The <code>+</code> operator is similar to the <code>&lt; &gt;</code> operator in that it is performed on the last list specified, e.g., both <code>mol:LAST+-1</code> and <code>mol:LAST:+-1</code> specify the last-but-one monomer defined in molecule mol. Note the use of <code>+1</code>. There is no complementary <code>--</code> operator, since this character is often used in monomer types, etc.</td>
</tr>
<tr>
<td><code>#symmetry</code></td>
<td>Used to find the symmetry image of an atom for a system containing symmetry atoms. For example, <code>:*:*%111#2</code> specifies the set of atoms generated by applying the second symmetry operator (#2) to all matches to Si* (silicon) atoms. Currently, the symmetry operator may not be applied to pseudoatoms, ghost (offset) atoms, e.g., <code>:*:*%111#2</code>, or other symmetry atoms, e.g., <code>:*:*%111#3</code>.</td>
</tr>
<tr>
<td><code>%offset</code></td>
<td>Used to find the cell offset of an atom for a system containing periodic atoms, such as a crystal structure. The offset may be specified in one of two ways by <code>#(x,y,z)</code> or by <code>%xyz</code>, where the <code>x</code>, <code>y</code>, <code>z</code> are integer unit cell offsets along the unit cell vectors <code>a</code>, <code>b</code>, <code>c</code>, e.g., <code>:*:*%1-5,13</code> or by <code>%xyz</code>, where the unit cell offsets <code>xyz</code> are in the range -9 to 9, e.g., <code>:*:*%100,:*:*%10</code>. With the offset operator, a new ghost (offset) atom is created if the specified image atom does not already exist, e.g., <code>:*:*%120</code> would create the (1,2,0) unit cell image atoms for those atoms matching <code>:*:*</code> the first time these atoms are specified. Unlike the symmetry operator, the offset operator may be applied to pseudoatoms, ghost (offset) atoms, e.g., <code>:*:*%111%202</code>, or symmetry atoms, e.g., <code>:*:*%2%202</code>.</td>
</tr>
<tr>
<td><code>&quot;&quot;</code></td>
<td>Used (in matching pairs) to quote literal strings inside specifications, for example, &quot;molecule 3:2&quot;:<em>:</em> specifies all the atoms in molecule molecule 3:2. Since strings pass through the Tcl interpreter, it may be necessary to quote the &quot; characters, e.g., <code>:*:*&quot;Cu_2+t&quot;</code>.</td>
</tr>
</tbody>
</table>
A. Btcl Language and Commands—Standalone Mode - Continued

<table>
<thead>
<tr>
<th>Character sequence</th>
<th>Description/examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>{}</td>
<td>Used for multiple specifications, for example, {} {<em>:2:H} specifies two separate lists, * and <em>:2:H, and (</em>) (</em>:2:H) specifies only a single list. Multiple specifications are allowed only for certain commands, e.g., subset, and are not allowed to span systems, as would occur for {sys1:<em>} {sys2::</em>:2:H}. The select command does not allow multiple specifications.</td>
</tr>
</tbody>
</table>

**Specification operator precedence:**

<table>
<thead>
<tr>
<th>Precedence</th>
<th>Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>{}</td>
</tr>
<tr>
<td></td>
<td>::</td>
</tr>
<tr>
<td></td>
<td>,</td>
</tr>
<tr>
<td></td>
<td>:</td>
</tr>
<tr>
<td></td>
<td>;</td>
</tr>
<tr>
<td></td>
<td>_</td>
</tr>
<tr>
<td></td>
<td>&lt; &gt; # %</td>
</tr>
<tr>
<td></td>
<td>!</td>
</tr>
<tr>
<td></td>
<td>..</td>
</tr>
<tr>
<td></td>
<td>@</td>
</tr>
<tr>
<td></td>
<td>+</td>
</tr>
<tr>
<td>V</td>
<td>( )</td>
</tr>
</tbody>
</table>

**Background**

The select command and object specification syntax are used to quickly access a system database to select lists of fragments, monomers (residues or sidechains), and atoms. These lists are very useful in the Discover program for defining geometric properties of fragments and residues, like distances and angles, so that they may be monitored, analyzed, or altered. They are also very useful for defining constraints or restraints on parts of models, etc., to reduce the amount of work in, for example, a minimization or dynamics simulation.
Example 1

**BTCL** > `select ats (CRN:2,PRI:3):CA`

In this example, a Btcl object variable called ats is created, which is the list of alpha-carbon atoms (CA) in the “2” monomer of the CRN model and in the “3” monomer of the PRI model.

Example 2

**BTCL** > `select ats MYSYS::CRN:!(GLY_*)`

In this example, a list of all the monomers in the fragment CRN in system MYSYS that are not of type GLY is returned in the object variable ats.

Example 3

**BTCL** > `select ats CRN:(1..4,6,8..11)`

In this example, the specification generates all the monomers of the CRN model with names 1, 2, 3, 4, 6, 8, 9, 10, and 11. This gives the desired list because of the operator precedence rules shown above.

Example 4

**BTCL** > `select ats crn:!3..15@3:CA..(CA+5)`

This last example again relies on the precedence rules to generate a specific list. Since “..” has higher precedence than “!” and “+”, the specification string is the simplest form of `(crn):(!(3..15@3)):(CA..(CA+5))`. The example produces a list of all the atoms from CA to CA+5, i.e., six in all, for all monomers in model crn except 3, 6, 9, 12, and 15.

Example 5

**BTCL** > `select ats "mysys::*:*:nonbond_group;c3"`

In this example, a list is made of all the c3 nonbond groups in system database mysys. Every character in the specification is lowercase, since table names, column names, and target strings are case-insensitive. However, system, table, and column names are not
allowed to contain wildcards, since it is not possible to create objects that span databases; and table columns may only be accessed individually. In this example, the resulting object ats is empty, since the table nonbond_group is not accessible through the system database hierarchy, even though it is part of the system database. Note that the spec argument is quoted in this example because of the ‘;’ character.

Example 6

\[
\text{BTCL} \rightarrow \text{select ats1 "Xtal::*:*:Si1#(1,2)"}
\]
\[
\text{BTCL} \rightarrow \text{select ats2 "Xtal::*:*:Si1#2..LAST"}
\]

In this example two atom lists, ats1 and ats2, are defined from a system called Xtal, which contains symmetry atoms. Assuming that this system contains one model and one monomer, then ats1 contains 2 atoms which are the images of the Si1 atom when operated on by symmetry operators 1 (the identity) and 2. Atom list ats2 specifies all symmetry images of the atom named Si1 other than the identity.

Note: the specification Si1# is equivalent to Si1.

Example 7

\[
\text{BTCL} \rightarrow \text{select ats1 "Xtal::*:*:Si1#2%101"}
\]
\[
\text{BTCL} \rightarrow \text{select ats2 "Xtal::*:*:Si1%(1..3,0,(1,3))"}
\]

In this example, two atom lists, ats1 and ats2, are defined from a periodic (infinite) system called Xtal, which also contains symmetry atoms. Assuming that this system contains one model and one monomer, then ats1 contains one atom, which is the 101 unit cell offset image (ghost atom) of the second symmetry image of atom Si1. Atom list ats2 specifies a number of periodic images of the atom Si1 using the range, grouping (…) and concatenation (,) operators. In fact, the expanded list is equivalent to the following specification: Xtal::*:Si1(#101,#201,#301,#103,#203,#303).
Note: the specification \texttt{Si1\%000}, or \texttt{Si1(0,0,0)}, is equivalent to \texttt{Si1}.

### subset

#### Purpose

The \texttt{subset} command is used to create or access subsets of objects in a database in the Discover environment.

#### Syntax

\texttt{subset operation arg1 ?arg2 ...?}

*The subset command operations:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Arguments</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{get}</td>
<td>\texttt{varName spec}</td>
<td>Get the list of items that the subsets contain, which are specified by \texttt{spec}, into an object handle \texttt{varName}. If a subset contains subsets, then these sub-subsets are also expanded into their component lists and so on. The command returns a string list of data types (table names) that \texttt{spec} expands into.</td>
</tr>
<tr>
<td>\texttt{define}</td>
<td>?allow_empty \texttt{Bool}? \texttt{Table path:name} (\texttt{spec}</td>
<td>\texttt{objHandle1} ?\texttt{objHandle2} ...?)</td>
</tr>
</tbody>
</table>
Description

A database subset is a named list of items (row numbers) taken from a particular database in the Discover environment. Unlike Btcl objects, which may also specify a list of items in a database, subsets are actually created within a database and can be accessed through the usual database operations and atom specifications (refer to the select command). A subset may contain a list of any database items, including other subsets.

The subset get command is used to retrieve the contents of subsets that are specified by an atom specification or an object handle.

The subset define command is used to create subsets in a database, which can be accessed through atom specifications and subset get commands. Ordinary subsets are simply defined by a list of object items specified by an atom specification string or an object handle, e.g., $obj. Subsets may also be made up of subsets or objects that are made up of multiple items, such as distances and angles. When a subset of, for instance, distance objects is defined, the subset define command first creates these objects from the supplied arguments, which must be either a multiple atom specification string or a number of object handles, e.g., two for distances. The Table argument is used to specify how many definition objects are required (see the table below).

Further explanations of the subset define and subset get commands are provided in the form of examples.

Tables available for subset definitions:

<table>
<thead>
<tr>
<th>Table</th>
<th>Number of points</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subset</td>
<td>1</td>
</tr>
<tr>
<td>Distance</td>
<td>2</td>
</tr>
<tr>
<td>Angle</td>
<td>3</td>
</tr>
<tr>
<td>Torsion</td>
<td>4</td>
</tr>
<tr>
<td>OutOfPlane</td>
<td>4</td>
</tr>
</tbody>
</table>

Example 1

```
BTCL > subset define Subset "rna::*:mons" "*"
```
This example uses the `subset` command to define subsets from the rna system database at the molecule level with the name mons. Each subset contains a list of all the monomer rows for that molecule. Note that the last two arguments are quoted only for emphasis in this example. Also note that the `rna::` part of the `path` argument is required only if the rna system is not the default system database.

**Example 2**

```bash
BTCL > subset define Subset "rna::*:*:hydrogens" "H*"
```

This example defines subsets from the rna system database at the monomer level with the name hydrogens. Each subset contains a list of all hydrogen atoms contained in that monomer. There are as many subsets defined at this level as there are valid levels, i.e., if there is only one model in the rna system with 30 monomers (residues), then 30 subsets at the monomer level are defined with the name hydrogens.

**Example 3**

```bash
BTCL > subset define allow_empty false Subset "rna::*:(1..5):nitrogens" "N*"
Error: subset cannot be created for all ATS path specs.
while executing
"subset define allow_empty false Subset "rna::*:(1..5):nitrogens" "N*"
"N*"
```

This example is similar to the previous one except that the subset level spans only the first five monomers and we are considering nitrogen atoms. If, for example, the third monomer did not contain any nitrogen atoms, only four subsets named nitrogen could be defined at the monomer level. Usually this would be fine, but, since the `allow_empty` flag is set to `false` (the default being `true`), an error is reported.

**Example 4**

```bash
BTCL > subset define Subset "rna::mySet" "*:mons, *:*:hydrogens, *:*:N*"
```

In this example, a subset called mySet is defined at the system level. The subset contains all the mons subsets at the molecule
A. Btcl Language and Commands—Standalone Mode - Continued

level, all the subsets named hydrogens at the monomer level, and a list of all the nitrogen atoms in the rna system.

Example 5

BTCL > subset define Subset "rna::*:mySet" "mons, *:hydrogens, *:N*"

This example is almost the same as the previous example, but the mySet subsets are defined at the molecule level for all molecules in the rna system. Note that this new definition does not replace the previous one since it is defined for a different level. Also note that a mySet subset could not be defined for a lower level, e.g., at the monomer level, since the mons subsets would exist at a higher level and could not be specified in the spec argument.

Example 6

BTCL > subset define Angle "rna::*:GLY_*:myAng" "{CA} {N} {HN}"

This example of the subset define command shows the definition of subsets named myAng at the monomer level for all the GLY type monomers which refer to rows in the Angle table. These rows are created by this command where, effectively, a set of new “angles” is defined between the CA, N, and HN atoms in the specified monomers. Note that the last argument contains 3 separate atom specs, contained between braces. This is because the Angle table contains three point columns to fill in when a new row is added. If the Table argument were Distance, the number of specs would be 2 and for Torsion it would be 4, etc. (refer to table above).

Example 7

BTCL > subset define Angle "rna::*:myAngAll" "(GLY_*:CA) (GLY_*:N) (GLY_*:HN)"

BTCL > subset define Subset "rna::*:myAngSub" "*:myAng"

These two examples of the subset define continue from the previous example to define new subsets called myAngAll and myAngSub. The definition of myAngAll differs from that of myAng in that, rather than having a subset for each matching monomer level, myAngAll defines one subset for each molecule that spans an identical set of angles. It would be specified by *:myAngAll
rather than *:*:myAng and would be more efficient if you always wanted to consider all the specified angles at once.

The definition of myAngSub creates a subset that contains all the myAng subsets. Effectively, this subset would be equivalent to myAngAll in usage and would be specified by *:myAngSub.

An alternative to the subset define Subset command above would be the following:

```
BTCL > subset define Angle "rna::*:myAngSub" "*:myAng"
```

The only difference between these commands is that the specified subset "*:myAng" must be an Angle subset. For example, the following would produce an error:

```
BTCL > subset define Angle "rna::*:myAngSub" "*:myAng, *:*:H*"
```

whereas the following would not:

```
BTCL > subset define Subset "rna::*:myAngSub" "*:myAng, *:*:H*"
```

**Example 8**

```
BTCL > subset get dbSub "rna::*:(1..4):hydrogens"
Object
```

This example shows the use of the subset get command. Here, the Btcl object dbSub contains a list of all the hydrogen atoms contained in the subsets called hydrogens at the monomer level for the first four monomers. The output from the command is Object, since these subsets refer only to items in the Atom table. (See Example 9.)

**Example 9**

```
BTCL > subset get dbSub "rna::(myset, *:*:myAng)"
Object Angle
```

This example shows that the subset get command can get lists of row numbers from more than one subset specification at a time. Here, the Btcl object dbSub contains all the rows in the myset subset at the system level, which contains all the rows in the mons and hydrogens sub-subsets, and all the nitrogen row numbers (see Example 4), plus all the angle rows in the myAng subset at the
A. Btcl Language and Commands—Standalone Mode - Continued

monomer level. Note that the output is Object Angle, since the rows refer to either the Atom or Angle tables. In fact, any item that does not belong to the Distance, Angle, Torsion, or outOfPlane tables is referred to as an Object. The output from the subset get command is very useful for filtering the items in the resulting object and/or for error trapping.

Example 10

```btcl
BTCL > select objList "*,:*"

BTCL > subset define Subset mol:1:list $objList
```

This example shows the definition of a single subset, called list, at the atom level for monomer 1 of molecule mol, using an object handle $objList as the definition of its contents. Note that $objList contains a list of all the molecules and all the monomers in the default system database, whereas usually the objects would have to be at the same hierarchy level as the subset being defined, i.e., at the atom level. This feature may be useful at times but should be used with care.

---

**subStructure**

**Purpose**

The `subStructure` command is used to return a list of all the atoms connected to a given atom which are not also bonded to a second given atom.

**Syntax**

```
subStructure varName atom1 ?atom2?
```

**Description**

In the above syntax, `atom1` and `atom2` are single atom specifications. If these atoms are directly bonded, then a list of atoms composed of `atom1` and all the atoms connected to `atom1`, which are not also connected to `atom2`, is returned to the object handle `varName`. If `atom2` is not given as an argument or is in a different molecule than `atom1`, then all the atoms in the molecule containing `atom1` are
subStructure

returned to object handle varName. If the two atoms specified are not in the same database, or are in the same molecule but not directly bonded, or are both members of the same ring, then the subStructure command reports an error.

Example 1

```
BTCL > select atom1 "ethanol:1:C1"
BTCL > select atom2 "ethanol:1:C2"
BTCL > subStructure list1 $atom1 $atom2
BTCL > subStructure list2 $atom2 $atom1
BTCL > database handle dbh ethanol
BTCL > $dbh get names1 Name $list1
BTCL > $dbh get names2 Name $list2
BTCL > object print $names1
  type: STRING
  items: 4
  elements: 1
  contents:
    "C1"
    "H4"
    "H5"
    "H6"

BTCL > object print $names2
  type: STRING
  items: 5
  elements: 1
  contents:
    "C2"
    "O3"
    "H7"
    "H8"
    "H9"
```

In this example, it is assumed that we have loaded a system database that contains an ethanol model. The first two select commands are used to create objects that refer to the two carbons in the ethanol model (atom1 and atom2). Two subStructure commands
are then executed to get the lists of atoms attached to both carbon atoms (list1 and list2). The following `database handle` command is used to create a database handle (dbh) to the ethanol model, which is then used with two database handle `get` operations to get the atom names corresponding to the atoms in list1 and list2, names1 and names2, respectively. Finally, the contents of the names list objects are printed out using the `object print` command.

Notice that atom C1 (atom1) is the methyl group carbon, whereas atom C2 (atom2) is the methoxy group carbon. The first command is a `database handle` command, which is used to create a handle ($dbh) to the ethanol model system.

**Example 2**

```bash
BTCL > $dbh get v1 Coord $atom1
BTCL > $dbh get v2 Coord $atom2
BTCL > vector vBond subtract $v1 $v2
BTCL > vector vBond multiply $vBond 0.5
BTCL > $dbh get coords Coord $list2
BTCL > vector coords add $coords $vBond
BTCL > $dbh set $coords Coord $list2
```

This example shows how lists of atoms created with the `subStructure` command can be used to alter model geometries. Here, the bond length of the C2–C1 bond is increased by 50% by adding half the bond vector to all the atom coordinates of the methyl group (containing the C1 atom). Continuing from the previous example, the atomic coordinates of the carbon atoms (atom1 and atom2) are extracted from the database using database handle `get` operations, as vectors v1 and v2. Two `vector` operations are then performed (a subtraction and a multiply by 0.5) to obtain half the bond vector ($vBond$) from atom2 to atom1. Another database handle `get` command is then performed to extract coordinates of the group of atoms at atom1, i.e., the methyl group list1. The half bond vector, vBond, is then added to the methyl atom coordinates using a vector `add` operation. Finally, the new coordinates for the methyl
group (list1) are set for the methyl group atoms using a database handle set operation.

vdWTailCorrection

Purpose

The vdWTailCorrection command is used in periodic systems to calculate contributions to the potential energy and pressure/stress arising from van der Waals interactions between atoms separated by distances larger than the nonbond cutoff.

Syntax

vdWTailCorrection cutoff_value
vdWTailCorrection cutoff_value -printonly
vdWTailCorrection off

Description

Using nonbond cutoffs in simulations of periodic systems containing liquids or amorphous solids results, for a typical system of 1000–2500 atoms, in errors in the potential energy and pressure of 50–200 kcal mol⁻¹ and 0.05–0.1 GPa (500–1000 atm.) respectively. For simulations such as constant-pressure dynamics to be effective when nonbond cutoffs are used, it is important to apply a correction for these neglected interactions. In the Discover program, this can be achieved using the vdWTailCorrection command.

The vdWTailCorrection command is used in either of two ways, depending on whether the tail corrections are to be incorporated into subsequent dynamics and minimization calculations.

When you want to apply the correction during simulations, you first issue the command:

```
BTCL > vdWTailCorrection cutoff_value
```

where cutoff_value is the value of the cutoff currently applied to the van der Waals interactions. This corresponds either to the value specified in the most recent invocation of the forcefield command or to the default value of 9.5 Å.
A. Btcl Language and Commands—Standalone Mode - Continued

After issuing the `vdWTailCorrection` command, all subsequent dynamics and minimization stages include the energy and pressure corrections. To revert to the original state, the `vdWTailCorrection` command is issued again in the following form:

`BTCL > vdWTailCorrection off`

When you simply want to obtain the values of the energy and pressure corrections based on the current system volume, without influencing subsequent dynamics or minimization, the syntax is:

`BTCL > vdWTailCorrection cutoff_value -printOnly`

The energy and pressure corrections are returned by the `vdWTailCorrection` command as strings, in the Discover program’s default units of kcal mol\(^{-1}\) of cells and GPa, respectively. The two values may then be passed to other Btcl procedures or to external programs.

**Example 1**

```
BTCL > set cutoff 8.5

BTCL > forcefield nonbond \ 
   +separate_coulomb \ 
   vdw \ 
   summation_method = atom_based \ 
   cutoff = $cutoff spline_width = 0 buffer_width = 0.5 \ 
   coulomb \ 
   summation_method = group_based \ 
   cutoff = $cutoff spline_width = 0 buffer_width = 0.5

BTCL > vdWTailCorrection $cutoff

BTCL > dynamics \ 
   time = 10000 timestep = 1.0 \ 
   initial_temperature = 298.2 +boltzmann \ 
   ensemble = nvt temperature_control_method = Andersen \ 
   integration_method = Velocity_verlet \ 
   cell_mass = 20 \ 
   temperature = 298.2 temperature_window = 10 \ 
   deviation = 5000
```
The vector command is a general-purpose command for the efficient creation, review, and manipulation of binary lists of scalars,
vectors, and matrices to be created and manipulated using a large range of functions.

**Syntax**

vector varName ?definition?
vector varName operation ?-keyword | -keyword varName2? arg1 ?arg2? ?arg3?

Note: If you are new to the vector command, it is most helpful to review the Background section before reading the descriptions of each operation.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Argument types (number)</th>
<th>Description of vector returned</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic operations:</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>add</strong></td>
<td>int or real (2)</td>
<td>arg1 + arg2 (per element).</td>
</tr>
<tr>
<td><strong>addMultiply</strong></td>
<td>real (3)</td>
<td>Return arg1 * arg2 * arg3.</td>
</tr>
<tr>
<td><strong>degree</strong></td>
<td>real (1)</td>
<td>Convert a vector of radians to degrees (scale elements by 180/π).</td>
</tr>
<tr>
<td><strong>divide</strong></td>
<td>int or real (2)</td>
<td>arg1 / arg2 (per element).</td>
</tr>
<tr>
<td><strong>increment</strong></td>
<td>real (1 or 2)</td>
<td>varName += arg1 (?arg2?).</td>
</tr>
<tr>
<td><strong>initialize</strong></td>
<td>?-keyword int or real (1, 2 or 3)</td>
<td>If only arg1 is given, return a vector or scalar containing the first arg1 positive integers. If arg2 is also given, return the arg1_th through the arg2_th positive integers. If arg3 is also given, return the arg1_th through the arg2_th positive integers in steps of arg3. If any of the arguments contains a decimal point, return the results as real.</td>
</tr>
<tr>
<td><strong>keyword</strong></td>
<td>sequential</td>
<td>Initialize a vector.</td>
</tr>
<tr>
<td><strong>prime</strong></td>
<td></td>
<td>If only arg1 is given, return a vector or scalar containing the first arg1 prime numbers. If arg2 is also given, return the arg1_th through the arg2_th prime numbers. If arg3 is also given, return the arg1_th through the arg2_th prime numbers in steps of arg3.</td>
</tr>
<tr>
<td><strong>gaussian</strong></td>
<td></td>
<td>Return arg1 number of normalized Gaussian points. Optional mean and standard deviation can be given in arg2 and arg3.</td>
</tr>
<tr>
<td><strong>random</strong></td>
<td></td>
<td>Same as the vector operation random.</td>
</tr>
<tr>
<td><strong>diagonalMatrix</strong></td>
<td></td>
<td>Same as the vector operation diagonalMatrix.</td>
</tr>
<tr>
<td><strong>replicate</strong></td>
<td></td>
<td>Same as the vector operation fill.</td>
</tr>
<tr>
<td><strong>modulus</strong></td>
<td>int or real (2)</td>
<td>Remainders of arg1 / arg2 (per element).</td>
</tr>
<tr>
<td>Operation</td>
<td>Argument types (number)</td>
<td>Description of vector returned</td>
</tr>
<tr>
<td>--------------------</td>
<td>----------------------------</td>
<td>-------------------------------------------------------------</td>
</tr>
<tr>
<td>multiply</td>
<td>int or real (2)</td>
<td>( \text{arg1 \times \text{arg2}} ) (per element).</td>
</tr>
<tr>
<td>multiplyIncrement</td>
<td>real (2)</td>
<td>( \text{varName} \text{+= arg1 \times \text{arg2}} ).</td>
</tr>
<tr>
<td>multiplyMultiply</td>
<td>real (3)</td>
<td>Return ( \text{arg1 \times arg2 \times \text{arg3}} ).</td>
</tr>
<tr>
<td>multiplyMultiplyIncrement</td>
<td>real (3)</td>
<td>( \text{varName} \text{+= arg1 \times \text{arg2 \times \text{arg3}} ).</td>
</tr>
<tr>
<td>negate</td>
<td>real (1)</td>
<td>( \text{varName} *= -1 ) (per element).</td>
</tr>
<tr>
<td>radian</td>
<td>real (1)</td>
<td>Convert a vector of degrees (scale elements by ( \pi/180 )).</td>
</tr>
<tr>
<td>reciprocal</td>
<td>real (1)</td>
<td>( \text{varName} = 1 / \text{arg1} ) (per element).</td>
</tr>
<tr>
<td>subtract</td>
<td>int or real (2)</td>
<td>( \text{arg1} - \text{arg2} ) (per element).</td>
</tr>
</tbody>
</table>

**Conversion operations:**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Argument types (number)</th>
<th>Description of vector returned</th>
</tr>
</thead>
<tbody>
<tr>
<td>abs</td>
<td>real (1)</td>
<td>Absolute values (per element).</td>
</tr>
<tr>
<td>boolean</td>
<td>int or real (1)</td>
<td>A vector of Booleans of (char) 0 or 1 corresponding to the input elements that are 0 or not 0.</td>
</tr>
<tr>
<td>ceiling</td>
<td>real (1)</td>
<td>Smallest integers not less than the vector elements (returned as reals).</td>
</tr>
<tr>
<td>real</td>
<td>Boolean or int (1)</td>
<td>A vector of reals of (double) 0 or 1 corresponding to the input elements that are 0 or not 0.</td>
</tr>
<tr>
<td>floor</td>
<td>real (1)</td>
<td>Largest integers not greater than the vector elements (returned as reals).</td>
</tr>
<tr>
<td>integer</td>
<td>Boolean or real (1)</td>
<td>Return a vector of integers of 0 or 1 for a Boolean vector input and a vector of integers (truncated) for a real vector input.</td>
</tr>
<tr>
<td>pack</td>
<td>real (1)</td>
<td>Return a symmetric matrix in packed form if the input is a symmetric matrix in unpacked form; return a copy of the input if it is a symmetric matrix in packed form.</td>
</tr>
<tr>
<td>unpack</td>
<td>real (1)</td>
<td>Return a symmetric matrix in unpacked form if the input is a symmetric matrix in packed form; return a copy of the input if it is a symmetric matrix in unpacked form.</td>
</tr>
</tbody>
</table>

**Geometric operations:**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Argument types (number)</th>
<th>Description of vector returned</th>
</tr>
</thead>
<tbody>
<tr>
<td>xRotate</td>
<td>real (2)</td>
<td>If ( \text{arg2} ) is a matrix (3 x 3 or 4 x 4), then a rotation of ( \text{arg1} ) is performed on ( \text{arg2} ) about the x-axis. If ( \text{arg1} ) is a matrix and ( \text{arg2} ) the angle, then a post-multiply of the rotation matrix is performed.</td>
</tr>
<tr>
<td>yRotate</td>
<td>real (2)</td>
<td>As for xRotate, but about the y-axis.</td>
</tr>
<tr>
<td>zRotate</td>
<td>real (2)</td>
<td>As for xRotate, but about the z-axis.</td>
</tr>
<tr>
<td>threeDot</td>
<td>real (2)</td>
<td>Dot product of a 3 x 1 vector and a 4 x 4 matrix. Special case—the fourth row and column of the matrix are ignored.</td>
</tr>
</tbody>
</table>
### A. Btcl Language and Commands—Standalone Mode - Continued

<table>
<thead>
<tr>
<th>Operation</th>
<th>Argument types (number)</th>
<th>Description of vector returned</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>translate</strong></td>
<td>real (2)</td>
<td>Translation by a 3 x 1 vector, ( \text{arg}1 ) or ( \text{arg}2 ), of the 4 x 4 matrix, ( \text{arg}2 ) or ( \text{arg}1 ).</td>
</tr>
</tbody>
</table>

**Logical operations:**
- **not** | Boolean (1) | Logical negation of \( \text{arg}1 \) (per element). |
- **and** | Boolean (2) | Logical “and” operation of \( \text{arg}1 \) and \( \text{arg}2 \) (per element). |
- **or** | Boolean (2) | Logical “or” operation of \( \text{arg}1 \) and \( \text{arg}2 \) (per element). |
- **xor** | Boolean (2) | Logical “exclusive or" operation of \( \text{arg}1 \) and \( \text{arg}2 \) (per element). |
- **eq** | int or real (2) | Logical 1 if \( \text{arg}1 = \text{arg}2 \), else 0 (per element). |
- **ne** | int or real (2) | Logical 1 if \( \text{arg}1 \neq \text{arg}2 \), else 0 (per element). |
- **gt** | int or real (2) | Logical 1 if \( \text{arg}1 > \text{arg}2 \), else 0 (per element). |
- **lt** | int or real (2) | Logical 1 if \( \text{arg}1 < \text{arg}2 \), else 0 (per element). |
- **ge** | int or real (2) | Logical 1 if \( \text{arg}1 \geq \text{arg}2 \), else 0 (per element). |
- **le** | int or real (2) | Logical 1 if \( \text{arg}1 \leq \text{arg}2 \), else 0 (per element). |
- **?** | Boolean (1); Boolean, int or real (1 or 2) | \( \text{arg}1 \) ? \( \text{arg}2 \) : (\( \text{arg}3 \) ? \( \text{arg}3 \) : 0) (per element). |

**Statistical operations:**
- **average ?-keyword?** | real (1 or 2) | (1) If only \( \text{arg}1 \) is provided, return average of the elements. (2) Use \( \text{arg}2 \) as a set of weights to the elements of \( \text{arg}1 \) and return the weighted average.

\[
\text{average} = \frac{\sum_{i=1}^{\text{arg}_1} \text{arg}_{1i} \cdot \text{arg}_{2i}}{\sum_{i=1}^{\text{arg}_2} \text{arg}_{2i}}.
\]

If \text{bygroup} is specified as a keyword, return the averages within the groups (see the \text{object} command for more information on grouping).
- **count** | Boolean, int or real (1) | The number of items (i.e., scalars, vectors, or matrices) in \( \text{arg}1 \). |
- **min ?-keyword?** | int or real (1) | Minimum values for each corresponding element in the input vector items. \text{keyword} can be \text{index} or \text{bygroup}. If \text{bygroup} is specified, the vector returned contains the minimum elements within each group (see the \text{object} command on how groups are defined). If \text{index} is specified, the vector returned is an integer vector of the indices of the minimum elements (rather than the minimum elements themselves). |
<table>
<thead>
<tr>
<th>Operation</th>
<th>Argument types (number)</th>
<th>Description of vector returned</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>max</strong>? -keyword?</td>
<td>int or real (1)</td>
<td>Maximum values for each corresponding element in the input vector items. <strong>keyword</strong> can be <strong>index</strong> or <strong>bygroup</strong>. If <strong>bygroup</strong> is specified, the vector returned contains the maximum elements within each group (see the <strong>object</strong> command on how groups are defined). If <strong>index</strong> is specified, the vector returned is an integer vector of the indices of the maximum elements (rather than the maximum elements themselves).</td>
</tr>
<tr>
<td><strong>product</strong>? -keyword?</td>
<td>int or real (1 or 2)</td>
<td>(1) If only <strong>arg1</strong> provided, return the product of the items in <strong>arg1</strong>. (2) Use <strong>arg2</strong> as a set of weights to the elements of <strong>arg1</strong> and return the weighted product = ( \prod_{i} arg1_{i}^{arg2_{i}} ). If <strong>bygroup</strong> is specified as a keyword, return the products within the groups (see the <strong>object</strong> command for more information on grouping). The <strong>bygroup</strong> option does not support the input of optional weights.</td>
</tr>
<tr>
<td><strong>random</strong></td>
<td>int or real (1, 2 or 3)</td>
<td>Vector of random numbers (between 0 and 1). The arguments are scalar integer values and the returned vector contains <strong>arg1</strong> random scalars, or <strong>arg1</strong> random unit vectors of dimension ((arg2 * 1)) if <strong>arg2</strong> is supplied, or <strong>arg1</strong> matrices of dimension ((arg2 * arg3)) of random numbers.</td>
</tr>
<tr>
<td><strong>standardDeviation</strong>? -keyword?</td>
<td>real (1)</td>
<td>Standard deviations for each element over all the items ( n ) in <strong>arg1</strong>, i.e., ( \sigma_{m} = \sqrt{\frac{1}{n(n-1)} \sum_{i=1}^{n} (arg1_{i} - arg1\overline{m})^{2}} ). If <strong>bygroup</strong> is specified as a keyword, return the averages within the groups (see the <strong>object</strong> command for more information on grouping).</td>
</tr>
<tr>
<td><strong>sum</strong>? -keyword?</td>
<td>int or real (1 or 2)</td>
<td>(1) If only <strong>arg1</strong> is provided, return the sum of the items in <strong>arg1</strong>. (2) Use <strong>arg2</strong> as a set of weights to the elements of <strong>arg1</strong> and return the weighted sum = ( \sum_{i} arg1_{i}^{arg2_{i}} ). If <strong>bygroup</strong> is specified as a keyword, perform the summation within the groups (see the <strong>object</strong> command for more information on grouping). The <strong>bygroup</strong> option does not support the input of optional weights.</td>
</tr>
</tbody>
</table>

**Transcendental and algebraic operations:**

| **acos**      | real (1)          | Arc cosines (in radians) (per element). |
### A. Btcl Language and Commands—Standalone Mode - Continued

<table>
<thead>
<tr>
<th>Operation</th>
<th>Argument types (number)</th>
<th>Description of vector returned</th>
</tr>
</thead>
<tbody>
<tr>
<td>asin</td>
<td>real (1)</td>
<td>Arcsines (in radians) (per element).</td>
</tr>
<tr>
<td>atan</td>
<td>real (1)</td>
<td>Arctangents (in radians) (per element).</td>
</tr>
<tr>
<td>atan2</td>
<td>real (2)</td>
<td>Arctangents (in radians) as results of the C function, atan2(arg1, arg2) (per element).</td>
</tr>
<tr>
<td>cos</td>
<td>real (1)</td>
<td>Cosines (in radians) (per element).</td>
</tr>
<tr>
<td>exp</td>
<td>real (1)</td>
<td>Exponentials (base e) (per element).</td>
</tr>
<tr>
<td>log</td>
<td>real (1)</td>
<td>Logarithms (base e) (per element).</td>
</tr>
<tr>
<td>log10</td>
<td>real (1)</td>
<td>Logarithms (base 10) (per element).</td>
</tr>
<tr>
<td>sin</td>
<td>real (1)</td>
<td>Sines (in radians) (per element).</td>
</tr>
<tr>
<td>tan</td>
<td>real (1)</td>
<td>Tangents (in radians) (per element).</td>
</tr>
<tr>
<td>power</td>
<td>real (2)</td>
<td>arg1 raised to the power of arg2. Results of the C function pow(arg1, arg2) (per element).</td>
</tr>
<tr>
<td>sqrt</td>
<td>real (1)</td>
<td>Square roots (per element).</td>
</tr>
</tbody>
</table>

**Vector and matrix operations:**

- **cross**
  - real (2)
  - Cross product of two 3 x 1 vectors.

- **diagonalize ?-eigen vectors varName2?**
  - real (1)
  - Eigenvalues and optional eigenvectors, are returned to object varName2.

- **diagonalMatrix**
  - int (1) ?real (1)?
  - Single-banded diagonal matrix of dimension arg1. If vector is arg2, this is used to fill in the elements along the diagonal, otherwise zeros are used.

- **dot**
  - real (2)
  - Dot product for vector * vector, vector * matrix, matrix * vector, or matrix * matrix. Also supports the special case where the vector is 3 x 1 and the matrix is 4 x 4.

- **norm**
  - real (1 or 2)
  - Return the p-norm of a vector arg1 where p is given by arg2; returns the Euclidean norm if arg2 is not specified.

- **fill**
  - Boolean, int, or real (1); ?int or real (1)?
  - Vector composed of arg2 copies of vector arg1 if arg2 is specified, else returns arg1.

- **identity**
  - int (1)
  - Identity matrix with dimension arg1. Equivalent to operation diagonalMatrix arg1 1.

- **index**
  - Replaced by objectrange command (page 217).

- **inverse**
  - real (1)
  - Inverse of arg1.

- **linearEquationSolver**
  - real (2)
  - Solve a system of simultaneous, inhomogeneous linear equations with the matrix in arg1 and the inhomogeneous terms as a vector arg2.

- **normalize**
  - real (1)
  - Normalized vector. Does not support matrices.

- **outer**
  - real (2)
  - Outer product of two vectors.
Btcl vector variables refer to data organized as lists of scalars, vectors, or matrices composed of Boolean, integer, or real values. Where the term “number of items” is used, it refers to how many scalars, vectors, or matrices are listed in the vector object. The term “elements” refers to the individual values within an item. In general, a vector object can be thought of as having three dimensions $n, i, j$, where $n$ is the number of items, $i$ is the number of rows, and $j$ the number of columns. The total number of elements is thus $n \times i \times j$. Hence, a collection of scalars has dimensions $n, 1, 1$; a collection of vectors, $n, i, 1$; and a collection of matrices, $n, i, j$.

The definition of some operations depends on the data stored. Where the term “per element” is used, the operation applies to corresponding individual elements in each vector object argument, irrespective of the underlying organization of the list as scalars, vectors, or matrices.

For some operations (such as the binary arithmetic operations add, subtract, multiply, and divide), the vector command supports input of vector objects with different dimensions. For example, in general, if $\text{arg1}$ is of dimensions $n, i, j$, and $\text{arg2}$ of dimensions $n', i', j'$,
the add operation accepts these arguments if $n'$ equals $n$, or $n$ equals 1, or $n'$ equals 1, and the resulting vector has the maximum of $n$ and $n'$ as its item dimension. In other words, mapping between 1 and $n$ or between $n'$ and $n$ items is allowed. The same is true for the row and column dimensions. The vector command expands the argument that has the smaller value in any one of its dimensions accordingly, to match the corresponding larger dimensions. Please note that arguments with dimensions like 2,1,1 and 1,3,4 are allowed in this scheme.

Unlike ordinary Tcl variables, which can contain lists of numbers or lists of lists, etc., as strings, Btcl vector variables actually contain a string code, which is used internally by Btcl to refer to arrays of numbers stored in memory. The Btcl vector command allows access to and manipulation of the values referred to by a Btcl vector variable. In general, a vector variable may refer to a great number of values—manipulating these values with the vector command instead of standard Tcl strings provides very fast math operations on collections of scalars, vectors, or matrices.

A vector variable can be created only with the vector command but can be deleted using the Btcl unset command or by reassigning the variable name using a set or another vector command. When this occurs, the internal data associated with that vector object are also deleted. It is also possible to copy the value of a vector variable, i.e., the string code used to refer to the actual data, into an ordinary Tcl variable and use this variable in a vector command as if it were the actual vector variable. However, deleting or reassigning the Tcl variable has no effect on the original vector variable, and, conversely, deleting the original vector variable makes the string code stored in the Tcl variable invalid as a reference to internal values. This distinction is made clearer in the description and Example 4 below.

Description

The vector command takes positional arguments rather than named parameter arguments. It acts like the set command in that, to create a vector variable $v$, the syntax would be:

```
BTCL > vector v {1 2 3}
```

or

```
where, in this case, \texttt{v} is set to a string code which refers internally to a list of three scalars. To find out what a vector variable \texttt{v} holds, the syntax is:

\begin{verbatim}
BTCL > vector v
1.0 2.0 3.0
\end{verbatim}

In general, a vector variable might refer to a great many values and so, unlike the \texttt{set} command, the data values just stored are not echoed back to you in interactive mode. Lists of scalars, vectors, or matrices are created, depending on the number of braces enclosing each sublist. The following are valid vector definitions:

\begin{verbatim}
BTCL > vector v1 "{1 0 0}" ; vector v1
{1.0 0.0 0.0}
BTCL > vector v2 "{1 0 0} {0 1 0} {0 0 1}" ; vector v2
{1.0 0.0 0.0} {0.0 1.0 0.0} {0.0 0.0 1.0}
BTCL > vector v3 "{1} {2} {3}" ; vector v3
1.0 2.0 3.0
\end{verbatim}

\texttt{v1} refers to a list of length 1 containing a vector of length 3; \texttt{v2} refers to a list of length 3 containing vectors of length 3; \texttt{v3} refers to a list of length 3 containing scalars. Note that, in the last example, a list of 1D vectors is considered to be a list of scalars. The following are valid matrix definitions:

\begin{verbatim}
BTCL > vector m1 "{\{1 0 0\} {0 1 0} {0 0 1\}}" ; vector m1
{{1.0 0.0 0.0} {0.0 1.0 0.0} {0.0 0.0 1.0}}
BTCL > vector m2 "{\{1 0\} \{0 1\}} \{\{0 1\} \{-1 0\}\}" ; vector m2
{{1.0 0.0} {0.0 1.0}} {{0.0 1.0} {-1.0 0.0}}
BTCL > vector m3 "{\{1\} \{2\} \{3\}}" ; vector m3
{{1.0}} {{2.0}} {{3.0}}
\end{verbatim}

\texttt{m1} refers to a list of length 1 containing a 3 x 3 matrix; \texttt{m2} refers to a list of length 2 containing 2 x 2 matrices; \texttt{m3} refers to a list of length 3 containing 1 x 1 matrices. Note that, in this last example, single-element matrices are still considered to be matrices. This is because, under certain operations, they can act as scaling matrices. When you create lists of vectors or matrices, all elements of the list
must have exactly the same dimensions (as in the above examples). However, there is one special matrix definition that allows you to specify a symmetric matrix, for example:

```
BTCL > vector sym "{{1} {0 1} {0 0 1} {0 0 0 1}}" ; vector sym
       {{1.0} {0.0 1.0} {0.0 0.0 1.0} {0.0 0.0 0.0 1.0}}
```

The output of `vector varName` is suitable for defining a new vector:

```
BTCL > vector v1 {{1 0 0}}

BTCL > vector v2 [vector v1]; vector v1
       {1.0 0.0 0.0}
```

The contents of a normal Tcl variable can also be used to create/specify the contents of a Btcl vector variable.

```
BTCL > set a {{4 5 6}}
       {4 5 6}

BTCL > vector v4 $a ; vector v4
       {4.0 5.0 6.0}
```

However, when copying vectors, it is most efficient to use the `vector` command:

```
BTCL > vector v5 $v4 ; vector v5
       {4.0 5.0 6.0}
```

The Btcl `vector` command also takes several operations (listed in the table above), which themselves take arguments that are either string definitions or vector variables of compatible dimensions. Refer to the examples below.

### Example 1

```
BTCL > vector v1 {{1 2 3}}

BTCL > vector v2 add $v1 "(4 5 6)"

BTCL > vector v2
       {5.0 7.0 9.0}
```

In this example, the first line performs an assignment of the vector variable `v1` from a string definition. The second command line creates a vector `v2` from the vector sum of vector `v1` and a string def-
inition. The third command line uses the vector command to echo the value of the vector variable v2.

Example 2

```
BTCL > vector v1 "\{1 1 1\} \{1 1 1\}"
BTCL > vector v2 "\{2 3 4\} \{4 5 6\}"
BTCL > vector v3 dot $v1 $v2
BTCL > vector v3
9.0 15.0
```

In this example, vector variables v1 and v2 are both assigned as lists containing two vectors. Vector variable v3 is then assigned as the dot product of these two vectors. Notice that v3 is a list of two scalars corresponding to two results.

Example 3

```
BTCL > vector sym "\{(1) (1 2) (1 2 3)\}"
BTCL > vector eigen diagonalize -eigenvectors eigenVec $sym
BTCL > echo eigen values = [vector eigen], eigen vectors =
\n[vector eigenVec]
eigen values = \{0.307979 0.643104 5.04892\},
eigen vectors =
\{(0.591009 -0.736976 0.327985) (0.736976 0.327985 -0.591009) (0.327985 0.591009 0.736976)\}
```

In this example, a symmetric matrix vector variable called sym is first created. It is then diagonalized using the vector diagonalize operation to assign the eigenvectors to a vector variable called eigen. This operation may also take an optional keyword eigenvectors, which specifies that the next argument is a vector variable name to which the eigenvectors are to be assigned. Currently, the diagonalize operation is the only vector operation that can take a named parameter in this manner. Notice that the eigenvectors are returned as a matrix.

Example 4: Vector variables in procedures and functions

Because of the way Btcl vectors are pointers to actual vector data, you must be careful when defining Tc1 procedures. Below is an example of a Tc1 procedure taking vector variable arguments.
In this example, the vector arguments are passed like any ordinary procedure arguments. The result of the function in this case is returned as a string. Because vector variable $c$ is created locally, the “vector pointer” $c$ would not be valid outside this procedure, since local variables are deallocated at the end of procedures. For this example, passing the string value back is acceptable, since in most cases you would be expecting a single value back from the procedure. In general, it is a bad idea to pass around the string values of vector variables, since they may be very long, and the process of string conversion may introduce some inaccuracy (if tcl_precision is not set to 17).

To create a procedure that returns a general vector result, i.e., a vector function, it is far better to pass a variable name for the result, as in the following example:

```
BTCL > proc v_add {sum v1 v2} {
    upvar 1 $sum c
    vector c add $v1 $v2
}

BTCL > v_add x "1 2 3" "0 1 2"

BTCL > vector x
1.0 3.0 5.0
```

Here, we have used the advanced Tcl command upvar which tells the procedure to associate the local variable $c$ with the variable $sum$ in the context of the calling code (in this case at the global level). In this example, setting the variable $c$ inside the procedure is equivalent to setting the variable $x$ outside the procedure.
vibrationalAnalysis

Purpose

The vibrationalAnalysis command is used to perform normal mode analysis on the molecules in the current system, to selectively obtain the mass-weighted normal modes of vibration and the corresponding zero-point vibrational frequencies and their intensities.

Syntax

vibrationalAnalysis ?path? ?keyword = value?...

<table>
<thead>
<tr>
<th>Path/keyword</th>
<th>Values</th>
<th>Default</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>normal_modes</td>
<td>Boolean</td>
<td>false</td>
<td>Define whether the calculated normal modes (eigenvectors) are to be output.</td>
</tr>
<tr>
<td>mode_intensities</td>
<td>Boolean</td>
<td>true</td>
<td>Define whether the vibrational frequency intensities are to be output.</td>
</tr>
<tr>
<td>output</td>
<td>Boolean</td>
<td>false</td>
<td>Path to output control parameters.</td>
</tr>
<tr>
<td>print</td>
<td>Boolean</td>
<td>false</td>
<td>Define whether output is to be directed to the output stream.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>If no other output mode is defined, this defaults to true.</td>
</tr>
<tr>
<td>hessian</td>
<td>Boolean</td>
<td>false</td>
<td>Define whether output is to be written to a Hessian file.</td>
</tr>
<tr>
<td>file_name</td>
<td>string</td>
<td>—</td>
<td>Define the name of the Hessian file for output.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>If no value is given, the default is run_name.hessian.</td>
</tr>
<tr>
<td>modes</td>
<td>string</td>
<td>—</td>
<td>If supplied, the value is the name of a Btcl vector variable to which the normal modes are to be output.</td>
</tr>
<tr>
<td>frequencies</td>
<td>string</td>
<td>—</td>
<td>If supplied, the value is the name of a Btcl vector variable to which the vibrational frequencies are to be output.</td>
</tr>
<tr>
<td>intensities</td>
<td>string</td>
<td>—</td>
<td>If supplied, the value is the name of a Btcl vector variable to which the vibrational frequency intensities are to be output.</td>
</tr>
</tbody>
</table>

Description

The vibrationalAnalysis command is used to find the internal vibrational modes and frequencies of a molecule. The calculation is performed by diagonalizing the Hessian matrix generated from
the second derivatives of the internal potential energy of the molecule.

The first six frequencies and modes (five for linear models) represent the translation and rotation of the model as a whole and should have frequencies of approximately zero. If they do not, then the original model was probably not properly minimized with respect to the current system forcefield, and all results will be dubious.

The basic output from the normal mode calculation is the vibrational frequencies and their intensities, although you can select to also output the normal modes or not to output the intensities.

The output can be written to the output stream (usually run_name.out), to a Hessian file, or directly to Btcl variables for subsequent analysis. If the output goes to the output stream, the vibrational free energy and its components are also generated and output.

Example 1

vibrationalAnalysis

If the vibrationalAnalysis command is configured with its default parameter values (listed in the table above), this causes a normal mode analysis calculation to be performed on the currently selected model(s). Vibrational frequencies, their intensities, and free energy results are written to the output stream.

Example 2

vibrationalAnalysis +normal_modes output +print +hessian

   filename = vib

This example causes a vibrational (normal mode) analysis to be performed with frequencies, intensities, and normal modes being written both to the output file and to a Hessian file called vib.hessian.

Example 3

vibrationalAnalysis +normal_modes -mode_int output

   frequencies = freqs modes = modes
This example causes a vibrational analysis to be performed on the current system, with the results being written to Btc1 variables named freqs for the vibrational frequencies and modes for the vibrational normal modes. Note that freqs is a vector and modes is a matrix composed of the normal mode vectors.

writeFile

Purpose

The writeFile command is used to write a data file so that it can be read by subsequent Discover, Cerius², or Insight runs (e.g., a coordinate snapshot to an archive file).

Syntax

writeFile filetype ?keyword = value?...

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Values</th>
<th>Default</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>filetype</td>
<td>molecular_system Archive coordinate or coordinate frame restart forcefield</td>
<td>molecular_system</td>
<td>Specify the type of file(s) to write data to. One of these values must be the first argument after writeFile, since the filetype is specified by a positional argument.</td>
</tr>
<tr>
<td>binary</td>
<td>Boolean</td>
<td>false</td>
<td>Write the binary form of the file. Currently available only for the forcefield filetype (which must be binary).</td>
</tr>
<tr>
<td>filename</td>
<td>string</td>
<td>run_name.extension</td>
<td>Specify name of file to write to. Default is run_name.extension, where extension is the extension appropriate to the file (see below).</td>
</tr>
<tr>
<td>frame</td>
<td>integer</td>
<td>1</td>
<td>For an archive file, specify which frame to write. If the value is zero, writeFile appends to the end of the archive file.</td>
</tr>
</tbody>
</table>
A. Bcl Language and Commands—Standalone Mode - Continued

**Description**

The `writeFile` command writes data to a particular file. If the `filename` parameter is not specified, the filenames used depend on the `filetype` parameter and are as follows:

- **archive**
  - `run_name.arc`

- **coordinate**
  - `run_name.car`

- **molecular_system**
  - `run_name.car` and `run_name.mdf`

- **dynamics_restart**
  - `run_name.xdyn`

(In the above file types, `run_name` is your name for the calculation.)

- **forcefield**
  - `$FORCEFIELD.xfrc` (with `binary` option), where `$FORCEFIELD` is the value of the `$FORCEFIELD` environment variable.

**Note**—If the `filename` parameter is given without an extension, the appropriate extension is added.

The `dynamics_restart` file contains data needed to restart a dynamics run.

**Example 1**

```
writeFile archive frame = 3
```

Write the third frame of the archive file `run_name.arc`.

**Example 2**

```
writeFile molecular_system filename = test
```

Write the structural data and coordinate files `test.mdf` and `test.car`.
Example 3

writeFile forcefield +binary filename = cff91

Write the binary forcefield file cff91.xfrc. The binary file is read faster by Discover.

Example 4

writeFile car file = test1

Write the coordinate file test1.car.

Example 5

writeFile dynamics_restart filename = crn.xdyn

Write the dynamics restart file crn.xdyn.
B

Databases and Tables

Introduction to databases and tables

Introduction

What you most need to know about CDiscover tables can be summarized in two points:

♦ All the data representing atoms, bonds, models, and so on, are stored in tables: a separate table for each kind of item to be described. For example, each atom is one row of the Atom table, and each characteristic of atoms is a column of the Atom table: atom type, formal charge, Cartesian coordinate, and so on.

♦ There are straightforward Btcl script commands—the same commands for all tables—that enable you to print, retrieve, search, or change the information stored in the tables.

Why should you care that CDiscover stores its data in tables?

Traditionally, software:

♦ Stores its data interwoven with the program.

♦ Can operate on data only in the ways that are programmed in.

♦ Requires writing of new C or C++ code to do anything new or different.

By contrast, CDiscover:

♦ Stores its data separately from the program, not interwoven with it.

♦ Provides a comprehensive script language for printing, extracting, selecting, or storing any data in a table.

♦ Enables you to analyze your data or control the simulation in new ways by retrieving or modifying the data in tables.
To illustrate, here are a few examples of tables and script commands. Below is part of the *Atom* table for a simple model, a trimer of alanine, cysteine, and glycine, with hydrogens deleted to make the table shorter for the example. Only four of the columns are shown (the first column is the table row number of each atom).

```
Atom
{
  Row Name    Monomer  Charge  Coord
  --- ---- ----- ------- ------------------------
  0) N      0 -0.5000  { 0.039 -0.028  0.000 }
  1) CA     0 0.1200  { 1.499 -0.043  0.000 }
  2) C      0 0.3800  { 2.055  1.361  0.000 }
  3) O      0 -0.3800  { 1.321  2.356  0.011 }
  4) CB     0 -0.3000  { 3.167  1.462 -1.217 }
  5) N      1 -0.5000  { 6.075  1.462  0.000 }
  6) CA     1 0.1200  { 4.043  2.755  0.013 }
  7) C      1 0.3800  { 5.543  2.582  0.010 }
  8) O      1 -0.3800  { 6.075  1.462 -0.002 }
  9) CB     1 -0.3000  { 3.346  3.556  1.233 }
 10) SG     1 0.0000  { 4.209  5.237  1.208 }
 11) N      2 -0.5000  { 7.712  3.675  0.020 }
 12) CA     2 -0.1000  { 8.268  5.079  0.035 }
 13) C      2 0.3800  { 7.540  6.067  0.046 }
 14) O      2 -0.3800  }
}
```

This listing of the *Atom* table is produced by the Btcl script command:

```
$mol print Atom.(Name,Monomer,Charge,Coord)
```

All the table commands are prefixed by a previously defined database handle to a collection (database) of interrelated tables; here, $mol specifies the collection that contains the *Atom* table. The database handle print command lists information from all columns of a table (when only the table name is given), from a single column (table and column names joined by a period are given), or from selected columns (table name, period, parenthesis, and comma-separated list of column names). An optional array variable can be used at the end to specify particular rows of the table; when it is not present all rows of the table are printed.

Note the different types of data that columns can hold: *Name* is a string, *Charge* is a floating-point number, and the (Cartesian) *Coord* is a triplet of floating-point numbers in a single column, but two dimensional: each entry is a three-element array, as signified by {} braces.
Introduction to databases and tables

To retrieve data from a table into a Btcl array variable, you simply use a command like:

```
$mol get coord Atom.Coord
```

The database handle `get` command creates a new array variable (third word in the command) and retrieves information from a column of a table (fourth word, table and column names joined by a period). An optional fifth word can be used to specify particular rows of the table; when it is not present the entire column (that is, all rows of it) are retrieved.

Now look at the `Monomer` column of the `Atom` table. To take advantage of well researched and powerful relational database techniques, we express many-to-one relationships such as atoms in a monomer by storing the row number of the “one” in a column of the “many” table. Thus, there is a column in the `Atom` table to store which monomer each atom is part of, but the `Monomer` table does not mention atoms:

```
Monomer {
    Row  Molecule  Number  Type
    ---  --------  ------  ----
    0)         0  1       ALAn
    1)         0  1B      CYSH
    2)         0  1C      GLYN
}
```

It may seem backwards initially, but you’ll find that the Btcl script language makes it easy to look at relationships from either point of view, whether you want to know which monomer an atom is in or to find all the atoms of a monomer. For example:

```
$mol select 1 Atom.Monomer atomList
```

creates a new array variable called `atomList`, which contains `{ 5 6 7 8 9 10 }`—these are the row numbers (in the `Atom` table) of all the atoms in monomer 1.

By the same many-to-one principle, the `Bond` table stores the two atoms that make up each bond. Atoms can have one or more bonds, but each bond has exactly one atom at each end:
B. Databases and Tables

Bond
{

Row    Atom-1   Atom-2   Order     Bibond
---  ------  ------  -------  -----
0)       0       1   1.0000       1
1)       1       0   1.0000       0
2)       1       2   1.0000       4
3)       1       4   1.0000       8
4)       2       1   1.0000       2
5)       2       3   2.0000       7
6)       2       5   1.5000      10
...

Each bond is stored twice, in both directions, to make it possible to find all the bonds of an atom by selecting only the Atom-1 column. For example, to get all the bonds of atom 2:

$mol select 2 Bond.Atom-1 bondList

This command creates the variable bondList, which contains { 4 5 6 }—the row numbers in the Bond table. We can use the result of a database handle select or get command as a row-list specification for a chained get. For example, to get the names of the atoms bonded to atom 2:

$mol get bondedAtom Bond.Atom-2 $bondList
$mol get bondedName Atom.Name   $bondedAtom

These commands create the variables bondedAtom, which contains { 1 3 5 }, and bondedName, which contains { CA O N }.

Lastly, note that the Bond row number of the complementary bond is stored in the last column of the Bond table itself. This is an example of rows of a table being related to each other. Same-table relationships can be one-to-one as they are here, or many-to-one, such as might be used to express a derivation or inheritance hierarchy of rows within a single table.

Data types for columns

The allowable data types for table columns (object attributes) are listed in the BTCL object types table on page 218.
Special notes:
In addition to the scalar data types (data types with one element), vector data types, which are collections of \( n \) values with the same type, can be used as the data type of a column. The most common example of this is Cartesian coordinates, where each row entry in a coordinates column is really a 3-vector of doubles.

The \texttt{rid} data type is specific to the database itself and has no meaning outside a database. A \texttt{rid} is a reference to an object somewhere in the database. That is, it refers to one row in one table in the database. Which table and row this is, is encoded in the value of the \texttt{rid}. This information is used by the database and script language to resolve references across tables. A \texttt{rid} is a global reference within a single database, but it cannot be used to refer to rows (objects) in another database.

The \texttt{OBJ\_ARRAY} data type is an \( n \)-vector of values of some type. In fact, coordinates, which are described as 3-vectors of doubles, are \texttt{OBJ\_ARRAY}s of type double and length three. The term \texttt{OBJ\_ARRAY} is used because the type of an \texttt{OBJ\_ARRAY} can also be \texttt{rid}, which means that a valid type for a column in a table is a list or array of objects (rows) in the database.

---

**System database**

**Introduction**

A system database is created whenever a \texttt{begin}, \texttt{reset}, or \texttt{readFile} command is encountered in a Btcl script (.inp file). This database is a collection of tables containing information describing a model system. Unlike general-purpose databases, a Discover database is stored in memory rather than on a peripheral device. This is essential for performance reasons.

The purpose of a system database is to provide you with access to important system data. Any quantity stored in the database can be recovered via Btcl commands and manipulated or output as desired. For example, you might want to change the values of atom coordinates or add columns or tables for your own data.
B. Databases and Tables

System databases are deleted automatically when the Discover program is finished running or when a reset or begin command is issued.

The Discover program maintains a table of all databases known to it, and another table of “current” databases of every type (where the current database of a given type is the one to be used by default when there is not an explicit specification). A system database is of type System in the CurrentDatabase table. A system created with a begin becomes the current system database, but a system created with a readFile does not.

Contents of the system database common to all systems

These tables are always created in a system database, regardless of periodicity or symmetry considerations:

Atom
Monomer
Molecule
NonbondGroup
Bond

The names Atom and Bond are not the actual names of the tables in question. The real names are MainCell/Atom and MainCell/Bond. However, Atom and Bond are aliases and can be used as if they were the actual names. The aliases are used in the remainder of this discussion.

Atom table

The Atom table always has the following twelve columns, regardless of periodicity or symmetry considerations:
In addition, an optional *Velocity* column, containing velocities that are read from history files, may be present. This column is created when a `readFile history` command (page 255) is issued.

A row of the *Atom* table corresponds to an atom.

The entry in the *Chirality* column has the following interpretation:

- 0 = neither chiral nor prochiral
- 1 = prochiral, lightest (or lowest priority) bonded atoms are equivalent
- 2 = prochiral, intermediate size (or priority) bonded atoms are equivalent
- 3 = prochiral, heaviest (or highest priority) bonded atoms are equivalent
- 4 = chiral
- 8 = not determined
- 9 = unable to determine

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Monomer</td>
<td>rid(^a)</td>
<td>Specifies the monomer to which the atom belongs. It is a reference to the Monomer table.</td>
</tr>
<tr>
<td>Name</td>
<td>string</td>
<td>The atom name appearing in the .car file.</td>
</tr>
<tr>
<td>NonbondGroup</td>
<td>rid</td>
<td>Specifies the nonbond group to which the atom belongs. It is a reference to the NonbondGroup table.</td>
</tr>
<tr>
<td>Type</td>
<td>string</td>
<td>The forcefield atom type appearing in the .car file.</td>
</tr>
<tr>
<td>Charge</td>
<td>double</td>
<td>The forcefield-dependent partial charge.</td>
</tr>
<tr>
<td>Chirality</td>
<td>byte</td>
<td>See below.</td>
</tr>
<tr>
<td>AtomicNumber</td>
<td>byte</td>
<td>Number of protons in the atom.</td>
</tr>
<tr>
<td>FormalCharge</td>
<td>short</td>
<td>Formal charge on the atom.</td>
</tr>
<tr>
<td>IsotopeNumber</td>
<td>short</td>
<td>0 (for the most common isotope) or the isotopic number (number of protons plus number of neutrons).</td>
</tr>
<tr>
<td>OutOfPlane</td>
<td>byte</td>
<td>See below.</td>
</tr>
<tr>
<td>Coord</td>
<td>double(3)</td>
<td>The atom coordinates.</td>
</tr>
<tr>
<td>Mass</td>
<td>double</td>
<td>The atomic mass.</td>
</tr>
</tbody>
</table>

\(^a\)rid = row identifier
B. Databases and Tables

In the prochiral cases, priority is established on the basis of atom mass, and on the basis of the masses of connected atoms and groups when the initial masses are the same.

The entry in the OutOfPlane column has the following interpretation:

For non-ESFF forcefields:
0 = no oop energy
1 = oop energy

For ESFF:
0 = no oop energy, not an axial atom
1 = oop energy, not an axial atom
2 = no oop energy, axial atom
3 = oop energy, axial atom

Monomer table

A monomer (also known as a residue in proteins) is a sequence of atoms within a model. For many crystalline systems (e.g., zeolites) where this breakdown does not make sense, the entire model is considered to be one monomer. That is, the Monomer table contains one entry, and all the P1 atoms in the MainCell/Atom table refer to this single “monomer”.

The Monomer table always has the following columns:

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Molecule rid</td>
<td>string</td>
<td>Specifies the model to which the monomer belongs. It is a reference to the Molecule table.</td>
</tr>
<tr>
<td>Number</td>
<td>string</td>
<td>The monomer number appearing in the .car file.</td>
</tr>
<tr>
<td>Type</td>
<td>string</td>
<td>The monomer type appearing in the .car file—e.g., ALA for an alanine amino acid.</td>
</tr>
</tbody>
</table>

A row of the Monomer table corresponds to a monomer.

Molecule table

The Molecule table always has the following columns:
A row of the Molecule table corresponds to a molecule.

**NonbondGroup table**

The NonbondGroup table always has the following columns:

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Monomer</td>
<td>rid</td>
<td>Specifies the monomer to which the nonbond group belongs. It is a reference to the Monomer table. A nonbond group can be assigned to a monomer because all atoms of a nonbond group belong to the same monomer.</td>
</tr>
<tr>
<td>Name</td>
<td>string</td>
<td>The nonbond group name appearing in the .mdf file.</td>
</tr>
<tr>
<td>Switching Atom</td>
<td>rid</td>
<td>The atom at which the group begins. It is a reference to the Atom table. The switching atom is used to construct nonbond neighbor lists.</td>
</tr>
</tbody>
</table>

A row of the NonbondGroup table corresponds to a neutral group to be used in a group-based calculation of nonbond energies.

For non-protein systems, no use is made of NonbondGroups.

**Bond table**

The Bond table always has the following columns:

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Atom-1</td>
<td>rid</td>
<td>Specifies one atom of the pair. It is a reference to the Atom table.</td>
</tr>
<tr>
<td>Atom-2</td>
<td>rid</td>
<td>Specifies the other atom of the pair. It is a reference to the Atom table.</td>
</tr>
<tr>
<td>Order</td>
<td>float</td>
<td>The bond order (e.g., single, double, ...).</td>
</tr>
<tr>
<td>Bibond</td>
<td>rid</td>
<td>Specifies the row in the Bond table, which contains the same bond but with the identities of Atom-1 and Atom-2 reversed.</td>
</tr>
</tbody>
</table>
B. Databases and Tables

A row of the Bond Table corresponds to a bond or connection between a pair of atoms in a molecule.

Each bond is stored twice in the Bond table to facilitate selection. Thus, to find all bonds involving a particular atom, you can just identify all rows of the Bond table having the given atom in the Atom-1 column.

One could store bonded atoms in additional columns of the Atom table rather than providing a separate table for bonds. For some operations, e.g., operations involving a fixed number of columns, this design would be slightly more efficient (however, the Discover program uses acceleration algorithms to make table searches quite efficient). For one-to-many relationships, however, representation of relationships in a separate table is preferable. This keeps table and column structure simple and also simplifies pattern matching.

Contents of the system database for periodic systems

Introduction to symmetry and periodicity

The Discover program can perform simulations on periodic systems (crystals). Periodic systems are represented by a set of atoms in a unit cell. When a set of translation operators (space group operators) is applied to the atoms of the unit cell, space is filled, forming a periodic lattice. The unit cell may be two- or three-dimensional, and its shape is described by a set of cell parameters discussed later. In general, the origin of the unit cell is at (0,0,0), and at least one cell edge lies along the x, y, or z axis.

Systems with periodicity and symmetry include additional representational data in the System database tables. The following terms are used in describing these additional data columns for periodic systems:

Unit cell:

The local reference coordinate system for periodic systems. The cell axes are the intervals of periodicity for each dimension.
Asymmetric atoms:
The input-independent atom set known as the asymmetric unit. This atom set forms the original basis for generation of the periodic system.

Symmetry group:
The set of matrix transforms used to generate additional atoms in the system. These transforms form a mathematical group and are defined in terms of the Unit Cell local coordinate system.

Symmetry atoms:
The unique set of atoms generated by applying the symmetry group to the asymmetric unit of independent atoms. Any non-unique symmetry atoms are not included in the system Atom table.

P1 atoms:
The collection of all asymmetric and symmetry atoms in the system atom table—sometimes referred to as the unit volume. This atom set is the basis for infinite translation by the unit cell vectors to form the periodic system.

Lattice offset:
An ordered triple representing multiples of the cell vector lengths which are added to P1 Atom coordinates to perform translations. The lattice offsets form an infinite set and give rise to the infinite nature of periodic systems.

Ghost atoms:
The set of atoms resulting from lattice translation of P1 Atoms using lattice offsets. In the Discover program, Ghost Atoms are typically generated only on demand, as required to fill out the bonding of the system for a particular process, such as minimization.

Differences from simple systems
For a periodic system, two tables are used in addition to the basic set of tables:
B. Databases and Tables

In addition, two tables acquire some new columns:

Bond
Atom

The name P1Bond is not the actual name of the table in question. The real name is MainCell/P1Bond. However, P1Bond is an alias and can be used as if it were the actual name. This alias is used in the remainder of this discussion.

Descriptions of new and augmented tables

Cell table

For periodic systems, the Cell table has the following columns:

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Periodicity</td>
<td>short</td>
<td>2 or 3, representing 2D or 3D periodicity, respectively.</td>
</tr>
<tr>
<td>AtomTable</td>
<td>string</td>
<td>MainCell/Atom in all cases. It indicates the name of the atom table associated with this cell definition.</td>
</tr>
<tr>
<td>BondTable</td>
<td>string</td>
<td>MainCell/Bond in all cases. It indicates the name of the bond table associated with this cell definition.</td>
</tr>
<tr>
<td>Axes</td>
<td>short</td>
<td>See below.</td>
</tr>
<tr>
<td>CellParameter</td>
<td>double(6)</td>
<td>See below.</td>
</tr>
<tr>
<td>ParameterMatrixPosition</td>
<td>short(12)</td>
<td>Specifies a cell parameter matrix permutation array used internally to handle different axis orientations.</td>
</tr>
<tr>
<td>CellPermutation</td>
<td>short(3)</td>
<td>Specifies a coordinate permutation array used internally.</td>
</tr>
<tr>
<td>Fractional2Cartesian</td>
<td>double(9)</td>
<td>See below.</td>
</tr>
<tr>
<td>Cartesian2Fractional</td>
<td>double(9)</td>
<td>See below.</td>
</tr>
<tr>
<td>Origin</td>
<td>double(6)</td>
<td>Not currently used. Currently assumed to be (0,0,0).</td>
</tr>
<tr>
<td>Orientation</td>
<td>double(9)</td>
<td>Not currently used. Currently assumed to be the identity matrix.</td>
</tr>
<tr>
<td>P1BondTable</td>
<td>string</td>
<td>MainCell/P1Bond in all cases. It indicates the name of the P1Bond table associated with the cell.</td>
</tr>
</tbody>
</table>

A row of the Cell table corresponds to a unit cell.

The entry in the Axes column specifies alignment of cell axes as follows:
0 = zyx; i.e., c axis is aligned with the z axis, b axis is in the z-y plane
1 = zxy; i.e., c axis is aligned with the z axis, a axis is in the z-x plane
2 = xyz; i.e., a axis is aligned with the x axis, b axis is in the x-y plane
3 = xzy; i.e., a axis is aligned with the x axis, c axis is in the x-z plane
4 = yxz; i.e., b axis is aligned with the y axis, a axis is in the y-x plane
5 = yzx; i.e., b axis is aligned with the y axis, c axis is in the y-z plane

The current default value is 2, corresponding to the xyz alignment in the Insight program.

The entry in the CellParameter column specifies cell vectors in terms of the Cartesian axes. Cell vectors are the three edges of the unit cell that are attached to the vertex at the origin. They are generally labeled a, b and c. The form of the entry depends upon the axis convention chosen as follows:

zyx = (a.x a.y b.y a.z b.z c.z)
zxy = (b.y b.x a.x b.z a.z c.z)
xyz = (c.z c.y b.y c.x b.x a.x)
xzy = (b.y b.z c.z b.x c.x a.x)
yxz = (c.z c.x a.x c.y a.y b.y)
yzx = (a.x a.z c.z a.y c.y b.y)

The entry in the Fractional2Cartesian column specifies the matrix which converts fractional coordinates (coordinates with respect to the cell vectors) to Cartesian coordinates in the format: (a.x a.y a.z b.x b.y b.z c.x c.y c.z). Thus \( F \cdot F_{2C} = C \), where \( F \) represents fractional coordinates, \( C \) represents Cartesian coordinates and:

\[
F_{2C} = \begin{bmatrix}
    a.x & a.y & a.x \\
    b.x & b.y & b.z \\
    c.x & c.y & c.z
\end{bmatrix}
\]

Clearly the rows of \( F_{2C} \) represent the cell vectors. For the xyz axis system the matrix reduces to:

\[
F_{2C} = \begin{bmatrix}
    a.x & 0 & 0 \\
    b.x & b.y & 0 \\
    c.x & c.y & c.z
\end{bmatrix}
\]
B. Databases and Tables

where:

\[ a_x = a \]
\[ b_x = b \cos \gamma \]
\[ b_y = b \sin \gamma \]
\[ c_x = c \cos \beta \]
\[ c_y = c \cos \alpha - \cos \beta \cos \gamma \] / \sin \gamma
\[ c.z = \sqrt{\left( c^2 - (c.x)^2 - (c.y)^2 \right)} \]

\[ \gamma = \text{angle between } a \text{ and } b \]
\[ \beta = \text{angle between } a \text{ and } c \]
\[ \alpha = \text{angle between } b \text{ and } c \]

In a number of circumstances it is convenient to represent atom positions in fractional coordinates. Determination of atom position with respect to the cell is particularly straightforward with fractional coordinates. The fraction can be viewed as the fraction of the corresponding cell vector which the atom would traverse in moving from the origin to its current position, e.g., \((0.5, 0.5, 0.5)\) is at the geometric center of the cell.

The entry in the \textit{Cartesian2Fractional} column specifies the matrix which converts Cartesian coordinates to fractional coordinates in the format:

\[(C2F11 \ C2F12 \ C2F13 \ C2F21 \ C2F22 \ C2F23 \ C2F31 \ C2F32 \ C2F33)\).

Thus \(C \cdot C2F = F\), where \(C\) represents Cartesian coordinates, \(F\) represents fractional coordinates and:

\[
C2F = \begin{bmatrix}
C2F11 & C2F12 & C2F13 \\
C2F21 & C2F22 & C2F23 \\
C2F31 & C2F32 & C2F33
\end{bmatrix}
\]
For periodic systems, the \textit{P1Bond} table has the following columns:

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Atom-1</td>
<td>rid</td>
<td>Same as for \textit{Bond} table for nonperiodic systems.</td>
</tr>
<tr>
<td>Atom-2</td>
<td>rid</td>
<td>Same as for \textit{Bond} table for nonperiodic systems.</td>
</tr>
<tr>
<td>Order</td>
<td>float</td>
<td>Same as for \textit{Bond} table for nonperiodic systems.</td>
</tr>
<tr>
<td>Bibond</td>
<td>rid</td>
<td>Same as for \textit{Bond} table for nonperiodic systems.</td>
</tr>
<tr>
<td>XYZLatticeOffset</td>
<td>short (periodicity)</td>
<td>Specifies an offset for \textit{Atom-2} such that \textit{Atom-1} and the offset version of \textit{Atom-2} are bonded. The offset can be trivial—((0\ 0\ 0))—which implies that \textit{Atom-1} and \textit{Atom-2} are bonded to one another. When the offset is nontrivial, no &quot;ghost atom&quot; corresponding to the offset is created initially, so the bond is initially implicit (i.e., not yet in the \textit{Atom} table).</td>
</tr>
</tbody>
</table>

For periodic systems, the \textit{P1 Bond} table is created in addition to the system \textit{Bond} table described on page 313. The \textit{P1 Bond} table contains an additional column for a lattice offset which is applied to \textit{Atom-2} to give the bondmate for \textit{ Atom-1}. In this table, both atoms are \textit{P1 Atoms}.

Similarly, the system \textit{Atom} table is augmented to include two additional columns used to identify the \textit{P1 Atom} parent and lattice offset for \textit{Ghost Atoms} in the system. \textit{P1 Atoms} all have NULL/(0,0,0) entries in these columns.

A P1 bond (i.e., an entry in the \textit{P1Bond} table) represents an “infinite bond” consisting of the specified atom pair and all pairs constructed by applying an identical lattice offset to the atoms of the specified pair.

For periodic systems, the \textit{Bond} table is created by making a copy of the first four columns of the \textit{P1Bond} table and zeroing out any \textit{Atom-2} references corresponding to nontrivial lattice offsets. A \textit{P1Bond} column (of type \texttt{rid}) is added to the \textit{Bond} table so that each entry in \textit{Bond} has a reference to its associated P1 bond. Thus, if you need an explicit bond, but find that \textit{Atom-2} is zero, you can use the related P1 bond to make a new ghost atom.

Given any P1 atom (i.e. an atom specified explicitly in a periodic system), you can find its bond mates by looking under \textit{Atom-1} in the \textit{Bond} table. Its bond mates are either already there under \textit{Atom-2}, or can be constructed using the \textit{P1Bond} table. Now sup-
B. Databases and Tables

pose you have a ghost atom formed by adding a lattice offset, L, to a parent P1 atom, P. How do you find its bond mates? We have established how to determine the bond mates of the P1 atom P. Applying L to each of these yields the bond mates of the original ghost atom. Thus, if Atom-2 + Loff is a bondmate of P obtained from P1Bond, then Atom-2 + Loff + L is bonded to the ghost atom.

Atom table augmentation for periodic systems

For periodic systems, a P1Atom column (type rid) and an XYZLatticeOffset column (type short(3)) are added to the Atom table. Entries in these columns are null and (0 0 0), respectively, for the P1 atoms. For ghost atoms, a reference to the associated P1 atom is stored in the P1Atom column, and the associated offset is stored in the XYZLatticeOffset column.

Contents of the system database for systems with symmetry and periodicity

Differences from simpler systems

For systems involving symmetry and periodicity, the following tables are used in addition to those described for periodic systems without symmetry:

SymmetryObject
SymmetryOperator
MainCell/NAME/AsymmetricBond

where NAME specifies the symmetry group. Thus the statement

@group (PNA21) [ORIGIN_1]

in an .mdf file gives rise to a table named MainCell/PNA21/ORIGIN_1/AsymmetricBond.

The following tables acquire additional columns for systems involving symmetry and periodicity:

Cell
Atom

Descriptions of new and augmented tables

SymmetryObject table

The SymmetryObject table has the following columns:
A row of the SymmetryObject table represents an instance of a symmetry group (point, plane, or space, depending upon whether the system is nonperiodic, 2D-periodic, or 3D-periodic).

**SymmetryOperator table**

The SymmetryOperator table has the following columns:

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cell rid</td>
<td>rid</td>
<td>Specifies the associated cell. It is a reference to the Cell table.</td>
</tr>
<tr>
<td>GroupLabel</td>
<td>string</td>
<td>A group label such as PNA21.</td>
</tr>
<tr>
<td>GroupQualifier</td>
<td>string</td>
<td>A group qualifier such as ORIGIN_1.</td>
</tr>
<tr>
<td>NumberSymmetryMatrices</td>
<td>short</td>
<td>The number of symmetry operators (i.e., rows in the SymmetryOperator table) in the symmetry group.</td>
</tr>
<tr>
<td>Change</td>
<td>short</td>
<td>A flag indicating that positions of asymmetric atoms (base atoms specified explicitly) have changed (as in a minimization step) and that the positions of the full atom set (atoms specified through application of symmetry operators to asymmetric atoms) must therefore be regenerated.</td>
</tr>
<tr>
<td>DegreesFreedom</td>
<td>unsigned</td>
<td>The number of degrees of freedom of the associated system, diminished by symmetry constraints.</td>
</tr>
<tr>
<td>AsymmetricColumn</td>
<td>string</td>
<td>Contains the string “AsymmetricAtom”. This is the name of the extra column in the Atom table that identifies the asymmetric atom associated with a given atom.</td>
</tr>
<tr>
<td>SymmetryMatrixColumn</td>
<td>string</td>
<td>Contains the string “SymmetryMatrix”.</td>
</tr>
<tr>
<td>AsymmetricBondTable</td>
<td>string</td>
<td>Contains “MainCell/NAME/AsymmetricBond” (e.g., “MainCell/PNA21/ORIGIN_1/AsymmetricBond”).</td>
</tr>
</tbody>
</table>

A row of the SymmetryObject table represents an instance of a symmetry group (point, plane, or space, depending upon whether the system is nonperiodic, 2D-periodic, or 3D-periodic).
B. Databases and Tables

A row of the SymmetryOperator table corresponds to a particular symmetry operator.

**MainCell/NAME/AsymmetricBond table**

The MainCell/NAME/AsymmetricBond table has the following columns:

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Atom-1</td>
<td>rid</td>
<td>Same name as in the P1Bond table.</td>
</tr>
<tr>
<td>Atom-2</td>
<td>rid</td>
<td>Same name as in the P1Bond table.</td>
</tr>
<tr>
<td>Order</td>
<td>float</td>
<td>Same name as in the P1Bond table.</td>
</tr>
<tr>
<td>Bibond</td>
<td>rid</td>
<td>Same name as in the P1Bond table.</td>
</tr>
<tr>
<td>XYZLatticeOffset</td>
<td>short(periodicity)</td>
<td>Same name as in the P1Bond table.</td>
</tr>
<tr>
<td>SymmetryMatrix</td>
<td>short</td>
<td>See below.</td>
</tr>
</tbody>
</table>

If the system also has symmetry, a third asymmetric Bond table is created, which includes an additional column to hold the symmetry operator identifier to be applied to Atom-2 in order to produce the bondmate of Atom-1. In this table both Atom-1 and Atom-2 are asymmetric atoms. Note that the asymmetric bond table can hold asymmetric atom identifiers plus symmetry and lattice offsets—this information corresponds to the connectivity data in the .mdf file for full-symmetry systems.

When an .mdf file is read in, the MainCell/NAME/AsymmetricBond table captures the .mdf file information as-is. For each row, the symmetry operation specified in the SymmetryMatrix column is applied to Atom-2 to generate a bond for inclusion in the P1Bond table.

**Cell table augmentation for systems with symmetry**

For systems with symmetry, a CurrentSymmetryObject column (of type rid) is added to the Cell table. An entry in this column is a reference to the SymmetryObject table. (The Cell table contains explicit cell data and, if symmetry is present, a pointer to the SymmetryObject table, which contains or points to the symmetry data.)
For systems with symmetry, the following columns are added to the Atom table:

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Asymmetric Atom rid</td>
<td>Null</td>
<td>Null for asymmetric atoms, and, if not null, indicates the asymmetric atom from which the given atom is derived.</td>
</tr>
<tr>
<td>Symmetry Matrix short</td>
<td>short</td>
<td>The number of the symmetry operation used to produce the given atom from the associated asymmetric atom.</td>
</tr>
<tr>
<td>Projection Matrix double</td>
<td>double</td>
<td>The transformation used to maintain symmetry in the course of dynamics, which would break symmetry if left to itself.</td>
</tr>
<tr>
<td>Degrees Freedom short</td>
<td>short</td>
<td>Degrees of freedom associated with the given atom. In general, there are 3 degrees of freedom; however, when an atom is confined by symmetry to a plane, line or point, this number is reduced accordingly (to 2, 1, or 0).</td>
</tr>
</tbody>
</table>

**Subset tables**

A system database may also contain subset information. When this is so, the database has a Subset table. A subset is an arbitrary collection of one or more objects. The objects can be of many different types, e.g., atoms, distances, angles, etc. A Subset table has the following columns:

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Context</td>
<td>rid</td>
<td>Specifies the context in which the subset is defined, i.e., a particular model or monomer. That is, if a subset is defined using atoms in one monomer, it is in the monomer context; if the atoms are in several monomers, it is in the molecule context.</td>
</tr>
<tr>
<td>Name</td>
<td>string</td>
<td>Specifies the subset name. Subsets in different contexts may have the same name. If two subsets in the same context have the same name, subset select commands for that context and name retrieve the contents of both. They are separate subsets only in the sense that they correspond to different rows of the Subset table.</td>
</tr>
</tbody>
</table>

A row of the Subset table corresponds to a subset.
B. Databases and Tables

Subset contents are stored in the SubItem table. The SubItem table has the following columns:

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subset</td>
<td>rid</td>
<td>Specifies the row number in the Subset table of the subset which contains the item.</td>
</tr>
<tr>
<td>Item</td>
<td>rid</td>
<td>Specifies the item itself. It could be an atom or a distance, angle, torsion, or out-of-plane object created in the course of the subset definition.</td>
</tr>
</tbody>
</table>

A row of the SubItem table corresponds to an item in a subset.

PseudoAtom table

A system database may also contain pseudoatom information. When this is so, the database has a PseudoAtom table. A pseudoatom does not correspond to a real atom, but rather is just a point specified explicitly or by the weighted-average coordinates of other atoms. It is called a pseudoatom because it can be used in many operations that take real atoms, e.g., distance measurements and energy restraints. A PseudoAtom table has the following columns:

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Context</td>
<td>rid</td>
<td>Specifies the context in which the pseudoatom is defined, i.e., a particular molecule or monomer. That is, if a pseudoatom is defined using atoms in one monomer, it is in the monomer context; if the atoms are in several monomers, it is in the molecule context.</td>
</tr>
<tr>
<td>Name</td>
<td>string</td>
<td>Specifies the pseudoatom name. Multiple pseudoatoms can share a given name and context.</td>
</tr>
<tr>
<td>Atom</td>
<td>rid</td>
<td>Identifies the row of the Atom table corresponding to the pseudoatom.</td>
</tr>
<tr>
<td>Subset</td>
<td>rid</td>
<td>Identifies a subset containing the component atoms of the pseudoatom. “Fixed” pseudoatoms, which are defined by fixed coordinates, have no component atoms and hence no entry in the Subset column.</td>
</tr>
<tr>
<td>Weight</td>
<td>OBJ, AR-RAY or string</td>
<td>See below.</td>
</tr>
</tbody>
</table>

A row of the PseudoAtom table corresponds to a pseudoatom.
System database

The Weight column can take several forms:

- NULL = pseudoatom coordinates are the geometric centroid of the component atom coordinates
- OBJ_ARRAY = object array of weights used in computing pseudoatom coordinates (should have a weight for each component atom)
- string = name of a column in the Atom table with values used as weights in computing pseudoatom coordinates; e.g., mass (for center of mass pseudoatom), charge, or a user-defined column— for center of mass:

\[
\text{Coors} = \frac{\sum w_i r_i}{\sum w_i}
\]

where \( i \) runs over all atoms in the pseudoatom and \( w_i \) represent the atomic weights.

Internal energy exclusion

For some applications it may be desirable to exclude the energy computation between specific atoms. For example, you may want to calculate a special bond energy for a specific bond or set of bonds and supply the contribution to the Discover program via interprocess communication (IPC). In this case, you can set the appropriate internal energy exclusion flags to tell the Discover program not to calculate such a bond. Note that this is different from the atomMovability flags which affect which atoms are movable: a set of atoms can be excluded from the energy computation (presumably because the values are calculated elsewhere) and still be movable for minimization and dynamics.

Exclusion flags

To exclude specific atoms from the internal energy computation, create a column of type short in the Atom table of the System database. This column must be named InternalEnergyExclude. A non-zero setting means that that particular atom is to be excluded. Note that an internal energy term is excluded only if all the atoms involved in calculating that internal energy term are excluded. For
B. Databases and Tables

instance, both atoms for a bond term need to be excluded before the bond term is not calculated.

**Btcl commands**

To facilitate setting the internal energy exclusion flags, a Btcl procedure called System_SetExcludeInternalEnergy has been provided. This procedure checks if the Atom.InternalEnergyExclude column exists, creates it if necessary, and sets the value in the column to be 1 (exclude) or 0 (include) for the atoms specified. The procedure is contained in the file $BIOSYM/data/discover/script/systemAtom.tcl.

**Example**

```bsh
#BIOSYM btcl 3
#testing acenm for internal energy exclusion functionalities

set PROJECT acenm
begin

database handle dbh System.

#energy for the whole system
energy print energies = 1

#exclude some of the atoms for internal energy calculations
System_SetExcludeInternalEnergy "ACENM:ACE_1:(CA,HA1)"

#energy for the whole system with 2 atoms excluded
energy print energies = 1

#exclude these atoms
atomMovability set excluded ex_1 "ACENM:ACE_1:(HA2,HA3,C,O), ACENM:N-M_2:*

#include the atoms excluded back into the system
System_SetExcludeInternalEnergy "ACENM:ACE_1:(CA,HA1)" Include

#energy for the system with only 2 atoms
energy print energies = 1

The output from calculating the energies of the system is:
```

The output from calculating the energies of the system is:
System database

<table>
<thead>
<tr>
<th>Energy components</th>
<th>kcal/mol</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total:</td>
<td>-18.530128</td>
</tr>
<tr>
<td>Internal:</td>
<td>5.607232</td>
</tr>
<tr>
<td>Bond:</td>
<td>2.758628</td>
</tr>
<tr>
<td>Angle:</td>
<td>2.848604</td>
</tr>
<tr>
<td>Torsion:</td>
<td>0.000000</td>
</tr>
<tr>
<td>OutOfPlane:</td>
<td>0.000000</td>
</tr>
<tr>
<td>Nonbond:</td>
<td>-24.137360</td>
</tr>
<tr>
<td>Vdw:</td>
<td>3.727446</td>
</tr>
<tr>
<td>Repulsive:</td>
<td>8.964310</td>
</tr>
<tr>
<td>Dispersive:</td>
<td>-5.236864</td>
</tr>
<tr>
<td>Electrostatic:</td>
<td>-27.864806</td>
</tr>
<tr>
<td>Hydrogenbond:</td>
<td>0.000000</td>
</tr>
</tbody>
</table>

The output after excluding some of the atoms and recalculating the energies is:

<table>
<thead>
<tr>
<th>Energy components</th>
<th>kcal/mol</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total:</td>
<td>-18.743014</td>
</tr>
<tr>
<td>Internal:</td>
<td>5.394346</td>
</tr>
<tr>
<td>Bond:</td>
<td>2.545742</td>
</tr>
<tr>
<td>Angle:</td>
<td>2.848604</td>
</tr>
<tr>
<td>Torsion:</td>
<td>0.000000</td>
</tr>
<tr>
<td>OutOfPlane:</td>
<td>0.000000</td>
</tr>
<tr>
<td>Nonbond:</td>
<td>-24.137360</td>
</tr>
<tr>
<td>Vdw:</td>
<td>3.727446</td>
</tr>
<tr>
<td>Repulsive:</td>
<td>8.964310</td>
</tr>
<tr>
<td>Dispersive:</td>
<td>-5.236864</td>
</tr>
<tr>
<td>Electrostatic:</td>
<td>-27.864806</td>
</tr>
<tr>
<td>Hydrogenbond:</td>
<td>0.000000</td>
</tr>
</tbody>
</table>

The output after excluding all but two atoms and recalculating the energies is:

<table>
<thead>
<tr>
<th>Energy components</th>
<th>kcal/mol</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total:</td>
<td>0.212886</td>
</tr>
<tr>
<td>Internal:</td>
<td>0.212886</td>
</tr>
<tr>
<td>Bond:</td>
<td>0.212886</td>
</tr>
<tr>
<td>Angle:</td>
<td>0.000000</td>
</tr>
<tr>
<td>Torsion:</td>
<td>0.000000</td>
</tr>
<tr>
<td>OutOfPlane:</td>
<td>0.000000</td>
</tr>
<tr>
<td>Nonbond:</td>
<td>0.000000</td>
</tr>
<tr>
<td>Vdw:</td>
<td>0.000000</td>
</tr>
<tr>
<td>Repulsive:</td>
<td>0.000000</td>
</tr>
<tr>
<td>Dispersive:</td>
<td>0.000000</td>
</tr>
<tr>
<td>Electrostatic:</td>
<td>0.000000</td>
</tr>
<tr>
<td>Hydrogenbond:</td>
<td>0.000000</td>
</tr>
</tbody>
</table>
B. Databases and Tables

Note that the internal energy from the second and third calculations add up to the same internal energy as in the first calculation.

Nonbond energy exclusion

Exclusion flags

To exclude the energies of some nonbond interaction pairs, create a column of flags of type short, called NonbondEnergyExclude, in the MainCell/Atom table of the System database. The flags in the column should be set to either 0 or 1. The nonbond interaction energies between any two atoms that are both labeled by 1 are not calculated during the energy calculation. Note that whether the nonbond interaction energy between an atom labeled 0 and an atom labeled 1 should be calculated is not affected by this column of flags. It is similar to the fixed-atom model, but is used for excluding only some nonbond interactions.

Example

#BIOSYM btcl 3
#testing acenm for internal energy exclusion functionalities

set PROJECT acenm
begin

database handle dbh System.

#energy for the whole system
energy print energies = 1

database handle en1 Energy.
$en1 select “Nonbond” Values.Name el
$en1 get nb1 Values.Value $el

#exclude atoms in residue 1
System_SetExcludeNonbondEnergy “ACENM:1:atom;**”

#energy for the whole system with 2 atoms excluded
energy print energies = 1
database handle en2 Energy.
$en2 select "Nonbond" Values.Name e2
$en2 get nb2 Values.Value $e2

$dbh print Atom.NonbondEnergyExclude
$dbh print Atom.Movability

#include back atoms in residue 1
System_SetExcludeNonbondEnergy "ACENM:1:atom;*" Include

$dbh print Atom.NonbondEnergyExclude
$dbh print Atom.Movability

#exclude atoms in residue 2
atomMovability set excluded ex_1 "ACENM:2:atom;*"

echo nonbond energies for the full system= [vector nb1]
echo nonbond energies for the with excluded atoms= [vector nb2]
echo nonbond energies for the excluded atoms along= [vector nb3]

Consensus dynamics

To do consensus dynamics, you need to create a column of flags of type integer, called Consensus, in the Main Cell/Atom table of the System database. See details on page 175).
B. Databases and Tables

Energy database

Introduction

A read-only energy database is created whenever a `begin` or `reset` command is encountered in a Btcl script (the .inp file). The energy database is of type `Energy` in the `CurrentDatabase` table. The purpose of the `Energy` database is to provide script-level access to important energy-related quantities in the program. Any quantity stored in the database can be recovered via Btcl commands and manipulated or output as you want. Complete energy breakdowns by bond, angle, torsion, and out-of-plane angle are available.

Contents of the energy database

Two tables, `Values` and `Atom`, are always created in the energy database. Additionally, a `Misc` table is created to store the virial coefficients and the Hessian if either is created during the current energy computation. Additional tables named `EnergyControl`, `NonbondInteractionTable`, and `NonbondInteraction-` are created if nonbond analysis is requested (see the `analyzeNonbond` command).

Values table

There are four (4) columns in the `Values` table:

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>string</td>
</tr>
<tr>
<td>Parent</td>
<td>rid</td>
</tr>
<tr>
<td>Value</td>
<td>double</td>
</tr>
<tr>
<td>Print</td>
<td>short</td>
</tr>
</tbody>
</table>

Thus, a row specifies the name of an energy term, its immediate parent, and the numerical value of the energy term. The `Value` column is updated after every energy calculation. The `Print` column
Energy database

is used to control the output of rows with an energy value of 0.0 (see the examples below).

The purpose of the Parent column is to define a hierarchy of energies. For example, Bond energy is a child of Internal energy. All children of Internal energy are summed to form the total value for Internal energy. Therefore, only leaves on the tree (i.e., final ends, or rows with no children referring to them) are independent energy values. These values are extracted from the current energy calculation and recursively summed into their parents to finally provide the Total energy row.

Atom table

The Atom table always contains four (4) columns:

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coord</td>
<td>double</td>
</tr>
<tr>
<td>Gradient</td>
<td>double</td>
</tr>
<tr>
<td>Nonbond-A</td>
<td>double</td>
</tr>
<tr>
<td>Nonbond-B</td>
<td>double</td>
</tr>
</tbody>
</table>

These columns contain the coordinates, gradients, and the non-bond parameters A (or r) and B (or ε). The columns Repulsive, Dispersive and Electrostatic are created (all of type double) specifying the repulsive, dispersive, and electrostatic energies, respectively, for each atom in the system after the analyzeNonbond command is executed with the on keyword for an energy calculation. The per-atom energy is defined such that the sum of all per-atom energies equals the total nonbond energy of the system.

van der Waals repulsive and dispersive energies and Coulomb and hydrogen bond energies on a per-atom basis are accessible from the Atom table of the Energy database. Given the atom interaction energies, per-monomer, per-molecule, and even per-subset energies can be calculated.
B. Databases and Tables

Accessing the energy database

Currently, the energy database is intended to be read-only. Changing values (for example an energy term) has no effect on the current calculation. Energy database values are overwritten with new values from the internal energy structures at each energy evaluation.

The energy database is deleted when the Discover program is finished running or when a reset or begin command is issued.

Example 1—Print commands

The Values table can be obtained and printed out by simple Btcl commands. A sample output may be:

```
BTCL > database handle ener_h Energy.
BTCL > energy
BTCL > $ener_h print Values
"Values"
{
  | Row | Name             | Parent | Value   | Print |
--- | -----------------| ------ | ------- | ------|
  0) Total                        .  -18.530  0
  1) Internal                     0    5.607  0
  2) Bond                         1    2.759  0
  3) Angle                        1    2.849  0
  4) Torsion                      1    0.000  0
  5) OutOfPlane                   1    0.000  0
  6) Cross                        1    0.000  1
  7) BondBond                     6    0.000  0
  8) BondAngle                    6    0.000  0
  9) EndBondTorsion               6    0.000  0
10) MiddleBondTorsion            6    0.000  0
11) AngleTorsion                 6    0.000  0
12) AngleAngleTorsion            6    0.000  0
13) BondBond-1-3                 6    0.000  0
14) AngleAngle                   6    0.000  0
15) Nonbond                      0  -24.137  0
16) Vdw                          15    3.727  0
17) Repulsive                    16    8.964  0
18) Dispersive                   16  -5.237  0
19) Electrostatic                15  -27.865  0
```
Energy database

20) Hydrogenbond  15  0.000  0
21) Restraint     0  0.000  1
22) External     0  0.000  1
23) Diffraction  22  0.000  0
}

Other commands that request energy printout, e.g., **energy print energies = 1**, use the data stored in the **Values** table to form a hierarchy of energy values:

```
Energy components          kcal/mol
Total:               -18.530128
   Internal:         5.607232
         Bond:       2.758628
         Angle:      2.848604
         Torsion:    0.000000
         OutOfPlane: 0.000000
   Nonbond:         -24.137360
         Vdw:       3.727446
         Repulsive: 8.964310
         Dispersive: -5.236864
         Electrostatic: -27.864806
   Hydrogenbond:    0.000000
```

Additionally, the **Print** column dictates how 0.0 values are collapsed:

<table>
<thead>
<tr>
<th>If the value is:</th>
<th>Then</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Always print the term.</td>
</tr>
<tr>
<td>1</td>
<td>Look at the term and its descendants and print them out if any of them are not zero.</td>
</tr>
<tr>
<td>2</td>
<td>Do not print the term; do not look at its descendants.</td>
</tr>
</tbody>
</table>

**Example 2—Accessing data**

You can access the Hessian, stored in the **Misc** table, and perform a diagonalization by using the Btcl script procedure **Utility_CastHessianMatrix** (this script is found in the file $BIOSYM/data/discover/script/energyAtom.tcl) and the Btcl **vector** command.

```
BTCL > database handle ener_h Energy.
BTCL > energy print first_deriv = 1 print second_deriv = 1
BTCL > $ener_h get hess Misc.Hessian
```
Example 3—Specialized output

In this example, the user wants to output the electrostatic energy every 10 steps of a minimization. Additionally, the output is to be placed in a separate file created by the script.

```tcl
BTCL > database handle ener_h Energy.
BTCL > set fileHandle [open electrostatic.list w]
BTCL > proc getElectrostaticEnergy {ener_h} {
    $ener_h select Electrostatic Values.Name row_n
    $ener_h get e_energy .Value $row_n
    return [object e_energy]
}
BTCL > minimize \execute frequency = 10 \command = {puts $fileHandle [getElectrostaticEnergy $ener_h]}
BTCL > close $fileHandle
```

Example 4—Decomposition of internal energy

Internal energies and derivatives can be decomposed into the more basic components of individual bonds, angles, torsions, and out-of-plane angles by setting the `internalEnergyDecomposition` name in the `Global` table. This can be activated through the Btcl command `System_SetGlobal` (this script is found in the file `$BIOSYM/data/discover/script/systemAtom.tcl`):

```tcl
BTCL > System_SetGlobal internalEnergyDecomposition 1
```

Any subsequent energy evaluations tabulate the individual energies and derivatives in the `Bond`, `Angle`, `Torsion`, and `Oop` tables of the `Energy` database.
BTCL > database handle edb Energy.

BTCL > $edb print Bond

"Bond"
{
  Row  Atom1  Atom2  Energy    dE/dBnd
  ---  -----  -----  --------  -------
   0)      0      1   0.04539    8.046
   1)      0      2   0.00180    1.816
   2)      0      3   0.00023    0.644
   3)      1      4   0.00458    2.498
   4)      1      5   0.03620    6.403
   5)      1      8   0.00117    1.230
   6)      5      7   0.00386    3.082
   7)      5      6   0.00006   -0.297
   8)      8      9   0.00088    1.093
   9)      8     10   0.00167    1.507
  10)      8     11   0.00155    1.191
  11)     11     12   0.00070    0.875
  
}

The example above shows the decomposition of the bond internal energy. These energy values include the bond energy as well as the contributions from the bond crossterms. Each crossterm energy is distributed completely into the internals that make up the crossterm. For example, a bond/angle crossterm contributes half of its energy to the bond and the other half of its energy to the angle. Below is a summary of how each of the crossterms’ energies are factored:

<table>
<thead>
<tr>
<th>crossterm</th>
<th>apportioning of energy</th>
</tr>
</thead>
<tbody>
<tr>
<td>bond–bond</td>
<td>1/2 bond1 1/2 bond2</td>
</tr>
<tr>
<td>bond–angle</td>
<td>1/2 bond 1/2 angle</td>
</tr>
<tr>
<td>angle–angle</td>
<td>1/2 angle1 1/2 angle2</td>
</tr>
<tr>
<td>bond1-torsiona</td>
<td>1/2 bond 1/2 torsion</td>
</tr>
<tr>
<td>bond2-torsionb</td>
<td>1/4 bond1 1/4 bond2 1/2 torsion</td>
</tr>
<tr>
<td>angle-torsion</td>
<td>1/2 angle 1/2 torsion</td>
</tr>
<tr>
<td>angle–angle–torsion</td>
<td>1/3 angle1 1/3 angle2 1/3 torsion</td>
</tr>
</tbody>
</table>

a middle-bond–torsion crossterm
b bend bond–torsion–end bond crossterm
B. Databases and Tables

There is a little overhead in bookkeeping for the decomposition; therefore, when the energy breakdown is no longer needed, the option should be unset with:

```
BTCL > System_SetGlobal internalEnergyDecomposition 0
```

## Minimize database

### Introduction

A minimize database is created when the `minimize` command is invoked from Btcl. The `Minimize` database is of type `Minimize` in the `CurrentDatabase` table. The purpose of the minimize database is to record the series of steps taken during the minimization process. It is also possible to access the minimize database during the minimization in order, for example, to implement particular stopping strategies.

The `Iteration` table is always created along with the database. It has eight (8) columns:

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Method</code></td>
<td>string</td>
<td>The current method used (steepest descents, conjugate gradients, Newton).</td>
</tr>
<tr>
<td><code>SubMethod</code></td>
<td>string</td>
<td>The specific sub-method used (BFGS, DFP, etc.).</td>
</tr>
<tr>
<td><code>StepsTaken</code></td>
<td>int</td>
<td>The number of iterations taken for the method.</td>
</tr>
<tr>
<td><code>EnergyCalls</code></td>
<td>int</td>
<td>The number of energy calls taken for the method.</td>
</tr>
<tr>
<td><code>InitialMaximumDerivative</code></td>
<td>double</td>
<td>The initial maximum derivative when the minimization begins for the method.</td>
</tr>
<tr>
<td><code>ToleranceMaximumDerivative</code></td>
<td>double</td>
<td>The convergence tolerance specified for the method.</td>
</tr>
<tr>
<td><code>CurrentMaximumDerivative</code></td>
<td>double</td>
<td>The current maximum derivative when the minimization is being performed for the method.</td>
</tr>
<tr>
<td><code>TimeInMethod</code></td>
<td>double</td>
<td>The elapsed CPU time (in seconds) taken by the method.</td>
</tr>
</tbody>
</table>

A new row is created every time a new method of minimization is specified.

Additionally, when BFGS is specified as the Newton method, a `Misc` table is created under the `Minimize` database, which has one
Minimize database

(1) column, called InverseHessian, of type OA_DOUBLE, which contains the inverse Hessian. Only the latest Hessian calculated is kept in the Misc table, meaning that the Misc table has, at most, one (1) row at any time.

**Example 1**—Access to the minimize database during a minimization

To access the Minimize database, first create a database handle. This can be done any time that minimization is running but it may be more efficient to do so once, before the minimization starts. This is accomplished with the first execute statement in the minimize command:

```
BTCL > minimize \
    iteration_limit = 20 \n    execute +before frequency = 0 \n    command = {database handle min_h Minimize.; database print $min_h} \n    execute frequency = 1 before = 1 after = 1 \n    command = {$min_h print Iteration; print output energy_summary = 1 \n        internal_energy = 1 nonbond_energy = 1}
```

The second execute statement uses the database handle to print out the Iteration table (in addition to performing other commands to output energy summaries).

A sample output of the Iteration table looks like:

```
<table>
<thead>
<tr>
<th>Row</th>
<th>Method</th>
<th>SubMethod</th>
<th>StepsTak</th>
<th>EnergyCa</th>
<th>InitialM</th>
<th>Toleranc</th>
<th>CurrentM</th>
<th>TimeInMe</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SteepestDescent</td>
<td></td>
<td>0</td>
<td>1</td>
<td>71.95</td>
<td>1000.0</td>
<td>71.95</td>
<td>0.3000</td>
</tr>
<tr>
<td>1</td>
<td>ConjugateGradient</td>
<td>PolakRibiere</td>
<td>4</td>
<td>8</td>
<td>71.95</td>
<td>10.0</td>
<td>7.62</td>
<td>0.3500</td>
</tr>
<tr>
<td>2</td>
<td>NewtonMethod</td>
<td>NewtonRaphson</td>
<td>3</td>
<td>4</td>
<td>7.62</td>
<td>0.0</td>
<td>0.00</td>
<td>0.5300</td>
</tr>
</tbody>
</table>
```

**Example 2**—Access to the inverse Hessian after a minimization

For the Broyden–Fletcher–Goldfarb–Shanno minimization method, an inverse Hessian is produced; it can be accessed as follows:

---
B. Databases and Tables

```bsh
BTCL > minimize
   iteration_limit = 1000 movement_limit = 0.200
   sd
   convergence = 1000.0 line_search_precision = 0.100
   final_convergence = 1000.0
   cg
   convergence = 10.0 method = polak
   line_search_precision = 0.100 final_convergence = 10.0
   newton
   convergence = 1.0e-13 method = bfgs
   line_search_precision = 0.900 max_atoms = 200
   final_convergence = 1.0e-13

BTCL > database handle min_h Minimize.

BTCL > database print $min_h

BTCL > $min_h get i_hess Misc.InverseHessian

BTCL > Utility_CastHessianMatrix i_hess
```

where `Utility_CastHessianMatrix` is a Btcl script to cast the inverse Hessian into a matrix object, called `i_hess`, which is suitable as input for other Discover commands.

---

**Dynamics database**

*Introduction*

A Dynamics database is created when the `dynamics` Btcl command is invoked. The Dynamics database is of type Dynamics in the CurrentDatabase table.

Two (2) tables are always created along with the database:

- The `Properties` table contains two (2) columns, Name (type = string) and Value (type = double) which give the name and the value of some of the properties in a dynamics run. Some of the properties reported are number of degrees of freedom, kinetic energy, total energy, temperature, and, for periodic systems, volume, density, pressure, stress, strain, and cell parameters.

- The `Coord` table contains two (2) columns, Velocity (type = double) and Acceleration (type = double), which give the velocities and accelerations of the asymmetric, movable atoms in the system.
Example — Accessing the dynamics database during a dynamics run

First create a database handle. This can be done any time that dynamics is running, but may be best to do so before the dynamics run starts, for reasons of efficiency.

```
BTCL > dynamics time = 10 \
execute +before frequency = 0 \
  command = {database handle dyn_h Dynamics.; database print $dyn_h} \
execute frequency = 1 before = 1 after = 1 \
  command = {$dyn_h print Properties; $dyn_h print Coord}
```

In this example, we get the dynamics database handle before dynamics starts and print it out. We also print out the Properties table before, during (every step), and after dynamics.

Sample output of the Properties and Coord tables are:

The Properties table:

```
{  
  Row | Name           | Value  
  --- | -------------- | ------  
  0)  | DegreesFreedom | 24.00  
  1)  | KineticEnergy  | 7.10   
  2)  | TotalEnergy    | -132.78 
  3)  | Temperature    | 298.00 
  4)  | Volume         | 112.99 
  5)  | Density        | 2.65   
  6)  | Pressure       | 10.19  
  7)  | StressXX       | -12.78 
  8)  | StressYY       | -9.75  
  9)  | StressZZ       | -8.03  
 10)  | StressXY       | 1.54   
 11)  | StressXZ       | -0.23  
 12)  | StressYZ       | -0.40  
 13)  | StrainXX       | 0.00   
 14)  | StrainYY       | 0.00   
 15)  | StrainZZ       | 0.00   
 16)  | StrainXY       | 0.00   
 17)  | StrainXZ       | 0.00   
 18)  | StrainYZ       | 0.00   
  9)  | A              | 4.91   
 10)  | B              | 4.91   
 11)  | C              | 5.41   
 12)  | Alpha          | 90.00  
 13)  | Beta           | 90.00  
 14)  | Gamma          | 120.00 
 15)  | Boltzmann      | 1.00   }
```
B. Databases and Tables

The Coord table:

<table>
<thead>
<tr>
<th>Row</th>
<th>Velocity</th>
<th>Acceleration</th>
</tr>
</thead>
<tbody>
<tr>
<td>0)</td>
<td>{ 0.0000000 0.0000000 0.0000000 }</td>
<td>{ 0.000000 0.000000 0.000000 }</td>
</tr>
<tr>
<td>1)</td>
<td>{ 0.0000000 0.0000000 0.0000000 }</td>
<td>{ 0.000005 0.000000 -0.000001 }</td>
</tr>
<tr>
<td>2)</td>
<td>{ 0.0000000 -0.0006231 0.0000000 }</td>
<td>{ 0.0000000 -0.003234 0.000000 }</td>
</tr>
<tr>
<td>3)</td>
<td>{ 0.0004435 0.0002521 0.0000000 }</td>
<td>{ -0.0003330 0.000034 -0.001037 }</td>
</tr>
<tr>
<td>4)</td>
<td>{ -0.0000482 0.0000000 0.0006467 }</td>
<td>{ 0.0003144 0.000000 0.000188 }</td>
</tr>
<tr>
<td>5)</td>
<td>{ -0.0000549 -0.0000133 -0.0000301 }</td>
<td>{ 0.0000033 0.000057 0.000335 }</td>
</tr>
<tr>
<td>6)</td>
<td>{ -0.0000206 0.0004364 -0.0005672 }</td>
<td>{ 0.0003730 0.000752 0.012754 }</td>
</tr>
<tr>
<td>7)</td>
<td>{ -0.0001985 0.0000000 -0.0000706 }</td>
<td>{ 0.0004280 0.000000 0.000079 }</td>
</tr>
</tbody>
</table>

Miscellaneous examples

Example—Heat diffusion calculation

This section explains the input file that is used in Lesson 6: Using a Precoded Btcl Script for a Complex Simulation (supplied electronically). This file, which can be run by using the Pilot utility, illustrates several techniques:

- Defining Btcl procedures and using their arguments.
- Using the echo command (and custom Btcl procedures) to test scripts.
- Getting data from a table into an object array.
- Selecting a subset of the rows of a table based on the contents of a column.
- Using the vector and geometry commands for vector and matrix calculations.
- Setting data into a table from an object array.
- Automatically invoking a procedure at the beginning of a dynamics command.

First we define a convenient Btcl procedure to echo only a range of items in an object array—some of those in this example have hundreds of items, too many to conveniently echo them all. You may find procedures like this useful for testing and debugging your own Btcl scripts:
Miscellaneous examples

proc echoRange { obj first last } \
{
    # create rng containing obj items first to last
    object rng range $obj $first $last

    # print rng to a string, then echo that string
    echo [object rng print]
}

Define a Btcl procedure to multiply the temperature of all the
atoms within a given radius of the center of the periodic cell by a
factor of temperatureRatio, thus heating the fragments in the cen-
tral part of the cell in order to analyze heat diffusion during
dynamics:

proc raiseTemperature { temperatureRatio radius } \
{
    In the context of a Btcl script, database simply means a self-con-
tained collection of interrelated tables. A molecular system is a
description of a molecule or molecules as read in from the .car and
.mdf files, possibly modified thereafter by script commands. A
dynamics database is derived in part from a system, but contains
only the information necessary to simulate dynamics (for example,
the fixed atoms are removed). In this example, there is only one
system and one dynamics database, so there is no need for a data-
base name after the dot in the following database handle (page
156) commands:

    # create "handles" to the Molecular System and Dynamics databases
    database handle molsys    system.
    database handle dynamics dynamics.

    Next is an example of retrieving information from a table. The
$handle get (page 156) command (handle being replaced by a han-
dle to a particular database) creates a new object array (third word)
and reads information from a column of a table (fourth word, the
names joined by a dot as Table.Column) into it. An optional fifth
word can be used to specify particular rows of the table; when it is
not present the entire column (that is, all rows of it) are retrieved.

    Also note the temporary use of the echo (page 124) command to
follow the progress of the script during development. The vector
matrix (page 285) command is executed first—this is the effect of
B. Databases and Tables

the brackets (page 127)—and the result (here, a string consisting of the contents of matrix) is printed:

```csharp
# get coordinates of periodic cell
$molsys get matrix Cell.Fractional2Cartesian
echo [vector matrix]
```

The `vector` (page 285) command provides a large selection of arithmetic operations on vectors and matrices. The second word is a new variable name that will be created to hold the result of the operation. The third word is the desired operation, here, the inner (dot) product of matrix with the vector `{ 0.5 0.5 0.5 }`, which as the product of a 3 x 3 matrix and a 3-element vector produces a 3-element coordinate for the center of the periodic cell. (We again use `echo` to show the interim result during script development.)

Note the "{  }" notation in Btcl to represent an array or vector of any desired length. The double quotes are necessary to prevent the braces from being interpreted as a conditional or loop (if, for, etc.) script command:

```csharp
# compute its center
vector center dot $matrix "{ 0.5 0.5 0.5 }"
echo [vector center]
```

The `select` (page 268) command searches a column of a table for a given value or values and returns a list of the rows that contained that value. The third word is the value or an array of values being selected for. The fourth word is the `Table.Column` to look into. The fifth word is the returned array of row numbers; since it is not prefixed by a $ to indicate an existing variable, a new variable is created for the result.

As mentioned above, the database handle `get` (page 156) command can take an optional fifth word, the list of rows whose values should be retrieved. In this case, we use $atom, the list of `Atom` table rows returned by the database handle `select` command, so that we retrieve the atom coordinates (`Atom.Coord`) only for those atoms whose `Movability` column contains 1.

We use the `echoRange` procedure, instead of the `echo` command, to verify that the first few entries of $coord are what we expect. In this example, $coord contains hundreds of entries, so it would not be convenient to print them all:
# get coordinates of all movable (i.e., not fixed) atoms
$molsys select 1 Atom.Movability atom
$molsys get coord Atom.Coord $atom
echoRange $coord 0 10

Similar to the vector command, the geometry (page 189) command can perform a large variety of geometrical calculations on vectors and matrices. The second word is a new variable name that will be created to hold the result of the operation. The third word is the desired operation, in this case the Euclidean distance between (fourth word) the array of the atoms’ 3D Cartesian coordinates and (fifth word) the center of the periodic cell. We again use the echoRange procedure to make part of the interim results visible:

# compute their distance from the cell center
geometry fromCenter distance $coord $center
echoRange $fromCenter 0 10

Among the vector operations are equality and inequality comparisons. Here, we want to know which of the atoms are within $radius angstroms of the periodic-cell center, so we use the comparison operation le (page 288), putting the result into the new variable inSphere:

# make a true/false vector: which are within ”radius” of center
vector inSphere le $fromCenter $radius
echoRange $inSphere 0 10

The object (page 213) command performs operations on the value-arrays or row-lists returned by table commands such as the database handle get or select commands; and many of the operations can also be applied to vectors and matrices. In this example, the object filter command removes any entries in the atom list for which the corresponding entry in the $inSphere list is false. Note that in the object command, the second word has a special syntax: it is never preceded by $.

# prune the list of atoms according to the true/false vector
object atom filter $inSphere

Although we use “database” in the CDiscover context to mean a self-contained collection of interrelated tables, in fact there are sometimes relationships between tables in different databases. For example, the rows of the Coord table in a Dynamics database are derived from the non-fixed atoms in the Atom table in a corre-
B. Databases and Tables

sponding Molecular System databases—each row in the former is related to a specific row in the latter.

CDiscover does not directly support interdatabase relationships. The next two lines, using the object cast command are a workaround to convert Atomtable row numbers in the Molecular System database to their corresponding Coord table row numbers in the Dynamics database. The numbers are converted from row identifiers to integers, then back to row identifiers, but in a different database and table. You do not need to do this often, but with knowledge of the correspondences between tables in separate databases, you can make reliable conversions by this method:

# convert the list of atoms in the Molecular System database to a list of rows in the Coord table in the Dynamics database
object dynCoord get $atom
object dynCoord cast int
object dynCoord cast -database $dynamics rid Coord

The remaining operations in the procedure should be easy to understand, since most are variations of the commands described above. The new elements are the built-in sqrt (page 129) function, and the database handle set (page 156) command, whose syntax is similar to that of the database handle get command:

# get the current velocities
$dynamics get velocity Coord.Velocity $dynCoord
echoRange $velocity 0 10

# temperature is proportional to the square of the velocity
$velocityRatio = sqrt($temperatureRatio)

# multiply by velocityRatio and set results back into the table
vector velocity multiply $velocity $velocityRatio
echoRange $velocity 0 10
$dynamics set $velocity Coord.Velocity $dynCoord

Next, the execution of CDiscover dynamics begins (a begin command is not needed, because the Insight interface adds one before it sources this script), after the above procedures have been read in and stored for later use. For this example, the most interesting point below is the execute +before command = option in the dynamics (page 167) command. It is here that the raiseTemperature procedure is invoked before the beginning of the dynamics simulation. [frequency = 0 means that the command is not
repeated during the simulation; the next line, **execute frequency = 50** is an example of a Btcl command that is to be executed at intervals, in this case every 50 femtoseconds.]

```tcl
# run dynamics, executing the raiseTemperature{} procedure
# before beginning ("frequency = 0" means that the procedure
# will not be repeated during dynamics)
dynamics \
  time = 100.0 timestep = 0.25 \
  execute +before frequency = 0 command = { raiseTemperature 16 5.0 } \
  execute frequency = 50 command = { print history } \
  initial_temperature = 298.0 +boltzmann \n  ensemble = nve \n  deviation = 500000

writeFile coordinate filename = .cor
```

---

**Example—Reading and writing machine-independent binary databases**

Here is how to write a database:

```tcl
set f [open "$name" w]
database handle dbH
database save $dbH $f
close $f
```

and then to read it:

```tcl
set f [open "$name"]
database open dbH $f
close $f
```

where *name* is the filename that you want.

You can read compressed databases by replacing the open statement with:

```tcl
set f [open "|gunzip -c $name.gz"]
```

or whatever the filename is.
B. Databases and Tables
Discover Interprocess Communication (IPC)

Purpose

The `ipc` command and related commands (see Background) provide the means to exchange data and Tcl code between CDiscover and user programs running on the same or different hosts.

Background

Through the IPC package, CDiscover can establish bidirectional communication with external FORTRAN/C/C++ programs written by Discover users. The purpose of this is to provide scientific application developers an efficient means of exchanging information such as atom coordinates, energies, energy gradients, and other information between CDiscover and their own programs. This is accomplished without linking code into CDiscover: communication is implemented via named pipes for the local host and via TCP/IP for remote connections. Some of the benefits and types of IPC applications which can be developed include:

- Algorithms that modify the behavior of the standard Discover program, e.g., modified minimizers, dynamics.
- Programs that add a special energy, gradient, stress, and/or Hessian to the normal Discover energy evaluation.
- Programs that use the Discover program as an “engine” for computing useful molecular mechanics properties.

In addition to communicating with user programs, CDiscover can also be programmed to communicate with additional instances of
C. Discover Interprocess Communication (IPC)

itself. This allows for a simple implementation of network parallelism.

---

IPC components

The IPC capabilities of the CDiscover program have two components: the Discover Btcl interface and the client application program interface (API).

Discover Btcl IPC commands

ipc procedure

Primary access to Discover IPC is through the ipc procedure (see ipc procedure). This procedure is used in Btcl (run_name.inp) scripts to initiate IPC connections to other programs.

object varName read/write operations

The read and write operations of the object command allow common Btcl objects to be read or written via the IPC connection.

discoverIpc command

This command implements the basic IPC functions for the Discover program. In general, you do not need to use this command directly; all common IPC functionality is encapsulated in the ipc procedure. However, for some users, the discoverIpc command allows more flexible configuration of the Discover program’s IPC capabilities.

discoverIO command

As with the discoverIpc command, discoverIO (see Interprocess communication for Discover—discoverIO) is generally not used directly by the script writer. During a Discover run’s energy evaluation, the discoverIO command is used to coordinate fast exchanges of atomic coordinates and energy-related values between the Discover program and an external program.
MSI IPC client application program interface

The client API consists of a small number of C and FORTRAN functions which client programs can use to initiate an IPC connection to the Discover program and then exchange data. The complete source code for the client-side IPC is provided without restrictions on its use and with examples.

ipc procedure

The `ipc` procedure is implemented in Tcl and encapsulates the Discover program’s IPC capabilities into a single procedure.

Note: The distinction between a command and a procedure in Tcl is in how the Tcl verb is implemented. Verbs implemented in C code are usually called commands, while verbs implemented as a body of Tcl are called procedures. Syntactically, a command and a procedure are identical, and so we will drop the distinction and just refer to the “ipc command”.

Syntax

The `ipc` command is similar in structure to the `object` command:

```
ipc varName operation ?-option ...? args ?args ...?
```

`varName` is the name of a Tcl variable that is used as a reference to the IPC object in question. The `varName` Tcl variable set by the `ipc varName init ...` command; subsequent IPC operations use this variable to determine which IPC connection to operate on.

For many `ipc` commands, `varName` may be a Tcl variable that refers to an IPC object or a Tcl file handle. Where the command behavior differs based on whether the handle is an IPC object or Tcl file handle, the descriptions of each operation will indicate.

`operation` is one of the following (which are described in more detail below):

*Common IPC operations*

- `init`

  Initializes an IPC connection to another program.
C. Discover Interprocess Communication (IPC)

info
Determine characteristics of an IPC connection.

shutdown
Close an IPC connection.

Advanced IPC operations

accept
Accept a connection request on a listening socket.

connect
Open a connection to a TCP/IP host/port.

eval
Get data from an IPC connection and evaluate it as a Btcl command string.

listen
Start a listening socket so that other programs including other instances of the Discover program can connect to this running Discover.

poll
Determine if input is available on a connection or a connection is pending on a listening socket.

send
Send a raw string over an IPC connection.

server
Put the Discover program into “server” mode.
Common IPC operations

init

`ipc varName init ?-options ...? ?args ...?`

Used to establish a link between the Discover program and another program. The external process may be running on the same host or a remote host. The external process may be another instance of Discover. This mechanism can be used to implement network parallelism.

Generally, the external process uses the BiosymIpc client library to establish a connection back to the Discover program. As soon as the external process calls the `BiosymIpc_Init` C function or the `BiosymIpc_Initf` FORTRAN function, the connection is established and the Discover program moves on to the next Btcl command.

The `args` specify the external process to run. These arguments are passed to the Tcl `exec` command. The input and output of the external process can be redirected to specific files.

Upon establishing the connection, a handle is created in the caller’s variable named `varName`. This variable must be used in all subsequent operations on the newly created IPC channel. When `varName` is unset, either explicitly or by going out-of-scope, the IPC connection is shut down.

options

`-autoEnergy atomSpecification`

A common use for an external program is to add an energy component to the Discover energy evaluation routine. The `-autoEnergy` option takes care of the details involved in establishing this external energy contribution using the Btcl `restraint` command. During an energy evaluation, the coordinates of the atoms specified in `atomSpecification` are transferred to the external program. The Discover program then expects to receive an energy contribution, gradients, and possibly other information. See the `discoverIO` command for more details.
C. Discover Interprocess Communication (IPC)

-connect host:port | host port

Directs the currently running Discover program to establish an IPC connection to the specified host and port, rather than invoke a new executable. The Discover program generally uses this when it is invoked by another instance of Discover to act as a server. See Advanced IPC operations for details on other ways to use the -connect option to the init operation.

-energyName string

Optionally used in conjunction with -autoEnergy to provide a name for the energy contribution given by the external process. This name will appear in the Energy database under the major heading “External”.

-port port

Optionally used in conjunction with the -connect option to specify a port number to attempt to connect to.

-remote hostName

This option specifies that the external process is to be run on a remote host. You must have permission to perform an rsh to the specified remote host.

info

ipc varName info ?-option ...?

Returns information about the IPC connection specified. If no options are given, all information is returned. Use info with no options to see what options are available. The data for each IPC connection are stored in a global associative array called IPC. The keys into this array consist of a unique “handle” name for each
connection, concatenated with a period and a member name as follows:

<table>
<thead>
<tr>
<th>Keys to IPC array</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>IPC(handle.inFH)</td>
<td>Discover input Tcl input file handle.</td>
</tr>
<tr>
<td>IPC(handle.outFH)</td>
<td>Discover input Tcl output file handle.</td>
</tr>
<tr>
<td>IPC(handle.remoteVersion)</td>
<td>BiosymIpc version in use by external.</td>
</tr>
<tr>
<td>IPC(handle.localVersion)</td>
<td>BiosymIpc version in use by Discover.</td>
</tr>
<tr>
<td>IPC(handle.restraint)</td>
<td>optional energy restraint object.</td>
</tr>
<tr>
<td>IPC(handle.command)</td>
<td>external program invocation.</td>
</tr>
<tr>
<td>IPC(handle.atomList)</td>
<td>optional atom specification.</td>
</tr>
</tbody>
</table>

**shutdown**

ipc varName shutdown

Closes the specified IPC connection. The same effect can be achieved with unset varName.

**Advanced IPC operations**

Many of the following operations are used internally by the ipc command. Although documented here, most users will not find it necessary to use these operations unless they are developing very special-purpose Discover communication tasks.

**accept**

ipc varName accept ?hostname?

Accept a connection request on a previously opened listening socket specified by varName. varName may contain a normal Tcl file handle or an IPC object handle. If an IPC object is specified, the inFH (input file handle) member of the IPC object is used to refer to the listen socket.
C. Discover Interprocess Communication (IPC)

If the optional hostname is present, the peer attempting to connect must have originated from the specified host.

The result is an ordered list of two Tcl file handles. The first handle is to be used for reading on the connection; the second handle is to be used for writing on the connection.

**connect**

`ipc varName connect hostname port`

Attempt to make a connection to the specified host and TCP/IP port. The port can be a service name or a numerical port specification. The result is an ordered list of two Tcl file handles. The first handle is to be used for reading on the connection; the second handle is to be used for writing on the connection.

**eval**

`ipc varName eval ?-nowait?`

Read input from an IPC handle or Tcl file handle and evaluate the incoming data as a Btcl command. Normally, the eval operation waits until input is available on the specified connection. If the -nowait option is specified, the command returns an empty string if no input is available on the specified connection.

The result that is returned is the result of the evaluation of the input data.

The incoming data is handled differently depending on whether varName contains an IPC handle or a simple Tcl file handle:

**IPC handle**

The actual Tcl file handle used for input is obtained from the `inFH` member of the IPC object referred to. (Use `ipc varName info inFH` to obtain the input Tcl file handle for an IPC object.) The file handle is read and interpreted as an IPC frame which must be of type STRING. The strings in the incoming IPC frame are concatenated and then evaluated in the Btcl interpreter (via Tcl’s eval command).

**Tcl file handle**

A single line of input is read. If the result is not a complete Tcl command, then additional lines are read until a complete command.
has been composed. The input is then evaluated in the Btcl interpreter (via Tcl’s `eval` command).

---

#### listen

`ipc varName listen ?port?`

Establish a listen socket. If the optional port number is given, then an attempt to bind the listen socket to that port is made.

The result is an ordered list of length two. The first list element is a readable Tcl file handle which can be used to poll the listen socket. The second list element is the port number actually assigned to the listen socket. Typically, this is transmitted to an external program via an environmental variable or command line option in order to let the external program know where it should connect back to upon startup.

---

#### poll

`ipc varName poll ?timeoutMilliseconds?`

Determine if input is available or a connection is pending. The input source is either an IPC handle or a Tcl file handle contained in `varName`. If the optional timeout value is given, the `poll` command will wait up to the specified number of milliseconds before returning.

The result is 1 if data is available for reading or a connection request has appeared, and 0 if not.

---

#### send

`ipc varName send ?-nonewline? ?args?`

Write strings (usually Btcl/Tcl commands) to the IPC handle or Tcl file handle contained in `varName`. Normally, `args` is a Tcl list of strings. Each string is sent with a newline appended unless the `-nonewline` option is used.

There is no returned result (empty string).
C. Discover Interprocess Communication (IPC)

The outgoing data is handled differently, depending on whether `varName` contains an IPC handle or a simple Tcl file handle:

**IPC handle**

The actual Tcl file handle used for output is obtained from the `outFH` member of the IPC object referred to. (Use `ipc varName info outFH` to obtain the output Tcl file handle for an IPC object.) The arguments are packed into a temporary Btcl string object, and the `object varName write` operation is used to send the data.

**Tcl file handle**

Each argument list element is sent as a separate line using the standard Tcl `puts` command.

---

**server**

```tcl
discover IO
```

The `server` operation implements a simple loop using the `eval` command. Input is collected from the IPC handle or Tcl file handle and evaluated until the connection is closed.

---

**Interprocess communication for Discover—discoverIO**

**Purpose**

The `discoverIO` command is used by the `ipc` command primarily to read/write coordinates, energies, gradients, stresses and Hessians from/to a Tcl file handle into/out of the System database and internal energy structures. When the `-autoEnergy` option is used with the `ipc init` operation, a Btcl `restraint` procedure that uses the `discoverIO` command is automatically created. This is used to exchange energy calculation results between an external program and the Discover program’s energy evaluation code.

However, the `discoverIO` command may also be used directly by Btcl experts to implement specialized data exchanges between Discover and other programs running on the network during Discover energy evaluations.
Syntax

discoverIO -read tclFileHandle ?-options args?

discoverIO -write tclFileHandle ?-options args?

Any number of -read tclFileHandle ?-options args? groups may be used on the same discoverIO command to specify several data exchanges in succession.

tclFileHandle

A Tcl file handle referring to a file/pipe/socket. This argument must immediately follow the -read or -write option. The target file must be readable (for -read) or writable (for -write).

Subsequent options and arguments further specify the data to be exchanged as follows:

-atom spec

spec is a string or the name of an object containing references to atoms of interest.

The -atom option allows specification of a collection of atoms to be operated on for data elements (see below) for which this applies (Coord and Gradient). The -atom option affects the previous -read or -write option prior to the appearance of the -atom option (call this previous operation “P”). Additionally, the -atom option affects the operation before “P” (call this “NP”) if:

1. operation NP exists, and
2. an -atom option was not specified already for operation NP, and
3. operation NP is the opposite of operation P in a -read/-write sense

For example, suppose the variables outFH and inFH are Tcl file handles to open channels for writing and reading, respectively. Then, the command:

discoverIO -write $outFH -read $inFH -atom "@:*:Atom;""

writes data (Coord) for all atoms and then reads data (e.g., Gradient) for all atoms. That is, the -atom option refers to the previous -read and also to the -write before that.
C. Discover Interprocess Communication (IPC)

-data list

Allows specification of the logical collections of items to be transferred. Currently recognized list elements are:

   Coord  
   Cartesian2Fractional  
   Fractional2Cartesian  
   CellParameters  
   Energy  
   Gradient  
   Stress  
   Hessian

If no -data option is specified and the operation is -write, the default -data corresponds to -data Coord if the system is nonperiodic and to -data [Cartesian2Fractional Fractional2Cartesian Coord] if the system is periodic.

If Coord is requested and no atom specification is given, then the following coordinates are exchanged:

<table>
<thead>
<tr>
<th>Request</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nonperiodic:</td>
<td>all coordinates</td>
</tr>
<tr>
<td>Periodic: write</td>
<td>all P1 atoms</td>
</tr>
<tr>
<td>Periodic: read</td>
<td>all asymmetric atoms</td>
</tr>
</tbody>
</table>

If Gradient is requested and no atom specification is given, then the following gradients are exchanged:

<table>
<thead>
<tr>
<th>Request</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nonperiodic:</td>
<td>all gradients</td>
</tr>
<tr>
<td>Periodic: write</td>
<td>all asymmetric atom gradients</td>
</tr>
<tr>
<td>Periodic: read</td>
<td>all P1 atom gradients</td>
</tr>
</tbody>
</table>

CellParameters sends a,b,c, alpha, beta, gamma
Cartesian2Fractional and Fractional2Cartesian send over transformation matrices as 9 real*8 values ordered as: 1,1,2,1,3,2,1,2,2, etc.

-expect ?tag?

The -expect option applies only to the immediately previous -read or -write.

If -expect option is used, then discoverIO either sends a string-type IPC frame (if the -expect applies to a -write) containing the list of items the client should expect next; or, if the -expect applies to a -read, discoverIO expects a string IPC frame containing the names of the data items that the Discover program should expect in forthcoming IPC frames.

The optional tag argument specifies a message entry for the IPC frame containing the expected data. This tag is used on -write and expected on -read. If no tag is provided, the tag Expect is sent or expected. If the tag must have a leading hyphen, use two hyphens (--).

Outgoing Expect frames always generate a flush.

-request ?tag?

The -request option applies only to the immediately prior -read. However, it sends its information out via the previous -write Tcl file handle.

The -request option is similar to the -expect option in that it helps the Discover program and clients negotiate what information is to be exchanged. If -request is specified, then before waiting for the data specified in the -read, Discover sends a Request message with strings indicating what information is desired from the client. If no -expect option applies to the -read in question, then Discover waits for the information requested. If an -expect does apply to this read, then before waiting for the data, Discover waits for an Expect message from the client. The contents of the Expect message from the client may modify what data Discover attempts to read next.

The optional tag argument specifies a message entry for the IPC frame containing the request data. If no tag is provided, the tag Request is sent. If the tag must have a leading hyphen, use two hyphens (--).
Request frames always generate a flush.

-energyName name

Specifies that a new child of the Energy database “External” energy row be created (in the table Values, column Name). The incoming energy components are made children of this new name, instead of direct children of the “External” row.

If -energyName is absent, then a new child of the “External” row is created using the name of the tclFileHandle specified on the associated -read action.

For example, if -energyName MOPAC appears, then the Energy database will look like:

```
  External         ...
  MOPAC           ...
```

All IPC frames with the message “Energy” will be summed into the MOPAC row. Furthermore, if “Energy” IPC frames are returned from the remote application with a second word in the message, then these distinguished energies are made children of the energy row specified by -energyName. For example, suppose -energyName MOPAC appears (as before) and that the application sends back two energy IPC frames with messages:

```
  "Energy firstPart"
  "Energy secondPart"
```

Now, the Energy database is set up as follows:

```
  External         ...
  MOPAC           ...
  firstPart       ...
  secondPart      ...
```

When the energies are printed, “firstPart” and “secondPart” are summed into “MOPAC”, which will in turn be summed with any other rows at that level of the hierarchy, into “External”. Note that, because the “MOPAC” row now has children, no energy IPC frames should arrive with only an “Energy” message; if this happens, the energies stored into “MOPAC” are overwritten when the children (“firstPart” and “secondPart”) are summed into “MOPAC”.

-energyName applies to the previous read action.
Tcl results

If the -read option is specified, the Tcl result is a list of the message records from each input IPC frame.

If the -write option is specified, the Tcl result is a list of the message records sent for each output IPC frame.

discoverIpc

Purpose

Used primarily by the ipc command to perform low-level inter-process communication operations. This command may be used directly by Btcl experts to implement specialized communications between the Discover program and other programs running on the network.

Syntax

discoverIpc operation ?args?

The possible operations are as follows:

eval

Purpose

Reads Btcl commands from tclFileHandle and evaluates them in the global scope.

Syntax

discoverIpc eval ?-nowait? tclFileHandle

?-nowait?

Option not to wait if no input is available. The default is to block until input is available.

tclFileHandle

A Tcl file handle referring to a file/pipe. The target file must be readable.

poll

Purpose

Determine if the file descriptor associated with the Tcl file handle has readable data after an optional timeout.
C. Discover Interprocess Communication (IPC)

Syntax

discoverIpc poll tclFileHandle ?timeoutMilliseconds?

tclFileHandle
A Tcl file handle referring to a file/pipe. The target file must be readable.

timeoutMilliseconds
Optional amount of time to wait for input to become available. If absent, or 0, do not wait.

---

**connect**

**Purpose**
Connect to a remote host/process and return a Tcl file handle opened for reading and writing on the connection.

**Syntax**

discoverIpc connect hostName portName

**hostName**
Host where remote process is listening for a connection. An empty hostName means to use “localhost”.

**portName**
Port number or service name of remote process. The result is an ordered list of two Tcl file handles. The first handle is to be used for reading on the connection; the second handle is to be used for writing on the connection.

---

**listen**

**Purpose**
Establish a listening socket on a specified or system-chosen port.

**Syntax**

discoverIpc listen ?portNumber?

**portNumber**
Port to which listen socket should be bound. The result is a list of length 2. The first element is a Tcl file handle connected to the listen socket. This can be used with the poll command to see if a connection on that socket is being requested. The second element is a
copy of the port number assigned for listening. This will be the same as the specified port if the bind was successful.

accept

Purpose
Accept a connection request on a listening socket specified by the Tcl file handle. If the connection is successfully accepted and created, a new Tcl file handle is created. The Tcl file handle for the read/write connection is returned.

Syntax
discoverIpc accept tclFileHandle ?hostname?

tclFileHandle
File handle on which to try the accept.

hostname
Optional remote host description. If present, this argument requires that the peer have the host description/name specified. The result is an ordered list of two Tcl file handles. The first handle is to be used for reading on the connection; the second handle is to be used for writing on the connection.

MSI IPC C application program interface

Background
This is the C language interface to the MSI interprocess communication facilities implemented by the Discover program. A FORTRAN interface is also available (MSI IPC FORTRAN application program interface). See MSI IPC client application program interface for an overview of the Discover program’s use of the IPC facilities.

MSI IPC allows long, double, and character data to be exchanged. All data exchanges are done via IPC Frames which consist of a header describing the data which follow.

To use these routines you need to compile the file ipc.c (C) and link the resulting object (ipc.o) with your program. Note that ipc.c also requires the file ipc.h.
C. Discover Interprocess Communication (IPC)

**Parallel Note:** On shared-memory parallel machines, use of these functions must be single-threaded and locked to one node, due to static data structures.

### C functions for IPC

#### Biosym IPC_Init

**Function**

Initialize the IPC package. Provide input and output file names. If these are not specified (given as NULL), use the values in the environmental variables BiosymIpcIn and BiosymIpcOut.

The order in which the files are opened is important, since the Discover process that opens these named pipes at the other end does so in this order.

```c
int BiosymIpc_Init(char *inputFilename, char *outputFilename)
```

#### Biosym IPC_Shutdown

**Function**

Close the file handles associated with the Discover IPC connection.

```c
void BiosymIpc_Shutdown(void)
```

#### Biosym IPC_Read

**Function**

Read a data entry from the inputFile filehandle. This function blocks until data is available (i.e., until the Discover program needs work from the client). If end-of-file is detected (i.e., Discover has exited or shut down the connection), the endOfFile variable is set to 1 and a 0 is returned. This signals that the client should then shut itself down.

Upon entry, “data” must be non-NULL and point to a BiosymIpc-Data structure; “endOfFile” must be non-NULL and point to an integer.

Upon successful return, the “data” union member of the BiosymIpcData structure points to allocated memory containing the data received. It is the caller’s responsibility to free this storage when it is no longer needed.

```c
int BiosymIpc_Read(BiosymIpcData *data, int *endOfFile)
```
**MSI IPC C application program interface**

**BiosymIpc_Write**

*Function* Write a data entry to the outputFile filehandle. The last data block of a session should flush by setting the flush flag to 1. Flushing the data is necessary to ensure that the Discover program receives it.

```c
int BiosymIpc_Write(BiosymIpcData *data, int flush)
```

**BiosymIpc_Print**

*Function* ASCII print the contents of a BiosymIpcData structure. This is intended primarily for debugging purposes.

```c
int BiosymIpc_Print(BiosymIpcData *data, FILE *fp)
```

**BiosymIpc_VarWrite**

*Function* Package a variable-length list of strings to be sent via IPC.

```c
int BiosymIpc_VarWrite(char *message, int flush, char *p, ...)
```

*message:* string to place in IPC frame message header field

*flush:* if non-zero, flush after done

*p:* one or more strings to send

*Return value* 0 on success, nonzero on failure.

**BiosymIpc_WriteRawBtcl2**

*Function* Send all strings from a variable-length list to the IPC output channel. Newlines are optionally placed after each string; a final flush is optionally performed. An IPC frame is not constructed.

```c
int BiosymIpc_WriteRawBtcl2(int noNewLine, int flush, char *p, ...)
```

*noNewLine:* if non-zero, do not append newlines to strings

*flush:* if non-zero, flush after done

*p:* one or more strings to send

*Return value* 0 on success, nonzero on failure.

**BiosymIpc_WriteRawBtcl**

*Function* Send all strings from a variable-length list to the IPC output channel. Newlines are placed after each string; a final flush is performed. An IPC frame is not constructed.

```c
int BiosymIpc_WriteRawBtcl(...)```
C. Discover Interprocess Communication (IPC)

Return value 0 on success, nonzero on failure.

**BiosymIpc_SetChannel**

*Function* Specify a new pair of FILE pointers for the BiosymIpc functions. The previous input and output FILE pointers are returned through the addresses provided.

**Note:** to open more than one channel, the procedure is as follows:

```c
{ 
    FILE *fpIN = NULL; 
    FILE *fpOUT = NULL; 
    BiosymIpc_SetChannel(&fpIN, &fpOUT); 
    BiosymIpc_Init("newInputFile", "newOutputFile"); 
}
```

That is, the IPC channel pointers must be set to NULL, NULL before initializing a new channel.

```c
void BiosymIpc_SetChannel(FILE **in, FILE **out)
```

**BiosymIpc_GetError**

*Function* Return a pointer to the local error message buffer.

```c
char *BiosymIpc_GetError(void)
```

**BiosymIpc_GetLocalVersion**

*Function* Return a pointer to the version of BiosymIpc being used in the program.

```c
char *BiosymIpc_GetLocalVersion(void)
```

**BiosymIpc_GetRemoteVersion**

*Function* Return a pointer to the version of BiosymIpc being used in the remote program.

```c
char *BiosymIpc_GetRemoteVersion(void)
```
Background

This is the FORTRAN interface to the MSI interprocess communication facilities implemented by the Discover program. A C Language Interface is also available (MSI IPC C application program interface). See MSI IPC client application program interface for an overview of the Discover program’s use of the IPC facilities.

MSI IPC allows INTEGER, REAL*8, and CHARACTER data to be exchanged. All data exchanges are done via IPC Frames, which consist of a header describing the data which follow.

The routines to write data out (usually to Discover) consist of single function calls. The arguments to the functions describe the data to be sent.

The routines to read data consist of two calls. The first call is always “BiosymIpc_ReadHeaderf”, which returns information about the data that will follow. The program uses this information to make the appropriate follow-up call that puts the transferred data into a local array.

The few remaining calls perform tasks such as initializing the connection, closing the connection, and determining version information.

Many read and write functions contain numItem and numElement arguments. These arguments are merely descriptive of how the data is organized. For example, coordinates are a common thing to exchange. The Discover program will send coordinates as collections of (x,y,z) with numElement == 3 and numItems == the number of points to be transferred.

To use these routines you need to compile the file ipcf.f (FORTRAN) and the file ipc.c (C). Both resulting objects (ipcf.o and ipc.o) must be linked with your program. Note that ipc.c requires the file ipc.h, and ipcf.f requires the file ipcf.inc.

Parallel Note: On shared-memory parallel machines, use of these functions must be single-threaded and locked to one node, due to static data structures.
C. Discover Interprocess Communication (IPC)

Subroutines

BiosymIpc_Initf

SUBROUTINE BiosymIpc_Initf()

Initialize an IPC connection. There is no input or output to the routine. The environmental variables BiosymIpcIn and BiosymIpcOut are set up by the Discover program before the client program is executed; these are used to establish the connection.

BiosymIpc_ReadHeaderf

SUBROUTINE BiosymIpc_ReadHeaderf(type, numItem, numElement, numByte, message, messageMax, messageLen, endOfFile, status)

Read an IPC frame header.

Input

INTEGER messageMax
maximum characters to transfer from message in IPC frame to message character variable

Output

INTEGER type
data type, 1 = integer
2 = real*8
3 = string

INTEGER numItem
number of items in data block

INTEGER numElement
number of elements per item

INTEGER numByte
total bytes in data block
CHARACTER*(*) message
    message from IPC frame
INTEGER messageLen
    actual character length of message, but not greater than messageMax
INTEGER endOfFile
    1 if end of file occurred
INTEGER status
    1 is OK, 0 is error

SUBROUTINE BiosymIpc_ReadString128f(string, iString, iiString, status)

Read an IPC string frame into an array of character*128 variables.

Input
   INTEGER iString(*)
       Work space to hold input characters. Must be dimensioned to hold all input characters including NULs at the end of each string.
   INTEGER iiString(*)
       Work space to hold string offsets into iString. Must be dimensioned to the largest number of strings that will be read.

Output
   CHARACTER*128 string(*)
       An array of strings each of which is no longer than 128 characters.
   INTEGER iiString(*)
       Contains the length of each string.
   INTEGER status
       1 is OK, 0 is error.
C. Discover Interprocess Communication (IPC)

**BiosymIpc_WriteIntegerf**

```plaintext
SUBROUTINE BiosymIpc_WriteIntegerf(data,
  &     numItem,
  &     numElement,
  &     message,
  &     messageLen,
  &     flush,
  &     status)

Write an IPC frame of integers.

**Input**

- **INTEGER data(*)**
  
  Array containing integers to be sent.

- **INTEGER numItem**
  
  The number of logical groups of integers in data.

- **INTEGER numElement**
  
  The number of integers in each logical group in data.

- **CHARACTER(*) message**
  
  A message to place in the outgoing IPC frame header.

- **INTEGER messageLen**
  
  Number of valid characters in message.

- **INTEGER flush**
  
  1 means flush the output pipe;
  0 means do not flush the output.

**Output**

- **INTEGER status**
  
  1 is OK, 0 is error.

**BiosymIpc_WriteReal8f**

```plaintext
SUBROUTINE BiosymIpc_WriteReal8f(data,
  &     numItem,
  &     numElement,
  &     message,
  &     messageLen,
  &     flush,
  &     status)
```
Write an IPC frame of real*8 values.

**Input**

- **REAL*8** data(*)
  - Array containing integers to be sent.
- **INTEGER** numItem
  - The number of logical groups of integers in data.
- **INTEGER** numElement
  - The number of integers in each logical group in data.
- **CHARACTER**(*) message
  - A message to place in the outgoing IPC frame header.
- **INTEGER** messageLen
  - Number of valid characters in message.
- **INTEGER** flush
  - 1 means flush the output pipe;
    0 means do not flush the output.

**Output**

- **INTEGER** status
  - 1 is OK, 0 is error.

**BiosymIpc_WriteStringf**

```fortran
SUBROUTINE BiosymIpc_WriteStringf(data, &
   numItem, &
   numElement, &
   message, &
   messageLen, &
   flush, &
   status)
```

Write an IPC frame of arbitrary-length strings.

**Input**

- **INTEGER** data(*)
  - Array of integers containing the characters of the strings
    packed together with each string 0 terminated.
- **INTEGER** numItem
  - The number of logical groups of integers in data.
C. Discover Interprocess Communication (IPC)

INTEGER numElement
    The number of integers in each logical group in data.

CHARACTER(*) message
    A message to place in the outgoing IPC frame header.

INTEGER messageLen
    Number of valid characters in message.

INTEGER flush
    1 means flush the output pipe;
    0 means do not flush the output.

Output
    INTEGER status
        1 is OK, 0 is error.

BiosymIpc_WriteString128f

SUBROUTINE BiosymIpc_WriteString128f(string,
&     lenString,
&     iString,
&     numItem,
&     numElement,
&     message,
&     messageLen,
&     flush,
&     status)

Write a series of char*128 variables in strings with lengths given in
the corresponding lenString.

Input
    CHARACTER*128 string
        Array of character*128 values to be written in an IPC frame.

    INTEGER lenString(*)
        Number of characters to use from each corresponding entry in
        string.

    INTEGER iString(*)
        Buffer (must have space for at least the total number of charac-
        ters (sum of lenString) plus nString.
MSI IPC FORTRAN application program interface

INTEGER numItem
   Number of groupings.
INTEGER numElement
   Number of elements per item.
   numItem * numElement is the total number of strings.
CHARACTER(*) message
   Message tag for output IPC header.
INTEGER messageLen
   Number of characters in message.
INTEGER flush
   Nonzero means flush output stream.
INTEGER status
   1 = no error, 0 = error.

SUBROUTINE BiosymIpc_WriteBtcl128f(command,
   & nCommand,
   & noNewline,
   & iString,
   & flush,
   & status)

Write a series of char*128 variables in command. The strings will be interpreted by Discover as a series of Btcl commands to be executed. Discover concatenates all the strings into a single string and evaluates this.

Discover must be waiting in an evaluation loop, collecting IPC frames, and evaluating the contents. See the Discover IPC documentation for how to do this in a Discover script.

Input

   CHARACTER*128 command(*)
      Array of character*128 values to be written in an IPC frame.
   INTEGER nCommand
      Number of command strings in command.
C. Discover Interprocess Communication (IPC)

INTEGER noNewline
Normally each string is given a newline so that it corresponds to one Btcl command; if noNewline is nonzero, newline characters are not appended to the strings. This allows commands longer than 128 characters to be built up.

INTEGER iString(*)
Buffer (must be dimensioned at least 130*nCommand = nCommand*128 + 2*nCommand to accommodate all string data plus possible newlines for each command and a mandatory 0 terminator).

INTEGER flush
Nonzero means flush output stream.

INTEGER status
1 = no error, 0 = error.

BiosymIpc_WriteRawBtcl128f

SUBROUTINE BiosymIpc_WriteRawBtcl128f(command, & nCommand, & noNewline, & iString, & flush, & status)

Write an array of character*128 variables which should contain Btcl code. The write is done without an IPC frame header. Discover must be waiting in an evaluation loop, collecting lines, and evaluating them as they form complete commands. See the Discover IPC documentation for how to do this in a Discover script.

The strings are concatenated into one large string for writing as indicated by the noNewline flag.

Input

CHARACTER*128 command(*)
Array of character*128 values.

INTEGER nCommand
Number of strings in command.

INTEGER noNewline
MSI IPC FORTRAN application program interface

If 1, do not place newlines between strings: just concatenate and output one big string;
if 0, place newline characters between each string and after the last string and output the concatenated string.

INTEGER iString(*)
Buffer (must be dimensioned at least 130*nCommand = nCommand*128 + 2*nCommand to accommodate all string data plus possible newlines for each command and a mandatory 0 terminator).

INTEGER flush
Nonzero means flush output stream.

INTEGER status
1 = no error, 0 = error.

**BioSymIpc_GetError**

SUBROUTINE BioSymIpc_GetErrorf(err, maxChar)
Return the last BioSymIpc error message.

*Input*
INTEGER maxChar
The maximum number of characters that can be transferred back into the “err” variable.

*Output*
CHARACTER(*) err
Contains the error string.

**BioSymIpc_GetLocalVersion**

SUBROUTINE BioSymIpc_GetLocalVersionf(ver, maxChar)
Return the version string for the BioSymIpc code being used in this program.

*Input*
INTEGER maxChar
The maximum number of characters that can be transferred back into the “ver” variable.
C. Discover Interprocess Communication (IPC)

**Output**

CHARACTER(*) ver

Contains the version string.

**BiosymIPC_GetRemoteVersionf**

SUBROUTINE BiosymIPC_GetRemoteVersionf(ver, maxChar)

Return the version string for the BiosymIPC code being used by the remote program.

**Input**

INTEGER maxChar

The maximum number of characters that can be transferred back into the “ver” variable.

**Output**

CHARACTER(*) ver

Contains the version string.

**BiosymIPC_Int2Charf**

SUBROUTINE BiosymIPC_Int2Charf(c, ia, n, l)

Utility to convert a string from a zero-terminated array of integers into a character variable.

**Input**

INTEGER ia(*)

Zero-terminated integer array of characters.

INTEGER n

Largest number of integers/chars to transfer.

**Output**

CHARACTER(*) c

Filled to n characters and space padded.

INTEGER l

Filled with the actual number of nonzero characters or n if the limit is reached first.

**BiosymIPC_Char2Intf**

SUBROUTINE BiosymIPC_Char2Intf(ia, c, n)
Utility to convert a character variable into a zero-terminated array of integers.

**Input**

- CHARACTER *(*) c
  A character variable.
- INTEGER n
  The largest number of chars to transfer into ia.

**Output**

- INTEGER ia(*)
  Filled to n integers. ia(n) is always 0.

**BiosymIpc_ReadIntegerf**

```fortran
SUBROUTINE BiosymIpc_ReadIntegerf(data, & status)
Read an IPC integer frame into an array of integers.
```

**Output**

- INTEGER data(*)
  Incoming integers.
- INTEGER status
  1 is OK, 0 is error.

**BiosymIpc_ReadReal8f**

```fortran
SUBROUTINE BiosymIpc_ReadReal8f(data, & status)
Read an IPC 8-byte real frame into an array of 8-byte reals.
```

**Output**

- REAL*8 data(*)
  Incoming real*8 values.
- INTEGER status
  1 is OK, 0 is error.

**BiosymIpc_ReadStringf**

```fortran
SUBROUTINE BiosymIpc_ReadStringf(data, & status)
```
C. Discover Interprocess Communication (IPC)

Read an IPC string frame into an array of integers representing the strings. Use BiosymIpc_Int2Charf to convert to FORTRAN character variables.

Output

INTEGER data

An array where all the incoming characters are stored as integers. Each individual string is separated by a 0 integer.

INTEGER index

An array of Fortran indices indicating the starting positions of each string in data.

INTEGER retNString

The number of strings read in from the IPC frame.

INTEGER status

1 is OK, 0 is error.

MSI IPC C language structures

Background

The following describes the data structures used by the MSI IPC Application Program Interfaces. The file ipc.h must be used at the top of a client C language program using #include "ipc.h" in order to define prototypes and structures used by the C API.

IPC frames

MSI IPC data is exchanged using a simple protocol consisting of a header followed by (binary) data. The following define the keywords used in IPC frame headers:
The header keywords have the following meanings:

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>#</td>
<td>Comment character; ignore line.</td>
</tr>
<tr>
<td>BiosymIpcVersion</td>
<td>Indicates the version of IPC protocol used.</td>
</tr>
<tr>
<td>Message</td>
<td>Arbitrary message, (&lt;BIOSYM_IPC_MESSAGE_MAX) bytes, currently 128.</td>
</tr>
<tr>
<td>DataType</td>
<td>See table below.</td>
</tr>
<tr>
<td>NumItem</td>
<td># of distinct items, e.g., coordinate triples.</td>
</tr>
<tr>
<td>NumElement</td>
<td># of data per item, e.g. 3 for coordinates.</td>
</tr>
<tr>
<td>NumByte</td>
<td>Total # of bytes in the subsequent data block.</td>
</tr>
<tr>
<td>EndBiosymIpcHeader</td>
<td>Binary data begins immediately after the newline.</td>
</tr>
</tbody>
</table>

No ordering is imposed on the header information with the exception that EndBiosymIpcHeader must appear last. Not all header records are needed for a particular IPC frame; the byte count will not appear for non-string data.

The following enumerated type is used to specify the type of data to be exchanged:

```c
typedef enum {
    BiosymIpcType_Integer, /* long */
    BiosymIpcType_Real8, /* double */
    BiosymIpcType_String
} BiosymIpcType;
```

The following define the keyword strings corresponding to each BiosymIpcType:

```c
#define BIOSYM_IPC_TYPE_INTEGER "integer"
#define BIOSYM_IPC_TYPE_REAL8 "real8"
#define BIOSYM_IPC_TYPE_STRING "string"
```
C. Discover Interprocess Communication (IPC)

The following define various constants and limits used in the MSI IPC code.

```c
#define BIOSYM_IPC_VERSION_CUR "94.0"  /* max length of remove version string */
#define BIOSYM_IPC_VERSION_LEN 64       /* max length of remove version string */
#define BIOSYM_IPC_MESSAGE_MAX 128      /* max IPC frame message length */
#define BIOSYM_IPC_HEADER_MAX 256       /* max length of any ASCII header line */
```

The BiosymIpcData structure encapsulates all the information needed to encode an IPC frame to be sent. Conversely, a BiosymIpcData structure is filled when an incoming IPC frame is decoded.

```c
typedef struct {
    char message[BIOSYM_IPC_MESSAGE_MAX];
    BiosymIpcType type;    /* Data type for this entry */
    long numItem;          /* # of items for this entry */
    long numElement;       /* # of data elements per item (e.g., 3 for coordinates) */
    long numByte;          /* # of bytes total in data block (only needed for strings) */
    union {
        void *data;
        long *integer;
        double *real8;
        char **string;
    } data;
} BiosymIpcData;
```
**Files**

**Introduction**

This appendix explains the purpose and format of the files used by CDiscover. “Generic” files (i.e., files used by several MSI programs) are described in separate File Formats documentation at our website:

http://www.msi.com/doc/

Files specific to CDiscover are described in this Appendix. Samples of some files are included in this appendix. Since they are used or output by several programs, generic files may contain some information that is irrelevant to CDiscover.

**Input files**

- Structural Data File (.mdf) ......................... (generic)
- Cartesian Coordinate File (.car) ....................... (generic)
- Restraints File (.rstrnt) ........................... (generic)
- Command input file (.inp)
- Forcefield parameter file (.frc)

**Output files**

- Cartesian Coordinate Archive File (.arc) ............. (generic)
- Output Coordinate File (.cor) .......................... (generic)
- Standard output file (.out)
- CDiscover output files (.arc, .xfrc, .out, .tbl, .xdyn, user-named)
- Discover dynamics restart files (.xdyn)
- Automatic potential parameter assignment
- ESFF forcefield parameter file (.epa)
D. Files

ESFF structure output file (.esf)
Torsion information
Dynamics Trajectory History Files (.his and .fhis) .......... (generic)
Dynamics tabular output for energies (.gre)
Dynamics tabular output for thermodynamic states (.grp)
Dynamics scratch file (.pre)

Command input file (.inp)

Contents of .inp file

The command input file contains the command language program that controls logical flow through the simulation run. Valid commands and syntax for this file are described under *Btcl Language and Commands—Standalone Mode*. Input files for CDiscover can be any length.

Sample CDiscover .inp file

```
#BIOSYM btcl 3
#
# Input File For Discover Generated By Insight Version 2.95.0
# Date:           Tue Sep 5 13:06:33 1995
# Host Name: iris90    Host Type: iris
#
begin
#
#
# minimize \  
iteration_limit = 300 movement_limit = 0.200 \  
sd \  
   convergence = 1000.0 line_search_precision = 0.100 \  
cg \  
   convergence = 10.0 method = polak \  
      line_search_precision = 0.100 \  
newton \  
   convergence = 0.001 method = bfgs \  
      line_search_precision = 0.900 max_atoms = 200 \  
final_convergence = 0.001 \  
execute +before +after \  
   command = (print output \  
          +total_energy \  
          +total_nonbond_energy \  
          +coulomb_energy \  
          +vdw_energy \  
          +repulsive_vdw_energy \  
          +dispersive_vdw_energy \  
```
Forcefield parameter file (.frc)

+internal_energy \n+bond_energy \n+angle_energy \n+torsion_energy \n+oop_energy
#
writeFile coordinate filename = .cor

---

**Forcefield parameter file (.frc)**

---

**Format of .frc file**

The forcefield parameter file consists of sections:

1. MSI file identification
2. Forcefield version
3. Atom types
4. Equivalence table
5. Hydrogen bond donors and acceptors
6. Quadratic bond-stretching potential
7. Quartic bond-stretching potential
8. Morse bond-stretching potential
9. Quadratic angle-bending potential
10. Quartic angle-bending potential
11. Quadratic bond–bond interaction potential
12. Quadratic bond–angle interaction potential
13. One-term torsion potential
14. Three-term torsion potential
15. Four-term torsion potential
16. Angle–angle–torsion interaction potential
17. Peripheral bond–torsion interaction potential
18. Central bond–torsion interaction potential
19. Three-term angle–torsion interaction potential
20. Out-of-plane potential, improper torsion definition
D. Files

22. Out-of-plane interaction potential using improper torsion definition
23. Angle–angle interaction potential
25. 6–9 Nonbond (van der Waals) potential
26. 6–12 Nonbond (van der Waals) potential
27. 10–12 Hydrogen bond potential
28. Bond increments

Content and format of .frc file

The potential function file contains all the information about the forcefield needed by CDiscover. The first line of the file must be a standard MSI file identification line. Thereafter, the data in the file is contained in sections delimited by lines starting with a # character and a section identifier. Each section contains specific information, but the sections may appear in any order. As a general rule, the contents of the parameter file are case-insensitive, i.e., it does not matter whether text is capitalized or not, except for the atom type names. The file is also free-format, meaning that individual items—atom types, parameters, etc.—do not have to be in particular columns but rather must be separated by one or more blanks and/or tabs. Any line starting with an exclamation point (!) is considered a comment and is ignored. Within each section, lines beginning with > are special comments associated with that section.

The information in the file and its format are considerably different from those of earlier forcefield files. The intention is that the file should be not only machine-readable, but also be easily read and understood by users.

Version and reference numbers

It has often been difficult to track changes made to forcefields and also difficult to document the parameters themselves. Each parameter line in the current parameter file starts with a version number and a reference number, which, if used properly, can aid in solving these problems.
Important

Do not attempt to change a .frc file without reading this section first!

Version numbers and changing parameters in a .frc file

Version numbers are numbers such as 5.3, composed of a major release number (e.g., 5) and a revision number (e.g., 3).

If there is more than one line containing the same data, for instance the parameters for a c–c bond then, by default, the one with the highest version number is used and all other lines are ignored.

How to change a .frc file entry

For example, a parameter file may have a section like:

```
#quartic_bond     cff91
> E = K2 * (R - R0)^2 + K3 * (R - R0)^3 + K4 * (R - R0)^4
!Ver  Ref     I     J          R0         K2          K3          K4
!---- ---    ----  ----     -------    --------   ---------    --------
1.0   1     c     h         1.1010    314.0000   -691.8900    844.6000
...
```

If it is realized, for example, that the K2 parameter was transcribed incorrectly and should have been 341 instead of 314, the procedure to change it would be:

1. Copy the incorrect line.
2. Correct the parameter(s) in the copied line.
3. Increase the version number.

In this example, the version number is changed from 1.0 to 1.1:

```
#quartic_bond     cff91
> E = K2 * (R - R0)^2 + K3 * (R - R0)^3 + K4 * (R - R0)^4
!Ver  Ref     I     J          R0         K2          K3          K4
!---- ---    ----  ----     -------    --------   ---------    --------
1.0   1     c     h         1.1010    314.0000   -691.8900    844.6000
1.1   1     c     h         1.1010    341.0000   -691.8900    844.6000
...
```

Leaving the incorrect parameter line in the file documents that an error was made and later corrected.

Remember, the Discover program looks for the parameters with the highest version numbers, regardless of their location in the file.

Reference numbers

Appropriate use of reference numbers allows you to additionally document, in as much detail as desired, exactly when, why, and by whom changes were made. The references are only for document-
D. Files

ing the parameters, changes, etc. and are ignored by the Discover program. A reference number refers to a reference section that has the following form:

#reference 1
Any text at all may be placed here. A line starting with a “#” signifies the end of this section (and perhaps the beginning of another section).

#

**MSI file identification**

The first line of a forcefield parameter line must be:

!BIOSYM forcefield          1

with capitalization exactly as shown. The number 1 identifies the format of the forcefield file. Currently, 1 is the only available format. Files without this file identification line are assumed to be older-format forcefield files and processed as such.

**Forcefield version**

The version of the forcefield parameters is given on a line of the form:

#version filename version date

The *filename* and *date* are for documentation purposes and are not interpreted by the Discover program, but the *version* is crucial. The version number should be equal to the highest version number used anywhere within the file. On adding or changing anything within the forcefield file, add a new #version line with a higher version number, for example:

#version cff91_2.frc     1.3     10-Feb-92

**Atom types**

The atom types for the forcefield are defined in this section, in the format:

#atom_types section_label

*version reference atom_type mass element comments…

…

#

Each line defines a new potential function atom type name, giving its atomic mass, element, and a description of the atom type. The
Discover program uses the atom types given in the .car and .mdf files to assign parameters from the forcefield file, matching the atom types against those found in the forcefield file, for example:

```
#atom_types cff91

> Atom type definitions for any variant of cff91
> Masses from CRC 1973/74 pages B-250.

<table>
<thead>
<tr>
<th>Ver</th>
<th>Ref</th>
<th>Type</th>
<th>Mass</th>
<th>Element</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>1</td>
<td>c</td>
<td>12.0</td>
<td>C</td>
<td>Aliphatic carbon</td>
</tr>
<tr>
<td>1.0</td>
<td>1</td>
<td>h</td>
<td>1.007825</td>
<td>H</td>
<td>Aliphatic hydrogen</td>
</tr>
</tbody>
</table>
```

**Equivalence table**

The equivalence table is central to the procedure used by the Discover program to assign parameters for a model. Its purpose is to reduce the number of distinct parameters that need to be defined in the parameter file. This is accomplished by assigning to each atom type the name of an atom type that has the same parameters for a particular term (bond, angle, torsion, etc.). If no equivalent parameters exist for a given parameter type, then the atom is set equivalent to itself for that parameter type. In the following example, the nonbond parameter equivalence name for both o’ (carbonyl oxygen) and oh (hydroxyl oxygen) is o’, meaning that their nonbond interactions with other atoms are calculated with the same parameters, although their bond interactions are different.

The format of this section is:

```
#equivalence section_label
version reference atom_type nonbond bond angle torsion oop
...
#
```

`atom_type` is the atom type for which equivalences are being defined. The rest of the line gives the equivalent atom type name for nonbond, bond, angle, torsion, and out-of-plane parameters:

```
#equivalence cff91

<table>
<thead>
<tr>
<th>Ver</th>
<th>Ref</th>
<th>Type</th>
<th>NonB</th>
<th>Bond</th>
<th>Angle</th>
<th>Torsion</th>
<th>OOP</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>1</td>
<td>h*</td>
<td>h*</td>
<td>h*</td>
<td>h*</td>
<td>h*</td>
<td>h*</td>
</tr>
<tr>
<td>1.0</td>
<td>1</td>
<td>hi</td>
<td>h*</td>
<td>h*</td>
<td>h*</td>
<td>h*</td>
<td>h*</td>
</tr>
</tbody>
</table>
```

---

CCSDiscover/September 1997 387
D. Files

Hydrogen bond donors and acceptors

Depending on the forcefield being used, the Discover program may or may not have enough information to find hydrogen bonds for the enclosure analysis or the print hydrogen bonds command. This section provides the information needed to define what is considered a hydrogen bond.

The format is:

```
#hbond_definition section_label
version reference distance distance
version reference angle angle
version reference donors atom_type atom_type ...
version reference acceptors atom_type atom_type ...
#
```

The four lines giving distance, angle, donors, and acceptors may appear in any order, and the distance and angle do not have to be defined if the default values of 2.5 Å and 0° are satisfactory. The distance is the maximum donor–acceptor distance in angstroms and the angle is the minimum donor–hydrogen–acceptor angle considered to be consistent with a hydrogen bond. The donor and acceptor lines list the appropriate atom types. If there are more than one donor or acceptor line, they are considered to be continuations of the first line.

Quadratic bond-stretching potential

The simple quadratic bond-stretching potential is:

\[
E = k_2 (r - r_0)^2
\]  

Eq. 21

where \( r \) is the current bond length and \( r_0 \) the reference bond length in angstroms, and \( k_2 \) is the force constant in kcal mol\(^{-1}\) Å\(^{-2}\).

The format of the quadratic bond-stretching potential section is:

```
#quadratic_bond section_label
version reference i j r_0 k_2 ...
#
```

where \( i \) and \( j \) are the atom types of the bonded atoms. For example:
Quartic bond-stretching potential

The quartic bond-stretching potential is:

\[ E = k_2 (r - r_0)^2 + k_3 (r - r_0)^3 + k_4 (r - r_0)^4 \]

Eq. 22

where \( r \) is the current bond length and \( r_0 \) the reference bond length in angstroms, and \( k_2, k_3, \) and \( k_4 \) are the coefficients for the quadratic, cubic, and quartic terms, respectively, in kcal mol\(^{-1}\) Å\(^{-2}\), kcal mol\(^{-1}\) Å\(^{-3}\), and kcal mol\(^{-1}\) Å\(^{-4}\).

The format of the quartic bond-stretching potential section is:

```
#quartic_bond section_label
version reference i j r0 k2 k3 k4
...
```

where \( i \) and \( j \) are the atom types of the bonded atoms. For example:

```
#quartic_bond cff91
> E = K2 * (R - R0)^2 + K3 * (R - R0)^3 + K4 * (R - R0)^4

<table>
<thead>
<tr>
<th>Ver</th>
<th>Ref</th>
<th>I</th>
<th>J</th>
<th>R0</th>
<th>K2</th>
<th>K3</th>
<th>K4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>1</td>
<td>c</td>
<td>h</td>
<td>1.1010</td>
<td>341.0000</td>
<td>-691.8900</td>
<td>844.6000</td>
</tr>
<tr>
<td>1.0</td>
<td>1</td>
<td>c</td>
<td>c</td>
<td>1.5330</td>
<td>299.6700</td>
<td>-501.7700</td>
<td>679.8100</td>
</tr>
</tbody>
</table>
```

Morse bond-stretching potential

The Morse bond-stretching potential is:

\[ E = D_b [1 - e^{-\alpha(r-r_0)}]^2 \]

Eq. 23

where \( r \) is the current bond length and \( r_0 \) the reference bond length in angstroms, \( D_b \) is the bond dissociation energy in kcal mol\(^{-1}\), and \( \alpha \) is the Morse anharmonicity parameter with units of Å\(^{-1}\).
D. Files

The format of the Morse bond-stretching potential section is:

```
#morse_bond section_label
version reference i j r0 Db alpha
```

where \(i\) and \(j\) are the atom types of the atoms involved in the bond.

The Morse bond potential has a very shallow slope at large bond distances, which can cause problems when minimizing very bad structures. To avoid this problem, the BTCL **forcefield** command has an option that causes CDiscover to temporarily replace the Morse potential with a quadratic potential using the following expression for the force constant:

\[
k_2 = D_b \alpha^2
\]

**Eq. 24**

For example:

```
#morse_bond cvff
> E = D * (1 - exp(-ALPHA*(R - R0))^2

<table>
<thead>
<tr>
<th>Ver</th>
<th>Ref</th>
<th>I</th>
<th>J</th>
<th>R0</th>
<th>D</th>
<th>ALPHA</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>1</td>
<td>C</td>
<td>H</td>
<td>1.1050</td>
<td>108.6000</td>
<td>1.7710</td>
</tr>
<tr>
<td>1.0</td>
<td>1</td>
<td>C</td>
<td>C</td>
<td>1.5260</td>
<td>88.0000</td>
<td>1.9150</td>
</tr>
</tbody>
</table>
```

**Quadratic angle-bending potential**

The quadratic angle-bending potential is:

\[
E = k_2 (\theta - \theta_0)^2
\]

**Eq. 25**

where \(\theta\) is the current bond angle, \(\theta_0\) is the reference bond angle in degrees, and \(k_2\) is the quadratic force constant in kcal mol\(^{-1}\) rad\(^{-2}\). The program automatically converts the angles to radians before carrying out the calculation.

The format of the quadratic angle-bending potential section is:

```
#quadratic_angle section_label
version reference i j k the0 the2
```

---

390  CDiscover/September 1997
where $i$, $j$, and $k$ are the atom types involved in the angle, with $j$ being the apex atom. For example:

```
#quadratic_angle    cvff
> E = K2 * (Theta - Theta0)^2

!Ver Ref I J K       Theta0         K2
!---- --- ---- ----    --------     -------
1.0 1 h c h       106.4000     39.5000
1.0 1 h c c       110.0000     44.4000
1.0 1 c c c       110.5000     46.6000
```

Quartic angle-bending potential

The quartic angle-bending potential is:

$$E = k_2(\theta - \theta_0)^2 + k_3(\theta - \theta_0)^3 + k_4(\theta - \theta_0)^4$$

*Eq. 26*

where $\theta$ is the current bond angle, $\theta_0$ is the reference bond angle in degrees, and $k_2$, $k_3$, and $k_4$ are the quadratic, cubic, and quartic force constants in kcal mol$^{-1}$ rad$^{-2}$, kcal mol$^{-1}$ rad$^{-3}$, and kcal mol$^{-1}$ rad$^{-4}$ respectively. The program automatically converts the angles to radians before carrying out the calculation.

The format of the quartic angle-bending potential section is:

```
#quartic_angle section_label
version reference i j k Theta0 k2 k3 k4
...
```

where $i$, $j$, and $k$ are the atom types involved in the angle, with $j$ being the apex atom. For example:

```
#quartic_angle    cff91
> Delta = Theta - Theta0
> E = K2 * Delta^2 + K3 * Delta^3 + K4 * Delta^4

!Ver Ref I J K       Theta0         K2         K3          K4
!---- --- ---- ----    --------     -------    --------    --------
1.0 1 h c h       107.6600     39.6410    -12.9210     -2.4318
1.0 1 c c h       110.7700     41.4530    -10.6040      5.1290
```

Quadratic bond-bond interaction potential

The quadratic bond–bond interaction potential, or cross term, is:
D. Files

\[ E = k_{bb'}(r - r_0)(r' - r'_0) \]  \hspace{1cm} \text{Eq. 27}

where \( r \) and \( r_0 \) are the current bond length and reference bond length of one bond in angstroms; \( r' \) and \( r'_0 \) are the equivalent values for the second bond; and \( k_{bb'} \) is the interaction force constant in kcal mol\(^{-1}\) Å\(^{-2}\).

The format of the quadratic bond–bond interaction potential section is:

\#bond-bond section_label
version reference i j k k_{bb'}

\#

where \( i \) and \( j \) are the atom types of the first bond, and \( j \) and \( k \) of the second. The reference bond lengths \( r_0 \) and \( r'_0 \) are taken from the appropriate bond section of the parameter file. For example:

\#bond-bond cff91
> E = K(b, b') \ast (R - R_0) \ast (R' - R'_0)

<table>
<thead>
<tr>
<th>Ver</th>
<th>Ref</th>
<th>I</th>
<th>J</th>
<th>K</th>
<th>K(b, b')</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>1</td>
<td>h</td>
<td>c</td>
<td>h</td>
<td>5.3316</td>
</tr>
<tr>
<td>1.0</td>
<td>1</td>
<td>c</td>
<td>c</td>
<td>h</td>
<td>3.3872</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Quadratic bond–angle interaction potential

The quadratic bond–angle interaction potential, or cross term, is:

\[ E = k_{b\theta}(r - r_0)(\theta - \theta_0) \]  \hspace{1cm} \text{Eq. 28}

where \( r \) and \( r_0 \) are the current bond length and reference bond length in angstroms, \( \theta \) and \( \theta_0 \) are the current bond angle and reference bond angle in degrees, and \( k_{b\theta} \) is the quadratic force constant in kcal mol\(^{-1}\) Å\(^{-1}\) rad\(^{-1}\). The program automatically converts the angles to radians before carrying out the calculation.

The format of the quadratic bond–angle interaction potential section is:

\#bond-angle section_label
version reference i j k k_{b\theta}

...
where \( i, j, \) and \( k \) are the atom types involved in the angle, with \( j \) being the apex atom. If the atom types \( i \) and \( k \) are different, two different bond–angle terms are present, with the bond referred to as \( b \) involving \( i \) and \( j \), and \( b' \) involving \( j \) and \( k \). If atom types \( i \) and \( k \) are the same, it is an error to supply the second force constant. The values of the reference bond length \( r_0 \) and reference angle \( \theta_0 \) are taken from the appropriate bond and angle sections of the parameter file. For example:

\[
\text{#bond-angle cff91} \\
> \quad E = K \cdot (R - R0) \cdot (Theta - Theta0) \\
!Ver Ref I J K K(b,theta) K(b',theta) \\
|---- --- ----  ----  ----     ---------- ----------- |
1.0 l h c h 18.1030 11.4210 \\
1.0 l c c h 20.7540 11.4210 \\
\]

**One-term torsion potential**

The single-cosine form of the torsion potential is:

\[
E = V[1 + \cos(n\phi - \phi_0)] \quad \text{Eq. 29}
\]

where \( V \) is half the barrier height in units of kcal mol\(^{-1}\), \( n \) is the periodicity of the torsion, and \( \phi \) is the current torsion angle and \( \phi_0 \) the reference torsion angle in degrees. The reference angle is usually 0° or 180°.

The format of the one-term torsion potential section is:

\[
\text{#torsion_1 section_label} \\
\text{version reference i j k l \phi_0 n V} \\
\]

The atoms are bonded to each other in the order \( i-j-k-l \). The Discover program recognizes an asterisk (*) as a wildcard matching any atom type, as in the first line of the example. Neither or both end atom types (\( i \) and \( l \)) must be wildcards. If both specific and wildcard parameters are found for a specific torsion, the specific parameters are used. The interpretation of the rotational barrier \( V \) also differs slightly for the two cases. The value of \( V \) for a specific parameter is the contribution of that single torsion to the total barrier for rotation about the central bond; for wildcard parameters the value of \( V \) is the total barrier for rotation about the central...
D. Files

bond. For example, ethane (C\textsubscript{2}H\textsubscript{6}) has nine individual H–C–C–H torsion terms that combine to give the barrier height for rigid methyl group rotation. If wildcards are used, as in the first line of the example below, \( V \) is half the total torsion-term contribution to the rotational barrier in ethane, here, 1.422 kcal mol\(^{-1}\). If, however, the specific torsion definition \( b–c–c–b \) is used, as in the second line of the example, then \( V \) is the individual contribution, which here is one ninth, or 0.158 kcal mol\(^{-1}\). Thus, the two sets of parameters in the example for ethane are identical. For example:

\[
\# \text{torsion}_1 \quad \text{cvff}
\]

\[> E = Kphi \times [1 + \cos(n*Phi - Phi0)] \]

\[
\begin{array}{ccccc}
\text{Ver} & \text{Ref} & \text{I} & \text{J} & \text{K} & \text{L} & \text{Kphi} & \text{n} & \text{Phi0} \\
1.0 & 1 & * & c & c & * & 1.4225 & 3 & 0.0000 \\
\end{array}
\]

# Three-term torsion potential

The three-term cosine expansion of the torsion potential is:

\[
E = V_1 [1 - \cos (\phi - \phi_{01})] + V_2 [1 - \cos (2\phi - \phi_{02})] + V_3 [1 - \cos (3\phi - \phi_{03})] \\
\text{Eq. 30}
\]

where \( V_1, V_2, \) and \( V_3 \) are the barrier heights in kcal mol\(^{-1}\), and \( \phi \) is the current torsion angle and \( \phi_{01}, \phi_{02}, \) and \( \phi_{03} \) the reference torsion angles in degrees. The reference angles are usually 0° or 180°.

The format of the 3-term torsion potential section is:

\[
\# \text{torsion}_3 \quad \text{section_label} \\
\text{version} \quad \text{reference} \quad i \quad j \quad k \quad l \quad V_1 \quad \phi_{01} \quad V_2 \quad \phi_{02} \quad V_3 \quad \phi_{03} \\
\]

#

The atoms are bonded to each other in the order \( i–j–k–l \). For example:

\[
\# \text{torsion}_3 \quad \text{cff91}
\]

\[> E = \sum(n=1,3) \{ V(n) \times [1 - \cos(n*Phi - Phi0(n))] \} \]

\[
\begin{array}{cccccccccccc}
\text{Ver} & \text{Ref} & \text{I} & \text{J} & \text{K} & \text{L} & V1 & \phi0 & V2 & \phi0 & V3 & \phi0 \\
1.0 & 1 & h & c & c & h & -0.2432 & 0.0 & 0.0617 & 0.0 & -0.1383 & 0.0 \\
1.0 & 1 & c & c & c & h & 0.0000 & 0.0 & 0.0316 & 0.0 & -0.1781 & 0.0 \\
\end{array}
\]

#
Four-term torsion potential

In addition to torsion_1 and torsion_3, CDiscover also supports torsion_4. This is required by the four-term torsions in the new AMBER forcefield. Torsion_4 supports up to four nonzero torsion terms with any initial angles. The four-term cosine expansion of the torsion potential is:

$$ E = \sum \{ V(n) \{ 1 + \cos(n\phi - \phi_0(n)) \} \} $$

Eq. 31

where $V$ are the barrier heights in kcal mol$^{-1}$, and $\phi$ is the current torsion angle and $\phi_0$ is the reference torsion angle in degrees.

Since torsion_3 supports only 0° and 180° initial angles, while torsion_4 does not have this requirement, the latter may be used to get around any initial angles in 3-term torsions that are not 0° or 180°. For example, in the old AMBER forcefield, you may want to create a torsion_4 section for the (60° and 240°) 3-term torsions.

The format of the 4-term torsion potential section is:

```
#torsion_4 section_label
version reference i j k l V1 \phi0 V2 \phi0 V3 \phi0 V4 \phi0
...
#
```

The atoms are bonded to each other in the order $i$–$j$–$k$–$l$. For example:

```
#torsion_4 amber
> E = SUM(n=1,4) \{ V(n) \{ 1 + \cos(n*Phi_i - Phi_0(n)) \} \}

<table>
<thead>
<tr>
<th>Ver</th>
<th>Ref</th>
<th>I</th>
<th>J</th>
<th>K</th>
<th>L</th>
<th>V1</th>
<th>Phi0</th>
<th>V2</th>
<th>Phi0</th>
<th>V3</th>
<th>Phi0</th>
<th>V4</th>
<th>Phi0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>4</td>
<td>BH</td>
<td>BC</td>
<td>OB</td>
<td>CS</td>
<td>0.0000</td>
<td>0.0</td>
<td>1.2500</td>
<td>240.0</td>
<td>0.0000</td>
<td>0.0</td>
<td>0.0000</td>
<td>0.0</td>
</tr>
<tr>
<td>1.1</td>
<td>4</td>
<td>BH</td>
<td>BC</td>
<td>OB</td>
<td>HY</td>
<td>0.0000</td>
<td>0.0</td>
<td>1.2500</td>
<td>240.0</td>
<td>0.0000</td>
<td>0.0</td>
<td>0.0000</td>
<td>0.0</td>
</tr>
<tr>
<td>1.1</td>
<td>4</td>
<td>AH</td>
<td>AC</td>
<td>OA</td>
<td>CS</td>
<td>0.0000</td>
<td>0.0</td>
<td>1.7500</td>
<td>60.0</td>
<td>0.0000</td>
<td>0.0</td>
<td>0.0000</td>
<td>0.0</td>
</tr>
<tr>
<td>1.1</td>
<td>4</td>
<td>AH</td>
<td>AC</td>
<td>OA</td>
<td>HY</td>
<td>0.0000</td>
<td>0.0</td>
<td>1.7500</td>
<td>60.0</td>
<td>0.0000</td>
<td>0.0</td>
<td>0.0000</td>
<td>0.0</td>
</tr>
</tbody>
</table>
```

Angle-angle-torsion interaction potential

The angle–angle–torsion interaction potential, or cross term, is:

$$ E = V(\theta - \theta_0)(\theta' - \theta'_0)\cos\phi $$

Eq. 32
D. Files

where $V$ is the force constant with units of kcal mol$^{-1}$ rad$^{-2}$, $\theta$ and $\theta_0$ are the current and reference values of one bond angle, $\theta'$ and $\theta'_0$ are the current and reference values of the second bond angle, and $\phi$ is the current torsion angle. All angles are in degrees. The program automatically converts the angles to radians before carrying out the calculation.

The format of the angle–angle–torsion interaction potential section is:

```
#angle-angle-torsion section_label
version reference i j k l V
...
#
```

where atoms $i$–$j$–$k$ define the angle associated with $\theta$ and $\theta_0$, atoms $j$–$k$–$l$ define the angle associated with $\theta'$ and $\theta'_0$, and atoms $i$–$j$–$k$–$l$ constitute the torsion angle. The values of the reference angles are obtained from the appropriate angle sections. For example:

```
#angle-angle-torsion_1    cff91
> E = K * (Theta - Theta0) * (Theta' - Theta0') * cos(Phi)

!Ver Ref I J K L K(Ang,Ang,Tor)
!---- --- ---- ---- ---- --------------
1.0 1 h c c h -12.5640
1.0 1 c c c h -16.1640
...
```

Peripheral bond–torsion interaction potential

The interaction potential between a three-term torsion and the end bonds of the torsion is:

$$E = (r - r_0)[V_1 \cos \phi + V_2 \cos 2\phi + V_3 \cos 3\phi] \quad \text{Eq. 33}$$

where $r$ is the current bond length and $r_0$ the reference bond length in angstroms; $V_1$, $V_2$, and $V_3$ are the force constants in kcal mol$^{-1}$ Å$^{-1}$; and $\phi$ is the current torsion angle in degrees.

The format of the peripheral bond–torsion interaction potential section is:

```
#end_bond-torsion_3 section_label
version reference i j k l V_1 V_2 V_3 V_1' V_2' V_3'
```
where \( i, j, k, \) and \( l \) are the atom types involved in the torsion; \( V_1, V_2, \) and \( V_3 \) are the force constants for the interaction of bond \( i-j \) with the torsion; and \( V'_1, V'_2, \) and \( V'_3 \) are the force constants for the interaction of bond \( k-l \) with the torsion. If the \( i-j \) and \( k-l \) bonds are identical, that is, the atom type of \( i \) is the same as that of \( l \) and \( j \) the same as \( k \), then \( V'_1, V'_2, \) and \( V'_3 \) must not be given, as is shown in the first line of the example. The reference bond lengths \( r_0 \) are obtained from the appropriate bond section. For example:

**Central bond-torsion interaction potential**

The interaction potential between a torsion and its central bond is:

\[
E = (r - r_0)[V_1 \cos \phi + V_2 \cos 2\phi + V_3 \cos 3\phi]
\]

*Eq. 34*

where \( r \) is the current bond length and \( r_0 \) the reference bond length in angstroms; \( V_1, V_2, \) and \( V_3 \) are the force constants in kcal mol\(^{-1}\) Å\(^{-1}\); and \( \phi \) is the current torsion angle in degrees.

The format of the section central bond–torsion interaction potential is:

```
#middle_bond-torsion_3 section_label
version reference i j k l V_1 V_2 V_3
```

where \( i, j, k, \) and \( l \) are the atom types involved in the torsion; and \( V_1, V_2, \) and \( V_3 \) are the force constants for bond \( j-k \) interacting with the torsion. The reference bond lengths \( r_0 \) are obtained from the appropriate bond section. For example:
D. Files

Three-term angle–torsion interaction potential

The interaction potential between a torsion and the angles defined by the same atoms is:

$$E = (\theta - \theta_0)[V_1 \cos \phi + V_2 \cos 2\phi + V_3 \cos 3\phi] \quad \text{Eq. 35}$$

where $\theta$ is the current bond angle and $\theta_0$ the reference angle in degrees; $V_1, V_2,$ and $V_3$ are the force constants in kcal mol$^{-1}$ rad$^{-1}$; and $\phi$ is the current torsion angle in degrees. The program automatically converts the angles to radians before carrying out the calculation.

The format of the 3-term angle–torsion interaction potential section is:

```plaintext
#angle-torsion_3 section_label
version reference i j k l V_1 V_2 V_3 V'_1 V'_2 V'_3
...
#
```

where $i, j, k,$ and $l$ are the atom types involved in the torsion; $V_1, V_2,$ and $V_3$ are the force constants for angle $i$–$j$–$k$ interacting with the torsion; and $V'_1, V'_2,$ and $V'_3$ are the force constants for angle $j$–$k$–$l$ interacting with the torsion. The values of the reference angles $\theta_0$ are obtained from the appropriate angle sections. For example:

```plaintext
#angle-torsion_3  cff91

\[ E = (\Theta - \Theta_0) \left( F(1) \cos \phi + F(2) \cos 2\phi + F(3) \cos 3\phi \right) \text{ Eq. 35} \]

where $\Theta$ is the current bond angle and $\Theta_0$ the reference angle in degrees; $F(1), F(2),$ and $F(3)$ are the force constants in kcal mol$^{-1}$ rad$^{-1}$; and $\phi$ is the current torsion angle in degrees. The program automatically converts the angles to radians before carrying out the calculation.

The format of the 3-term angle–torsion interaction potential section is:

```plaintext
#angle-torsion_3 section_label
version reference i j k l V_1 V_2 V_3 V'_1 V'_2 V'_3
...
#
```

where $i, j, k,$ and $l$ are the atom types involved in the torsion; $V_1, V_2,$ and $V_3$ are the force constants for angle $i$–$j$–$k$ interacting with the torsion; and $V'_1, V'_2,$ and $V'_3$ are the force constants for angle $j$–$k$–$l$ interacting with the torsion. The values of the reference angles $\Theta_0$ are obtained from the appropriate angle sections. For example:
Out-of-plane potential, improper torsion definition

This section supplies the parameters used for the potential when the out-of-plane coordinate is defined according to an improper torsion. An out-of-plane potential is usually applied to planar groups containing an sp² central atom bonded to three other atoms. Examples are amide nitrogens, amide carbons, and the carbon atoms in a benzene ring. The out-of-plane potential acts to keep the central atom in the plane defined by the other three atoms. The functional form is:

\[ E = V[1 + \cos(n\chi - \chi_0)] \]  

\text{Eq. 36}

where \( V \) is the force constant with units of kcal mol\(^{-1}\), \( n \) is the (dimensionless) periodicity of the improper torsion (always 2); \( \chi \) is the current improper-torsion angle in degrees; and \( \chi_0 \) the reference improper-torsion angle (always 180°).

The format of the out-of-plane potential (improper torsion definition) section is:

```
#improper_torsion section_label
version reference i j k l n \chi0 V
...
#
```

where \( i, j, k, \) and \( l \) are the atom types of the four atoms involved in the out-of-plane term, \( j \) being the central atom. This term is asymmetric with respect to the three outer atoms \( i, k, \) and \( l \). For example:

```
#improper_torsion  cvff
> E = V * [ 1 + \cos( n * \chi - \chi0) ]

!Ver Ref I J K L n Chi0 V
--- --- --- --- --- --- ----- ----- ----- ----- ----- ----- ----- -----
2.50 1 c c' n o' 2 180.0 10.0
2.50 1 c' n c hn 2 180.0 0.05
...
#
```

Out-of-plane, Wilson definition

This section supplies the parameters used for the potential when the out-of-plane coordinate is defined according to the angle between one bond from the central atom and the plane defined by the other two bonds. An out-of-plane potential is usually applied to planar groups containing an sp² central atom bonded to three
D. Files

other atoms. Examples are amide nitrogens, amide carbons, and the carbon atoms in a benzene ring. The out-of-plane potential acts to keep the central atom in the plane defined by the other three atoms. The functional form is:

\[ E = V\chi^2 \]  

Eq. 37

where \( V \) is the force constant in kcal mol\(^{-1}\) rad\(^{-2}\); and \( \chi \) is the current Wilson out-of-plane angle in degrees. The program automatically converts the angle to radians before carrying out the calculation.

The format of the out-of-plane potential (Wilson definition) section is:

```
#wilson_out_of_plane section_label
version reference i j k l \chi_0 V
...
#
```

where \( i, j, k, \) and \( l \) are the atom types of the four atoms involved in the out-of-plane term, \( j \) being the central atom; and \( \chi_0 \) the reference angle in degrees. This term is asymmetric with respect to the outer atoms \( i, k, \) and \( l \), but is made symmetric by summing over the three different out-of-planes defined by a trigonal center. For example:

```
#wilson_out_of_plane cff91
> E = K * (Chi - Chi0)^2

!Ver Ref I J K L   K Chi0
!----- --- ---- ---- ---- ----------- ----
1.0  1  c  c= c= h    2.0765  0.0  
1.0  1  c= c= h  h    2.8561  0.0  
...
#
```

**Out-of-plane interaction potential using improper torsion definition**

This section supplies the parameters used for the interaction potential between two out-of-plane coordinates defined according to improper torsions. The central atom of one out-of-plane must be bonded to the central atom of the other. The functional form is:

\[ E = V[1 - \cos2\chi]^{1/2}[1 - \cos2\chi']^{1/2} \]  

Eq. 38
where \( V \) is the force constant in kcal mol\(^{-1}\), and \( \chi \) and \( \chi' \) are the current improper torsion angles in degrees.

The format of the out-of-plane interaction potential (improper torsion definition) section is:

```
#improper_torsion-improper_torsion section_label
version reference i j k l m n V
...
```

where \( i \) through \( m \) are the atom types of the atoms involved in this interaction. The out-of-plane angle \( \chi \) is defined by atoms \( i-j-k-l \), where \( j \) is the central atom; and \( \chi' \) is defined by \( j-i-m-n \), where \( i \) is the central atom. For example:

```plaintext
#improper_torsion-improper_torsion cvff
> E = V * [ 1 - \cos( 2 * \chi) ]^{1/2} * [ 1 - \cos( 2 * \chi') ]^{1/2}

<table>
<thead>
<tr>
<th>Ver Ref</th>
<th>J'</th>
<th>I'</th>
<th>K</th>
<th>L</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.50</td>
<td>n</td>
<td>c'</td>
<td>c</td>
<td>c</td>
<td>0.01</td>
</tr>
<tr>
<td>2.50</td>
<td>n2</td>
<td>c'</td>
<td>c</td>
<td>hn</td>
<td>0.01</td>
</tr>
</tbody>
</table>
...
```

**Angle-angle interaction potential**

The interaction potential between two angles sharing a common bond is:

\[
E = V(\theta - \theta_0)(\theta' - \theta'_0)
\]

**Eq. 39**

where \( V \) is the force constant in kcal mol\(^{-1}\) rad\(^{-2}\); \( \theta \) and \( \theta_0 \) are the current and reference values in degrees for the first bond angle; and \( \theta' \) and \( \theta'_0 \) are the current and reference values in degrees for the second bond angle. The program automatically converts the angles to radians before carrying out the calculation.

The format of the angle–angle interaction potential section is:

```
#angle-angle section_label
version reference i j k l V
...
```

where \( i, j, k, \) and \( l \) are the atom types of the atoms involved in the two angles. Atoms \( i-j-k \) define the first angle, \( \theta \); and \( k-j-l \) define
D. Files

the second angle, $\theta'$. The values of the reference angles $\theta_0$ and $\theta_0'$ are obtained from the appropriate angle sections. For example:

```
#angle-angle  cff91
> E = K * (Theta - Theta0) * (Theta' - Theta0')

<table>
<thead>
<tr>
<th>Ver</th>
<th>Ref</th>
<th>I'</th>
<th>J'</th>
<th>K'</th>
<th>K</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>h</td>
<td>c</td>
<td>h</td>
<td>h</td>
<td>-0.3157</td>
</tr>
<tr>
<td>1.0</td>
<td>h</td>
<td>c</td>
<td>c</td>
<td>h</td>
<td>-0.4825</td>
</tr>
</tbody>
</table>
```

6-9 Nonbond (van der Waals) potential

The “6–9” nonbond interaction potential is:

$$ E_{ij} = \frac{A_{ij}}{r_{ij}^6} - \frac{B_{ij}}{r_{ij}^9} $$  \hspace{1cm} \text{Eq. 40} $$

where $A_{ij}$ and $B_{ij}$ are parameters with units of kcal mol$^{-1}$ Å$^9$ and kcal mol$^{-1}$ Å$^6$, respectively; and $r_{ij}$ is the distance between atoms $i$ and $j$ in angstroms.

A completely equivalent representation is:

$$ E_{ij} = \epsilon_{ij} \left[ 2 \left( \frac{r_{ij}^*}{r_{ij}} \right)^9 - 3 \left( \frac{r_{ij}^*}{r_{ij}} \right)^6 \right] $$  \hspace{1cm} \text{Eq. 41} $$

where $\epsilon_{ij}$ is the potential well depth in kcal mol$^{-1}$, and $r_{ij}^*$ is the interatomic distance in angstroms at which the minimum occurs.

Conversion between the two representations is straightforward:

$$ \epsilon_{ij} = \frac{4B_{ij}^3}{27A_{ij}^2} \quad A_{ij} = 2\epsilon_{ij}r_{ij}^* 9 $$

$$ r_{ij}^* = \left[ \frac{3A_{ij}^2}{2B_{ij}} \right]^{1/3} \quad B_{ij} = 3\epsilon_{ij}r_{ij}^* 6 $$

The Discover program can read parameters in the $(A, B)$ or $(\epsilon, r^*)$ form. These parameters, given in this section, are for homonuclear interactions, i.e., the diagonal $A_{ii}$ and $B_{ii}$ or $r_{ii}^*$ and $\epsilon_{ii}$. The off-diag-
Heteronuclear interactions are computed as geometric (Eq. 43), arithmetic (Eq. 44), or sixth-power averages (Eq. 45).

<table>
<thead>
<tr>
<th>geometric:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>[ A_{ij} = \sqrt{A_{ii}A_{jj}} ]</td>
<td>[ B_{ij} = \sqrt{B_{ii}B_{jj}} ]</td>
</tr>
<tr>
<td>[ r_{ij} = \sqrt{r_{ii}r_{jj}} ]</td>
<td>[ \varepsilon_{ij} = \sqrt{\varepsilon_{ii}\varepsilon_{jj}} ]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>arithmetic:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>[ \varepsilon_{ij} = \sqrt{\varepsilon_{ii}\varepsilon_{jj}} ]</td>
<td>[ r_{ij} = \frac{r_{ii}^* + r_{jj}^*}{2} ]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sixth-power:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>[ \varepsilon_{ij} = \sqrt{\varepsilon_{ii}\varepsilon_{jj}} ]</td>
<td>[ r_{ij} = \left( \frac{r_{ii}^* + r_{jj}^*}{2} \right)^{1/6} ]</td>
</tr>
</tbody>
</table>

This is controlled by instructions of the following form:

@combination *keyword*

where *keyword* is one of geometric, arithmetic or sixth-power, and of the form:

@type *keyword*

where *keyword* is either of A-B or r-eps.

Thus, the lines:

@combination sixth-power
@type r-eps

specify that \((r^*, \varepsilon)\) parameters are input and Eq. 45 is used to create the off-diagonal interactions. This is the default for the CFF91 potential and need not be specified.

The format of the 6–9 nonbond (van der Waals) potential section is:

#nonbond(9–6) section_label
version reference i \[ r_{ii}^* \] \[ \varepsilon_{ii} \]

...
D. Files

where \( i \) is the atom type and \( r_{ij}^*\) and \( \varepsilon_{ij} \) are parameters for that atom type. For example:

\[
\# \text{nonbond}(9-6) \quad \text{cff91}
\]

\[
E = \varepsilon_{ij} \left[ 2 \left( r_{ij}^*/(r_{ij})^{12} \right)^{9} - 3 \left( r_{ij}^*/(r_{ij})^{6} \right)^{6} \right]
\]

\[
\text{where} \quad r_{ij} = \left( (r(i)^{6} + r(j)^{6})/2 \right)^{1/6}
\]

\[
\varepsilon_{ij} = 2 \sqrt{\varepsilon(i) \cdot \varepsilon(j)} \cdot \frac{r(i)^{3} \cdot r(j)^{3}}{r(i)^{6} + r(j)^{6}}
\]

6-12 Nonbond (van der Waals) potential

The “6–12” nonbond interaction potential is:

\[
E_{ij} = \frac{A_{ij}}{r_{ij}^{12}} - \frac{B_{ij}}{r_{ij}^{6}} \quad \text{Eq. 46}
\]

where \( A_{ij} \) and \( B_{ij} \) are parameters with units of kcal mol\(^{-1}\) \( \text{Å}^{12} \) and kcal mol\(^{-1}\) \( \text{Å}^{6} \), respectively; and \( r_{ij} \) is the distance between atoms \( i \) and \( j \) in angstroms.

A completely equivalent representation is:

\[
E_{ij} = \varepsilon_{ij} \left[ \left( \frac{r_{ij}^*}{r_{ij}} \right)^{12} - 2 \left( \frac{r_{ij}^*}{r_{ij}} \right)^{6} \right] \quad \text{Eq. 47}
\]

where \( \varepsilon_{ij} \) is the potential well depth in kcal mol\(^{-1}\), and \( r_{ij}^* \) is the interatomic distance in angstroms at which the minimum occurs.

The conversion between the two representations is straightforward:

\[
\varepsilon_{ij} = \frac{B_{ij}^2}{4A_{ij}} \quad A_{ij} = \varepsilon_{ij} r_{ij}^{*12}
\]

\[
r_{ij}^* = \left[ \frac{2A_{ij}}{B_{ij}} \right]^{1/6} \quad B_{ij} = 2\varepsilon_{ij} r_{ij}^{*6}
\]
Again, the parameters given in this section are for homonuclear interactions, and the off-diagonal, heteronuclear interactions can be computed as geometric (Eq. 43), arithmetic (Eq. 44), or sixth-power (Eq. 45) averages.

The Discover program can read parameters in the \((A, B)\) or \((\epsilon, r^*)\) form and can use the geometric, arithmetic, or sixth-power combination rules. This is controlled by instructions of the following form:

@combination *keyword*

where *keyword* is one of geometric, arithmetic or sixth-power, and of the form:

@type *keyword*

where *keyword* is one of A-B or r-eps.

Thus the lines:

@combination geometric
@type A-B

specify that \((A, B)\) parameters are input and Eq. 43 is used to create the off-diagonal interactions. This is the default and need not be specified.

The format of the 6–12 nonbond (van der Waals) potential section is:

#nonbond(12-6) section_label
version reference i A_{ii} B_{ii}
...#

where \(i\) is the atom type and \(A_{ii}\) and \(B_{ii}\) are parameters for that atom type. For example:

#nonbond(12-6) cvff
@type A-B
@combination geometric
> E = A_{ij}/r^{12} - B_{ij}/r^6
> where A_{ij} = sqrt( A_{ii} * A_{jj} )
> B_{ij} = sqrt( B_{ii} * B_{jj} )
D. Files

<table>
<thead>
<tr>
<th>!Ver</th>
<th>Ref</th>
<th>I</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>1</td>
<td>h</td>
<td>7108.4660</td>
<td>32.87076</td>
</tr>
<tr>
<td>1.0</td>
<td>1</td>
<td>c</td>
<td>1790340.7240</td>
<td>528.48190</td>
</tr>
</tbody>
</table>

**10–12 Hydrogen bond potential**

The “10–12” hydrogen bond potential is:

\[ E = \frac{A}{r_{ij}^{12}} - \frac{B}{r_{ij}^{10}} \]  

Eq. 49

which is used in the old AMBER forcefield. \(A\) and \(B\) are coefficients with units of kcal mol\(^{-1}\) Å\(^{12}\) and kcal mol\(^{-1}\) Å\(^{10}\), respectively; and \(r_{ij}\) is the distance between atoms \(i\) and \(j\) in angstroms.

**Note:** Whether to calculate hydrogen bonding interactions using the 10–12 potential is determined by the existence of the hydrogen_bond section in the forcefield file. In other words, if there is no hydrogen_bond section in the forcefield file used, the 10–12 potential is not used in the calculation. The new AMBER forcefield should not have a hydrogen_bond section.

The format of the 10–12 hydrogen bond potential section is:

```
#10-12_hydrogen_bond
section_label
version reference i j A B
...
#
```

where \(i\) and \(j\) are the atom types of the hydrogen-bonding atoms.

For example:

```
#10-12_hydrogen_bond amber
> E = A/Rij*12 - B/Rij*10

<table>
<thead>
<tr>
<th>!Ver</th>
<th>Ref</th>
<th>I</th>
<th>J</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>1</td>
<td>H</td>
<td>NB</td>
<td>7557.0</td>
<td>2385.0</td>
</tr>
<tr>
<td>1.0</td>
<td>1</td>
<td>H</td>
<td>NC</td>
<td>10238.0</td>
<td>3071.0</td>
</tr>
</tbody>
</table>
```

```
#
```
Bond increments

The bond increments section supplies the bond increments used by the Insight program to assign charges to atoms. Discover ignores this information. The charge on an atom is given by:

\[ q_i = \sum \delta_{ij} \quad \text{Eq. 50} \]

where \( q_i \) is the charge on atom \( i \), and \( \delta_{ij} \) is the bond increment between that atom and another to which it is bonded. The sum is over all atoms bonded to atom \( i \).

The format of the bond increments section is:

```
#bond_increments section_label
version reference i j \delta_{ij} \delta_{ji}
```

where \( i \) and \( j \) are the atom types of the atoms bonded to each other, and \( \delta_{ij} \) and \( \delta_{ji} \) are the bond increments. Normally \( \delta_{ij} = -\delta_{ji} \); however, unequal magnitudes of the forward and backward bond increments can be used for atoms having a formal charge. Thus, in the CVFF forcefield the bond increment for n4–hn is -0.11, and for hn–n4 it is 0.36. In an ammonium ion (NH4+) each hydrogen would have a charge of +0.36, and the nitrogen’s charge would be \( 4 \times -0.11 = -0.44 \), giving a net charge to the ion of +1. For example:

```
#bond_increments     cff91
!Ver Ref I J DeltaIJ DeltaJI
!---- --- ---- --------- ---------
1.0 1 h c 0.0530 -0.0530
1.0 1 c c 0.0000 0.0000
```

Standard output file (.out)

The standard output file contains all the important formatted output from the simulation. The specific contents of the file depend on the actual calculations made. (See print command for information on controlling what CDiscover includes in the .out file.)
D. Files

**CDiscovey output files (.arc, .xfrc, .out, .tbl, .xdyn, user-named)**

Standard CDiscover output is in the form of archive (.arc), standard output (.out), Insight table (.tbl) files, and user-named files. Files with the extensions .xdyn (see Discover dynamics restart files (.xdyn)) and .xfrc can also be written. These files are specified by the Discover Dynamics Output control panel in the C2•Discover module, the Analyze/Output command in the Insight•Discover_3 module, and the print and writeFile commands in the stand-alone mode of CDiscover. The .tbl file is used by the graphing facility of the Cerius² and Insight interfaces.

**Discover dynamics restart files (.xdyn)**

For a system previously simulated with the CDiscover program, a restart file can be used to eliminate the necessity of locating all the internal coordinates for the system and assigning the appropriate potential function parameters to them. This process can otherwise be quite time consuming for large systems. This restart file has the extension .xdyn for CDiscover. The restart file is a binary file generated the first time a new system is simulated with CDiscover. Potential function parameters are stored in the restart file and can be used to replicate an earlier simulation. Alternatively, parameters can be selected from the potential parameter file (using the forcefield keyword with the begin command). This is useful when you need to make changes in the system or to the set of potential function parameters.

To continue a dynamics run from a previous CDiscover job, the dynamics restart file is read by putting the readFile dynamics_restart command right before the dynamics command.

The restart file contains most of the important data structures necessary to define a system. In addition to the forcefield parameters, it contains type identifiers for all atoms in a system, the residue sequence of the system, and the names of all named torsions in the system. The excluded-atoms list (see Forcefield-Based Simulations,
Automatic potential parameter assignment

published separately by MSI) is also contained in the restart file, so that it does not have to be regenerated at the beginning of a restarted simulation. Finally, the restart file contains pointers to various data structures necessary for system regeneration.

It is important to note that the restart file does not contain specific information pertaining to the states of any minimization or dynamics calculations carried out when the file was generated. The data required to restart a dynamics calculation where it was stopped in a previous program execution is contained in the dynamics restart file.

Automatic potential parameter assignment

The automatic potential parameter assignment information includes various interactions (bonds, bond angles, torsion angles, out-of-plane interactions, and angle–angle terms) whose potential function parameters (if any) have been assigned automatically. Parameters may be selected automatically in two circumstances:

♦ If the Discover program was directed to continue on encountering unfound parameters, and parameters for certain interactions were not defined in the potential file.

♦ If the system to be simulated was constructed with the fragment-building feature of the Insight program (available under a separate license), and no specific parameters for one or more interactions were found in the potential file.

For CDIscov, this information appears in the .out file or one with a user-defined name, if requested.

ESFF structure output file (.esf)

The default short-format output file (.esf) prints the van der Waals and charge parameters used in the forcefield calculation. If:

> setenv ESFF_DEBUG 1
is executed before the Discover job starts, long-format output, which contains the internal values and energies of the initial model structure, is generated.

### ESFF forcefield parameter file (.epa)

The parameter file (.epa) is generated by Discover program when the ESFF forcefield is chosen. It contains explicit internal parameters, which are used in energy evaluations.

#### Section 1, bond parameters

The Morse function is used for bond energy calculation, and the parameters for the bond dissociation energy, the anharmonicity parameter alpha, and the bond reference value are listed in columns 6–8 for each bond in the model. Column 9 is the internal value of model at the starting geometry. Columns 10 and 11 are flags for axial bonds.

<table>
<thead>
<tr>
<th>Bond atom names</th>
<th>Bond atom type</th>
<th>Bond Order</th>
<th>bond Energy</th>
<th>alpha</th>
<th>reference value</th>
<th>internal values</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASYM_1:FE1</td>
<td>ASYM_1:CL2</td>
<td>1.0</td>
<td>76.77236</td>
<td>1.63428</td>
<td>2.24307</td>
<td>2.34477</td>
</tr>
<tr>
<td>ASYM_1:FE1</td>
<td>ASYM_1:CL3</td>
<td></td>
<td>76.77236</td>
<td>1.63428</td>
<td>2.30311</td>
<td>2.34477</td>
</tr>
</tbody>
</table>

#### Section 2, angle parameters

Angle force constants and reference values are listed in columns 9 and 10 for each angle in the model. For perpendicular and linear angles, the reference values are 90° and 180°, respectively, and are not listed. For planar angles, the reference values are calculated based on the symmetry of the center atom and are also not listed.

<table>
<thead>
<tr>
<th>Angle atom names</th>
<th>Angle atom types</th>
<th>angle types</th>
<th>force constate</th>
<th>internal values</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASYM_1:CL2</td>
<td>ASYM_1:FE1</td>
<td>Perpendicular</td>
<td>0</td>
<td>24.81519</td>
</tr>
<tr>
<td>ASYM_1:CL2</td>
<td>ASYM_1:FE1</td>
<td>Linear</td>
<td>0</td>
<td>25.35004</td>
</tr>
<tr>
<td>ASYM_1:CL3</td>
<td>ASYM_1:O4</td>
<td>Planar</td>
<td>0</td>
<td>26.25131</td>
</tr>
<tr>
<td>ASYM_1:FE1</td>
<td>ASYM_1:C22</td>
<td>Free</td>
<td>0</td>
<td>48.71039 124.40968 136.06333</td>
</tr>
</tbody>
</table>

#### Section 3, torsion parameters

The force constant shown in column 10 for each type of torsion existing in the model is V/2. Note that the parameters for bonds, angles, and pseudoangles are listed according to the number of internals, whereas the parameters for torsions and out-of-planes are listed according to the number of internal types.

<table>
<thead>
<tr>
<th>Torsion angle atom types</th>
<th>center bond order</th>
<th>center angle type</th>
<th>force constant(k)</th>
<th>number of torsion angles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ko o' o'</td>
<td>n</td>
<td>2.0</td>
<td>6.34125</td>
<td>1</td>
</tr>
<tr>
<td>Ko o' o'</td>
<td>cp</td>
<td>2.0</td>
<td>6.34125</td>
<td>1</td>
</tr>
</tbody>
</table>

#### Section 4, out-of-Plane

<table>
<thead>
<tr>
<th>Out-of-plane angle atom type(center atom)</th>
<th>force constant</th>
<th>number of out of plane angles</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>1.30000</td>
<td>1</td>
</tr>
<tr>
<td>n=</td>
<td>9.50000</td>
<td>1</td>
</tr>
</tbody>
</table>
Section 5, pseudoangle parameters

This section lists the angle parameters for each pseudoangle in the model. Parameters bondo_1 and bondo_2 are not used explicitly in energy calculation. They are listed here just for comparison with the starting geometry. Only the force constant and \( \theta_{\text{ao}} \) are used in energy evaluations.

<table>
<thead>
<tr>
<th>Pseudo Angle 1</th>
<th>Angle_type</th>
<th>force constant</th>
<th>( \theta_{\text{ao}} )</th>
<th>bondo_1</th>
<th>bondo_2</th>
<th>internal_( \theta_{\text{ao}} )</th>
<th>internal_bond_1</th>
<th>internal_bond_2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal</td>
<td>21.51612</td>
<td>112.15383</td>
<td>2.01450</td>
<td>2.31688</td>
<td></td>
<td>105.68648</td>
<td>2.09207</td>
<td>2.32759</td>
</tr>
</tbody>
</table>

Center atom ASYM_1/TI1  T1044

Atoms involved in first pseudo atom:
1  ASYM_1:C4  c=
2  ASYM_1:C5  c=
3  ASYM_1:C6  c=
4  ASYM_1:C7  c=
5  ASYM_1:C8  c=

Atoms involved in second pseudo atom:
1  ASYM_1:CL2  c1

Torsion information

The named torsion rotation angles and corresponding energies appear in the .out or .tbl files or one with a user-defined name, if requested.

Dynamics tabular output for energies (.gre)

CDiscovet can send graph information during dynamics runs to .tbl or user-named files—See the Cerius\(^2\) or Insight online help and print command for information on controlling what Discover includes in these files and how often information is output during dynamics.

Dynamics tabular output for thermodynamic states (.grp)

CDiscove can send graph information during dynamics runs to .tbl or user-named files—See the Cerius\(^2\) or Insight online help and print command for information on controlling what Discover includes in these files and how often information is output during dynamics.
D. Files

**Dynamics scratch file (.pre)**

The dynamics scratch file is used as a buffer file for temporary storage of the thermodynamic state table generated during constant-pressure dynamics. This table is appended to the .out file after the energy table.
General Index

A

adiabatic compressibility, 64
analysis, 40
ANALYSIS card, 42
Analysis Input control panel, 43, 44, 45, 46
Analysis module, 40, 42
Analysis Output control panel, 47
Analysis Show Frames control panel, 45, 46
Analysis Statistics control panel, 47, 49, 50, 63
Analyze pulldown, 67
analyzeNonbond command, 143
Andersen, H. C., 31
angle–angle interaction potential, 401
angle–angle–torsion interaction potential, 395
angles, 210
  constraints, 20, 256
  defining, 197, 212
  measuring, 268
  quadratic potential, 390
  quartic potential, 391
  radians, 195
  restraints, 265
angle–torsion interaction potential, 398
array geometric objects, 194
Atom Constraints control panel, 19, 20
atom types, 386
  additional information, 16
  assigning, 15
  labels, 16, 17
  manual assignment, 15
  reassigning, 15, 16
  verifying assignments, 15
  when assigned, 16
  wildcarding, 393
atomMovability command, 149
atoms
  allowing to move, 19
  connectivity, 284
  constraining, 18
  coordinates, 210
  dipole calculations, 62
  excluding certain atoms from calculation, 325
  filtering lists of, 343
  finding bond partners, 308, 319
  finding those near another atom, 343
  fixing, 18, 19
  forces on, 25
  mean-square displacement, 51
  movability, 149
  positions, 318
  radial distribution, 58
  selecting, 276
  specification, 273
  typing, 15
  velocity autocorrelation, 54
  within a sphere, 343
atom-type charge, 16
autocorrelation function, 53
automatic parameters, 186
averages, types, 249

B

background jobs
  completion status, 74
  default host, 74
  default mode, 72
  electronic mail notification, 73
  execution mode, 73
  job number, 73
  job status, 75
  network queuing system, 72, 74
  notification window, 74
  preferred host, 73
  requirements for running on a remote host, 72
  stopping execution, 75
  submission to queue, 72
Background_Job pulldown, 68, 99
begin command, 152
binary databases, 345
binary representation of number, 139
Block Average Preferences control panel, 50, 65
bold type, meaning, 8, 9
bond increments, 407
bond–angle interaction, quadratic potential, 392
bond–bond interaction, quadratic potential, 391
bonds
  constraints, 256
  finding, 308, 314
  lengths, 210
  Morse potential, 389
  quadratic potential, 388, 390
  quartic potential, 389
Btcl
annealed_dynamics, example custom procedure, 132
arrays, 137
asterisks in keywords list, meaning of, 140
atom specification, 114
backslash, 125, 126
bold type in command descriptions, meaning of, 139
Boolean parameters, 137
Booleans, shorthand for, 137
braces, 125
break, 131
carriage return, 125, 130
cdlFetch, 136
cdlSetDefaults, 136
changing defaults, 135
commands, 123
comments in scripts, 126
continue, 131
control, 138
control, statements, 129
customized procedures, 132
debugging, 340
default values, 138, 187
defaults, viewing values, 136
dollar symbol, 126
echo, 124
ellipses in command descriptions, meaning of, 141
else, 130
elseif, 130
ending a session, 103, 106, 139
enum values, 137
eval, 127
exiting from a loop, 131
expr, 127
expressions, 124
for, 130
foreach, 131
global, 133
hash mark, 126
if, 129, 130
incr, 131
incrementing a variable, 131
integer arithmetic, 132
integer parameters, 137
interactive utility, 106
itself in command descriptions, meaning of, 139
keyword path, 134
keywords, list of, appropriate values, 140
keywords, list of, explanation of, 140
logical and arithmetic expressions, 128
looping commands, 130
objects, 222
parameters, defaults, 135
parameters, types, 137
parentheses, 127
path mechanism, 134
proc, 132
procedures, defining, 114
PROJECT variable, 138
question marks in command descriptions, meaning of, 141
quotes, 125
randomSeed variable, 138
real parameters, 137
reciprocals, 132
return, 133
semicolon, 125, 126
set, 124
set project, 106
space, 126
special characters, 128
strings, 137
strings, quoting, 130
subcommands, 127, 137, 174
syntax, 134
testing, 340
values, types of, 140
variable assignment, 114, 124
variable assignment from Insight interface, 85
variable substitution, 127
variables, comparing to strings, 130
variables, finding values of, 124
variables, global, 138
variables, undefined, 124
vector and scalar operations, 114
vertical line in command descriptions, meaning of, 141
Btvl
while, 130

C
Calculate pulldown, 67, 68, 97
calculations
  energy evaluation, 25
  gradient evaluation, 25
  monitoring progress, 244
  setting up, 24, 68
  stages, 67
  stopping for missing parameters, 24
types, 1, 67
CDiscover compared with FDiscover, 3
cdlConfig, 175
cell
  dimensions and angles, 208
  mass, 172, 173
  optimization, 27
  parameters, 174
cell multipole method
  and nonbond interactions, 145
cellParameter command, 153
central bond–torsion interaction potential, 397
cg (conjugate gradients) parameters, 205
charge groups
  additional information, 17
  definition, 17
  display, 17
  manually defining, 18
  removing definitions, 18
  tolerance, 18
  when defined, 17
charges
  assignment, 15
dipole calculations, 61
labels, 18
chemical functional groups, 17
chirality
  finding, 268
  restraints, 265
Color Selected Objects control panel, 19
commands, standalone, 123
Command_Comment Language_Control, 98
communication with external executables, 179
communication with external programs, 347
Completion_Status Background_Job, 99
compressibility, 56, 59
calculation
  expense, 18, 20, 21
time, 28, 146
conditional parameters, 69
conformational changes, 190
conformations
  comparing, 214
  controlling, 195, 198, 212, 213, 276
  finding, 194, 198, 213
  least-squares lines, 196
  least-squares planes, 196
  principal axes, 196
  principal moments, 196
  root-mean-square, 196
  transforming, 194
  weighting, 196
consensus dynamics, 176
constraints, 18, 256
  additional information, 19
  cancelling, 258
  changing, 258
  creating, 256, 276
dynamics, 20
  setting, 19
tolerance, 20
  verifying assignments, 19
  when to set, 18
Control_Bkgd_Job Background_Job, 99
coordinates
  Cartesian, 317
  fractional, 317
coulomb parameters, 184
Coulombic interactions, 187
cross terms, 391
  missing, 186
crystal optimization, 27
customizing the Discover program, 347

cyclo gly–pro–gly–ala–arg, tutorial, 77

C2•Analysis module, 40, 42
C2•Discover module, 12
C2•Tables module, 35, 47

D

database command, 154
databases
accessing, 340
accessing data, 158, 164, 276
Atom table, 310, 330, 331
Atom table example, 306
Axes column, 316
Bond table, 313
bonds, 307
Cartesian2Fractional column, 318
Cell table, 316
CellParameter column, 317
Chirality column, 311
compressed, 345
Coord table, 338
current, 310
CurrentSymmetryObject column, 322
default, 157
definition, 341
deleting, 156
Dispersive column, 331
Dynamics, 338
Electrostatic column, 331
Energy, 329
exchange with external programs, 356
excluding items from a calculation, 328
Fractional2Cartesian column, 317
Global table, 334
handle operations, 158
handles, 147, 156
hierarchical, 272
interdatabase relationships, 343
Iteration table, 336
machine-independent, binary, 345
MainCell/NAME/AsymmetricBond table, 321
manipulating, 340
manipulation, 145, 156
many-to-one relationships, 307
Minimize, 335
Misc table, 330, 336
molecular system, 272, 309

Molecule table, 312
Monomer table, 312
NonbondGroup table, 313
operations, 154
OutOfPlane column, 312
periodic systems, 314
printing, 156, 331
printing tables, 306
Properties table, 338
PseudoAtom table, 324
P1Bond table, 318
Repulsive column, 331
retrieving information, 307, 341
specification, 155, 160
SubItem table, 323
Subset table, 323
subsets, 278
symmetric periodic systems, 320
SymmetryObject table, 320
SymmetryOperator table, 321
table aliases, 316
table columns, 306
tables, 162
tables and script commands, 306
types, 155
Values table, 330
DeCipher module, 116
default settings, 11
defaults, previous stage, 67
Define Pseudo_Atom, 98
Delete Pseudo_Atom, 98
dielectric constant, 23, 64
diffraction command, 165
diffusion constant, 53
Ding, H. Q., 22
dipole autocorrelation, 62
Dipole control panel, 62, 63
dipole moment, 61, 62, 64, 163
dipole vector, 61, 62
dipole–dipole correlation function, 61, 62, 63
dipole–dipole power spectrum, 61, 62, 63
Discover Atom Typing control panel, 16, 17, 18
DISCOVER card, 12
DISCOVER card deck, 5, 12, 42
Discover Dynamics control panel, 20, 29, 30, 31, 32, 33
Discover Dynamics Output control panel, 34, 35
Discover in Cerius², 12
Discover Input File control panel, 33, 39, 40
Discover Job Control control panel, 35, 36, 37, 38
Discover Job Control Options control panel, 37
Discover Minimize control panel, 26, 27, 33
Discover Minimize Output control panel, 34, 35
Discover module, 12, 13
Discover Non-Bond control panel, 22, 23
Discover Parameters control panel, 24
Discover program, overview, 3
Discover Scaling control panel, 23
Discover Stress Constraints control panel, 31, 32

discoveHistory command, 167
Discover_3 module, 67
Disco_Error_List D_Run, 99
displacement analysis functions, 50
distances
  constraints, 20, 256
  measuring, 268
Distribution Preferences control panel, 50
dynamic properties, 53, 55
dynamics
  additional information, 29, 33
  Andersen, 31, 172
  averaging results, 35
  carrying sums to the next stage, 172
  clearing sums and averages, 171
  collision frequency, 172
  constant-energy, constant-volume ensemble, 172
  constant-pressure, 172
  constant-pressure/constant enthalpy, 30
  constant-pressure/constant-temperature, 30
  constant-stress, Parrinello method, 173
  constant-temperature, 172
  constant-temperature, constant-pressure ensemble, 172
  constant-volume/constant-energy, 29
  constant-volume/constant-temperature, 30
  constraints, 20
  database, 341
  data-collecting stage, 171
  energy changes, 174
  ensembles, 29, 172
  external hydrostatic pressure, 32
  graphs, 46
  heat bath, 31
  information during run, 338
  initialization, 171
  integrators, 31
  kinetic temperature, 30
  length of run, 174
  minimization incorporated into run, 33
  Nosé, 172, 173
  Nosé–Hoover, 31
  NPH, 30, 32
  NPT, 30, 32
  NVE, 29
  NVT, 30
  output, 34, 174, 249
  part of a model, 18
  periodic systems, 30
  prerequisites, 28
  pressure, 29, 30, 33, 173
  pressure “bath”, 32
  print control, 174
  RATTLE algorithm, 20
  restart files, 172, 408
  restarting, 172, 176, 408
  run length, 29
  setting up, 27
  specifying, 28
  stages, 33
  standalone mode of Discover program, 171
  statistical ensembles, 30
  strain, 174
  stress, 30, 32, 173
  stress “bath”, 32
  stress–strain relationship, 174
  tables, 47
  target temperature, 31, 32
  temperature, 29, 32
  temperature control, 32, 172
  temperature scaling, 31, 32
  temperature window, 31, 32
  thermodynamic temperature, 30, 31
  thermodynamics ensembles, 29
  time step, 174
  timestep, 28
  uses, 2
  velocity scaling, 31
  velocity, random, 171
  zeroing sums between stages, 172
Dynamics Calculate, 97
dynamics command, 168
D_Run pulldown, 68

E
eigenvectors, 299
Einstein relation, 52
electron scattering pattern, 165
electrostatic interactions, 144, 187
electrostatic properties, 64
energy
current, 25, 177
from external programs, 351, 356
output, 247
single evaluation, 25, 177
testing, 179
total, meaning of, 19
energy command, 177
energy expression, 13, 18
automatic parameter assignment, 182
constraints, 19
excluding default terms, 23
parameter scaling, 23
parameters, 23
preparing, 15, 20
scaling terms, 182
EnergyAnalysis Analyze, 98
energyContribution command, 179
ensembles
canonical, 172
constant pressure–constant enthalpy, 172
constant pressure–constant temperature, 172
constant volume–constant energy, 172
constant volume–constant temperature, 172
microcanonical, 172
enthalpy, 64
costant, 64
Equilibration Options control panel, 31
equivalence tables, 387
ESFF forcefield
editing parameters, 111
setting up, 70
Ewald sum method, 22, 145
geometric combination rule, 22
off-diagonal parameters, 22

F
file browsers, 14
files
.arc, 43, 249, 260, 303, 408
.archive, 246, 249, 260, 303
.automatic parameter assignment, 409
.background_job_hosts, 73
.car, 260, 303
.choosing output types, 35
 comando, 382
ccontrol of, 39
cordinate, 260, 303
creating input, 101
dynamics_restart, 172, 174, 260, 303, 408
editing input, 39
.err, 105
.forcefield, 152, 260, 303, 383
.forms, 3, 381
.FORTRAN history, 168
.frc, 383
.gre, 411
.grp, 411
.Hessian, 301
.history, 246, 260
.input, 3, 7, 101, 381
.input, checking, 87
.input, constructing, 108
.input, editing, 39
.input, example, 107
.input, merging, 39
.input, saving, 39
.input, using, 40
.ipc, 244
.machine-dependent history, 167
.machine-independent history, 167
.mdf, 260, 303
.molecular data, 152, 260, 303
.molecular description, 109
.MSIdiscoverstat, 38
.names, 39
.out, 105, 249, 407, 408
.output, 34, 105, 246, 249, 381, 407
overwriting, 249
.pkl, 176, 209, 243
.pek, 176, 209, 243
.pre, 412
.prm, 409
reading, 114, 259
reading frames, 261
rereading, 262
restart, 408
scratch, 412
table, 175, 246, 249
tables, 47
.tbl, 249, 408
trajectory, 34, 43
trajectory, analysis, 40
transfer, 36, 37, 38
user-named, 249, 408
writing, 85, 114, 302
writing frames, 304
.xdyn, 172, 260, 303, 408
.xfrc, 260, 304, 408
File_Control Language_Control, 98
Fix Specify, 97
Fluctuations control panel, 65
forcefield
  choosing, 14
forcefield command, 182
Forcefield Specify, 97
forcefields
  additional information, 14
  automated loading, 14
  choice, 13
  choosing, 152, 182
  combination rule, 403
  default, 14
  energy expression, 23
  filenames, 13
  nondefault, 15
  recommendations, 13
  specifying, 152, 182

G
Geometric Calculate, 97
GeometricSubset Analyze, 98
gometry command, 190
gometry manipulation, 190, 211, 265, 286
tutorial, 113

ghost atoms, 319
Goddard, W. A., 22
gradient evaluation, 25
graphs
  additional information, 47
  dipole moment, 63
  dipole vector, 63
  dipole–dipole correlation function, 63
  dipole–dipole power spectrum, 63
  MSD, 51, 53
  power spectrum, 56
  printing, 47
  properties, 48, 65
  RDF, 58
  results, 48
  simulations, 48
  structure factor, 61
  VACF, 54
Greengard, L., 22
Gruneisen parameter, 64

H
heat diffusion, 340
help command, 203
help, on-screen, 7
Hessian matrix, 330
  accessing, 333, 337
  diagonalization, 333
Hoover, W. G., 31
host
  local, 72
  remote, 72
hydrogen bonds
  information, 388
  potential, 406

I
IEEE format, 168
inaccessible parameters, 69
infrared spectra, 53, 55, 61
input files, editing, 39
interatomic vector lengths, 56
internal energy, 64
interprocess communication applications, 347
IPC
accept a connection, 350
advanced operations, 353
C functions, 364
C language interface, 363
characteristics of connection, 350
close a connection, 353
close connection, 350
establish a link between programs, 351
FORTRAN interface, 367
get data from a connection, 350
get data from a connection, 350
initialize connection, 349
language structures, 378
listening socket, 350
low-level operations, 361
open a connection, 350, 352
reading/writing frames, 360
remote host/process, 362
send over a connection, 350
shared-memory parallel machines, 363, 367
specialized communications, 361
specify a port number, 352
types of data exchanged, 363, 367

ipc commands, 347
isobaric heat capacity, 64
isoenthalpic Joule–Thomson coefficient, 64
isometric heat capacity, 64
isothermal compressibility, 64
isothermal Joule–Thomson coefficient, 64

italic type, meaning, 8, 9

K
Karasawa, N., 22
Kill_Bkgd_Job Background_Job, 99
Kirkwood–Frohlich equation, 64

L
Language_Control pulldown, 67
Lee, M. A., 22
line geometric object, 193
List Pseudo_Atom, 99
List Setup, 69, 96
Looping_Control Language_Control, 98

M
machine-independent binary databases, 345
Maxwell–Boltzmann distribution, 171
Mean Squared Displacement control panel, 51, 52, 53
mean-square displacement, 50
linear fit, 53
Measurements control panel, 49
minimization
accuracy, 208
additional information, 27
algorithms, 26, 207
Broyden–Fletcher–Goldfarb–Shanno, 26
Broydon–Fletcher–Goldfarb–Shanno, 207
convergence, 244
convergence criteria, 26
Davidon–Fletcher–Powell, 26, 207
Fletcher–Reeves, 26, 207
graphs, 46
iteration information, 337
methods, 207
Newton–Raphson, 26, 207
number of iterations, 27
output, 34, 249
part of a model, 18
periodic systems, 208
Polak–Ribiere, 26, 207
prerequisites, 24
print control, 208
quasi-Newton, 207
setting up, 25
smart minimizer, 26

J
jobs
background, 36
ccontrol, 35
directories, 37
from an existing input file, 40
interactive, 36
machines, 36
monitoring, 36, 38
NQS, 36, 37
password, 37
remote, 36, 37, 38
starting, 40
stopping, 36, 38
user ID, 37
specifying, 26
stages, 26
subcommands, 208
system size, 207
tables, 47
truncated Newton–Raphson, 26, 207
use of databases, 335
uses, 1
Minimize Calculate, 97
minimize command, 204
minimizers
  additional information, 26
  choice, 26, 207
  recommendations, 207
  sequence of use, 26, 207
models
  animation, 45, 46
  anisotropic, 51, 54
  building, 15
  collection of, as trajectory, 43
  conformation from graph, 46
  conformations, 45
  current, 15
  current energy, 25
  information, 46
  large, 22
  loading, 15
  minimization, 25
  nonperiodic, 22, 57, 59
  optimizing conformation, 25
  original structure, 34
  partially known structure, 18
  partially rigid, 18
  periodic, 22, 27, 52, 57, 60, 65
  preparing, 15
  updated at end of run, 34
molecular dynamics, see dynamics
Molecular Simulations, Inc.
  customer support, 8
  website, 8
molecular systems
  definition, 341
  manipulating, 309
molGeom command, 210
monitoring iterative processes, 243
Morse potential, 389

N
neutron scattering pattern, 165
newton parameters, 205
nonbond interactions, 21, 143, 186
  additional information, 21
  buffer width, 22
  cell method, 22
  cell multipole method, 22, 187
  cell-based method, 187
  components, 334
  cutoffs, 186
  cutoff distance, 21
  different treatment for van der Waals and Coulombic, 22
  Ewald calculations, 187
  Ewald sum method, 22
  excluding some from calculation, 328
  illustration of terms, 21
  nonbond list, 22
  spline width, 21
  switching function method, 21
  2D-periodic models, 22
nonbond parameters, 183
Nonbonds Specify, 96
normal mode analysis, 300
NQS Control control panel, 37
NQS Options control panel, 37

O
object command, 217
ordering, 56, 59
Ousterhout, J. K., 134
out-of-plane
  angle, defining, 197, 212
  potential, 399
out-of-plane–out-of-plane interaction potential, 400
output, 34, 105, 249
  dynamics, 34
  levels, 242
  minimization, 34
  specifying, 24
  standard, 106
  types of results, 35
  when specified, 34
Output Analyze, 68, 69, 98
output command, 241

P
packing, 56, 59
parameters
   automatic, 23, 186
   generic, 23
   missing, 23, 186
   scaling, 23, 187
parameters parameters, 182
Parrinello, M., 173
peek command, 243
peripheral bond–torsion interaction potential, 396
phase transitions, 56, 59
Phi_Psi_Map Strategy, 96
plane geometric object, 194
point geometric object, 193
polypropylene, tutorial, 78
power spectrum, 55
   frequency resolution, 55
   maximum frequency, 55
   resolution, 55
precision, 139
prerequisites, 6
pressure, 30, 64
   additional information, 31
   constant, 64
   hydrostatic, 32
   resetting, 33
print command, 174, 246
printing, control of, 174, 241
program dimensioning, 3
programming language, 123
properties, fluctuation, 63
pseudoAtom command, 250
pseudoatoms, 250
   coordinates, recalculating, 252
   defining, 252
   retrieving, 252
Pseudo_Atom pulldown, 67

R
radial distribution function, 56
cutoff distance, 57
Fourier transform, 59
groups, 56, 58
Rahman, A., 173
Raman spectra, 53, 55
random number generator, 138
random seed, setting, 138
rattle command, 256
Rattle control panel, 20
Ray, J., 173
RDF control panel, 58, 61
readFile command, 259
readFile dynamics_restart command, 172, 408
Rename Pseudo_Atom, 98
reset command, 262
restraint command, 263
Restraint Specify, 97
restraints
   chiral, 265
   clearing, 267
   cosine, 265
   creating, 263, 276
defaults, 265
   deleting, 264
disabling, 267
   energies, 268
   examining, 263, 267
   functional form, 266
   modifying, 263
   parameters, 266
   quadratic, 266
   relative, 265
   scaling, 267
   user-defined, 272
results, analysis, 40
   block average, 50, 65
dipole functions, 61
fluctuations, 63
   frequency distribution, 49
graphs, 46
   highest values, 50
   histogram, 49
   history, 49
   loading trajectory, 43
   lowest values, 50
   mean-square displacement, 50
   measurements, 49
   model display, 45
MSD, 50
properties, 49, 65
RDF, 56
running average, 50, 65
saving to table file, 47
selecting, 48
single conformation, 46
subset of results, 44
text window, 46
time-dependent profile, 49
VACF, 53
velocity autocorrelation, 53
Retrieve Discover Structure control panel, 41
RMS_Comparison Calculate, 97
Rokhlin, V. I., 22
Run Discover control panel, 14, 15, 16, 17, 20,
25, 28, 33, 34, 39, 40
Run D_Run, 68, 99
Save Model control panel, 46
scale parameters, 185
Scaling Specify, 97
Schmidt, K. E., 22
sd (steepest-descent) parameters, 205
select command, 272
Select Discover Forcefield control panel, 14, 15
self-diffusion constant, 50, 52
sessions, saving and loading, 14
setting up and starting a CDiscover program
run, 68
Setup pulldown, 67
Setup_Bkgd_Job Background_Job, 99
Simple_Minimize Strategy, 96
Simple_Min_Dyn Strategy, 96
sorting data, 118
sound, speed, 64
Specify pulldown, 67, 96
Spreadsheet icon, 117
Stage Setup, 69, 96
stages, 33, 67, 85
standalone and within Cerius² interface, 40
standalone mode of Discover program, 101
starting a standalone Discover run, 105, 110
starting Discover program, 5, 40, 67
statistics, 48, 249
Strategy pulldown, 67, 96
stress, 30
  additional information, 31
  components, 32
  normal, 32
  resetting, 33
  shear, 32
tensors, 32
structure factor, 59
  cutoff distance, 60
  k-space interval, 60
  resolution, 60
subset command, 278
subsets of atoms, 114, 213, 252
subStructure command, 284
symmetry, 15
Symmetry control panel, 15
System Setup, 68, 96
T
tables, Cerius², 47
  additional information, 35, 47
tables, internal, see databases
Tcl, 134
temperature, 64
  additional information, 31
  constant, 64
  control, 30, 173
  fluctuation, 173
  increasing for a subset of atoms, 340
  initial, 171
target, 171
template forcing, 266
tethering, 266
text editors, 39
thermal effects, 53, 55
thermal pressure coefficient, 64
thermodynamic properties, 64
torsions
  defining, 197, 212
  finding, 199
two-term cosine potential, 395
restraints, 83
setting, 82
single-cosine potential, 393
three-term cosine potential, 394
trajectory
additional information, 42
analysis, 40
animating, 90
collection of models as, 43
frame information, 46
frames, 44
graphs, 46
loading, 43
loading model first, 90
model display, 45
partial, 44
saving to table file, 47
searching, 50
single frames, 45
single frame, saving model, 46
stepping through frames, 45
typewriter font, 9
typographical conventions, 8

U
unnormalized distribution function, 57
Use_Existing_Input Strategy, 96

V
van der Waals
attractive interactions, 22
geometric combination rule, 22
heteronuclear interactions, 403
interaction potential, 402, 404
interactions, 144, 187
nonidentical atom types, 22
repulsive interactions, 22
variables in Insight parameter boxes, 69
vdw parameters, 183
vdWTailCorrection command, 286
vector command, 289
vectors
creating, 296
deleting, 296
manipulating, 299
retrieving, 296
Velocity Autocorrelation control panel, 54, 55, 56
velocity autocorrelation function, 53
Fourier transform, 55
velocity of sound, 64
Vibrational Calculate, 97
vibrational calculations
free energy, 301
frequencies, 300
vibrational properties, 53, 55
vibrationalAnalysis command, 300
volume, 64
constant, 64
expansivity, 64
rate of change, 32

W
warnings, 242
water
fixed-geometry model, 256
SPC, 256
TIP3P, 256
writeFile command, 302

X
X-ray diffraction pattern, 59, 165

Numerics
2,4-dimethylpentane, tutorial, 79