Impact Command Reference Manual

Impact: Integrated Modeling Program using Applied Chemical Theory

Version 6.7, May 2015

For inquiries about Impact

Schrödinger
101 SW Main St., Suite 1300
Portland, OR 97204
503–299–1150
503–299–4532 fax
Email: help@schrodinger.com

Impact Version 67009
May 2015

1 Introduction to Impact

Impact^{TM} (Integrated Modeling Program using Applied Chemical Theory) is an integrated program for molecular mechanics simulations.[1] It allows the user to define the simulation system (usually a protein or DNA molecule in aqueous solution) and to perform Monte Carlo or molecular dynamics simulations. In addition, the user has at her/his disposal a whole array of tools for analyzing the results of the simulations. Finally, Impact is the "driver" for the high-throughput ligand screening program Glide^{TM}, the Liaison^{TM} module for calculating ligand binding energies, and the mixed mode Quantum Mechanics/Molecular Mechanics program QSite^{TM}.

This is the *Impact Command Reference Manual*. It documents using Impact from the command-line, and all the keywords of Impact input files. Running Impact from Maestro, and discussion of the principal applications Glide, Liaison, and QSite, are more fully documented in other manuals:

- *Glide Quick Start Guide*

  A collection of tutorial examples that illustrate the use of Glide.
- *Glide User Manual*

  A description of Glide, focusing on its use from Maestro.
- *Glide Technical Notes*

  A collection of case studies elaborating on the scientific methods and results of Glide.
- *Liaison User Manual*

  A description of Liaison, including its use from Maestro, a tutorial, and notes on the scientific methods and results.
- *QSite User Manual*

  A description of QSite, including its use from Maestro, a tutorial, and notes on the scientific methods and results.

1.1 A Brief History of Impact

The current commercial version of Impact and the Glide, Liaison, and QSite products was developed from the academic Impact originally designed in the laboratory of Professor Ronald M. Levy at Rutgers University. The following people have contributed to the development of Impact:

1.1.1 Commercial Versions

- v5.0 (June 2008) Jay Banks, Yixiang Cao, Wolfgang Damm, Richard Friesner, Emilio Gallicchio, Thomas Halgren, Ronald Levy, Daniel Mainz, Rob Murphy, and Matt Repasky

[1] J. L. Banks et al., *J. Comp. Chem.*, **26**, *1752-1780* (2005)

- v4.0 (November 2005) Jay Banks, Yixiang Cao, Wolfgang Damm, Richard Friesner, Emilio Gallicchio, Thomas Halgren, Ronald Levy, Daniel Mainz, Rob Murphy, Matt Repasky, and Linda Zhang.
- v3.5 (January 2005) Jay Banks, Yixiang Cao, Wolfgang Damm, Richard Friesner, Emilio Gallicchio, Thomas Halgren, Ronald Levy, Daniel Mainz, Rob Murphy, Matt Repasky, and Linda Zhang.
- v3.0 (June 2004) Jay Banks, Yixiang Cao, Wolfgang Damm, Richard Friesner, Emilio Gallicchio, Thomas Halgren, Ronald Levy, Daniel Mainz, Rob Murphy, and Matt Repasky.
- v2.7 (October 2003) Jay Banks, Yixiang Cao, Wolfgang Damm, Richard Friesner, Emilio Gallicchio, Thomas Halgren, Ronald Levy, Daniel Mainz, Rob Murphy, and Matt Repasky.
- v2.5 (January 2003) Jay Banks, Yixiang Cao, Wolfgang Damm, Richard Friesner, Emilio Gallicchio, Thomas Halgren, Ronald Levy, Daniel Mainz, and Rob Murphy.
- v2.0 (June 2002). Jay Banks, Yixiang Cao, Wolfgang Damm, Richard Friesner, Emilio Gallicchio, Thomas Halgren, Ronald Levy, Daniel Mainz, and Rob Murphy.
- v1.8 (September 2001). Jay Banks, Yixiang Cao, Wolfgang Damm, Richard Friesner, Emilio Gallicchio, Thomas Halgren, Ronald Levy, Daniel Mainz, and Rob Murphy.
- v1.7 (March 2001). Jay Banks, Yixiang Cao, Richard Friesner, Emilio Gallicchio, Thomas Halgren, Ronald Levy, Daniel Mainz, Rob Murphy, and Ruhong Zhou.
- v1.6 (November 2000). Jay Banks, Michael Beachy, Yixiang Cao, Richard Friesner, Emilio Gallicchio, Ronald Levy, Daniel Mainz, Rob Murphy, and Ruhong Zhou.
- v1.0 (June 1999). Jay Banks, Richard Friesner, Emilio Gallicchio, Avijit Ghosh, Ronald Levy, Rob Murphy, Anders Wallqvist, and Ruhong Zhou.
- v0.95 (Nov 1998). Jay Banks, Richard Friesner, Emilio Gallicchio, Avijit Ghosh, Ronald Levy, Rob Murphy, Anders Wallqvist, and Ruhong Zhou.
- v0.9 (Aug 1998). Jay Banks, Mark Friedrichs, Richard Friesner, Emilio Gallicchio, Avijit Ghosh, Ronald Levy, Rob Murphy, Anders Wallqvist, and Ruhong Zhou.
- v0.8 (May 1998). Jay Banks, Chris Cortis, Shlomit Edinger, Mark Friedrichs, Richard Friesner, Emilio Gallicchio, Avijit Ghosh, Ronald Levy, Rob Murphy, Anders Wallqvist, and Ruhong Zhou.

1.1.2 Academic Versions

- V7.0 (August 1996). Jay Banks, Yanbo Ding, Gabriela Del Buono, Francisco Figueirido, Ronald Levy, and Ruhong Zhou.
- V6.0 (January 1994). Les Clowney, Francisco Figueirido, Ronald Levy, Lynne Reed, Maureen Smith-Brown, Asif Suri and John Westbrook.
- V5.8 (December 10, 1991). Les Clowney, Francisco Figueirido, Douglas Kitchen, Ronald Levy, Maureen Smith, Asif Suri and John Westbrook.
- V5.7 (December 17, 1990). Steve Back, Teresa Head-Gordon, Douglas Kitchen, Dorothy Kominos, Ronald Levy and John Westbrook.
- V5.5 and earlier (June 1990). Steve Back, Donna Bassolino, John Blair, Fumio Hirata, Douglas Kitchen, David Kofke, Dorothy Kominos, Ronald Levy, Asif Suri and John Westbrook.

1.2 Major Features

The major features of Impact include:

- Energy Minimization
- Molecular Dynamics
- Fast Multipole Method (FMM)
- Multiple Time-step Algorithm r-RESPA
- S-Walking/J-Walking Methods
- Explicit Solvation Model
- Poisson-Boltzmann Continuum Solvation (PBF)
- Surface Generalized Born Solvation Model (SGB)
- OPLS-AA with Automatic Atomtype Recognition
- Flexible Schemes for Freezing Part of System
- QSite: Mixed-Mode QM/MM Simulations for Reactive Chemistry
- Liaison: Calculating and Predicting Ligand Binding Energies
- Glide: High-Throughput Ligand-Receptor Docking

1.3 Hardware Requirements

Schrödinger tests and distributes Glide 6.7, Liaison 6.7 and QSite 6.7 for SGI IRIX, IBM AIX, and Intel-x86 compatible Linux-based machines at this time. Impact 6.7 is not distributed separately from these products. For current information on other platforms, please contact Schrödinger.

1.4 Installation

To install Glide, Liaison, or QSite, see the *Schrödinger Installation Guide*. A PDF version of this manual and product documentation should be on your product CD.

For those that want to get started quickly, installation is often as easy as running:

```
% /bin/sh INSTALL
```

from the CD, and following the prompts. But please see the *Installation Guide*.

After installation, in the directory specified by your `$SCHRODINGER` environment variable, there should be an Impact directory labelled with the current version number, at this printing, this is '`impact-v67009`'. In that directory, there are seven subdirectories:

`bin/`
: The executable binary and scripts for running all manner of Impact-based jobs. Since these are platform-dependent, these files are separated into further subdirectories with their platform's designation, e.g. `Linux-x86/`.

`data/`
: The database parameters for the OPLS series of force fields.

`docs/`
: Electronic versions of the *Impact Reference Manual* (this document) are located here.

`lib/`
: Platform-dependent shared libraries needed by Impact are kept here.

`disabled_lib/`
: Disabled shared libraries, moved from the '`lib/`' subdirectory should be kept here. Disabling libraries should only be done within Schrödinger's recommendations.

`samples/`
: The example files noted in this manual's appendices.

`tutorial/`
: Files that correspond to the instructional material in the *Glide Quick Start Guide*, *Liaison User Manual*, and *QSite User Manual* that walks you through various types of calculations.

A file '`compatibility`' is also in your '`impact-v67009`' directory, listing the minimum version numbers of other Schrödinger products compatible with this Impact release. All Schrödinger startup scripts will use this information automatically.

The single important environment variable each Impact user has to have is `$SCHRODINGER`. It should be set to your top-level installation directory for Schrödinger products, e.g. `/usr/local/bin/schrodinger`. If you plan on using some of the utility scripts from a command-line interface, you might like to add the directory `$SCHRODINGER/utilities` to your `PATH` enviroment variable, so that the scripts in this directory are accessible by name without the full directory name prepended. If your command-line shell is `sh`, `ksh`, or `bash`, this is done by:

```
(sh/ksh/bash)% export PATH=$PATH:$SCHRODINGER/utilities
```

and if your shell is `csh` or `tcsh`, then do:

```
(csh/tcsh)% setenv PATH $PATH:$SCHRODINGER/utilities
```

To run an Impact example, first make sure that `$SCHRODINGER` is set to your Schrödinger installation directory. Then `cd` to one of example directory and type:

```
% $SCHRODINGER/impact -i input_file -o log_file
```

This will read from the *input_file* and write the log file to *log_file*. If `-o` is not specified, Impact will set the log file name to be the same as your input file, but with a `.log` extension in place of `.inp`.

Note that the log file (*stdout*) is not the file specified in the top `write` command in the input file, which is usually more detailed than the log file. Just typing `impact` with no arguments is equivalent to typing `main1m`: the program then looks for an input file named '`fort.1`', and writes to standard output.

If an input file is specified but a log file is not, Impact constructs the log file name by appending the suffix `.log` to the input file name, after first removing the suffix `.inp` if it is present. Thus

```
% $SCHRODINGER/impact -i myfile
```

and

```
% $SCHRODINGER/impact -i myfile.inp
```

will both result in writing a log file called `myfile.log`.

1.5 Input Files

Instructions for Impact are placed in the *main input file*, which is then given as the `-i` argument to the `impact` execution script.[2] The program executes commands in the input file sequentially, or as directed by control structures in Impact's input scripting language, DICE. See Chapter 4 [Advanced Input Scripts], page 127, for details of control structures, variables, and advanced features of DICE. Here is a simple example:

```
WRITE file example.out -
   title Example *

CREATE
  build primary name species1 type auto read maestro -
     file "example.mae"
  build types name species1
QUIT

SETMODEL
   setpotential
      mmechanics consolv agbnp
   quit
   read parm file paramstd noprint
   energy parm cutoff 9.5 listupdate 10 diel 1.0 nodist
   energy rescutoff byatom all
   zonecons auto
QUIT
```

[2] Historically, the main input file had to be assigned to FORTRAN unit number 1, which usually as the filename '`fort.1`'. The name may be different on other machines.

```
DYNAMICS
  input cntl -
      nstep 1000 delt 0.001 stop rotations -
      constant totalenergy nprnt 50 tol 1.e-7
  run
  write maestro file "example_out.mae"
QUIT

END
```

The input file always begins with a description of where to write the output generated by Impact during its execution, and ends with the keyword `end` on a single line. The following meta-example is the simplest legal Impact program:

```
write file fname title your_favorite_title *
end
```

An optional `verbose` *value* argument before the `*` specifies the verbosity of output from various parts of Impact.

After the opening `write` statement, one specifies a sequence of *tasks* that Impact should execute. In Impact tasks correspond to a high-level description of the computer experiment. For example, the task `create` sets up the internal variables describing the molecular structure of the system of interest, while inside of task `dynamics` one runs a molecular dynamics simulation. Typically it is important that tasks are executed in the correct order, which is usually dictated by common sense (the least common of the senses).[3]

A task by itself does not produce any side effects. For instance, the fragment

```
create
quit
```

would do exactly nothing. When Impact begins executing a task it sets up a special environment, which is task-dependent. This environment exists until the keyword `quit` is encountered, closing the task. Within each of these environments different collections of commands (subtasks) are in effect. For instance, within the `create` task one can execute the subtask `build`, but it is not defined inside of the task `dynamics`. Trying to execute `build` inside of the latter task would lead to an error.

Impact requires that tasks (as well as their matching `quit`) be declared on a line by themselves. Subtasks, on the other hand, come in several flavors. They must always be the first non-blank word on a line and most often they are followed on the same line by a series of subtask-specific keywords and parameter values. A few, however, have the same formatting requirements as tasks do, and must be ended by the keyword `quit`.[4]

---

[3] For example, few people we know would run a dynamics simulation before setting the system up.

[4] They act like secondary level tasks.

In general, task and subtask names can be abbreviated by giving the first four characters of the full name. In addition, some special abbreviations are recognized. For example: `minimize` can be entered as `minm`; `energy` can be given as `enrg` (as illustrated above); . . .

Because Impact is written mostly in FORTRAN the implementation puts a limit on the maximum length of a line of 2000 characters. As the lines are scanned lowercase letters are automatically converted to uppercase, unless protected as shown below.[5] The following characters are special:

‘`"`’
: To protect a word and preserve the case. For example, if you want to open a file named ‘`/home/me/FooBar`’, you must write ‘`"/home/me/FooBar"`’.

‘`!`’
: An exclamation point ‘`!`’ flags a comment, and anything following it until the end of the line is not read or processed.

‘`-`’
: A hyphen at a line’s end indicates the command is continued on the next line of the input file. Note that there should be at least one space before the hyphen and that the sum of the lengths of the continued lines must not exceed the limit of 2000 characters.

‘`$`’
: String constants are delimited by this character as in ‘`$foo$`’.

‘`'`’
: The quote is used to delimit names of variables used in Impact input files, as in ‘`while 'foo' lt 10`’.

‘`*`’
: Sometimes *portions* of command lines are terminated with an asterisk. It is required wherever it appears in the examples.

The top level of Impact is the *task* level where the objects of primary interest are described, such as system creation, molecular dynamics or energy minimization. When describing *tasks* in this documentation, meta-examples are generally used, where the following conventions are followed. The order of the keywords inside a subtask is generally not important though, of course, a keyword cannot be separated from its value when one is required.

`keywords`
: that should be typed exactly as shown will appear in this font. Some keywords may be abbreviated by an initial portion of the word, and the examples in this manual contain some such abbreviations; but in the absence of such an example, use the entire keyword as shown.

`variables`
: are meta-keywords, that is, you must replace *variable* with the appropriate `keyword`, number, or filename.

`[ ]`
: is used to delimit keywords that are optional; an extra character, ‘`+`’ or ‘*’, may also be present. `[ ]+` means to repeat the contents

[5] File names that are not protected are actually converted back to lowercase before opening the file.

one or more times and `[ ]*` to repeat the contents zero or more times.[6] For example

```
[ foo | bar | baz ]
```

means that one of the keywords `foo` or `bar` or `baz` may be used in this location. If there are no '`|`' characters present the body is always optional, and if there is a a '`+`' immediately following the ']', as in '`[ foo ]+`', then repeat the contents 1 or more times (here 1 or more occurrences of `foo`).

nil
: stands for the "empty item," that is, no item at all, so that '`[` *`foo`* `|` nil `]`' is equivalent to '`[` *`foo`* `]`'.

`( )`
: in an example indicates that the contents of the parentheses is repeated as many times as indicated by the following expression. In the following expression the symbols '`foo bar baz`' are repeated four times.

```
( foo bar baz ) repeated four times
```

Using the above rules, the meta-example

```
You should [ run | debug ] Impact [ when it rains | nil ]

is expanded in any of the following statements

You should run Impact when it rains
You should debug Impact when it rains
You should run Impact
You should debug Impact
```

One instance of a meta-example for the minimization task is:

```
minimize
   read restart coordinates formatted file fname
   steepest dx0 value dxm value deltae value
   run
   write restart coordinates formatted file fname
quit
```

where *value* refers to the value to be assigned to the preceding keyword, and *fname* refers to a file name.[7]

Some keywords are common to many different tasks and subtasks, so they are described here.

`file`
: This keyword must be followed by the name of a file. In the meta-examples this is generally shown as *fname*.[8]

`name`
: This keyword must be followed by the name of a species. In the meta-examples this is generally shown as *spec*.

---

[6] The other potential uses of the square brackets are discussed in Section 4.1.1 [Lists (Background)], page 128.

[7] Value and number are **usually** equivalent to real and integer. *Val* or *num* are also used in this context.

[8] To refer to the file '`junk`' you would type '`file junk`'.

`resnumber`
: This keyword must be followed by the number (integer value) of a residue. In the meta-examples this is generally shown as *resn*. It should be noted that residue numbers supplied in the main input file have the following meanings: positive numbers mean the residue numbering used in the original PDB file; negative numbers mean the reordered Impact residue numbers (i.e., sequential, starting with 1); 0 means all applicable residues.

`atname`
: This keyword must be followed by the name (character string) of an atom. In the meta-examples this is generally shown as *atna*.

`fresidue`

`lresidue`
: These keywords should be followed by a number specifying the first and last residues of interest in the primary sequence.

`echoon`

`echooff`
: These keywords can appear at the task level, or the subtask level of task `analysis`. They turn on or off the printing of certain output. The default is `echoon`.

An aid to gauging the correctness of an input file is that, in general, as each command is processed it is deleted from the command line. When processing is finished, a check is made to see that no characters remain. The presence of extraneous characters indicates that the input file was incorrectly formed.

1.6 Structure File Formats

Via the `build primary type auto` (see Section 2.2.1.1 [Auto (primary type)], page 15) and `build types` (see Section 2.2.1.3 [Types (build)], page 18) commands, Impact can read and write Maestro, MDL SD, and PDB files.

The freely available program `Babel` is a program that converts different file formats, and currently supports input file formats:

```
Input file type

 1. Alchemy                      2. AMBER PREP
 3. Ball and Stick               4. MSI BGF
 5. Biosym .CAR                  6. Boogie
 7. Cacao Cartesian              8. Cambridge CADPAC
 9. CHARMm                      10. Chem3D Cartesian 1
11. Chem3D Cartesian 2          12. CSD CSSR
13. CSD FDAT                    14. CSD GSTAT
15. Dock PDB                    16. Feature
17. Free Form Fractional        18. GAMESS Output
19. Gaussian Z-Matrix           20. Gaussian Output
21. Hyperchem HIN               22. MDL Isis
23. Mac Molecule                24. Macromodel
```

```
25. Micro World                    26. MM2 Input
27. MM2 Ouput                      28. MM3
29. MMADS                          30. MDL MOLfile
31. MOLIN                          32. Mopac Cartesian
33. Mopac Internal                 34. Mopac Output
35. PC Model                       36. PDB
37. JAGUAR Input                   38. JAGUAR Output
39. Quanta                         40. ShelX
41. Spartan                        42. Spartan Semi-Empirical
43. Spartan Mol. Mechanics         44. Sybyl Mol
45. Sybyl Mol2                     46. Conjure
47. UniChem XYZ                    48. XYZ
49. XED                            50. M3D
```

and output file formats:

```
Output file type

 1. DIAGNOSTICS                     2. Alchemy
 3. Ball and Stick                  4. BGF
 5. Batchmin Command                6. Cacao Cartesian
 7. Cacao Internal                  8. CAChe MolStruct
 9. Chem3D Cartesian 1             10. Chem3D Cartesian 2
11. ChemDraw Conn. Table           12. MSI Quanta CSR
13. Dock Database                  14. Wizard
15. Conjure Template               16. CSD CSSR
17. Feature                        18. Fenske-Hall ZMatrix
19. Gamess Input                   20. Gaussian Cartesian
21. Gaussian Z-matrix              22. Gaussian Z-matrix tmplt
23. Hyperchem HIN                  24. Icon 8
25. IDATM                          26. Isis
27. Mac Molecule                   28. MacroModel
29. Micro World                    30. MM2 Input
31. MM2 Ouput                      32. MM3
33. MMADS                          34. MDL Molfile
35. Mopac Cartesian                36. Mopac Internal
37. PC Model                       38. PDB
39. JAGUAR Z-Matrix                40. JAGUAR Cartesian
41. Report                         42. Spartan
43. Sybyl Mol                      44. Sybyl Mol2
45. MDL Maccs file                 46. XED
47. UniChem XYZ                    48. XYZ
49. M3D
```

Before you run `babel`, you need to setup an environmental variable $BABEL_DIR:

```
% setenv BABEL_DIR your_babel_directory
% export BABEL_DIR= your_babel_directory
```

The easiest way to run `babel` is in manual mode:

```
% babel -m
```

and follow instructions to select desired input and output file formats. You can also run `babel` from the command line, as in

```
% babel -ix myfile.xyz -renum -oai myfile.dat "AM1 MMOK T=30000"
```

This will create a MOPAC input file with atom 1 from *myfile.xyz* as atom 1 in *myfile.dat*. For details of how to run `babel`, etc, consult the README files under the `babel` directory. `babel` also comes with Schrödinger's product Jaguar, and is accessible therein via the `jaguar babel` command.

1.7 Force Field

In molecular modeling there are several different force fields used to describe the interactions among atoms and molecules. Some of the well known ones are OPLS, MMFF, AMBER, MM3, CHARMm, and GROMOS. Impact currently supports OPLS-AA[9].

1.7.1 OPLS-AA

The OPLS-AA force field, which was developed by the Jorgensen group, is an effort to develop a parameterization that reproduces liquid state properties of molecules. Again this is a force field that uses experimental data from the liquid state and quantum mechanical calculations for intramolecular bond, angle, and torsion motions to set the constituent parameters. The intramolecular interaction is given as,

$$V_{\mathrm{intra}} = \sum_{\mathrm{bonds}} K_r (r - r_{eq})^2 + \sum_{\mathrm{angles}} K_\theta (\theta - \theta_{eq})^2 + V_{\mathrm{torsion}}$$

where V_{torsion} written as,

$$V_{\mathrm{torsion}} = \sum_i \frac{V_1^i}{2} [1 + \cos(\phi)] + \frac{V_2^i}{2} [1 - \cos(2\phi)] + \frac{V_3^i}{2} [1 + \cos(3\phi)] \,.$$

The non-bonded interaction is given as a van der Waals terms together with an electrostatic term (R is again the atom-atom distance),

$$V_{\mathrm{inter}} = \sum_{i<j} \left[4\epsilon_{ij} \left(\frac{\sigma_{ij}^{12}}{R_{ij}^{12}} - \frac{\sigma_{ij}^{6}}{R_{ij}^{6}} \right) + \frac{q_i q_j}{R_{ij}} \right].$$

Note that in this description the dielectric constant is set to its proper value of 1.0. For molecules containing atoms connected by a distance of more than 3 bond-lengths the atom-atom interaction is given by the V_{inter} -term. The (1,4)-interactions are scaled by a factor of 1/2. The non-bonded parameters ϵ and σ for each atom-pair is constructed from the atomic values by the geometric mean combination rule,

$$\epsilon_{ij} = \sqrt{\epsilon_i \epsilon_j}$$
$$\sigma_{ij} = \sqrt{\sigma_i \sigma_j}.$$

[9] W. L. Jorgensen, D. S. Maxwell, and J. Tirado-Rives, *J. Amer. Chem. Soc.*, **118**, *11225–11235* (1996)

It is also possible to use the partial charges read from a Maestro or MacroModel format structure file instead of those provided by OPLS-AA, using the `cmae` keyword documented in Section 2.2.1.1 [Auto (primary type)], page 15.

1.8 Online Documentation

Schrödinger publishes PDF versions of all product manuals at the website `http://www.schrodinger.com/Support/pdf.html`. An up-to-date copy of this manual, the *Impact Command Reference Manual*, along with other manuals, are linked there.

2 Setup System

This chapter describes tasks to set up Impact simulations: create system, and set up models, etc. This should be done before any real simulation tasks can be performed.

2.1 Set commands

These commands are not true tasks, in that they are completely specified on one line, with no subtasks and no `quit` keyword. They are used to specify conditions of the Impact execution that typically remain the same throughout the duration of the program, so they should usually occur at the beginning of the input file, either immediately after or even before the initial `write` command that specifies the main output file. In particular, `set ffield` may have unpredictable results if it occurs in the middle of an input script, or if two or more `set ffield` commands are issued in the same script.

2.1.1 Set Path

This command specifies a directory where Impact will look for input files specified in subsequent commands. The directory name is added to a list stored in memory. When Impact starts up, the list contains '`.`' (the current working directory), and a default directory that normally is '`$SCHRODINGER/impact-v6.7/data`'. The `set path` command adds **one** directory to the **end** of this list. Thus the specified directory will be searched only for files that cannot be found in the current working directory, the default directory, or directories specified by previous `set path` or `set ffield` commands. To specify more than one directory, use more than one `set path` command, one for each directory in the order you wish them to be searched.

- `set path` *`dirname`*

2.1.2 Set Ffield (or Set Force)

This command specifies the force field that Impact uses to calculate energies and forces. This has two consequences:

> A directory that contains the parameter and residue database relevant to the specified force field is added to the **beginning** of the search path, after only the current working directory. Thus the correct residue and parameter files will be used instead of the default ones.
>
> A flag is set that indicates which force field is being used. This flag determines the functional form used in energy and force calculations.

- `set ffield` *`ffname`*

Currently the values that can be used for *ffname* are `OPLS2001`,and `OPLS2005`.

`OPLS2001` generally uses pre-2000 OPLS force field parameters. `OPLS2005` is a new parameterization which includes optimized parameters for proteins[1] and ligands.[2]

2.1.3 Set Noinvalidate

Maestro files can embed properties, such as energies and structure identifiers, that implicitly only correspond to the particular structure, connectivity, or even precise Cartesian coordinates of the atoms. Maestro files can encode these *dependencies* in such a way to tell other Schrödinger software when they are invalid and should be deleted from the structure.[3]

For example, if an input structure already has a property `r_mmod_Potential_Energy-OPLS-AA`, this is an energy that corresponds to the particular geometry of the molecule. If any of the internal coordinates are changed, the energy value is no longer valid. Such properties are removed if and when geometries are modified, and upon output of the structure, they will not appear.

Sometimes, however, it is desired to retain all the input properties through a complicated workflow. Perhaps you have minimized a number of ligand structures with MacroModel, and then dock them with Glide using its internal conformation generator. Normally, when Glide does its conformation generation, it invalidates all the input properties known to depend on the internal coordinates of the structure, including the MacroModel energies. If you want your output PoseViewer files to keep these properties, even if they don't correspond to the coordinates anymore, and also have the Glide pose properties, which do correspond, then you must add this `set noinvalidate` property to your Glide input file.

- `set noinvalidate`

Caution: This option is a temporary measure. In the future, we intend to introduce an easy-to-use method in Maestro to tailor each property's invalidation setting, so you can clear invalid ones while fixing other ones, to your preference.

2.2 Task Create

The object of this task is to set up, modify and process the internal coordinates of the molecules in the simulated system. Very few things can be done without first setting up the system, so this task is typically among the first to be executed. Remember, however, that Impact input files should start with a line that identifies the name of the log file and a descriptive title. Thus, the typical Impact input file has the structure

[1] G. A. Kaminski, R. A. Friesner, J. Tirado-Rives and W. L. Jorgensen, *J. Phys. Chem. B*, **105**, *6474–6487* (2001)

[2] J. L. Banks et al., *J. Comp. Chem.*, **26**, *1752-1780* (2005)

[3] These dependencies are denoted by a `m_depend` block in Maestro files.

```
write file logfile title Some title *
set commands if desired
create
  Set up the simulation system
quit
setmodel
  Set up the model parameters
quit
Perform the calculations
end
```

2.2.1 Subtask Build

This subtask is used to initialize or modify the connectivity arrays, internal and cartesian coordinate arrays, residue arrays, and charge arrays for the molecule(s) specified by the user. The modification may be a conformational change (i.e., a change in secondary structure), or the insertion of connectivity information (for crosslinks), or the addition of a user defined residue into a molecule. '`Build primary`' must be called before any further calculations to fill the arrays.

2.2.1.1 Primary type Auto

- `build primary type auto name` *spec* `-`
  `[mole` *molname*`] [check] -`
  `[ gotostruct` *structnum* `| nextstruct ] -`
  `read [ maestro | pdb | sd ] file` *filename* `-`
  `[tagged` *tagname*`] [ cmae ] [ fos | fobo ] -`
  `[ notestff | notest ]`

The '`type auto`' option of the '`build primary`' command is generally used to interface Impact to the Maestro graphical front end. An Impact species of type '`auto`' contains internally all of the information necessary to produce a molecular file in Maestro format that can later loaded into the Maestro graphical front end. If the species is constructed using exclusively files in Maestro format it is ensured that graphical and other information originally contained in the input Maestro files is carried over to the Maestro file in output (see Section 3.1.6 [Read/write (minimize)], page 53). The '`build primary type auto`' command also supports input from PDB and SD files; in these cases Impact essentially converts these formats to Maestro format internally.

`name` Specifies the identifier *spec* of the species to be created or the of the existing species to which a new molecule is to be added.

`mole` Specifies the identifier *molname* of the molecule to be created.

`check` Instructs Impact to compare the molecular structures of the molecules currently loaded in the species with the ones being loaded. If the two sets are considered chemically identical, except perhaps for a conformational difference, the automatic atom typing of the molecules are not performed even if the `build types` (see Section 2.2.1.3 [Types (build)], page 18) is subsequently invoked. Otherwise all the molecules present in the species are deleted and replaced with the molecule being loaded and the '`build types`' will preserve its normal behavior.

The `check` keyword is necessary after the first structure when reading multiple structures sequentially into the same Impact species. Without it, the atoms of the new structure are appended to those already in the species, rather than replacing them. When reading multiple structures in a `while-endwhile` loop (see Section 4.3.1.1 [while (control)], page 140), the first `build primary` command must occur before entering the loop, without the `check` keyword, whereas the `build primary` command inside the loop must be `build primary check`. Such loops are standard procedure in the Glide docking module (see Section 3.5 [Docking], page 72).

`maestro` Specifies that the molecular file in input is in Maestro format (usually denoted by a `.mae` file extension). The '`tagged`' option is used to specify that only the subset of the atoms tagged with the specified tag *tagname* are to be loaded. Sets of atoms are sometimes tagged by the Maestro front end to identify special structures of the system (such as the ligand in a ligand-receptor complex, often tagged `LIG_`) in order to instruct Impact to handle them in special ways (such as loading the ligand in a different Impact species from the receptor).

`tagged` An option used with files in Maestro format. See note above.

`pdb` Specifies that the molecular file in input is in PDB format (usually denoted by a `.pdb` file extension).

`sd` Specifies that the molecular file in input is in `MOL` format (usually denoted by a `.mol` or `.sdf` file extension).

`gotostruct`
`nextstruct`
Used for multi-structure files, files that contain a sequence of structures rather than a single structure. '`gotostruct`' instructs to read the structure at the position *structnum* in the file. '`nextstruct`' reads the next available structure in the file starting from the last accessed position (or the first structure if the file has been accessed for the first time). The default is to read the current structure (the first structure or the last accessed structure). Note that Impact maintains only a record of the position of the current open file, so that if *file1* and then *file2* are accessed in sequence, the position information of *file1* is lost.

`cmae` Read partial charges for all atoms from Maestro files. These override charges that OPLS-AA would assign.

`fos` Use formal charges from Maestro or SD files for single atoms. This allows you to choose specific oxidation states for ions, e.g., Fe3+ instead of OPLS-AA's default for Iron, Fe2+.

`fobo` Use all formal charges and bond orders from the input Maestro or SD file, overriding the assignments that the OPLS-AA typer would make.

`notestff` The default behavior of `build primary auto` is to check the Lewis structure of the species and skip further processing of structures for which no valid Lewis structure could be generated. The '`notestff`' keyword allows processing of the species regardless of the validity of its Lewis structure. Accepting input structures that are not correct Lewis structures may be necessary in the QM region of mixed QM/MM calculations (see Section 2.3.8 [Subtask QMregion], page 39), where the Jaguar program will determine the correct structure. For additional information regarding Lewis structure checking see the '`lewis`' or '`ifo`' keywords.

CAUTION: we **strongly discourage** use of the '`notestff`' keyword for structures other than those that contain the QM region of QSite jobs, unless you are sure that the connectivity, bond orders, and formal charges of your input structure are correct. Forcing the program to process incorrect structures can lead to serious errors in results.

The keyword is applied to all species that undergo a `build types` command until the next `build primary auto` command where the default behavior is reverted to unless another '`notestff`' command is given.

2.2.1.2 Solvent

Impact distinguishes between species that are used primarily as *solvent* and those that are used as *solute*. This option should be used in the place of '`build primary`' to specify the nature of the solvent.[4] A typical although simplified use is given in the following example:

```
CREATE
   build primary name dipep type auto read maestro file "gly2.mae"
   build solvent name water type spc nmol 216 h2o
   build types name dipep
   build types name water
QUIT
```

[4] There can be only one solvent species in Impact.

If both solvent and solute are present, then Impact will automatically remove those solvent molecules that overlap the solute. The removal algorithm is based on safe default settings which however may cause the removal of too many solvent molecules, giving a total system density that is too low. These settings can be modified using the `mixture` subtask of the `setmodel` task (see Section 2.3.5 [Mixture (setmodel)], page 34).

- `build solvent name` *spec* `type [ spc | tips | tip4p ] nmol` *num* `h2o`

Builds the structural arrays for the solvent species *spec*. It can handle SPC, TIP3P and TIP4P water models. The parameter to `nmol` gives the initial number of molecules (which might be different from the final value (see Section 2.3.5 [Mixture (setmodel)], page 34) .

2.2.1.3 Types

- `build types name` *spec* `[pparam] [lewis` *int*`|ifo` *int*`] -`
  `[patype` *int*`] [plewis` *int*`]`

Assigns OPLS-AA atom types to species *spec*.

Most, but not all, of the Impact tasks require the ability to calculate the energy of the system using a force field. A force field is based on the assignment of an atom type to each atom. Impact provides a facility to automatically assign `OPLS-AA` atom types to a molecular system and to automatically recognize which bonds, bond angles and torsions are to be included in the energy calculation. This facility is invoked by the '`build types`' command. The automatic atomtyping procedure is time consuming especially for large molecules. For species built stepwise from individual molecules invoke the '`build types`' command only when the species is completed rather than after each build command. For example the sequence of commands

```
CREATE
  build primary type auto name complex mole receptor -
    read maestro file receptor.mae
  build types name complex
  build primary type auto name complex mole ligand -
    read maestro file ligand.mae
  build types name complex
QUIT
```

and

```
CREATE
  build primary type auto name complex mole receptor -
    read maestro file receptor.mae
  build primary type auto name complex mole ligand -
    read maestro file ligand.mae
  build types name complex
QUIT
```

will generate identical molecular systems with identical `OPLS-AA` atom types assignment, but the latter will execute in less time.

The Lewis structures of all species to be typed are, by default, checked prior to the assignment of atomtypes and force field parameters. If the species is found to have a valid Lewis structure, the species is passed to the automatic atomtyping routine. If the Lewis structure is found to be invalid, the Lewis structure refinement process is initiated and an attempt is made to generate a valid Lewis structure. If no valid Lewis structure is generated, further processing on the species is halted unless the '`notestff`' flag is employed in the '`build primary auto`' command. The behavior of the Lewis structure checking/refinement process is controlled via the '`lewis`' or '`ifo`' arguments as shown below.

- '`lewis 1`' - Use formal charges for isolated atoms from the input structure. Equivalent to setting the '`fos`' flag for a '`build primary auto`' command.
- '`lewis 2`' - Use formal charges and bond orders from the input structure. No Lewis structure check is performed. Equivalent to setting the '`fobo`' flag for a '`build primary auto`' command.
- '`lewis 5`' - Default behavior. First test if input structure is valid, if not then attempt to generate a valid Lewis structure.

To print the atom types and force field parameters assigned, add the `pparam` flag to the '`build types`' command. For more verbose printing from the automatic atomtyping process, use the `patype` flag with increasing verbiage in going from values of 1 to 6. For more verbose printing from the Lewis structure checking/refinement process, use the `plewis` flag which will output increasing verbosity in going from values of 1 to 6.

2.3 Task Setmodel

The object of this task is to process energy, structural and simulation parameters required for the following simulations:

- pure solute;
- pure solvent;
- mixed solute-solvent;
- crystal.

This task must be completed before calls to `minimize`, `dynamics`, or subtasks of `analysis` requiring energy evaluations.

2.3.1 Subtask Energy

Read in information needed to calculate force and energy in MM, MD and MC simulations, including boundary conditions, potential cutoff, constraints, and screening of Coulomb interactions. The following options are allowed in subtask `energy`.

2.3.1.1 Periodic

Sets up periodic boundary conditions for species *spec* based on the supplied `bx`, `by`, `bz` box dimensions, which should be in Å. Instead of specifying a species by name you can use the keyword `all`.

- `energy periodic [ name` *spec* `| all ] [ bx` *val* `by` *val* `bz` *val* `]`

2.3.1.2 Molcutoff/Rescutoff

- `energy [ molcutoff | rescutoff ] [ byatom | bycm ] [ all | none | name` *spec* `]`

Specifies that a molecular (`molcutoff`) or residue-based (`rescutoff`) group cutoff scheme should be used for species *spec*. The `byatom` and `bycm` options control the criteria according to which two atom groups (two molecules or two residues) are considered neighbors. Using `byatom` mode two atom groups are considered neighbors if any two atoms belonging to different groups are closer than the cutoff distance. Using `bycm` mode two atom groups are considered neighbors if the corresponding centers of mass are closer than the cutoff distance. If `byatom` is specified for species *spec1* and `bycm` is specified for *spec2* then an atom group of *spec1* is considered neighbor of an atom group of *spec2* if the distance between any atom of the first atom group and the center of mass of the second group is smaller than the cutoff distance. The default is `byatom` for the residue-based cutoff scheme (`rescutoff`) and `bycm` for the molecule-based cutoff scheme (`molcutoff`). The `all` option can be used to apply to all species the specified group cutoff scheme. If instead `none` is given, an atom-based cutoff scheme is applied to all species. If a group cutoff scheme is not specified for a species then an atom-based cutoff scheme is assumed.

The term *group cutoff* implies that, if two atom groups (molecules or residues) are considered neighbors, every atom in the first group are considered neighbors to every atom in the other group regardless of their interatomic distance. (In the non-bonded energy calculation the actual distance between each pair of neighboring atoms is used.) For simulations involving water, for example, molecular cutoffs should always be used in order to avoid splitting dipoles in the electrostatic energy calculation. With respect to molecular-based cutoffs a molecule is defined as a covalently linked set of atoms. A residue can not span more than one molecule so, for example, each water molecule is a separate residue. For proteins a residue-based cutoff scheme should be preferred over an atom-based cutoff scheme. In the OPLS force field each residue has a zero or integral total charge (a charge group) therefore a residue-based cutoff scheme avoids some of the major dipole splitting problems inherent in an atom-based cutoff scheme.

2.3.1.3 Constraints

Instruct Impact to read in bonds or distances that should be constrained during molecular dynamics using the `SHAKE` method. There are two ways of specifying constraints:

- `energy constraints read file` *`fname`*

will read the constraints from the given file (see below for a description of the format of the constraint file). Alternatively,

- `energy constraints (bonds [ water ] | lonepairs )`

constrains all bonds to their equilibrium values based on the bond parameters read in by `setmodel read`. Therefore, parameters must be read first for this option to work. Note that **all** species will be thus constrained. If the optional keyword `water` is present only the bond lengths of water molecules are constrained. The keyword `lonepairs` is a little more complicated. It finds all atoms whose names have the first two letters LP and adds the bonds and angles associated with them to the `SHAKE` constraints. Lone pairs move too much due to their low atomic weight and therefore this option should be used when the force field is AMBER86 and cysteines and methionines, which contain LP's on the sulfur, are present. The added constraints only apply to bonds made directly to the LP's (such as SG–LP) and the angles involving two LP's (such as LP–SG–LP). The command

- `energy constraints angles water`

constrains the H–H distance of water molecules to the value obtained from the equilibrium bond length and angle. The commands

```
energy constraints bonds water
energy constraints angles water
```

allow to perform MD simulations with rigid water models (SPC, TIP4P, and TIP3P) without constraining the other molecules in the system, without having to explicitly define a constraints file (see above) or in cases when a constraints file can not be used, such as when water molecules are part of a `type auto` species (see see Section 2.2.1.1 [Auto (primary type)], page 15). The commands

```
energy constraints bonds
energy constraints angles water
```

rigidify water molecules and constrain the bond lengths of all the other molecules in the system.

The maximum allowed number of iterations in the `SHAKE/RATTLE` algorithms can be controlled with the keyword `maxiter` (default: 1000)

- `energy constraints maxiter` *`num`*

2.3.1.4 Constraint file format

1. The file that contains the constrained distances is free format but the following lines are read in:
   - Number of constraints for a species.
   - Pairs of atoms constrained and constrained distance value.
     Caution: it is expected that constraints for all species are in one file and these are added to the list for the species, e.g.,

     ```
     energy constraints bond
     ```

     can be used first followed by

     ```
     energy constraints read file fname
     ```

where *fname* contains only the list of distances needed to constrain angles.

2. Sample constraint files
   - for H_2O constraining OH distances to 1.0 Å and HH distance to 1.633 Å:

     ```
     3
     1 2 1.0
     1 3 1.633
     2 3 1.0
     ```

   - If species 1 is unconstrained and species 2 is constrained water:

     ```
     0
     3
     1 2 1.0
     1 3 1.633
     2 3 1.0
     ```

Caution: If the option '`energy constraints bond`' is chosen and a constraint file is not read, all bonds in the molecule are constrained to their equilibrium values. This is done using the `SHAKE` algorithm.

(energy), Energy (setmodel)

2.3.1.5 Torsional Restraints

The following commands are useful to restrain torsional dihedral angles of the system near the current values or supplied values. These restraints are implemented as flat-bottom harmonic penalty potentials:

$$U(\phi) = \frac{k}{2}[\phi - (\phi_0 + \Delta)]^2 \quad \text{if } \phi > \phi_0 + \Delta$$

$$U(\phi) = \frac{k}{2}[\phi - (\phi_0 - \Delta)]^2 \quad \text{if } \phi < \phi_0 - \Delta$$

and 0 otherwise, where ϕ is the dihedral angle, ϕ_0 is the reference angle, Δ is the half-width of the flat-bottom region, and k is the force constant.
The command

- `energy restrain torsions all forcec` *`value`* `[range` *`value`*`]`

restrains all dihedral angles associated with a torsional potential energy term. The value of `forcec` is the force constant in kcal/mol/degrees^2 , the range parameter sets the half-width of the flat-bottom harmonic potentials in degrees. The range parameter can be omitted in which case it is set to zero (pure harmonic restraint).
To restrain specific dihedrals for a particular species use the command:

- `energy restrain torsions name` *`name`* `read file` *`file`*

The parameters of the restraining potential are read the specified file. Each line in this file represents a dihedral angle to be restrained. The format of each line is:

```
forcec  phi0    i    j    k    l     range
```

where `forcec` and `range` have the same meaning as above, `phi0` is the center of restraining potential, and `i`, `j`, `k`, and `l`, are the internal atom indexes of the atoms specifying the dihedral angle. Both types of commands can be given, in which case the restrains specified by the second command are added to the ones created by the first.

Torsional restrain parameters are reported in the output file with a verbose level of 3 or higher (see Section 1.5 [Input Files], page 5). The energy penalty of each individual restrained dihedral is reported in the output file at the end of a minimization task.

2.3.1.6 Parm

Read in parameters such as nonbonded cutoffs and nonbonded list update frequency, which are used by several energy manipulation tasks such as `dynamics`, `minimize`, `montecarlo`, `tormap`, and `potfield`.

- `energy parm cutoff` *`value`*

Sets a given cutoff distance to the length specified in *value*, which should be in Å. The keyword `cutoff` selects the nonbonded cutoff, which is used for both the Lennard-Jones and the electrostatic interactions (unless the Fast Multipole Method is used). This is a sharp cutoff which is meant to be used with either implicit solvation or with long range electrostatic treatments such as Ewald. The specification of a non-bonded cutoff value is necessary for systems, such as those with periodic boundary conditions, that require a non-bonded neighbor list. Conversely, the absence of the `parm cutoff` option in the input file turns off the use of non-bonded neighbor lists entirely; all non-bonded interactions are computed (excluded interactions such as 1,2 interactions are honored).

- `energy parm scr14` *`value`*

Sets the 1–4 nonbonded screening constant (2.0 by default).

- `energy parm [ dielectric` *`value`* `[ distance | nodistance ] ]`

Sets the value of the dielectric constant (1.0 by default). These options allow the choice of a distance-dependent or a constant dielectric function. One of these must be specified or the program will stop.

- `energy parm listupdate` *`num`*

Sets the number of steps between updates to the nonbonded (Verlet) list. If `listupdate` is not specified, it defaults to 10.

- `energy parm outcutoff` *`value`* `outlistupdate` *`num`*

Sets the cutoff radius and number of steps between updates for the outer neighbor list. When these optional parameters are specified an outer neighbor list is used. When the main non-bonded neighbor list is updated only the outer neighbor list is scanned rather than the entire system. If the outer neighbor list is updated more infrequently than the non-bonded neighbor list, using the outer neighbor list leads to a significant reduction of the time required to update the non-bonded neighbor list, particularly for large systems (>4,000 atoms).

- `energy parm hmass` *value*

Sets the mass of hydrogen atoms (in atomic mass units). Increasing the mass of hydrogen atoms from their physical value (1.008 amu) can be useful for improving the stability of the MD integrator and for possibly using longer MD time-steps. A value of 5 amu has been generally found to yield good results. Note that changing the mass of the system changes its kinetic properties. In classical mechanics however thermodynamic quantities are, in principle, strictly independent of the atomic masses.

- `energy parm print` *num*

Sets the frequencies at which the energy terms are printed to the output.

2.3.2 Subtask Read

This command is used to read in energy parameters from a separate file or from the main input file.

2.3.3 Subtask Setpotential

Read in information about the chosen potential function. Each option at the outermost level (as `mmechanics`) should be on its own line.

2.3.3.1 Mmechanics

Sets up a standard molecular mechanics potential function taking the following options.

```
• mmechanics [ all | name spec | nil ] -
    [ force | noforce | nil ] -
    [tail | notail | nil ] [ nobond ] [ noangle ] [ notors ] [ no14 ] -
    [ nohb ] [ novdw ] [ ewald [ kmax km ] [ alpha alfa ] ] -
    [ fmm level level maxpole poles [ smoothing ] ]
    [ consolv [ pbf | sgb | agbnp | nil ] consolv_options ]
```

`all` Use of `all` flags that the options `nobond`, `noangle` and `notors` refer to all species, otherwise use `species` *spec*.

`force`
`noforce` `Force`/`noforce` determine whether forces should be calculated. Forces are required for `minimization` and `dynamics`. (This is the default.) Currently this option is ignored if the Fast Multipole Method is used.

`tail`
`notail` Determines whether long-range corrections to the van der Waals energies due to cutoffs are made. `Tail` is needed for constant pressure simulations (the default is `notail`).

`nobond` Flag to turn off `bond` stretching term.

`noangle` Flag to turn off valence angle bending term.

`notors` Flag to turn off torsional twisting term.

`no14` Flag to turn off both 1-4 interaction term (`nonb14` and `noel14`).

`noel` Flag to turn off electrostatic term.

`nohb` Flag to turn off hydrogen bond term.

`novdw` Flag to turn off van der Waals (non-bonded) interaction term.

`ewald` Makes Impact use the Ewald summation method to handle the long-range electrostatic interactions. It only works if all species have periodic boundary conditions. To describe the parameters following the keywords `kmax` and `alpha` it is convenient to recall the definition of the Ewald potential (with 'conducting boundary conditions'):

$$\Phi(\mathbf{x}) = \sum_{\mathbf{n}} \frac{\mathrm{erfc}(\alpha\|\mathbf{x}+L\mathbf{n}\|)}{\|\mathbf{x}+L\mathbf{n}\|} + \sum_{\mathbf{k}\neq\mathbf{0}} \frac{4\pi}{L^3\|\mathbf{k}\|^2} \exp\left(-\frac{\|\mathbf{k}\|^2}{4\alpha^2} + i\mathbf{k}\cdot\mathbf{x}\right) - \frac{\pi}{L^3\alpha^2}.$$

This formula represents a solution to the Poisson equation for a unit charge under periodic boundary conditions (there is a negative background that renders the system neutral, as otherwise it can be shown that there is no solution) as a sum of two infinite series, both of which converge exponentially. The first, so-called 'real-space sum', converges faster the larger the value of α is. Conversely, the second sum converges faster the smaller this value. Impact restricts the first sum to the original copy, that is, it only considers the terms with $\mathbf{n} = \mathbf{0}$. The second sum, the 'reciprocal-space sum', is restricted to those values of $\mathbf{k}$ whose components are, in magnitude, less than or equal to the parameter specified by the keyword `kmax` (default: 5). The α parameter has by default the value $5.5/L$, where L is the linear dimension of the box (which must be cubic). The user can change this value, however, with the `alpha` keyword. Note, however, that changing this parameter might require changing the maximum number of reciprocal-space vectors also. A good reference for the Ewald summation method is the book by Allen and Tildesley, *Computer Simulation of Liquids*, Oxford University Press, 1991. For the mathematically inclined we recommend also the article: de Leeuw, Perram and Smith, *Simulation of electrostatic systems in periodic boundary conditions. I. Lattice sums and dielectric constants*, Proc. R. Soc. London, **A373**, 27–56 (1980).

`fmm` Selects the Fast Multipole Method (FMM) for the calculation of the electrostatic interactions. The number following `level` should be the desired number of levels in the hierarchical tree. Since the nodes of the tree correspond to subsequent subdivisions of the simulation box into halves along each direction, if level l is selected, the number of boxes at the lowest level will be 8^l and the linear dimension of each one box at that level will be $L/2^l$ with L being the linear dimension of the simulation box (which must be cubic).

The number following `maxpole` is the maximum number of multipole moments that will be used to approximate the potential and field produced by 'far' clusters. Currently a minimum of four (4) and a maximum of twenty (20) multipoles are allowed. The keyword `smoothing` determines whether a sharp or smooth cutoff are used to separate the direct forces into near and far components. It is only relevant when using the Reversible RESPA integrators (see Section 3.2.2 [Dynamics Subtask Run], page 58) with more than two stages. If periodic boundary conditions are in effect, the potential that gets computed coincides with the Ewald potential (see above), but the algorithm is completely different. One important restriction when using the FMM with periodic boundary conditions is that the system must be electrically neutral, i.e., the sum of all point charges must be zero. The main reference for the FMM is Greengard's thesis, *The Rapid Evaluation of Potential Fields in Particle Systems*, The MIT Press, Cambridge, 1988.

Because FMM calculations scale linearly with the total number of atoms, they can provide a significant speed advantage in calculating electrostatic interactions for large systems when it is not desirable to use cutoffs. Systems large enough for FMM to be advantageous may be large macromolecules or complexes of them, or smaller molecules with a large number of explicit solvent molecules. If it is possible to impose periodic boundary conditions, then the Ewald method (which requires such boundary conditions) tends to be faster than FMM for systems containing more than about 20000 atoms.

PLEASE NOTE: The Fast Multipole Method cannot currently be used with the truncated Newton minimization algorithm (`tnewton`) (see Section 3.1.3 [Subtask Tnewton], page 52), or with SGB continuum solvation (see below). It is available with PBF continuum solvation (see below), but the FMM is not applied to the continuum solvent itself. Unless the solute is quite large, therefore, it may not be advantageous to use FMM with continuum solvent.

`consolv [sgb]`

- `mmechanics consolv sgb [ cutoff` *`val`* `] -`
  `[ npsolv ] [ debug` *`val`* `]`

SGB, the default option for `consolv` is a surface area based version of the Generalized Born model, which can be proved to be a well-defined approximation to the boundary element formulation of the Poisson-Boltzmann (PB) equation[5]. The relationship of the surface area methodology to the volume-integration based approach of the original GB model[6] can be found in Ghosh et al.'s paper. With empirical corrections, SGB produces significant improvements in accuracy, as compared to the uncorrected GB model.

PLEASE NOTE: This solvation method cannot currently be used with the Fast Multipole Method FMM (see above).

`cutoff` The `cutoff` parameter specifies how far any atom must move from the coordinates used in the previous calculation before a new Reaction Field calculation is performed. The default value is 0.1 Å. If all atomic coordinates have moved less than this cutoff, then the previous calculated energy and forces are used for that step in the minimization. A relatively large value of `cutoff` can significantly reduce the required computational time at the expense of some loss in accuracy.

`npsolv` The `npsolv` keyword will turn on the properly parametrized dielectric radii and nonpolar parameters for SGB continuum solvent simulations. The parametrization was done by fitting the SGB calculated free energy coupled with a novel nonpolar function[7] against small molecule experimental solvation free energies.

`debug` Setting `debug` to a nonzero value causes diagnostic messages and files to be printed for each calculation.

The `consolv sgb` parameter files are in the directories

```
$SCHRODINGER/impact-v6.7/data/opls
$SCHRODINGER/impact-v6.7/data/opls2000
```

and all start with `sgb`. The files should not need to be modified by the user on an ongoing basis; most useful parameters can be

---

5 A. Ghosh, C. S. Rapp, and R. A. Friesner, *J. Phys. Chem. B*, **102**, *10983*, (1998)

6 Still, et al. *J. Am. Chem. Soc.*, **112**, *6127*, 1990

7 E. Gallicchio, L. Y. Zhang, and R. M. Levy, *J. Comput. Chem*, **23**, *517-529* (2002)

changed via the `sgbp` input file keyword (see Section 2.3.4 [Sgbp (setmodel)], page 33).

If the SGB model is activated, then the following line should appear in the output:

```
%IMPACT-I (mmstd): Using Surface Generalized Born Model
```

In the energy-decomposition printout provided by Impact during the course of a minimization, the continuum-solvent energy is provided under the heading '`RxnFld(Sgb)`'. These energies include the interactions between the atomic-point charges and the induced charges at the solute/solvent interface.

Examples:

- `mmechanics consolv sgb cutoff 0.1`
- `mmechanics consolv sgb nonpolar 1`

`consolv pbf`

- `mmechanics consolv pbf [ pbfevery` *val* `] [ cutoff` *val* `] -`
  `[ rxnf_cutoff` *val* `] [ cavity_cutoff` *val* `] -`
  `[ low_res | med_res | high_res ] [ debug` *val* `]`

PBF is a Poisson-Boltzmann Solver. It takes as input a set of atomic coordinates, their charges and radii, a solvent radius, and dielectric constants for the solute and solvent and computes the electrostatic potential from the resulting Poisson-Boltzmann equation. The reaction-field energy (electrostatic interaction of the fixed atomic charges with the induced surface charges at the solute/solvent interface) and gradient are then calculated. The reaction-field terms effectively represent the average interaction between the solute molecule(s) and the solvent. The advantage of this approach is that the large number of solvent molecules typically used in a solution-phase molecular simulation or minimization are not required, thereby dramatically reducing the computational expense. While treating the solvent as a continuum rather than a collection of discrete molecules is clearly an approximation, it has been shown to be a fairly good one for many types of calculations.

The novel feature of PBF over other algorithms used to solve the Poisson-Boltzmann equation is the use of a finite-element mesh with tetrahedron grids. This approach allows the density of grid points used in solving the discretized equations to be optimized such that accurate results may be achieved with a minimal number of grid points and hence with minimal computational effort. For example, a high density of points is required at the solute/solvent interface to compute a accurate and numerically stable reaction-field gradient. Other approaches using, for instance, a finite-difference method with cubic grids do not have this flexibility and must use a large number of points to obtain comparable accuracy. The use of a finite-element mesh also allows a high density of points to be used in a particular region of interest, e.g., a enzyme-binding site and a lower density of grid points elsewhere in the system, again minimizing the computational effort.

`pbfevery` This parameter sets the frequency in timesteps when a PBF calculation is performed. In between timesteps use the most recent PBF energies and forces.

`cutoff` The `cutoff` parameter specifies how far any atom must move from the coordinates used in the previous calculation before a new Reaction Field calculation is performed. The default value is 0.1 Å. If all atomic coordinates have moved less than this cutoff, then the previous calculated energy and forces are used for that step in the minimization. Preliminary results suggest that the pbf energy and gradient are slowly varying functions of the atomic coordinates, relative to the other energies and forces involved in a typical molecular mechanics calculation. A relatively large value of `cutoff` can significantly reduce the required computational time at the expense of some loss in accuracy.

`cavity_cutoff`
The keyword `cavity_cutoff` is used for cavity term recalculation. It is similar to the keyword `cutoff`.

`low_res` Use the low grid point resolution setting. This is the default.

`med_res` Use a medium grid point resolution setting.

`high_res` Use a high grid point resolution setting. This is the most expensive setting, but also the most accurate.

`debug` Setting `debug` to a nonzero value causes diagnostic messages and files to be printed for each calculation.

The `consolv pbf` parameter files are in the directories

```
$SCHRODINGER/impact-v6.7/data/opls
$SCHRODINGER/impact-v6.7/data/opls2000
```

and all start with `pbf`. The files should not need to be modified by the user on an ongoing basis. A few parameters, however, may need to be changed occasionally. For example, the dielectric constants used for the solutes and solvent can be changed in the '`pbf.com`' file. Also the solvent radius can changed by editing the same file.

If the PBF model is activated, then the following line should appear in the output:

```
%IMPACT-I (mmstd): Using Poisson-Boltzmann Model
```

In the energy-decomposition printout provided by Impact during the course of a minimization, the continuum-solvent energy is provided under the heading 'RxnFld(Pbf)'. These energies include the interactions between the atomic-point charges and the induced charges at the solute/solvent interface.

Because of the large memory requirements for medium-sized and larger proteins, PBF currently writes some arrays to disk and then reads them back in as needed. Currently only one file is being written to disk, 'zzZ_Ctbl_Pbf_Zzz'. Every effort is made to remove this file after a calculation has completed. However, if a calculation is aborted or something goes amiss, this file may be left on the disk.

Examples:

- `mmechanics consolv pbf cutoff 0.1`
- `mmechanics consolv pbf low_res cutoff 0.1 cavity_cutoff 0.9`

`consolv agbnp`

- `mmechanics consolv agbnp`

AGBNP is an analytical implicit solvent model based on the pairwise descreening (PD) Generalized Born (GB) model and a non-polar solvation free energy (NP) estimator which takes into account independently the work of cavity formation and the solute-solvent van der Waals interaction energy. The model and its derivation are described in detail in the following paper: E. Gallicchio, R. M. Levy, AGBNP: An Analytic Implicit Solvent Model Suitable for Molecular Dynamics Simulations and High-Resolution Modeling, J. Comput. Chem., 25, 479-499 (2004). AGBNP is unique among pairwise descreening GB models in that the overlap scaling coefficients depend on solute conformation and are computed from purely geometric considerations, rather than being fit to experimental and Poisson Boltzmann data. Hydrogen atoms do not contribute to descreening. The non-polar hydration free energy estimator is composed of two terms. The first, related to the cavity hydration free energy, is proportional to the solute surface area of each atom through surface tension parameters that depend on atom type. The surface area is defined as the van der Waals surface area obtained by increasing the van der Waals radius of each atom by 0.5 Å. The surface area of each atom is calculated using an analytical algorithm based on the same method used to calculate overlap scaling factors. Hydrogen atoms do not contribute to the solute surface area, that is they can be thought as of atoms of zero radius in this respect. The second component of the non-polar hydration free energy model is a solute-solvent van der Waals interaction energy estimator that depends on the Born radius and Lennard-Jones parameters of each atom. This estimator includes dimensionless scaling parameters for each atom type adjusted to better reproduce solute-solvent van der Waals energies obtained from explicit solvent simulations. In addition to the surface tension parameters and van der Waals scaling parameters, the other parameters of the model, atomic partial charges and van der Waals radii, are derived from the underlying force field without change (partial charges) or with small modifications (van der Waals radii).

The current AGBNP parameters are stored in a file called `agbnp.param` in the directories

```
$SCHRODINGER/impact-v6.7/data/opls
$SCHRODINGER/impact-v6.7/data/opls2000
$SCHRODINGER/impact-v6.7/data/opls2001
$SCHRODINGER/impact-v6.7/data/opls2005
```

depending on the active force field version. The format of the `agbnp.param` file is as follows:

Column	Content
1	Type index
2	OPLS symbolic type
3	van der Waals radius [Å]
4	non-polar gamma parameter [(kcal/mol)/Å^2]
5	non-polar alpha parameter [dimensionless]
6	non-polar delta parameter [kcal/mol]
7	correction gamma parameter [(kcal/mol)/Å^2]
8	correction alpha parameter [dimensionless]
9	correction delta parameter [kcal/mol]
10	screening parameter [dimensionless]

Lines that begin with '#' are comments. Lines beginning with `dielectric_in` and `dielectric_out` set the dielectric solvent of the solute and the solvent, respectively, and should precede any other non-comment line. `gamma` above refers to the surface tension parameters, `alpha` to the solute-solvent van der Waals scaling parameters, the values of the `delta` parameters should be left to their default values (zero). The values of the non-polar parameters used internally are the sum of the pure and correction values. However the non-polar energy derived from each is reported separately as a pure non-polar energy and a correction energy term. The correction energy term has the same expression as the non-polar estimator (this could change in the future) but it is calculated using the set of correction parameters rather than the pure non-polar parameters. The screening parameter in column 10, normally set to 1 for all atom types, is described in the following paper: A. K. Felts, Y. Harano, E. Gallicchio, and R. M. Levy. Free energy surfaces of beta-hairpin and alpha-helical peptides generated by replica exchange molecular dynamics with the AGBNP implicit solvent model. PROTEINS: Structure, Function, and Bioinformatics, 56, 310-321 (2004). To modify the AGBNP parameters edit a copy of the `agbnp.param` file in the working directory. The `agbnp.param` file in the working directory takes precedence over the `agbnp.param` file in the data directory.

If the AGBNP model is activated the following line should appear in the output:

```
%IMPACT-I: Using AGBNP: Analytical Generalized Born Model + Analytic
 Non-Polar Hydration Model
```

The running AGBNP energy components are reported under the labels `RxnFld(AGBNP)` and `NPolar(AGBNP)` in the output file, for the electrostatic and non-polar components (pure plus correction) respectively. The energy summary at the end of the output file lists the total AGBNP solvation free energy under `AGBNP Solvation Energy`, the electrostatic component of the solvation free energy under `AGBNP Solvation Energy (polar)`, the pure non-polar component under `AGBNP Solv. Energy (non-polar)`, and the correction term under `AGBNP Solv. Energy (correction)`.

There are no options associated with the `consolv agbnp` setting. AGBNP applies the same distance cutoff as specified by the `energy parm cutoff` command (see Section 2.3.1.6 [Parm (energy)], page 23) for the GB pair energies and for the pairwise descreening calculation of Born radii.

2.3.3.2 Weight

Change the weights of terms in the potential function. Unless otherwise indicated below, the weights are all initialized to 1.0 when `mmechanics` is used.

Caution: Despite the terminology below, intramolecular nonbond terms are affected both by *intramolecular* and *intermolecular* electrostatic and LJ weights. The total nonbond weight is the product of the intramolecular (within one species) and intermolecular (between species) weights.

- `weight intramolecular name` *`spec`* `-`
  `[ bond | angle | torsion | el14 | lj14 | elin | ljin | hbin ]` *`weight`*

The `intramolecular` keyword is used to change the weights of intramolecular terms (those within a single species). The `elin`, `ljin`, and `hbin` keywords change the weights for all included nonbond pairs within the molecule; `el14` and `lj14` change them only for "1-4" pairs, i.e., atoms at the outer ends of a quartet that defines a torsion angle. `hbin` is only used with the AMBER86 force field.

- `weight intermolecular -`
  `[ vdw | eel | hbond | hbelectrostatics ]` *`weight`*

The `intermolecular` keyword is used to change the weights of intermolecular terms within or between species, thus there is no `name spec` designation. `hbond` and `hbelectrostatics` are only used with the AMBER86 force field.

- `weight constraints name` *`spec`* `-`
  `[ noe | torsion | hbond ]` *`weight`*
- `weight constraints name` *`spec`* `buffer` *`weight`* `-`
  `[ halfwidth` *`sigma`* `]`

The `constraints` keyword defines the weights of various restraint force constants terms. The `torsion`, and `hbond` terms are zero by default and define distance and torsion restraint weights.

The `buffer` constraint energy is a harmonic term is applied to all "buffered" atoms specified via `zonecons` commands. See Section 2.3.7 [Zonecons (setmodel)], page 35. The default `buffer` is 25 kcal/(Å^2 mol). You can control the sigma halfwidth value via the `halfwidth` keyword, whose default is 0.0, equivalent to a harmonic constraint.

Caution: `buffer` is not a per-species parameter, but is applied to all buffered atoms in the system.

2.3.4 Subtask Sgbp

This keyword sets various SGB continuum solvent simulation parameters. It has no effect unless `mmechanics consolv sgb` is used in a preceding `setpotential` subtask to activate the SGB method.

- `sgbp grid_size` *`max`* `dock_grid_size` *`glide_max`* `-`
  `min_grid_size` *`min`* `printe [0|1] printf [0|1] -`
  `active_reg_incr` *`val`* `buffer_reg_size` *`val`* `accuracy` *`val`* `-`
  `epsout` *`val`* `hydrogen_radius` *`val`*

`grid_size`
: The maximum number of grid points each atom can have. The default value is 70.

`dock_grid_size`
: In a Glide calculation, the maximum number of grid points each atom can have, the default is 30.

`min_grid_size`
: The minimum number of grid points each atom can have. The default value is 20.

`printe`
: If set to 1, print the SGB energy. The default is 0.

`printf`
: If set to 1, print the SGB forces. The default is 0.

`active_reg_incr`
: When setting up the active region region, this amount is added to it. The default is 0.

`buffer_reg_size`
: This defines the buffer region size; the buffer region is located between the active region and the frozen region.

`accuracy`
: The threshhold value used with the `singlelong` multiple time scale scheme, and is related to the number of surface grid points used. The default value is 0.00001. Smaller values result in denser grids.

`epsout`
: The exterior (solvent) dielectric constant. The default is 80.0, a value typical of water simulations. (The *interior* dielectric constant is set by `enrg parm diel`, see Section 2.3.1.6 [Parm (energy)], page 23.)

`hydrogen_radius`
: The atomic radius of hydrogen, used in generating the surface. The default value is 1.0.

2.3.5 Subtask Mixture

- `mixture [ density` *`val`* `| keep` *`num`* `] [ overlap` *`val`*`]`

This command sets optional parameters for the removal of excess solvent molecules when solvent and solute are mixed. If `mixture` is not present then the default is to remove all solvent molecules that overlap (as defined below) with any solute atom. When the `mixture` command is issued only up to a maximum of N solvent molecules are removed. N is calculated in one of two ways. Either from the effective solute volume (which can be controlled using the `density` parameter) or from the number of solvent molecules not to be removed (the `keep` parameter). A molecule is considered for removal if the ratio of the distance d and the sum $R1 + R2$ of the van der Waals radii of any atom of the solvent molecule and any atom of the solute is smaller than a overlap threshold value (the `overlap` parameter). If the minimum distance d is larger than 10 Å a solvent molecule is not considered for removal regardless of the value of the overlap threshold value. If more than N solvent molecules are flagged for removal only the N solvent molecules with the smallest minimum distance d are removed. If instead the number of solvent molecules flagged for removal is less than N all flagged solvent molecules are removed.

`density`
: Keyword `density` is used to set the solute density. The default is 1 g/cm^3. The volume of solvent removed is equal to the effective volume of the solute. The effective solute volume is calculated from the solute mass and the solute density. The larger the solute density the smaller the effective solute volume and thus

the smaller the maximum number N of solvent molecules to be removed.

`keep` Keyword `keep` is used to set explicitly the minimum number of solvent molecules remaining after removal. The default is 0. The maximum number N of solvent molecules to be removed is set as the current number of solvent molecules minus the number of solvent molecules to keep. The `keep` option preempts the `density` option if both are given.

`overlap` The `overlap` option is used to set the overlap threshold value below which a solvent atom is considered to overlap with a solute atom. The default is 1. Decreasing the overlap parameter makes it less likely for two atoms to overlap.

2.3.6 Subtask Solute

This subtask is used to place solute molecules at certain positions in the container "box" of solvent used for the simulations.

2.3.6.1 Translate

The keyword `translate` brings the center of mass (COM) of the system of solute molecules to the origin (center of the box), and also finds the longest distance between atoms along the principal axis, which determines the box edge lengths. The option `skip` says to ignore the last *num* residues of the solute when performing the operation. With `rotate`, the solute is rotated so that the principal moments of inertia coincide with the x, y, z axis. The longest axis of the molecule is oriented along the z axis. `Skip` has the above meaning. If `rotate diagonal` is given on the command line the rotation is such that the principal moment of inertia lies along the diagonal of the simulation box (which must be cubic for this option to work).

- `solute translate [ rotate [ diagonal ] ] name` *`spec`* `[ skip` *`num`* `]`

Caution: `skip` *num* excludes residues that may not have meaningful coordinates yet (such as counterions) from the translation/rotation operation. This parameter may be read in for as many different species as necessary. The value given for `skip` means that the last *num* residues of the species are ignored in the translation/rotation of the solute.

2.3.7 Subtask Zonecons

This subtask is used to constrain (*freeze*) or restrain (*buffer*) various regions of a molecule based on options specified by the user.

- `zonecons [ auto | [ [freeze|genbuffer] | chain | resseq | -residue | atom | sphere ] name` *`spec sub-options`* `]`

There are seven types of `zonecons` subtasks described below. All but `zonecons auto` are additive, so you can use combinations of them. By default, all atoms are free to move, as if there are no `zonecons` subtasks at all.

Any buffered atoms are restrained using an harmonic potential centered on the original atom position. Any atom position can be restrained this way. A buffer zone is often used to to define an intermediate zone between a fixed region where the atom positions are frozen and the free region where the atom positions are not restrained. The buffer option is also often used to perform constrained minimizations. The force constant of the restraining harmonic potential is user selectable, see Section 2.3.3.2 [Weight (setpotential)], page 32.

2.3.7.1 Auto

Use the frozen/buffered settings from an input Maestro file.

- `zonecons auto`

Maestro files written by Maestro specifically for Glide, Liaison, or QSite jobs, or written as output from a Glide, Liaison, or QSite job, will contain an extra parameter (internally named `i_i_constraint`) for each atom. `Zonecons auto` uses this parameter in lieu of any other zonecons option, where the values 0, 1, and 2 correspond to free, frozen, and buffered, respectively.

2.3.7.2 Freeze/Genbuffer

Freeze or restrain (buffer) a specified group of atoms, e.g., all heavy atoms, all C atoms, all N atoms, all O atoms, or all atoms.

- `zonecons [freeze|genbuffer] name` *`spec`* `[all | allC | allN | allO | allheavy]`

This is the general freezing or restraining option, it can be used to freeze/restrain all atoms, all carbon atoms, all nitrogen atoms, all oxygen atoms, or all heavy atoms. The general restraining option is called `genbuffer` to differentiate it from the `buffer` designation available in some of the other `zonecons` options.

2.3.7.3 Chain

Chain-based scheme, select any chain in a protein to be in fixed, free, or buffer region

- `zonecons chain name` *`spec`* `[chainname` *`name`* `[fixed|free|buffer] ]+`

This is the chain option, which is used to classify the whole chain with *name* to be in fixed, free, or buffer regions.

2.3.7.4 Resseq

Residue sequence-based scheme, such as from residue number 20 to 50, to be in fixed, free, or buffer region

- `zonecons resseq name` *`spec`* `[resn` *`fres`* `to` *`lres`* `-`
  `[all | allC | allN | allO | allheavy] [fixed|free|buffer] ]+`

This is the residue sequence option, which states that in the specified residue sequence, starting from first residue *fres* to last residue *lres*, the specified atom types (all atoms, all carbons, etc.) are to be in fixed or free or buffer regions.

2.3.7.5 Residue

Residue-based scheme, such as backbone, sidechain, or amide of a residue to be in fixed, free, or buffer region

- `zonecons residue name` *spec* `[resn` *num* `-`
  `[all|backbone|sidechain|amide|Calpha|Ncap|Ccap] [fixed|free|buffer] ]+`

This is the residue option, which states that in the specified individual residue(s), with residue number(s) *num*, the specified atoms (all, backbone, sidechain, amide, α carbon, etc.) are to be in fixed or free or buffer regions.

2.3.7.6 Atom

Atom-based scheme, for any particular atom

- `zonecons atom name` *spec* `[atmn` *num* `[fixed|free|buffer] resadj [0|1] ]+`

The atom option, the lowest level option, which classifies each atom to be in the fixed or free or buffer regions.

The option `resadj` is used for residue-based adjustment; if it equals 1, then the whole residue associated with that particular atom will be classified in the the same region (in this case the residue becomes the basic operational unit). The default value for `resadj` is 0, which means no residue-based adjustment is performed.

2.3.7.7 Sphere

Sphere-based scheme, freeze/relax any atoms inside a sphere with a center and radius

- `zonecons sphere [center x` *val* `y` *val* `z` *val* `| name` *spec* `resn` *num* `atom-name` *name*`] -`
  `[freeze | relax ] rad` *rad* `buffrad` *buffrad* `resadj [1|0]`

This is the sphere option, which is used to relax or freeze a sphere with the center located at residue number *num* and atom name *name*, and a radius of *rad*. The *buffrad* is the radius for buffer, the shell between radius *rad* and *buffrad* becomes the buffer region. It should be noted that *buffrad* should be bigger or equal than *rad*.

The option `resadj` has the same meaning as in the atom option, except the default value here is 1, which means the residue-based adjustment is turned on in sphere option by default.

2.3.7.8 Example Zonecons Input

Here is an example for how to use the various options for zone constraints.

```
setmodel
   setpotential
     mmechanics
   quit
   read parm file paramstd.dat noprint
   enrg parm cutoff 20.0 -
     listupdate 100 diel 1.0 nodist print 1
   zonecons freeze  name hiv allheavy
   zonecons chain   name hiv chainname A free  chainname B fixed
   zonecons sphere  name hiv resn 20 atomname CA relax rad 10.0 buffrad 12.0
   zonecons residue name hiv resn 10 backbone fixed  resn 11 sidechain free
   zonecons resseq  name hiv resn 20 to 40 all buffer  resn 41 to 100 all fixed
   zonecons atom    name hiv atmn 45 free  atmn 50 fixed  atmn 52 buffer
quit
```

2.3.7.9 Zonecons Keywords

Some of the keywords used above for various `zonecons` subtasks have the following meanings. Not all keywords are appropriate for every `zonecons` option, see the above syntax diagrams for a list of those allowed.

`freeze` General freeze option, to freeze all atoms, all carbons or all heavy atoms.

`chain` Chain option, to freeze/relax/buffer proteins by chain name.

`resseq` Residue sequence option, to freeze/relax/buffer proteins by residue sequence.

`residue` Residue option, to freeze/relax/buffer a residue's backbone, sidechain, etc.

`atom` Atom option, to freeze/relax/buffer any particular atom.

`sphere` Sphere option, to freeze/relax a sphere with a center and a radius.

`free` Free to move.

`buffer` In the buffer region.

`fixed` In the frozen region.

`resadj` Residue based adjust, default value is 0 for atom level option, and 1 for sphere level option. If it equals 1, then the whole residue will share the same region with one or more atoms specified by the `zonecons` subtasks.

`allC` All carbon atoms.

`allN` All nitrogen atoms.

`allO` All oxygen atoms.

`allheavy` All heavy atoms, atoms except H.

`backbone` Backbone atoms in a residue.

`sidechain`
Sidechain atoms in a residue.

`amide` Amide group atoms in a residue.

`Calpha` Alpha carbon atom in a residue.

`Ncap` N-terminal cap in a residue (NH2, NH3+).

`Ccap` C-terminal cap in a residue (COOH, COO-).

`center` To read in the cartesian coordinates of a sphere center directly. The center can also be read in by specifying an atom name `atomname` in a residue `resn` in a specie `name` *spec*.

`rad` *value* Radius of frozen or free zone.

`buffrad` *value*
Radius of buffer zone. The value of *buffrad* should be bigger than *rad*.

`chainname` *name*
Chain name to be relaxed or fixed.

`atomname` *name*
Name of atom at center of sphere.

`resn` *fres* `to` *lres*
Starting from first residue *fres* and ending with last residue *lres*

Please note: `resn` (or `resnumber` or `rnumber`) residue numbers supplied in the main input file have the following meanings: positive numbers mean the residue numbering used in the original PDB file; negative numbers mean the reordered Impact residue numbers, i.e., sequential, starting with 1; 0 means all applicable residues.

Caution: The `zonecons` option alters many structural arrays. It is assumed that all bonds angles and torsions that lie completely in frozen regions will not change and therefore their entries in the structural arrays are deleted. Also, in later energy calculations non-bonded or hydrogen bond pairs for which both atoms are frozen are not stored or calculated.

2.3.8 Subtask QMregion (QSite)

The QSite module allows a section of a protein and/or whole ligand(s) to be treated quantum mechanically while the rest of the system is treated by OPLS-AA. Gas phase 6-31G* Hartree-Fock (HF) and DFT energies, minimizations, and transition state optimizations are currently implemented for all amino acids, ligands, ions, and bound waters. Single-point LMP2 calculations are also supported. QSite solvation using continuum solvent (PBF model) are possible as well.

2.3.8.1 QSite Overview

The QM/MM interface consists of a frozen localized single-bond QM molecular orbital at each QM/MM boundary.[8] The QM and MM regions interact via a Coulomb interaction (between MM charges and the QM wave function) and a van der Waals interaction (van der Waals parameters are employed for both the QM and MM atoms). In addition there are QM/MM hydrogen bonding terms. Specialized MM-like correction parameters are used for stretches, bends, and torsions involving atoms that touch or span the QM/MM interface. These parameters are fit to reproduce local-MP2/cc-pVTZ(-f) quantum chemical conformational energetics of each residue.

A QSite job requires both Impact and Jaguar input files. The job is initially launched using the Jaguar program driver script `jaguar`. Once Jaguar detects that it is doing a QSite job, it calls Impact, which then reads the main input file (with protein, ligand data) and the QM region specifications. Impact calculates the requisite MM energy/gradient terms and creates a Jaguar input file for the QM region only. Control is then passed back to Jaguar, which calculates the total QM portion of the QM/MM energy/gradient.

QSite geometry optimization uses an adiabatic approach. This means that a full minimization of the MM region is performed by Impact before each QM geometry step taken by Jaguar. During the QM step all of the MM region except for a few atoms at the QM/MM interface are frozen in the QM optimization/geometry steps and similarly the QM region is frozen in the MM optimization process.

In defining the QM region for a QSite job, it may be necessary to use an input structure that is not a correct Lewis structure. Ordinarily, Impact would reject such a structure, upon reading it in via the `build primary type auto` command. In order to bypass Lewis structure checking in such cases, use the `notestff` keyword in the `build primary` command for reading in the structure that will contain the QM region. See Section 2.2.1.1 [Primary type Auto], page 15 for details of this command and keyword.

The following subsections describe the Impact and Jaguar QM/MM inputs and illustrate the execution of a QSite run.

Here is the general syntax for the `qmregion` subtask:

- `qmregion [ residue name` *`spec`* `[ all | resn` *`num`* `chain` *`chainid`* `insert` *`insertion_code`* `molid` *`num`* `[ cutb` *`num`* `] ]`
- `qmregion atom name` *`spec`* `atom` *`num`*
- `qmregion ion name` *`spec`* `ionn` *`num`*

---

[8] D.M. Philipp and R.A. Friesner, *J. Comput. Chem.* **20**, *1468* (1999);
R.B. Murphy, D.M. Philipp, R.A. Friesner, *Chem. Phys. Lett.* **321**, *113* (2000); and
R.B. Murphy, D.M. Philipp, R.A. Friesner, *J. Comput. Chem.* **21**, *1442* (2000).

2.3.8.2 QM protein region

The `qmregion residue` command is used to specify parts of proteins, or entire molecules such as ligands or bound waters, as belonging to the QM region.

The QM region of a protein is specified by making QM/MM *cuts* or boundaries at the bonds emanating from the Cα carbon of any residue. In addition, whole residues can be designated as QM as long as they are inside the boundaries of QM/MM cuts at more distant residues. The 5 types of cuts and associated QM/MM regions are defined as follows and as depicted in the following figures.

Cut 1: The Cα-N bond forms the boundary, and the Cα atom and its attachments are in the QM region.

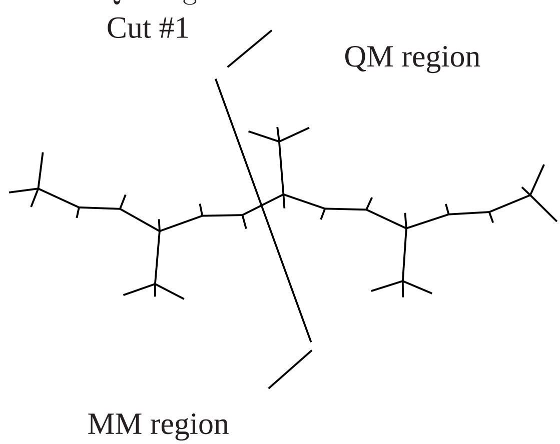

Figure QMMM−1; QM/MM regions
for backbone cut type 1.

Cut 2: The Cα-C bond forms the boundary, and the Cα atom and its attachments are in the QM region.

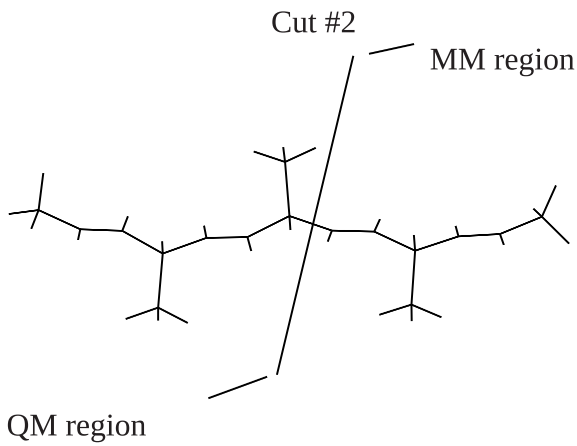

Figure QMMM−2; QM/MM regions
for backbone cut type 2.

Cut 3: The Cβ-Cα bond forms the boundary, and the side chain is the QM region.

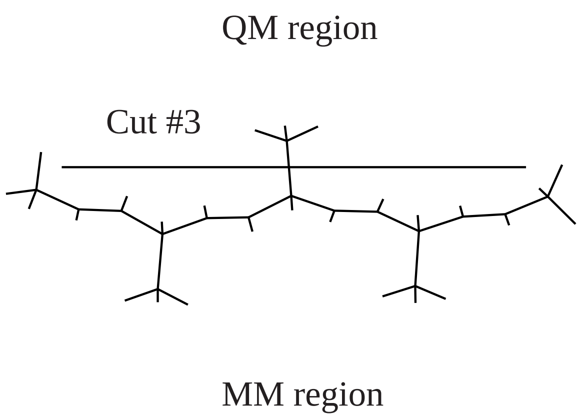

Figure QMMM−3; QM/MM regions
for side chain cut type 3.

Cut 4: The N-Cα bond forms the boundary, and the amide nitrogen (N) and its attachments are in the QM region.

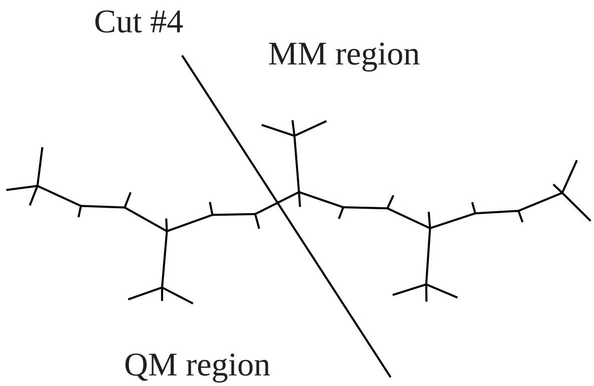

Figure QMMM−4; QM/MM regions
for backbone cut type 4.

Cut 5: The C-Cα bond forms the boundary, and the carbonyl carbon (C) and its attachments are in the QM region.

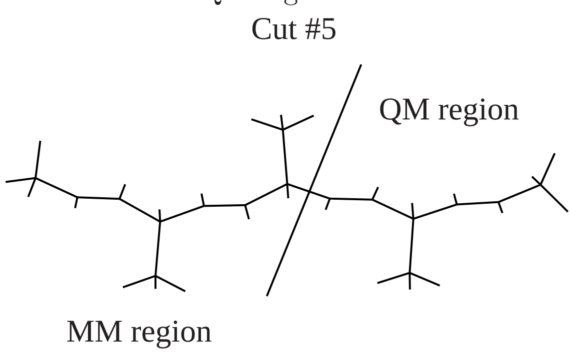

Figure QMMM−5; QM/MM regions
for backbone cut type 5.

Except for side chain cuts (type 3), the cut residue must be connected to another pure (no cut) QM residue. Placing backbone cuts in consecutive residues is not recommended because the boundary regions will interact too strongly.

Cuts in the following residues are *not* allowed, depending on the molecular mechanics force field in use: for OPLS2001 and later force fields, sidechain cuts in GLY, PRO, and ALA, and backbone cuts in PRO; for earlier force fields, sidechain cuts in ARG, SER, THR, PRO, GLY, and ALA, and backbone cuts in GLY and PRO. To treat these residues as QM regions, place backbone cuts on the adjacent residues on either side.

As an example, suppose the ala-gly-ser section of a . . . lys-ala-gly-ser-phe. . . protein is to be represented in a QM fashion, with OPLS1999 in use for the MM region. (The same reasoning would apply to the ala-gly section with OPLS2001.) In this case a cut of type 5 (or 1 to include the lys sidechain in the QM region) would be made in lys, and a cut of type 4 or (2 to include the phe sidechainin the QM region) in phe. In addition, residues ala-gly-ser would all be specified as fully QM, i.e. with no cuts. More commonly a set of sidechain cuts of type 3 might be made for residues that make important contacts with a ligand to allow the contact regions and the ligand all to be treated quantum mechanically.

Protein QM regions are specified in task `setmodel` with syntax like the following:

```
qmregion residue name prot resn 142 molid n cutb 3
```

This directive places the sidechain of residue 142 in species *prot*, molecule number *n* in the QM region. The integer following `cutb` specifies the type of cut to be made.

Alternatively, the whole residue can be made QM (no cut) by omitting the `cutb`-*value* pair:

```
qmregion residue name prot resn 142 molid n
```

The QM/MM interface requires that each protein segment of the QM region be defined either by a single cut of type 3, or by matching cut specifications for the N- and C-terminal residues of the segment in question. In the latter case, all intervening residues must explicitly be specified as QM in `qmregion` specifications.

Note that QSite requires that the whole system fit into one Impact species. This can be done by putting all molecules (proteins or ligands) into one species using the `mole` notation in the `build primary` commands, or by creating a single entry containing all the molecules in the Maestro Project Table or Workspace. QSite calculations can be carried out with PBF (but not SGB) implicit water or can be run with the *bound* waters typically found in PDB files. Solvent boxes, which require periodic boundary conditions, however, cannot be used.

A ligand or bound water molecule can be designated as a pure QM region with the same syntax as is used for an entire residue (between cuts, but not containing any cuts itself) in a protein:

```
qmregion residue name prot resn rnum molid molnum
```

where residue number *rnum*, in molecule number *molnum*, denotes the desired molecule in species *prot*. This syntax (with no `cutb` specification) designates the whole molecule as a QM region. Note that QM/MM boundaries cannot currently be made between ligand atoms.

2.3.8.3 Individual QM Atoms

The syntax

```
qmregion atom name spec atom num
```

indicates that the individual atom number *num* in species *spec* is to be included in the QM region.

2.3.8.4 QM Ions

Ions can be included in the QM region first by building the ion or ions. The following illustrates the placement of a Zn2+ ion:

```
CREATE
...
build newres zn2+ file zn
build primary ions name prot zn 1 xyz x 36.921 y 44.908 z -7.111 end
...
```

where `build newres` creates a Zn2+ residue with the name `zn` (the 1 following `zn` is a specification for one ion), and `build primary ions` adds the ion into the previously defined molecule of the species *prot* at coordinates (x,y,z). The specification of the ion as a QM region is done as follows:

```
qmregion ion name prot ionn 1
```

specifies that ion number (`ionn`) "1" of species *prot* should be treated as a QM ion. When multiple ions are present, one such `qmregion` directive should be given for each ion that is to be QM.

2.3.8.5 Basis set specifications.

All of the standard basis sets used in Jaguar are available for the QM region of a QSite setup. Then basis sets can be specified within the Impact input as follows.

- `basis name` *spec* `[ atom` *num* `| resnumber` *num* `| nil ] [ radius` *rad* `] basis` *bset*

The default basis used is 6-31G* (LACVP* for metals), which **must** be entered into the Jaguar input file (see below) regardless of other basis set specifications. To specify the basis on a particular residue the following syntax applies:

```
SETMODEL
  ..
  qmregion residue name dipep resn 2 cutb 3
  ..
  basis name dipep resnumber 2 basis cc-pvtz(-f)
  ..
QUIT
```

This will setup a cc-pvtz(-f) basis on the QM atoms of previously specified QM residue 2. Note that atoms comprising the QM/MM cut and their bonded neighbors will automatically stay at 6-31G*. This restriction is necessary since the QM/MM boundary region is parametrized with 6-31G*. The code will automatically keep the necessary 6-31G* basis sets regardless of basis set specifications made by the user.

The syntax for changing the basis set within a specified radius of a chosen atom is:[9]

```
SETMODEL
  ..
  qmregion residue name dipep resn 2 cutb 3
  ..
  basis name dipep atom 34 radius 5.0 basis cc-pvtz(-f)
  ..
QUIT
```

will change the basis set to cc-pvtz(-f) on atoms within 5 Å of atom number 34. This atom must be in a residue or a ligand in the QM region as specified by the `qmregion` commands.

2.3.8.6 QSite energy/minimization:

Single point QSite energies can be obtained using task `analysis` with the subtask `qmme`, e.g.,

```
ANALYSIS
  qmme
QUIT
```

will tell Impact to generate a QM/MM energy.

QSite geometry optimizations require the usual Impact MM minimization section, e.g.:

```
MINM
  conjugate dx0 0.05 dxm 3.0 rest 50
  input cntl mxcyc 10000 rmscut 1.9e-1 deltae 0.5
  run
QUIT
```

with no special QSite flags.

The following Impact example, and the Jaguar input example below, are for a small polypeptide with a water molecule. A threonine residue and water molecule constitute the QM region and are treated at the B3LYP level. The rest of the structure is treated with molecular mechanics.

[9] N.B.: The radius option is not available via Maestro, but you can add it by hand into the input file

```
CREATE
  build primary name species1 type auto read maestro file -
"qsite.mae"
  build types name species1
QUIT

SETMODEL
  setpotential
    mmechanics
  quit
  read parm file -
"paramstd.dat" -
  noprint
  energy parm dielectric 1 nodist -
   listupdate 10 -
    cutoff 12
  energy rescutoff byatom all
  zonecons auto
  qmregion residue name species1 resn 4  molid 1
  qmregion residue name species1 resn 691  molid 2
  basis name species1 resnumber 691 basis 6-31G
  qmregion residue name species1 resn 3 molid 1 cutb 5
  qmregion residue name species1 resn 5 molid 1 cutb 4
QUIT
```

The `CREATE` task above reads a Maestro file containing both the polypeptide chain and the water molecule, into the single species `species1`. Based on the connectivity data in this file, Maestro and Impact assign molecule numbers 1 to the peptide (because it includes the first atom listed in the file) and 2 to the water molecule (because it includes the next atom listed that has no covalent bonds to molecule 1).

The `qmregion` commands describe the cuts between the QM and MM region in the structure. All of residue number 4 in molecule number 1 is included in the QM region, as is residue number 691 in molecule number 2: this is the water molecule. The `basis` line tells Jaguar to treat residue number 691 with the 6-31G basis set rather than the default 6-31G*. The next line specifies a cut of type 5 in residue number 3 in molecule 1. Type 5 places the cut in the C-Cα bond with the sidechain in the MM region. Residue number 5 in molecule 1 has a cut of type 4, which is through the N-Cα bond with the sidechain in the MM region.

2.3.8.7 QSite Transition State Optimization

QSite can perform optimizations to transition state structures using three different methods. The method you choose will depend on what starting structures you have. See the *Jaguar User Manual* for more information on these methods.

- Standard method

If you only have an initial guess structure for the transition state, QSite can find the saddle-point closest to the starting structure by maximizing the energy along the lowest-frequency mode of the Hessian and minimizing the energy along all other modes.

- Linear Synchronous Transit (LST) method

  If you have structures for the reactant and product, then QSite can use a quasi-Newton method to search for the optimal transtion state geometry. Given the two endpoint structures, and an interpolation value between 0.0 ($\equiv$ reactant structure) and 1.0 ($\equiv$ product structure), QSite will try to construct an initial transition state structure at that point along the reaction coordinate.

- Quadratic Synchronous Transit (QST) method

  If you have structures for the reactant, product, and transition state guess, then QSite will use the same quasi-Newton method as LST does, but will use your initial guess for the transition state, rather than interpolating as in LST.

Impact input file keywords:

- `qmtransition [ reactant | product ] file` *`fname`* `[ gotostruct` *`number`* `]`

These keywords are necessary in the Impact input file when you have multiple structures to include in your calculation, as is required in both LST and QST. LST calculations require the reactant to be loaded in a normal `build primary` command, and the product structure to be defined with a `qmtransition` keyword thus. QST calculations require the transition state guess structure to be loaded by `build primary`, and both the reactant and product structures defined by `qmtransition`.

Jaguar input file keywords:

```
&gen
igeopt = 2
iqst = [ 0 | 1 | 2 ]
qstinit = interpolation_value
&
```

These keywords are actually Jaguar keywords; see the Jaguar documentation for more information. Briefly, `igeopt=2` tells Jaguar to do a transition state optimization rather than a minimization. `iqst` indicates which optimization method is to be used, standard, LST, or QST, respectively. The LST method calculates an initial guess structure by interpolating between the reactant and the product, the `qstinit` parameter indicates where along the reaction coordinate this structure should lay; the default is 0.5 (midway between).

2.3.8.8 Jaguar input section:

CAUTION: do not use the "qmme" energy option with a MINM section, they are not compatible and their simultaneous use will cause erroneous gradients.

QSite calculations also require a short Jaguar input file specifying options specific to the quantum region such as the charge and multiplicity of the quantum region.

The prototypical input file for running a gas phase QSite optimization looks like:

```
&gen
mmqm=1
basis=lacvp*
dftname=b3lyp
molchg=0
multip=1
iacc=1 vshift=1.0 maxit=100
&
```

where `mmqm=1` signifies to Jaguar that a QSite calculation is requested, `dftname=b3lyp` requests thar the B3LYP functional be used. Other DFT methods should not be used with QSite. The `basis` specification is mandatory and will be properly overriden by any basis set specifications made in the Impact input file as discussed above. `molchg=2` is the charge of the QM region, and `multip=1` is its multiplicity. The last three keywords are set in QSite jobs by default to aid convergence.

The QSite Jaguar input file for a solvation run consists of

```
&gen
mmqm=1
basis=6-31G*
igeopt=1
isolv=2
nogas=2
&
```

where `isolv=2` requests a PBF solvation calculation and `nogas=2` omits a preliminary gas phase optimization normally done in pure QM solvation geometry optimization calculations. The `nogas=2` option will be set automatically in Jaguar 4.1[10]. The `consolv pbf` keyword must also be present in the Impact input file as it is for pure MM solvation calculations.

2.3.8.9 Running QSite

QSite jobs can be run from the command-line by giving both input files to the `impact` script. The syntax for running a QSite job is then:

```
% impact -j job.jaguar.in -i job.impact.inp -o job.log
```

where *job.jaguar.in* is the Jaguar input file name (e.g. '`peptide.in`') and *job.impact.inp* is the Impact input file name.

The QM/MM output contains the QM and most of the MM output will appear in '`job.jaguar.out`' and the intermediate Jaguar output will appear in '`job.jaguar.log`' as the job runs in the scratch directory (the Jaguar scratch

[10] Jaguar v4.0 releases later than r21 will also set this automatically.

directory is set in the '`$SCHRODINGER/jaguar.hosts`' file. The QM/MM energy in the Jaguar output file has the heading;

```
Total QM-MM Energy:  -3390.09684895821 hartrees
```

Solvation energies also appear in the Jaguar output file as:

```
sfinal:  -2415.0483 kcal/mol
```

where `sfinal` is the solvation energy of the QM/MM system in water relative to the gas phase.

In addition the total QM/MM solution phase energy is specified in the Jaguar output as:

```
(P)  Solution phase energy........  -428.00832706556  (Q+R+S).
```

The solvation energies printed in the Impact output of a QM/MM run are not the QM/MM solvation energies.

The detailed requirements for running QSite are as follows. The QSite job is lauched as a Jaguar job using the `jaguar` run script which should be in the `$SCHRODINGER` directory. The Impact and Jaguar inputs should be in the same directory by default. If it is desired to keep the Impact information in a separate directory, the following lines should be added to the Jaguar input file

```
&impact
mmdir=/wherever/you/want/the/data
&
```

In general however, you will want to keep all your Schrödinger software grouped together.

3 Perform Simulations

This chapter describes tasks that perform Impact simulations such as energy minimization and molecular dynamics, as well as linear response binding affinity calculations and ligand docking.

3.1 Task Minimize

Minimize a system using either the steepest descent or the conjugate gradient method. This task may only be called after the structural arrays have been filled and after a potential energy function has been set using `setpotential`. This task is used in many of the included examples.

Results are printed every 10 steps by default, but this value can be adjusted via the `enrg parm print` keywords in the `SETMODEL` task (see Section 2.3.1.6 [Parm], page 23).

Example:

```
minimize
   read coordinates formatted file fname
   steepest dx0 value dxm value deltae value
   run
   plot indiv quit
   write coordinates formatted file fname
quit
```

3.1.1 Subtask Steepest

Use the steepest descent algorithm for energy minimization of a system.

- `steepest dx0 value dxm value`

`dx0` Initial step size (default = 0.05).
`dxm` Maximum step size (default = 1.0).

3.1.2 Subtask Conjugate

Use the conjugate gradient algorithm for energy minimization.

- `conjugate dx0 value dxm value maxit number`

`dx0` *step_size*
Set the initial step size (default = 0.05).

`dxm` *step_size*
Set the maximum step size (default = 1.0).

`maxit` *step_size*
Maximum number of iterations for line search (default = 3).

`rest` Frequency of restarting with steepest descent (default = number of atoms ×3).

3.1.3 Subtask Tnewton

Use the truncated Newton algorithm (copyright (c) 1990 by Tamar Schlick and Aaron Fogelson, updated November 1998 by Dexuan Xie and Tamar Schlick, used by permission)[1] for energy minimization.

PLEASE NOTE: This minimization algorithm cannot currently be used with periodic boundary conditions, distance and torsional restraints and the Fast Multipole Method.

- `tnewton [nfull` *`number`*`] [nhscale` *`number`*`] -`
  `[verbose` *`number`*`] [tncut` *`value`*`]`

`nfull` Number of minimization steps per update of the long-range forces (as defined by the `tncut` value). The default is 10, and values higher than 20 are not recommended. Setting `nfull` too high can result in unrealistic structures and/or failure of the minimization. The short-range forces are updated at every minimization step.

`nhscale` Scale factor for the size of the Hessian matrix. The amount of memory allocated for this matrix will be the `nhscale` value times the number of atoms in the system. The default is 50.

`verbose` Controls the amount of printing. The default is 0. A positive value will result in a large amount of output, and is not recommended in general.

`tncut` Cutoff distance between short-range and long-range forces. Forces between atoms more distant than this will be calculated only every `nfull` minimization steps, as opposed to every step for the short-range forces. The default is 10.0 Å.

3.1.4 Subtask Input

This subtask inputs parameters necessary for the minimizer.

- `input cntl [ mxcyc` *`num`* `] [ rmscut` *`val`* `] [ deltae` *`val`* `]`

`mxcyc` The maximum number of cycles for the minimization (default = 100).

`rmscut` Criteria for convergence of the RMS gradient (default = 0.01).

`deltae` Criteria for convergence of the change in energy for each atom, average over the whole system (default $= 1.0 \cdot 10^{-7}$).

Important Notes:

---

[1] For details, see Xie, D. and Schlick, T., “Remark on the Updated Truncated Newton Minimization Package, Algorithm 702,” ACM Trans. Math Softw., 25, 108-122, March 1999, and Xie, D. and Schlick, T., “Efficient implementation of the truncated-Newton algorithm for large-scale chemistry applications,” SIAM J. Opt., 10: 132-154, October 1999.

1. The values for both `rmscut` and `deltae` must be met before a run is converged.
2. The minimization will stop when the convergence criteria are met.

3.1.5 Subtask Run

This command signals the program to start running the minimization. All other parameters must be set correctly before `run` is executed.

3.1.6 Subtasks Read and Write

Impact provides the `write` command to save to a file the molecular system coordinates in several formats. The `write` and `read` commands also offer a simple way of saving a snapshot of the system (coordinates and, if so desired, velocities) and restoring it afterwards.

The following description applies not only to task `minimize` but also to `dynamics`, and `montecarlo`, although in some cases (to be discussed below) not all options would make sense. There are three types of file that can be used to hold snapshots of the system: PDB (`brookhaven` or `impact` format), Maestro, *residue template*, *restart* and *trajectory* files.

To write a PDB file use the following syntax:

- `write pdb [ brookhaven | impact | nil ] -`
  `name` *species_name* `file` *filename*

Note: only coordinates can be written to a PDB file. To read a PDB file you must do so inside the `create` task.

To write a Maestro file use the following syntax:

- `write maestro [ name` *spec1* `[ name` *spec2*`] ] -`
  `file` *filename*

If the species to be written to the Maestro file are of type '`auto`' the information from the original Maestro file (or as converted from a PDB or SD file) is preserved in the output of this command. If the species is of type other than '`auto`', Impact attempts to generate a valid Maestro file by creating a type '`auto`' temporary copy of the species before writing it to the file. If two species are specified, a temporary species of type '`auto`' obtained by merging the two species is written to the file. In absence of species specification the default is to merge both Impact species in the output file. To read a Maestro file you must do so inside the `create` task.

The `write restart` and `read restart` commands are used to save and restore the coordinates (and velocities) of all particles in the system. A restart file consists of a snapshot of the cartesian coordinates and, optionally, velocities of each atom of the system. When reading or writing restart files the behavior of Impact depends on the current task unless the files are written and read using the `external` keyword, in which case Impact honors all requests made on the command line.

- In task `minimize` only coordinates can be written or read. If the command line also specifies velocities Impact will not honor the request unless `external` format is used, although no error will be generated.
- In task `montecarlo` only coordinates can be written but both coordinates and velocities can be read.[2]
- In task `dynamics` velocities are always written to a restart file, even if they are not specified on the command line. The user can, however, choose not to read them back.

In all cases the usage is the same:

- `[ read | write ] restart coordinates [ and velocities ]`
  `[ box | nobox |` nil `] -`
  `[ formatted | unformatted | external |` nil `] -`
  `[ real8 | real4 | inte2 |` nil `] -`
  `file` *`filename`*

The meaning of the keywords is explained below.

A trajectory file contains a sequence of snapshots of the system (coordinates and, sometimes, velocities of all atoms). Normally trajectory files are read using the `table` subtasks `starttrack` and `stoptrack` but they can also be read wherever a restart file can be read.

- `write trajectory coordinates [ and velocities ] [ box | nobox |` nil `] -`

  `[ unformatted | external |` nil `] -`
  `[ real8 | real4 | inte2 |` nil `] -`
  `file` *`filename`* `-`
  `every` *`number_of_steps`*
- `read restart coordinates [ and velocities ] [ box | nobox |` nil `] -`
  `[ unformatted | external |` nil `] -`
  `[ real8 | real4 | inte2 |` nil `] -`
  `file` *`filename`* `-`
  `skip to` *`frame_number`*

Caution: reading a frame (snapshot) from a trajectory file using the last syntax shown should be done with care, since strange things may happen if the user mixes the coordinates with the velocities.

`formatted`
`unformatted`
`external` (default for restart and trajectory files) A `formatted` file is an `ASCII` file containing the list of coordinates (and velocities, if appropriate). The main advantage of these files is that they are human readable, but they usually occupy too much space. An `unformatted` file, on the other hand, is binary and thus much smaller. The main disadvantage is that files generated on one machine are usually not readily read on other machines. This prompted the development of the `external` way of writing

[2] Though velocities are not very meaningful in this case.

restart and trajectory files, which offers a compact (since it is binary), machine-independent representation. This is the default for trajectory files and it is strongly recommended (`unformatted` files may not be supported in the future). As mentioned above, if the keyword `external` is specified Impact honors all requests on the command line.

`inte2`
`real4`
`real8` (default)
: These keywords control the size of the data written to (read from) a binary restart or trajectory file. When reading an `unformatted` file they must be specified, but that is not necessary when reading an `external` file since the program can find this information from the file itself. The keyword `inte2` will be ignored when reading or writing an `external` file and `real4` will be substituted instead. The sizes are chosen as follows:

  `real8` Store the data as `real*8` numbers. This is the highest precision available and uses the most disk space.

  `real4` Stores the data as `real*4` numbers. This halves the storage requirements and also reduces the precision.

  `inte2` This option is somewhat more complicated. The numbers will be scaled by 1000. and stored as `integer*2` numbers. This will leave a maximum of 5 significant figures and maximum values of ±32.767.

`[ box | nobox | nil ]`
: Write (or don't write) the dimensions of the simulation volume with the coordinates (these dimensions are needed when performing constant pressure simulations). If a constant pressure simulation is being run, `box` is the default; otherwise it is `nobox`. This option applies to `trajectory` and `restart` files.

`every` *number_of_steps*
: Determines how often coordinate sets will be written.

`skip to` *frame_number*
: When reading a trajectory as a restart file one can specify which frame (snapshot) to read. Frame numbers start at 1 and should not exceed the number of frames that were written to the file.

3.2 Task Dynamics

The object of task `dynamics` is to perform a molecular dynamics (MD) simulation for a system prepared by tasks `create` and `setmodel`.

Please Note: Dynamics simulations may not give useful results, or may terminate with errors, if the initial structure has steric clashes or other problems. Even structures that have been minimized with other programs, or those produced by Maestro's build panel, may have such problems as measured with Impact's force fields. A short Impact minimization task prior to dynamics is useful for fixing such problems.

3.2.1 Subtask Input

Reads in program control parameters for the MD run.

- `input cntl nstep` *`steps`* `[ delt` *`time_step`* `]`
- `input cntl [ constant -`
  `[ temperature [ byspecies ] [ relax` *`value`* `] | totalenergy ] -`
  `[ pressure [ dvdp` *`value`* `] [ density` *`value`* `] | volume ]`
- `input cntl [ initialize temperature -`
  `[ forspecies ( name` *`spec`* `at` *`T_i`* `)` for all species `| -`
  `at` *`T_i`* `] [ seed` *`num`* `] ] -`
  `[ stop rotations ] [ nprnt` *`freq`*`) ] -`
  `[ tol` *`tolerance`* `] [metric` *`value`*`]`
- `input cntl [ statistics [ on | off ] ]`

Unless otherwise specified the default is to run MD simulations at constant temperature and volume. This results in coupling the system to an external heat bath (with a temperature that is independent of the species). Using the keyword `byspecies` results in velocity scalings that are independent for each species. In this case the user should specify an initial temperature for each species using the `forspecies` keyword, and all species should appear on the same (logical) line. Otherwise some of the species will end up with the default initial temperature. If '`constant totalenergy`' is specified instead there will be no scaling.[1]

Specifying '`constant pressure`', as opposed to '`constant volume`', results in coupling to a pressure bath using the algorithm of Berendsen et al. (*J. Chem. Phys.*, **81**, *3684* (1984)). Molecular center of mass coordinate rescaling is implemented. The distances between molecules change proportionally to the change in box size and intramolecular distances remain unchanged. Note that a "molecule" is defined as the entity created by a '`build primary`' command. Center of mass coordinate rescaling is ineffective for systems composed of a single molecule (systems built with only one '`build primary`' command). A solvent species is composed of as many molecules as created by the '`build solvent`' command.

[1] The total energy may actually not be conserved, due to the effects of a sharp cutoff. In most cases this will lead to an unstable simulation.

Independent of whether the simulation is run at '`constant temperature`' or '`constant totalenergy`' the user can initialize the temperature of all species (either the same for all or on a per-species basis) with the keywords '`initialize temperature`'. **Caution:** by default the temperature is not initialized since this could result in overwriting the velocities read from a restart file. Right after a `minimization`, the user *should* initialize the temperatures of all species to sensible values. The user *should not* use '`initialize temperature`' though, if there is an external restart file (with both coordinates and velocities) read in.

Several parameters can be specified in the '`input cntl`' line:

`nstep` Number of MD steps (*must* be larger than one!).

`nprnt` Gives the number of steps after which contributions to the energy will be printed out (5).

`delt` Gives the time step in picoseconds (0.001).

`relax` Relaxation time in ps for velocity scaling (if using '`constant temperature`') (0.01).

`seed` Seed to be used to start the random number generator when initializing the temperature of (any) species.

`taup` Relaxation time in ps for volume scaling (if using '`constant pressure`') (0.01).

`dvdp` Isothermal compressibility $1/V(dV/dP)$, in units of atm^{-1}. The default is the value for water: $4.96 \cdot 10^{-5}$ atm^{-1}. This quantity is needed for constant pressure simulations.

`density` Effective density (g/cm^3) of solute molecules. Needed to compute long-range corrections to the pressure (1.0).

`tol` Tolerance to be used when applying the constraints in `SHAKE` and `RATTLE` ($1.0 \cdot 10^{-7}$).

`stop rotations`
Flag for stopping the center of mass motion. Default is not to stop the center of mass motion.

`statistics on`
`statistics off`
Toggles collection of statistics on the fluctuations of the different energy terms during the simulation. In earlier versions this was always on; now it is off by default.

- `input target temperature` *`T_f`*
- `input target ( [ name` *`spec`* `] temperature` *`T_f`* `)` repeated for all species

Allows the specification of the final temperature (T_f) for the whole system or by species. The first form should be used only if the scaling is done on a species-independent basis. If the `byspecies` keyword was used, however, the second form must be used and all the species should appear on the same (logical) line. Multiple '`input target`' lines would result in conflicts.

The actual temperature will fluctuate about the desired value. At each MD step the kinetic energies will be scaled so the temperature will approach the desired value on a timescale determined by the `relax` parameter.

- `input target pressure` *P_f*

Reads in the final pressure (*P_f*) of the system. The same comment as in the previous paragraph applies, *mutatis mutandis*.

3.2.2 Subtask Run

Performs the actual molecular dynamics run. The temperatures are initialized at this step, not when the values are read from the '`input cntl`' line. The user can choose among three different algorithms for the integration of the equations of motion: the Verlet algorithm, which is the default; and two based on the reversible RESPA (r-RESPA) of Tuckerman, Berne and Martyna, *J. Chem. Phys.*, **97** (1992). Currently at most three inner stages are allowed and the frequency with which the corresponding forces are updated is controlled by the parameters *freqf* (fast forces), *freqm* (medium and slow forces) and *freqs* (slow forces). Currently *freqm* and *freqs* only have meaning if the FMM (fast multipole) code is used. On the other hand, *freqf* can be used with or without the FMM since it controls only the bonding forces. If the FMM is used and *freqs* is present, the forces are separated in three pieces: those arising from nearby bodies; those arising from bodies in the first and second neighbors that are not very close, and those coming from the local expansions. If *freqs* is not present but *freqm* is, the second and third are collected together.

- `run [ verlet | rrespa fast` *freqf* `[ medium` *freqm* `[ slow` *freqs* `] ] ]`

3.2.3 Subtasks Read and Write

Read or Write a) a `restart` file containing final coordinates, and velocities (forces could also be written) or b) a `trajectory` file (see Section 3.1.6 [Read/write (minimize)], page 53).

3.2.4 Subtask Convert

This subtask is provided to ease the transition to the new, default, `external` binary format (see Section 3.1.6 [Read/write (minimize)], page 53).

- ```
  convert -
    from [ unformatted | external ] file filename -
    to [ unformatted | external ] file filename -
    [ real4 | real8 | inte2 ] [ box | nobox ] -
    [ first start last end ]
  ```
  ```

Reads a trajectory file written in one format and writes it out in another. The keywords `box`, `nobox`, `real8`, `real4` and `inte2` apply only to the output file and allow the user to specify the corresponding options differently from the ones used when the input file was written (see Section 3.1.6 [Read/write (minimize)], page 53). Note that `inte2` is the same as `real4` when using the `external` format.

The parameters `start` and `end` allow the user to convert only a portion of the trajectory file. Since both input and output formats can be the same this is a handy way of extracting a consecutive sequence of frames.

3.3 Task Hybrid Monte Carlo (HMC)

The Hybrid Monte Carlo (HMC) method is often called "bad MD but good MC". Even though HMC is regarded as a Monte Carlo method, it uses Molecular Dynamics to perform the conformation-space search. Thus, in many respects, HMC's subtasks can be compared to those for Molecular Dynamics, as both usually call the same functions. Since molecular dynamics is only used for generating new conformations, a much larger time step can usually be used (this is why it is called bad MD), with the Metropolis criterion determining which moves to accept or reject.

3.3.1 HMC Methodology

The J-Walking and S-Walking methods are also implemented on the basis of the HMC protocol, and can be turned on by specifying subtasks. Since HMC performs the same simulation as does constant temperature molecular dynamics, many input controls for constant temperature MD are also suitable for HMC or are very similar for it, as you can see from the example shown below.

The following is a brief description of the S-walking (Smart Walking) method proposed by R. Zhou and B. J. Berne.[1] The S-Walking method is closely related to the J-Walking method proposed by Frantz et al.[2] Like the J-Walking method, the S-Walking method runs two walkers, one at the temperature of interest, the other at a higher temperature that more efficiently generates ergodic distributions. Instead of sampling from the Boltzmann distribution of the higher temperature walker as in J-Walking, S-Walking first approximately minimizes the structures being jumped into, and then uses the relaxed structures as the trial moves at the low temperature. By jumping into a relaxed structure, or a local minimum, the jump acceptance ratio increases dramatically. This makes the protein system easily undergo barrier-crossing events from one basin to another, thus greatly improving the ergodicity of the sampling. The method approximately preserves detailed balance provided the time between jumps is large enough to allow effective sampling of conformations in each local basin.

Here is a very simple example of a HMC calculation that uses S-Walking

[1] *J. Chem. Phys.*, **107**, *9185* (1997)
[2] *J. Chem. Phys.* **93**, *2769* (1990)

```
HMC
   input cntl mxcyc 10000 nmdmc 5 delt 0.0015 relax 0.01 seed 101 -
         nprnt 100 tol 2.0e-7
   input cntl swalk cycgap 5000 cycrec 20 minstep 100 -
         jtemp 500.0 jrate 0.1
   input target temperature 300.0
   write trajectory coordinates and velocities every 10 -
         external file pentpep.trj
   run
   write  restart coordinates and velocities formatted file pentpep.rst
   write pdb brookhaven name pentpep file pentpep_swk.pdb
QUIT
```

3.3.2 Subtask Input

Reads in program control parameters for the HMC run.

- `input cntl mxcyc` *`cycles`* `[ nmdmc` *`num`* `] [delt` *`time_step`* `] -`
  `[ relax` *`val`*`] [ seed` *`num`* `] [ stop rotations ] [ nprnt` *`freq`* `] -`
  `[ tol` *`tol`* `] [metric` *`value`*`]`
- `input cntl [ statistics [ on | off ] ]`
- `input cntl [ swalk | jwalk ] [ cycgap` *`cycles`* `] [cycrec` *`cycles`* `] -`
  `[ jtemp` *`temp`* `] [ jrate` *`rate`* `] [ minstep` *`steps`* `] [metric` *`num`* `]`
- `input target temperature` *`T_f`*

HMC samples the conformation space with the canonical ensemble. Thus the underlying molecular dynamics by default is constant temperature constant volume MD. This results in coupling the system to an external heat bath with a temperature that is specified by '`target temperature`'. Note that unlike dynamics, there is no '`initialize temperature`' option for HMC. Instead, HMC initializes velocities to a distribution based on '`target temperature`' at the beginning of each HMC step.

Several parameters can be specified in the '`input cntl`' line:

`mxcyc` Number of HMC cycles to be performed.

`nmdmc` Number of MD steps per HMC cycle (5). The total number of MD steps will be equal to (mxcyc * nmdmc).

`nprnt` Number of MD steps after which contributions to the energy will be printed out (5).

`delt` Time step in picoseconds (0.001).

`relax` Relaxation time in ps for velocity scaling (if using '`constant temperature`') (0.01).

`seed` Seed to be used to start the random number generator when initializing the velocities for any species.

`tol` Tolerance to be used when applying the constraints in `SHAKE` and `RATTLE` ($1.0 \cdot 10^{-7}$).

`jwalk` Turn on the `jwalk` option. This option performs J-Walking with other parameters specified by following items. It runs an extra high-temperature walker for barrier crossing, so the total MD steps will be doubled.

`swalk` Turn on the `swalk` option. This option performs S-Walking with other parameters specified by following items. It also runs an extra high-temperature walker for barrier crossing, so the total MD steps will be doubled. The difference between `swalk` and `jwalk` is that `swalk` option performs a rough local minimization for high-temperature conformations, while the `jwalk` option does not.

`cycgap` Number of HMC cycles for the high-temperature walker or low-temperature walker before they switch (1000). The two walkers are run in tandem.

`cycrec` Number of HMC cycles between records written of the high temperature-walker's configuration (20), where cycgap/cycrec = number of records stored in file *highT.cnf*.

`jrate` Trial jump rate (1.0%).

`jtemp` Jump-S/Jwalker's (high-temperature walker) temperature (500.0 K).

`minstep` Steepest decent minimization steps in S-walking (100)

`metric` Parameter for ergodicity analysis (0). metric = 1, perform ergodic metric calculation; metric = 0, no metric calculation.

`stop rotations`
Flag for stopping the center of mass motion. Default is not to stop the center of mass motion.

`statistics on`
`statistics off`
Toggles collection of statistics on the fluctuations of the different energy terms during the simulation. In earlier versions this was always on; now it is off by default.

- `input target temperature` *`T_f`*

Allows the specification of the final temperature (T_f) for the whole system. The actual temperature will fluctuate about the desired value. At each MD step the kinetic energies will be scaled so the temperature will approach the desired value on a timescale determined by the `relax` parameter.

3.3.3 Subtask Run

Performs the actual molecular dynamics run, as described in the Molecular Dynamics Run subection (see Section 3.2.2 [Run (dynamics)], page 58). The temperatures are initialized at this step, not when the values are read from the '`input cntl`' line. The user can choose among three different algorithms for the integration of the equations of motion: the Verlet algorithm, which is the default; and two based on the reversible RESPA (r-RESPA) of Tuckerman, Berne and Martyna, *J. Chem. Phys.*, **97** (1992). Currently at most three inner stages are allowed and the frequency with which the corresponding forces are updated is controlled by the parameters *freqf* (fast forces), *freqm* (medium and slow forces) and *freqs* (slow forces). Currently *freqm* and *freqs* only have meaning if the FMM (fast multipole) code is used. On the other hand, *freqf* can be used with or without the FMM since it controls only the bonding forces. If the FMM is used and *freqs* is present, the forces are separated in three pieces: those arising from nearby bodies; those arising from bodies in the first and second neighbors that are not very close, and those coming from the local expansions. If *freqs* is not present but *freqm* is, the second and third are collected together.

- `run [ verlet | rrespa fast` *`freqf`* `[ medium` *`freqm`* `[ slow` *`freqs`* `] ] ]`

3.3.4 Subtasks Read and Write

Read or Write a) a `restart` file containing final coordinates, and velocities (forces could also be written) or b) a `trajectory` file (see Section 3.1.6 [Read/write (minimize)], page 53).

3.3.5 Subtask Convert

This subtask is provided to ease the transition to the new, default, `external` binary format (see Section 3.1.6 [Read/write (minimize)], page 53).

- ```
  convert -
    from [ unformatted | external ] file filename -
    to [ unformatted | external ] file filename -
    [ real4 | real8 | inte2 ] [ box | nobox ] -
    [ first start last end ]
  ```

  Reads a trajectory file written in one format and writes it out in another. The keywords `box`, `nobox`, `real8`, `real4` and `inte2` apply only to the output file and allow the user to specify the corresponding options differently from the ones used when the input file was written (see Section 3.1.6 [Read/write (minimize)], page 53). Note that `inte2` is the same as `real4` when using the `external` format.

  The parameters `start` and `end` allow the user to convert only a portion of the trajectory file. Since both input and output formats can be the same this is a handy way of extracting a consecutive sequence of frames.
  ```

3.4 Task Linear Response Method (Liaison, LRM, or LIA)

Liaison, embodied in the `LRM` or `LIA` task, is Schrödinger's implementation of the Linear Response Method (LRM), also called the Linear Interaction Approximation (LIA), a method of combining molecular mechanics calculations with experimental data to build a model scoring function for the evaluation of ligand-protein binding free energies.

3.4.1 Liaison Overview

LRM-type methods were first suggested by Aqvist (J. Aqvist, C. Medina and J. EA. Samuelsson, *Protein Eng.* **7**, 385-391, 1994; T. Hansson and J. Aqvist, *Protein Eng.* **8**, 1137-1144, 1995), based upon approximating the charging integral in the free energy perturbation formula with a mean value approach in which the integral is represented as half the sum of the values at the endpoints, namely the free and bound states of the ligand. Since then they have been pursued by a number of research groups including that of Jorgensen (D. K. Jones-Hertzog and W. L. Jorgensen, *J. Med. Chem.*, **40**, 1539-1549, 1997), who has reported very good results for a number of ligand binding data sets. From a computational standpoint, this approximation has a number of highly attractive features:

1. In contrast to free energy perturbation (FEP), where a large number of intermediate *windows* must be evaluated, the LIA requires simulations of only the ligand in solution and the ligand bound to the protein. The idea is that one views the binding event as a replacement of the aqueous environment of the ligand with a mixed aqueous/protein environment.
2. Again in contrast to FEP, one can study disparate ligands as long as they have similar binding modes. FEP allows only very small changes between ligands to be investigated; the differences in the data sets we have examined up to this point are much more significant.
3. Only interactions between the ligand and either the protein or the aqueous environment enter into the quantities that are accumulated during the simulation; the ligand-ligand, protein-protein and protein-water interactions are part of the "reference" Hamiltonian and hence are used to generate configurations in the simulation (via either Monte Carlo or molecular dynamics) but are not used as descriptors in the resulting model for the binding free energy (see below). This eliminates a considerable amount of noise and systematic uncertainties in the calculations, for example arising from different conformations of the protein obtained from cocrystallized structures of different ligands.
4. The method as implemented by Jorgensen et al. contains three terms in the empirical formula for the binding energy: electrostatic, van der Waals, and solvent accesible surface area (SASA):

$$\Delta G = \alpha(\langle U^b_{elec}\rangle - \langle U^f_{elec}\rangle) + \beta(\langle U^b_{vdw}\rangle - \langle U^f_{vdw}\rangle) + \gamma(\langle U^b_{SASA}\rangle - \langle U^f_{SASA}\rangle)$$

$\langle ... \rangle$ means ensemble average from a Monte Carlo or Molecular Dynamics simulation, and all terms are evaluated only for interactions between ligand and its "environment". Aqvist et al. used only two terms in their original work, i.e., electrostatic and van der Waals interaction. However, Jorgensen et al. found that it is necessary to add one more term for larger data sets, and the third term was also proposed to be just a constant term. In our implementation as discussed later, the third term is based on the cavity energy in the SGB continuum solvent model.

If the linear response approximation was rigorously valid, the coefficient of the electrostatic term would be 0.5, corresponding to the mean value approximation to the charging integral. In fact, one can recover a value very close to this for less complex systems, such as solvation of small molecules in water. However, some of the steps involved in the binding event, such as the removal of water from the protein cavity and subsequent introduction of the ligand, are unlikely to be accurately described by a linear model. Therefore, in practice, optimization of fitting parameters yields electrostatic coefficients that are significantly different from the ideal value of 0.5. By allowing this empirical element, one is sacrificing generality; the method probably requires that the ligands have similar binding modes, and new parameters must be developed for each receptor. In return, however, one can obtain a reasonable level of accuracy (reflected in cross-validation studies as well as the overall fitting accuracy) with a modest expenditure of CPU time, under assumptions that are quite reasonable for many structure-based drug design projects.

We have developed an implementation of the LIA, in the context of the Impact program, using the generalized Born continuum solvation model and the OPLS-AA force field of Jorgensen and coworkers. To our knowledge, this is the first commercially available version of the LIA and the first version of any type to utilize continuum solvation. Key features of the Schrödinger implementation are as follows:

1. First, we replaced the solvent accessible surface area term in Jorgensen's LIA formulation by the cavity term in the continuum solvent model:

   $$\Delta G = \alpha(\langle U^b_{elec}\rangle - \langle U^f_{elec}\rangle) + \beta(\langle U^b_{vdw}\rangle - \langle U^f_{vdw}\rangle) + \gamma(\langle U^b_{cav}\rangle - \langle U^f_{cav}\rangle).$$

   We think it makes sense to use such a term in the context of a continuum solvent model. Indeed, it is not clear why the solvent accessible surface area is needed in an explicit solvent model, since waters are explicitly represented already.

2. The use of a continuum model provides much more rapid convergence of the simulations. The statistics on the various interaction terms are significantly better converged than in an explicit solvent simulation, and the required CPU time is much smaller.
3. We have implemented an automatic atom typing scheme for the OPLS-AA force field that assigns charges, van der Waals, and valence parameters with no human intervention. A key feature of OPLS-AA is excellent reproduction of condensed phase properties, obtained via fitting to liquid state simulations. Over the past years Jorgensen and coworkers have rapidly extended the functional-group coverage of OPLS-AA to include a larger number of pharmaceutically relevant species. This work will be continued and expanded at Schrödinger and at Columbia University (Prof. Richard Friesner) in collaboration with Professor Jorgensen. We intend in the coming year to increase both the accuracy and coverage of OPLS-AA substantially.
4. The Maestro interface to Liaison produces scripts that allow a series of Liaison jobs to be run automatically. This makes it convenient to use the method in the context of an industrial structure-based drug design effort, in which a large number of molecules need to be examined.

Here is a very simple LRM example that uses the SGB continuum solvent model

```
LRM
 assign ligand name drug
 input cntl average every 10 file lrm_bound.ave
 sample dynamics
 input cntl nstep 10000 delt 0.001 relax 0.01 nprnt 100 seed 101 -
   constant temperature
 input target temperature 300.0
 run rrespa fast 2
 write restart coordinates and velocities formatted file cmpx_lrm.rst
 write pdb brookhaven name prot file prot_lrm.pdb
 write pdb brookhaven name drug file lig_lrm.pdb
QUIT
```

3.4.2 Subtask Assign

Specifies the LRM or LIA ligand in the LRM simulation. This *ligand* can in fact be any entity; it could be a single ligand, a pair of ligands from a ternary complex, or even a protein, as long as all the components reside in a single species.

- `assign ligand name` *spec*

`name` *spec* determines the LRM ligand. The program thus will calculate and collect all interactions between this ligand and its "environment" (protein or water), but not the interactions within ligand itself or the protein (water) itself. In the continuum solvent model, this means that we need to separate the single and pairwise energies in the Generalized Born model into proper partial contributions to represent the LIA interaction between ligand and protein.

3.4.3 Subtask Param

Specifies LRM or LIA parameters, i.e., α, β, γ in the LRM simulation.

- `param elec` *`val`* `vdw` *`val`* `cavity` *`val`*

As mentioned above, the current method requires that new parameters be developed for each receptor, so this option is not actually used at present. Schrödinger's Maestro user interface generates scripts, as described below, that automate the LRM simulations on various ligands with known binding energies, and perform the requisite data collection. Then the user can run another script to calculate the LRM parameters and report the goodness of the fit to the experimental binding energies. Finally, the user can apply these parameters to predict the binding energies of new systems.

3.4.4 Subtask Input

Reads in program control parameters for the LRM simulation.

- `input cntl average every` *`num`* `file` *`filename`*

This command controls options for collection of the LRM statistics. It specifies how often the average LRM interaction energies are to be calculated and which file to use to print out the ensemble averages. (Other LRM-specific options may also be specifiable here in the future.)

`every` Calculate the LRM ensemble average every *num* steps.

`file` Write out the ensemble averages to file *filename*.

3.4.5 Subtask Sample

Selects a sampling method for the LRM simulation, such as Molecular Dynamics or Hybrid Monte Carlo.

- `sample [ dynamics | HMC ]`

The commands that follow the choice of sampling method are identical to those that would be needed if that method were invoked as a standalone task. This is illustrated in the previous example, where `dynamics` was chosen as the sampling method; all commands after `dynamics` are identical to those expected for the `dynamics` task. The following example uses `HMC` as the sampling method:

```
LRM
 assign ligand name drug
 input cntl average every 10 file lrm_bound.ave
 sample HMC
 input cntl mxcyc 10000 nmdmc 5 delt 0.0015 relax 0.01 seed 101 -
       nprnt 100 tol 2.0e-7
 input cntl swalk cycgap 5000 cycrec 20 minstep 100 -
       jtemp 500.0 jrate 0.1
 input target temperature 300.0
 run
 write restart coordinates and velocities formatted file cmpx_lrm.rst
 write pdb brookhaven name prot file prot_lrm.pdb
 write pdb brookhaven name drug file lig_lrm.pdb
QUIT
```

3.4.6 Scripts for Liaison simulation and fitting

Because generating fitting data for Liaison typically involves running similar simulations on a number of different systems (the *training set*), we recommend setting up these simulations, and the parameter-fitting job based on their results, from the Maestro user interface. (See the *Liaison User Manual* for examples of setting up such jobs.) To set up a Liaison simulation job from Maestro, it is necessary to provide an overall job name and the structures that constitute the training set, which may be one receptor and several ligands. Under the *current working directory* (CWD) from which you run Maestro, it sets up a directory with the overall job name ('`fit_lia`' in the following example), and a subdirectory under that for each ligand structure in the training set ('`pose1_H15`', etc.):

```
hal9000% ls -l
total 912
-rw-r--r--    1 banks    glidegrp     119 Jul 20 11:19 bindE.expt
-rwxr-xr-x    1 banks    glidegrp     374 Jul 20 11:19 change_sgbparam_fit_lia*
-rwxr-xr-x    1 banks    glidegrp     312 Jul 20 11:19 fit_fit_lia*
drwxr-xr-x    7 banks    glidegrp     116 Sep 10 10:27 fit_lia/
-rw-r--r--    1 banks    glidegrp  430687 Jul 20 11:19 fit_lia.mae
-rw-r--r--    1 banks    glidegrp    1170 Jul 20 11:19 liafit_fit_lia.out
-rwxr-xr-x    1 banks    glidegrp     452 Jul 20 11:19 simulate_fit_lia*
hal9000% ls -l fit_lia
total 64
drwxr-xr-x    2 banks    glidegrp    4096 Sep 10 10:27 pose1_H15/
drwxr-xr-x    2 banks    glidegrp    4096 Sep 10 10:27 pose2_H16/
drwxr-xr-x    2 banks    glidegrp    4096 Sep 10 10:27 pose3_H17/
drwxr-xr-x    2 banks    glidegrp    4096 Sep 10 10:27 pose4_H12/
drwxr-xr-x    2 banks    glidegrp    4096 Sep 10 10:27 pose5_H11/
```

```
hal9000% ls -l fit_lia/pose1_H15
total 1864
-rw-r--r--    1 banks    glidegrp    1170 Jul 20 11:19 bound.inp
-rw-r--r--    1 banks    glidegrp     799 Jul 20 11:19 free.inp
-rw-r--r--    1 banks    glidegrp     558 Jul 20 11:19 pose1_H15.bound.ave
-rw-r--r--    1 banks    glidegrp   12979 Jul 20 11:19 pose1_H15.bound.log
-rw-r--r--    1 banks    glidegrp   33587 Jul 20 11:19 pose1_H15.bound.out
-rw-r--r--    1 banks    glidegrp     186 Jul 20 11:19 pose1_H15.free.ave
-rw-r--r--    1 banks    glidegrp   12205 Jul 20 11:19 pose1_H15.free.log
-rw-r--r--    1 banks    glidegrp   35752 Jul 20 11:19 pose1_H15.free.out
-rw-r--r--    1 banks    glidegrp   10167 Jul 20 11:19 pose1_H15_lig.mae
-rw-r--r--    1 banks    glidegrp    9059 Jul 20 11:19 pose1_H15_lig_min.mae
-rw-r--r--    1 banks    glidegrp  430687 Jul 20 11:19 pose1_H15_rec.mae
-rw-r--r--    1 banks    glidegrp  363077 Jul 20 11:19 pose1_H15_rec_min.mae
```

In each of the ligand subdirectories, Maestro sets up simulation jobs for that ligand alone (‘`free.inp`’), and the ligand-receptor complex (‘`bound.inp`’), whose results give the energy terms in the LIA expression for ΔG above, for which the α, β, and γ coefficients are then fit to experimental binding energies for the systems in the training set. The command script `simulate_`*jobname* (in this case `simulate_fit_lia`) runs the simulations in each directory (either sequentially, or if the user specifies multiple processors, in parallel on the available processors), and renames the output files by prepending the name of each ligand, e.g. ‘`pose1_H15.bound.log`’.

For the parameter-fitting component of Liaison, Maestro sets up the script `fit_`*jobname*, which runs a least-squares fitting program to fit the output of the simulations to experimental data, which it reads from the file ‘`bindE.expt`’ in this case. The fitting program prints its output to the file ‘`liafit_`*`jobname`*`.out`’. (Headers, ligand names, and intercolumn spaces are abridged here to fit on the page.)

```
Input energy components:
Ligand   vdw_f  coul_f   rxn_f   cav_f   vdw_b  coul_b   rxn_b   cav_b    Expt
1_H15    0.000   0.000 -29.979   3.775 -51.264 -23.280   6.290   1.104  -
9.350
2_H16    0.000   0.000 -30.520   3.941 -51.035 -27.165   1.046   1.095 -
11.190
3_H17    0.000   0.000 -23.622   3.959 -56.821 -26.490   9.024   1.095 -
12.160
4_H12    0.000   0.000 -25.415   3.735 -50.892 -17.000  -6.610   1.093  -
9.930
5_H11    0.000   0.000 -18.047   3.756 -56.033 -16.753  -1.967   1.094 -
11.890

Liaison SVD-fitted parameters: alpha*Dvdw + beta*Delec + gamma*Dcav:
alpha =     0.145880   +-   0.018366
beta  =     0.031038   +-   0.004276
gamma =     1.517949   +-   0.383891

Chi-square:   202.172089
```

```
Binding energies fitted by SVD:
Ligand-Name     SVD-Fitted   Experiment
pose1_H15          -10.005       -9.350
pose2_H16          -10.648      -11.190
pose3_H17          -11.433      -12.160
pose4_H12          -10.795       -9.930
pose5_H11          -11.737      -11.890

RMSD error for binding energies =   0.636
```

3.4.7 Scripts for Liaison binding energy prediction

After fitting the LRM coefficients to experimental data for the training set, predicting binding energies for one or more new systems is a simple matter of running simulations on the new systems (bound and free, as for the training set) to obtain the required energy terms, which are then multiplied by the fit coefficients. In a prediction job, the Maestro interface sets up a script to run the simulations, again called `simulate_jobname`, in the *jobname* directory, where *jobname* may be different from that for the simulations on the training set. (If it's the same, the result will be to overwrite the previous `simulate_jobname` script, but there may be advantages to keeping both the training set and the predicted set under the same *jobname* directory. Here we use the job name `predict_lia` for the prediction run.) Maestro also sets up the script `predict_jobname` to calculate the predicted binding energies of one or more new ligands, using coefficients obtained from the previous fitting job. The following example is for a single ligand.

```
hal9000% ls -l
-rwxr-xr-x    1 banks         382 Jul 20 11:19 change_sgbparam_predict_lia*
-rw-r--r--    1 banks         310 Jul 20 11:19 liapredict_predict_lia.out
drwxr-xr-x    3 banks          54 Sep 10 10:27 predict_lia/
-rw-r--r--    1 banks      374748 Jul 20 11:19 predict_lia.mae
-rwxr-xr-x    1 banks         498 Jul 20 11:19 predict_predict_lia*
-rwxr-xr-x    1 banks         426 Jul 20 11:19 simulate_predict_lia*
hal9000% ls -l predict_lia
drwxr-xr-x    2 banks        4096 Sep 10 10:27 H06_altered_predict/
hal9000% ls -l predict_lia/H06_altered_predict
-rw-r--r--    1 banks         558 Jul 20 11:19 H06_altered_predict.bound.ave
-rw-r--r--    1 banks       13245 Jul 20 11:19 H06_altered_predict.bound.log
-rw-r--r--    1 banks       33572 Jul 20 11:19 H06_altered_predict.bound.out
-rw-r--r--    1 banks         186 Jul 20 11:19 H06_altered_predict.free.ave
-rw-r--r--    1 banks       11883 Jul 20 11:19 H06_altered_predict.free.log
-rw-r--r--    1 banks       30762 Jul 20 11:19 H06_altered_predict.free.out
-rw-r--r--    1 banks      374748 Jul 20 11:19 H06_altered_predict_lig.mae
-rw-r--r--    1 banks       10327 Jul 20 11:19 H06_altered_predict_lig_min.mae
-rw-r--r--    1 banks      374748 Jul 20 11:19 H06_altered_predict_rec.mae
-rw-r--r--    1 banks      364939 Jul 20 11:19 H06_altered_predict_rec_min.mae
-rw-r--r--    1 banks        1228 Jul 20 11:19 bound.inp
-rw-r--r--    1 banks         819 Jul 20 11:19 free.inp
```

The prediction script `predict_`*jobname* writes its output to the file '`liapredict_`*jobname*`.out`':

```
LIA prediction: predict_lia

Input data:
Van der Waals term coefficient (alpha) : 0.14588
Electrostatic term coefficient (beta)  : 0.031038
Cavity term coefficient (gamma)        : 1.51795

Calculated results:
Ligand-Name              Binding Energy (Kcal/mol)
H06_altered_predict             -12.780
```

3.5 Task Docking (DOCK or GLIDE)

The DOCK task, also called Glide (for Grid-based LIgand Docking with Energetics), is the heart of Schrödinger's Glide product. The docking algorithm searches for favorable interactions between a (typically) small ligand molecule and a (typically) larger receptor molecule, usually a protein. The ligand and receptor typically occupy separate Impact species, though they may also be separate molecules in the same species. The ligand must be a single Impact molecule, while the receptor may include more than one molecule, e.g. a protein and a cofactor. Because of the relative complexity of this task, several examples of its use are included in this section, in addition to the usual meta-examples under each subtask or command.

3.5.1 Description of the Docking Algorithm

The docking procedure for a given ligand molecule runs through two stages, which we refer to as *rough scoring* and *grid energy optimization*. Each stage relies on grids representing the receptor binding site, but the grids for one stage are not the same as for the other. As in other docking programs such as DOCK (E.C. Meng, B.K. Shoichet and I.D. Kuntz, *J. Comput. Chem.* **13**, 505, 1992) and Autodock (G.M. Morris, D.S. Goodsell, R.S. Halliday, R. Huey, W.E. Hart, R.K. Belew and A.J. Olson, *J. Comput. Chem.* **19**, 1639, 1998), the grids can be precomputed and stored on disk, so it is unnecessary to read in the receptor molecule, and perform computations on it, repeatedly for multiple ligands or multiple conformers of the same ligand. Using grids also makes computing the ligand-receptor interaction energy an $O(nlig)$ rather than $O(nlig * nprot)$ process, where $nlig$ is the number of atoms in the ligand and $nprot$ is the number of atoms in the receptor.

In a typical project, the user will set up the grids in one Glide run, and dock ligands in one or more subsequent runs, as described below. It is not currently possible to set up grids and dock ligands in the same run. (See "Important Operational Notes" in the *Glide Technical Notes*.) In all cases, the user should specify saving the grids to disk whenever calculating them.

In the current version of Glide, there are two possible ways to incorporate ligand flexibility: include multiple conformers of a given ligand in the input to Impact, or use the program's internal conformation generator starting with a single conformer of a given ligand. We strongly recommend the latter. It covers conformational space systematically, and by clustering conformers that have a common "core," it runs much faster than docking the same number of externally generated conformers. In conjunction with internal conformation generation, Glide also allows ligand torsional flexibility during the optimization of the ligand-receptor interaction energy, and we recommend using this feature. Future versions of Glide will allow for receptor flexibility; for now, scaling of the van der Waals radii of receptor atoms (also available for ligand atoms) mimics some possible motions of the receptor, such as "breathing" to fit a larger ligand than the one present in a particular co-crystallized structure.

In addition to generating or processing multiple conformations of a given molecule, Glide can also dock, and compare the predicted binding affinities of, multiple ligand molecules in a single Impact run, using a loop in the input scripting language (DICE). In the case of externally generated conformers, the same loop can run over a list of input structures that includes both different molecules and different conformers of each, using Impact's `build primary check` syntax to determine which is which. (The input structures for internal conformation generation can in principle also include multiple conformers of the same ligand, but there is no reason to do so, and we do not recommend it.)

The first stage of the algorithm, known as *screening* or *rough scoring*, measures the geometric "fit" between the ligand and receptor molecules, and approximations to specific interactions between them such as hydrogen bonds. The grids for the rough-scoring stage contain values of a *rough score function* representing how favorable or unfavorable it would be to place ligand atoms of given general types (e.g. polar hydrogens, hydrogen bond acceptors, hydrophobic heavy atoms) in given elementary cubes of the grid. These grids have a constant spacing, which defaults to 1 Å. The *rough score* for a given *pose* (position and orientation) of the ligand relative to the receptor is simply the sum of the appropriate grid scores for each of its atoms. By analogy with energy, favorable scores are negative, and the lower (more negative) the better.

The screening stage is actually a hierarchical series of filters that drastically narrow down the set of poses that are considered candidates for docking. A given pose is defined by three Cartesian coordinates of the ligand *center*, and three Euler angles. The ligand center is taken to be the midpoint of the *diameter*, which in turn is taken to be the longest line segment connecting two ligand atoms. Although some of the commands in the docking task use the abbreviation `cm` in keywords to refer to this point, this definition is very different from the *centroid* or "center of mass" of the ligand atom positions. Note also that it may be far from the actual position of any ligand atom. (In fact, if the ligand "wraps around" a convex portion of the receptor surface, the ligand center may be inside the receptor.) The Cartesian coordinates of the center position are defined relative to the origin of coordinates in the receptor coordinate file. The Euler angles ψ and θ are defined relative to an orientation in which the ligand diameter points along the z-axis; the ϕ angle (rotation of the ligand about its diameter) is taken to be zero in the input coordinates of the ligand. This biases one of the six coordinates in favor of its input value, but we have not found this to be a problem even when the input is the "correct answer", e.g., a co-crystallized ligand-receptor complex. It is also possible to choose the grid points to include the ligand center coordinates in the input, which introduces additional bias. The ligand poses that constitute the search space for the screening step correspond to discrete values of these six coordinates. The ligand center is placed at selected points on the rough-score grid, with the default being every other point. The ψ and θ angles are taken from the polar coordinates of a set of points uniformly distributed on the unit sphere (by default, a set of 302 such points from the file '`grid.pts`'), and ϕ is distributed evenly between 0 and 360 degrees, with the default being 25 values at intervals of 14.4 degrees.

Early filters in the screening stage are purely geometric, weeding out sites for the ligand center that have no chance of being good docking positions, because they are too far from the receptor or have no chance of shape complementarity. The later filters involve evaluating the rough-score function on subsets of the ligand atoms, such as those near the diameter (whose scores should be independent of ϕ, so ruling them out for one value of ϕ kills 25 poses based on as few as 2 ligand atoms), or hydrogen-bonding atoms (or others expected to make major contributions to favorable scores, so that if the score is not favorable for the subset, there's no point in evaluating it for the rest of the ligand). Effective application of the filters can rapidly reduce the number of poses to be considered from hundreds of thousands or millions to a few dozen (or less), before evaluating the full rough-score function on all the ligand atoms in any pose.

By default, and by our recommendation, the rough-scoring function is defined on a 1 Å grid. In the interest of execution speed, the default sites for the ligand center occupy a 2 Å grid consisting of alternating points of the rough-score grid. The default rough-score function is based on counting receptor atoms of various types within certain distances of grid points, and thus has a step-function character, and can vary considerably from one grid point to the next. Therefore a pose that gets an unfavorable score may be very close in space to one that would get a favorable score, and possibly would minimize to a good docked configuration. If the favorable score occurs for a pose with the ligand center on a skipped grid point, it might never be found. This is particularly likely for receptors with tight binding pockets.

To address this potential problem, Glide allows two enhancements of the rough-score function, which we call *greedy scoring* and *pose refinement*. Both involve examining scores at grid points surrounding the current positions of ligand atoms, but avoid the considerable expense of moving every atom of every pose through a 3x3x3 set of neighboring points.

Greedy scoring involves setting up alternative rough-score grids, which at each grid point incorporate some "influence" of the most favorable score in the 3x3x3 neighborhood of the central grid point. To construct a "greedy grid" given the original rough-score grid, the algorithm first finds the most favorable (lowest or most negative) score in the 3x3x3 neighborhood. The value stored in the greedy grid at the given grid point is then a linear combination of the original grid value and the best neighboring one: greedy = x

* best + $(1-x)$ * original. The default is $x = 0.33$, but the user may specify any value between 0 (the same as non-greedy scoring) and 1, inclusive.

Pose refinement is a method for evaluating the rough-scores of selected poses on a finer translational grid than the default. The refinement step takes each pose that passed all the screening tests, and moves the ligand center to neighboring grid points. The default step size for these moves is one grid point (1 Å), which with the default spacing of ligand center sites means that all the poses it covers other than the central one were skipped in the original search. If any of these "refined" poses gets a better score than the original (central) one, the algorithm passes the best such pose on to subsequent steps, instead of the central one.

Greedy scoring adds computational overhead for reading (and the first time, computing and writing) the greedy grid, and also, in our tests, about 10–20% to the CPU time for screening poses of a given conformation (presumably because more poses pass some of the filters). Pose refinement adds a negligible amount of time to a multiple-conformation or multiple-ligand run, and tends to *decrease* the number of poses that need to be passed to minimization. Because they significantly enhance the likelihood of finding good poses, we recommend using both features.

In a run with multiple externally-generated conformations of a given ligand, the program executes most efficiently (in both time and memory use) if it performs the (greedy) rough-score calculation for all the conformers first, keeps some specified total number of best poses over all the conformers, and then proceeds to pose refinement (and subsequent steps) only on those best overall poses of the given ligand. For internal conformation generation, the rough-scoring algorithm treats all the conformers for a given input ligand in tandem, so it automatically does pose refinement only on the best poses over all conformers.

The second stage of the docking algorithm begins with evaluation and minimization of a grid approximation to the nonbonded interaction energy between the ligand and the receptor. The grids store the values of the electrostatic potential due to the receptor atoms (with a constant or linear dielectric, at user option), and the attractive and repulsive parts of the Lennard-Jones energy. The docking algorithm is implemented only for the OPLS-AA force field. Attempting to use it with a different force field will result in an error exit from Impact.

The energy values are defined on an *adaptive* grid, with a finer spacing close to the receptor for accuracy where the potential energy is changing rapidly, and coarser far from the receptor to save time and space where the potential varies slowly (and contributes less to the total in any case). The default for the finest grid spacing is 0.4 Å, increasing to 3.2 Å in three steps. At user option, the grid energy also incorporates *smoothing functions* that eliminate the singularity in the potential energy at zero distance, and thus soften the hard walls that could otherwise trap the algorithm in local minima. We recommend starting the grid-energy minimization on the smoothed potential surface, and *annealing* to the full OPLS-AA grid energy. To accomplish this, include the subtask `smooth anneal 2` in the `DOCK` task.

The energy evaluations and minimizations use a continuous function for the energy, obtained by linear interpolation among the values at the corners of the cube of grid points surrounding each ligand atom position. The position and orientation coordinates of the ligand are varied continuously during the minimization. With Glide's internal conformation generation feature, we also provide, and recommend, the option of varying ligand dihedral angles during the minimization.

Glide performs its calculations in the context of two concentric rectangular boxes, representing different aspects of the receptor active site. The *bounding box* (or "ligand center box") delimits the space in which the ligand center (as defined above) can move. The size of this box determines the size of the space that the algorithm explores, and thus the amount of computer time (and to some extent memory) it takes to execute, so to optimize performance, it should be as small as the user's knowledge of the binding site will allow. Around this bounding box, the *enclosing box* is the space in which Glide defines and calculates the grid values for the rough-score and energy functions. The algorithm rejects a candidate site for the ligand center if any conformation and pose of the ligand, with its center at that site, would have any atom outside the enclosing box. Therefore it is important to make the enclosing box large enough relative to the bounding box so that the ligand will fit inside it at all likely sites for its center. Memory restrictions, unfortunately, limit the size of the enclosing box to 50 Å on a side.

The location and dimensions of the bounding and enclosing boxes are either calculated from the coordinates of the receptor atoms in residues that the user specifies as `active`, taken directly from user specifications via the `box` keyword in the `receptor` and/or `screen` subtasks, or read from grid files previously stored to disk.

3.5.2 Example 1: Set up grids

The following example sets up grids based on the receptor in the cocrystallized thrombin-inhibitor complex contained in PDB entry 1ETS. Subsequent examples dock ligands to this receptor, as represented by these grids. In the text accompanying these examples, we briefly explain the subtasks of the `DOCK` task. In later sections devoted to each subtask, we provide more detailed descriptions, and information about overriding defaults for parameters or options not shown here. It is important to note that all of the subtasks except `confgen`, `simil`, and `run` simply set up the specifications and parameters for the docking run; except for `confgen`, which immediately generates conformations, and `simil`, which immediately generates or reads similarity weights, Impact does not perform any docking calculations until it encounters `run`. Thus every invocation of the `DOCK` task must end with the `run` subtask. Note also that every subtask of this task occupies a single *logical line* of the Impact input file. Thus it is crucial to include the hyphens to indicate continuation of the command (subtask) on the next physical line. Furthermore, it is important to remember that each *physical* line of the Impact input file is truncated after 132 characters. For this reason, all file names in the examples shown here are on separate physical lines (with hyphens for continuation as needed). Users must insure that all their file pathnames (including directories) are short enough to fit in this limit, which typically means 128 or 130 characters in order to leave room for quotation marks and/or hyphens. The Maestro user interface will refuse to write an Impact input file, or start the corresponding job, if the user specifies a pathname that is too long. We recommend that users who have complicated directory structures should either run Impact in directories close to where their files are located, or if this is not practical, use such `Unix` system features as symbolic links or environment variables to shorten the names to be written to the Impact input file.

It will be noted that unlike most Impact input files, none of the examples in this section contains a `setmodel` task. This is because Glide computes energies differently from other tasks such as `minimize` and `dynamics`. It does so by precomputing receptor grids using the OPLS-AA force field, and reading (and interpolating) energies from them for ligand atoms, rather than looping over atom pairs. For this reason, this task does not require `setmodel` to specify features and parameters of the energy function.

```
write file "1ets_single_grid.out" -
      title "1ets_single_grid" *

CREATE
  build primary name recep type auto -
    read maestro file -
"1ets_single_grid.mae" -
    tag REC_
  build types name recep
QUIT

DOCK
  smooth anneal 2
  receptor name recep -
   writef -
"1ets_single_grid" -
   protvdwscale factor 0.900000 ccut 0.250000 -
   box center read xcent -37.510494 ycent -28.946030 zcent 44.411289 -
   boxxrange 27.346889 boxyrange 27.346889 boxzrange 27.346889 -
   actxrange 27.346889 actyrange 27.346889 actzrange 27.346889
  screen greedy -
   box center read xcent -37.510494 ycent -28.946030 zcent 44.411289 -
   ligxrange 12.000000 ligyrange 12.000000 ligzrange 12.000000 -
   writescreen -
"1ets_single_grid.save" -
   writegreed -
"1ets_single_grid_greedy.save"
  parameter clean
  final glidescore
  run
QUIT

END
```

`smooth` Indicates that the calculation of the energy grids should incorporate short-distance smoothing functions. `anneal 2` indicates that the grids should include two different potential-energy surfaces, one with smoothing and one without. In a `DOCK` task to do grid-energy optimization, `smooth anneal 2` means that the optimization should start on the smoothed surface and end on the unsmoothed one. Alternatively, a subsequent `DOCK` task could include `smooth anneal 1` to use only the smoothed surface, or omit the `smooth` subtask in order to use only the unsmoothed surface; but we strongly recommend using `smooth anneal 2` in all cases.

`receptor` Specifies the receptor molecule(s) and its active site.

`name recep`
Indicates that the receptor is in the Impact species designated `recep` in the preceding `CREATE` task. If

this species contained more than one molecule, then by default the receptor would include all molecules in the species; specifying `mole` *mol* in this subtask would restrict the receptor to that single molecule.

`writef 1ets_single_grid`

Indicates that the energy grids will be written to files whose names are built from the base `1ets_single_grid`. Specifically, '`1ets_single_grid.grd`' will contain structural information about the adaptive grid itself (size and coordinates of each grid box), '`1ets_single_grid_vdw.fld`' will contain the Lennard-Jones energy grid, '`1ets_single_grid_coul.fld`' will contain the Coulomb potential with a dielectric constant of 1, and '`1ets_single_grid_coul2.fld`' will contain the Coulomb potential with a distance-dependent dielectric of $1 * r$. In addition, Impact will write the receptor structure to a Maestro format file, '`1ets_single_grid_recep.mae`', for use in subsequent Glide jobs. (To compute and write just one of the Coulomb files and not the other, use the keyword `writecdie` for the constant dielectric or `writerdie` for the r-dependent dielectric. `writerdie` overrides `writecdie`, so if you specify both, only the r-dielectric will be computed and written. To specify a dielectric other than 1 or $1*r$, use the `dielco` keyword in the `minimize` subtask.) NOTE: The files read and written by Glide can be very large (tens of megabytes). To save space on user disks, and also to save time (network latency) in environments where the user disk is on a server other than the local CPU, we recommend reading and writing these files on local "scratch" disks while running Impact, and transferring them to more "permanent" locations separately.

`protvdwscale`

Specifies a scale factor (`factor`) for the van der Waals radii of nonpolar receptor atoms. All atoms whose partial charge (absolute value) is less than `ccut` are considered nonpolar for this purpose. Specifying `factor` < 1.0, by effectively making receptor atoms seem smaller to ligands, is a way of letting the receptor "breathe" to accommodate larger ligands than the one that happened to be in the cocrystalized complex from which the receptor structure was taken. Omitting this keyword will result in no scaling (equivalent to `factor 1.0`), but we recommend using some scaling factor such as 0.9 (which the Maestro interface writes to input files). See the Glide Technical Notes for further discussion of vdW scaling factors.

`box` Specifies the rectangular (in this case cubic) box in which the rough-score and energy grids are defined. (This is sometimes called the *enclosing box*). `center read` indicates that the coordinates (in Angstroms) of the center of the box are given by the following `xcent` *val* `ycent` *val* `zcent` *val* keyword-value pairs. `boxxrange` *val*, etc., give the lengths (in Angstroms) of the box edges, which are always parallel to the coordinate axes. The rough-scoring algorithm rejects a ligand center site if *any* orientation of the ligand at that site would have any atoms outside the grid box, so it is important to make `boxxrange` large enough so as not to exclude any ligand positions that may be desirable with some orientations of the ligand but outside the box with others. If `actxrange`, etc., are specified, they indicate that any residues with any atoms in a box of that size (and the given center) are counted as contributing to the receptor *surface*, a set of points on the van der Waals surface of the specified atoms, which is used to determine distances of grid points or boxes from the receptor. We strongly recommend `actxrange` = `boxxrange`, etc., but problems with the surface-generation algorithm require `actxrange`, etc., no greater than 50.0. In such cases it is acceptable to use `boxxrange > actxrange`, etc., but in fact `boxxrange >` 50.0 is probably not necessary except for unusually large ligands or broad binding regions.

`screen` Requests the rough-score screening phase of the calculation (in this case, just setting up the rough-score grids), and specifies parameters for its performance.

`greedy` Use the greedy-scoring algorithm.

`box` Specifies the box in which the ligand center is moved. (Sometimes called the *bounding box*.) As in the `receptor` subtask, `center read` indicates that the coordinates of the box center are to be read from the following specification. In order to leave equal space for ligand atoms on all sides of the bounding box, its center should be the same as that of the "enclosing box" specified in the `receptor` subtask; but for historical reasons, Impact will accept specification of different centers for the two boxes. `ligxrange 12.0 ligyrange 12.0 ligzrange 12.0` indicates that the ligand center should move in a box of dimensions 12 Å on a side (i.e., 6 Å in each positive and negative direction from the center of the box).

`writescreen`
Write the rough-score grids to the indicated file.

`writegreed`
Write the greedy-score grids to the indicated file.

`parameter`
This subtask specifies various general parameters and conditions for running the `DOCK` task. `clean` tells Impact to delete various dynamically-allocated arrays after the task is completed. If there were subsequent `DOCK` tasks in this job, they would need the data stored in those arrays, so `clean` would not appear here.

`final` Specifies the "final" scoring function that Glide is to use for ranking ligands. `glidescore` indicates Schrödinger's proprietary *GlideScore* (tm) scoring function, adapted from the ChemScore function found in the literature.[1] `noglidescore` would indicate using just the minimized grid energy (Coulomb + vdW), which in general is inadequate for comparing different ligand molecules. The `final glidescore` subtask is needed here, even though this task does not dock any ligands, because GlideScore requires information about the receptor molecule that may not be available in the actual docking task. Glide writes this information to a file called *`basename`*`.csc`, where *basename* is the name specified with `receptor writef`, in this case `1ets_single_grid`.

`run`

Run the calculation. The output consists of the grid and receptor data files, for use in subsequent docking tasks or

[1] Eldridge et al. *J. Comput. Aided Mol. Design*, **11** p. 425–445, 1997

jobs. In this case, they will be '1ets_single_grid.grd', '1ets_single_coul.fld', '1ets_single_grid_coul2.fld', '1ets_single_grid_vdw.fld', '1ets_single_grid.save', '1ets_single_grid_greedy.save', '1ets_single_grid_recep.mae' (receptor data for use by the report subtask in a subsequent job or DOCK task), and '1ets_single_grid.csc'. The '.grd' and '.fld' files are binary, the rest are ASCII.

3.5.3 Example 2: Single Ligand, Single Conformation

The following example uses the receptor data and grid files that the previous one wrote, to dock a single ligand, which happens to be the cocrystallized ligand from the same "1ets" thrombin-inhibitor complex as the receptor. This example shows rigid docking of a single conformation of the ligand. The next (multi-ligand) example will show internal conformation generation, and torsional flexibility in the energy optimization stage.

This example contains four different DOCK tasks, for different stages of the calculation. Some of these could be combined for this particular run, but are separated either because that's the way they would appear in a multi-ligand run (some within a WHILE loop, others outside it), or in order to illustrate different options for the commands included in the DOCK task.

```
write file "1ets_single_dock.out" -
      title "1ets_single_dock" *

DOCK
  smooth anneal 2
  receptor rdiel readf -
"1ets_single_grid"
  screen readscreen -
"1ets_single_grid.save" -
   greedy readgreed -
"1ets_single_grid_greedy.save" -
   maxkeep 1000 scorecut 100.000000
  ligand multiple maxat 100 maxrot 15 -
   ligvdwscale factor 0.800000 ccut 0.150000
  parameter setup save maxconf 1
  final glidescore
  report setup by glidescore nreport 500 -
   maxperlig 1 rmspose 0.500000 delpose 1.300000
  run
QUIT

CREATE
  build primary name lig type auto read maestro file -
"1ets_single_dock.mae" -
   tag LIG_ gotostruct 1
  build types name lig
QUIT
```

```
DOCK
  ligand name lig
  screen
  parameter save
  run
QUIT

DOCK
  smooth anneal 2
  ligand keep
  screen noscore refine maxref 100
  parameter save
  final glidescore read -
"1ets_single_grid.csc"
  minimize itmax 100 dielco 2.000000
 scoring ecvdw -25.000000 hbfilt -0.700000 metalfilt 0.000000 -
 hbpenal 3.000000
  report collect -
   rmspose 0.500000 delpose 1.300000
  run
QUIT

DOCK
  parameter clean final
  report -
   rmspose 0.500000 delpose 1.300000 write filename -
"1ets_single_dock"
  run
QUIT

END
```

The first `DOCK` task above (sometimes called the *setup* task) is somewhat similar to the one in the previous example, except that it reads rather than writes files, and that it indicates (through the `ligand` subtask) that one or more ligand structures are to be docked in this job.

`receptor`

The `readf` keyword indicates reading energy grids from files with the base name given, which in this case are the ones written in the previous example. `rdiel` means use the Coulomb potential computed with the r-dielectric (and stored in '`1ets_single_grid_coul2.fld`') for all energy calculations. Since everything is read from files, no other information about the receptor (active site, box size, etc.) is needed here.

`ligand` In subsequent `DOCK` tasks in this job, this subtask gives information about the ligand(s) to be docked. In this "setup" task, however, it simply indicates that there will be ligands, so that Glide can set up arrays to hold them. Even though there is only one ligand in this case, the `multiple` keyword must precede `maxat` and `maxrot`, which give the maximum number of atoms and rotatable bonds allowed in any ligand molecule in the current job. If we were indeed looping over multiple ligands, any one that exceeded these limits would be skipped. In addition, `maxat` is used in allocating storage for the ligand atom coordinates. The `ligvdwscale` keyword invokes scaling of the ligand vdW radii used in energy calculations, similar to `protvdwscale` above. As for the protein, omitting this keyword results in setting `factor 1.0` (no scaling), but we recommend using a scale factor `<` 1.0, and the Maestro interface writes `factor 0.8`, as shown. Again, see the Glide Technical Notes for further discussion.

`parameter`
: The `setup` keyword indicates that no actual calculations are to be done in this invocation of the task. Instead, the receptor and ligand data are simply read in and stored in dynamically allocated arrays. (The sizes of most of these arrays are read from the same grid files that contain their contents.) The `save` keyword indicates that these arrays should be retained in memory for use by subsequent invocations of the task. The `maxconf` keyword gives the dimension of dynamically allocated arrays that, in general, store information for multiple ligands or (externally generated) conformations. In this case, `maxconf 1` indicates a single ligand structure.

`screen`
: As with `readf` above, `readscreen` and `readgreed` here mean read the rough-score grids from the indicated files, and we don't need a `box` specification because it's in the same files. The following additional parameters give details of the rough-score screening task to follow.

  `maxkeep`
  : Indicates the maximum number of ligand poses to be passed to the energy minimization. The number actually kept may be less than this, because fewer poses pass the various rough-score filters.

  `scorecut`
  : Rough-score window for passing poses to grid-energy optimization. A pose survives if its rough-score is within `scorecut` of the best pose accumulated so far.

`report`
: Gives instructions for the "reporting" (output) of docked ligand *poses* (A pose is the structure of a single conformation of a single ligand, in a single position and orientation relative to the receptor). The setup task requires some information about what is to be reported and how.

`setup` Indicates that we're specifying the reporting function here. Of course we can't actually collect data for the report (much less write it to output files) until we've actually docked the ligands. But we need to allocate space for the report data, etc.

`by glidescore`
Indicates that the poses to be reported will be sorted in order of the GlideScore scoring function.

`nreport` The maximum number of poses to report. (The actual number may be smaller because fewer pass all screening or scoring tests, or because of the `maxperlig` keyword.

`maxperlig`
The maximum number of poses to report for any given ligand molecule. `maxperlig 1` is particularly useful for rapid screening of large databases, producing one pose for each of the `nreport` best-scoring ligands, which can then be subjected to more detailed calculations.

`rmspose`
`delpose` The rough-score and energy-optimization stages of a Glide may generate poses for a given ligand that are similar to each other. In order to avoid duplication in the report, these keyword-value pairs indicate that two poses of the same ligand are to be considered distinct (and thus both reported if they otherwise qualify) only if the RMS deviation of their atomic positions exceeds the `rmspose` value, or the maximum deviation for any atom exceeds `delpose`. These keyword-value pairs must appear in *every* occurence of the `report` subtask in a given Glide input file.

The second `DOCK` task above runs the rough-score screening (except for pose refinement). Glide knows that it should do this (rather than just allocate arrays) because there is no `setup` keyword in the `parameter` subtask.

`ligand name lig`
Copy the indicated Impact species into the Glide ligand arrays.

`screen` Run the rough-score screening using the parameters and information specified in the previous `DOCK` task.

The third `DOCK` task runs pose refinement and grid-energy optimization.

`smooth anneal 2`
: Needed here to tell Glide to use both the smoothed and "hard" potential energy surfaces in the actual minimization. It's possible to use `smooth anneal 2` in the first task in order to calculate or read both surfaces, but `smooth anneal 1` here to use only the smoothed one, or leave out the `smooth` subtask here to use only the hard surface.

`ligand keep`
: Continue to run calculations on the ligand structure used in the previous `DOCK` task, rather than reading in a new one.

`screen`

- `noscore` Don't do the whole rough-score process here, because we did it in a previous task.
- `refine` Use pose refinement.
- `maxref` Maximum number of poses to keep after pose refinement.

`minimize` Minimize the Coulomb+vdW interaction energy (interpolated on the grids) for each ligand pose that survives through the rough-score and refinement steps.

- `itmax` Maximum number of conjugate-gradient iterations
- `dielco` Dielectric coefficient. If `cdiel` appears in the `receptor` subtask above, this is the dielectric constant. If `rdiel`, the dielectric is this number multiplied by the interatomic distance in Angstroms.

`scoring` Various filters for keeping poses after energy minimization.

- `ecvdw` Reject any pose whose minimized Coul+vdW energy is greater (in this case, less negative) than this number.
- `hbfilt` Reject any pose for which the hydrogen-bond contribution to GlideScore is greater than this number.
- `metalfilt` Reject any pose for which the metal-binding contribution to GlideScore is greater than this number
- `hbpenal` Assign this penalty in GlideScore for each buried polar interaction.

`report collect`
: After minimization, and in this case GlideScore evaluation, collect data on top poses for final output. For a single ligand, this

could be combined with the `report write` subtask in the next task. But for a loop over multiple ligands, collection is done inside the loop for each ligand, and final output is done once at the end of the job, outside the loop.

The fourth `DOCK` task writes the final output.

`parameter clean final`
: Delete dynamically allocated arrays at the end of the task. The `final` keyword insures that the Glide report function is executed even if the last ligand's structure was problematic.

`report ... write filename ...`
: Write the best poses (up to `nreport` of them, but subject to `maxperlig` and survival through all scoring filters) to the output files. For `filename` *base*, write the receptor structure and the ligand pose structures to *base*_pv.mae, and a summary of the poses and their scores to *base*.rept. The user can view the poses on screen, in conjunction with the receptor, by using the `Glide Pose Viewer`, available from the Maestro "Analysis" menu.

3.5.4 Example 3: Multiple Ligands, Flexible Docking

The above example treats a single conformation of a single ligand, to find the most favorable pose for docking to the given receptor. Probably the more common use of Glide is to determine which of a number of conformations, or which ligand of a number of candidates, has the most favorable interaction with the receptor. The `DOCK` task can be invoked repeatedly to handle multiple input ligand structures, as in the loop shown below using the DICE scripting language. (See Chapter 4 [Advanced Input Scripts], page 127 for details of DICE.) We recommend using a loop as shown here, over multiple ligands in a single file (Maestro or MDL SD format), with each structure a different ligand, and using Impact's internal conformation generator (subtask `confgen`) and torsional flexibility during grid-energy optimization (`flex` keyword in `minimize` subtask) to sample the conformational space of each ligand in turn.

After the example, we describe the ways in which this example differs from the single-structure example above.

```
write file "1ets_example_mult.out" -
      title "1ets_example_mult" *

PUT 0 INTO 'buildcheck'
PUT 1 INTO 'startlig'
PUT 0 INTO 'endlig'
PUT -1 INTO 'strucseq'


DOCK
```

```
  smooth anneal 2
  ligand multiple maxat 100 maxrot 15 -
   ligvdwscale factor 1.000000 ccut 0.150000
  receptor rdiel readf -
"1ets_single_grid"
  screen readscreen -
"1ets_single_grid.save" -
   greedy readgreed -
"1ets_single_grid_greedy.save" -
   maxkeep 5000 scorecut 100.000000
  parameter setup save maxconf 1000
  final glidescore
  report setup by glidescore nreport 500 -
   external file -
"1ets_example_mult.ext" -
   maxperlig 1 rmspose 0.500000 delpose 1.300000
  run
QUIT

CREATE
  build primary name lig type auto -
   read sd file -
"many.mol" -
   gotostruct 1
  build types name lig
QUIT

DOCK
  ligand reference name lig
  screen noscore
  parameter save
  run
QUIT

PUT 'startlig' INTO 'strucseq'
CREATE
  build primary check name lig type auto -
   read sd file -
"many.mol" -
   gotostruct 'startlig'

  build types name lig
QUIT
IF 'buildcheck' LT 0
  IF 'buildcheck' EQ -1
    PUT -
$"END OF LIGAND FILE:"$ -
INTO 'outmsg'
  ENDIF
  IF 'buildcheck' EQ -2
     PUT -
```

```
$"ERROR READING LIGAND FILE:"$ -
INTO 'outmsg'
  ENDIF
  SHOW 'outmsg'
PUT -
$"many.mol"$ -
INTO 'filemsg'
SHOW 'filemsg'
    PUT $"No ligands read; aborting."$ INTO 'outmsg'
    SHOW 'outmsg'
    GOTO ABORT
ENDIF

PUT 'startlig' INTO 'i'
WHILE ('endlig' LT 1 OR 'i' LE 'endlig')

DOCK
  ligand name lig
  screen
  parameter save
  confgen name lig -
   ecut 12.000000
  run
QUIT

DOCK
  smooth anneal 2
  ligand keep
  screen noscore refine maxref 400
  parameter save
  final glidescore read -
"1ets_single_grid.csc"
  minimize flex itmax 100 dielco 2.000000
 scoring ecvdw -25.000000 hbfilt -0.700000 metalfilt 0.000000 -
 hbpenal 3.000000
  report collect -
   rmspose 0.500000 delpose 1.300000
  run
QUIT

PUT 'i' + 1 INTO 'strucseq'
CREATE
  build primary check name lig type auto -
   read sd file -
"many.mol" -
   nextstruct
  build types name lig
QUIT

IF 'buildcheck' LT 0
  IF 'buildcheck' EQ -1
```

```
    PUT -
$"END OF LIGAND FILE:"$ -
INTO 'outmsg'
  ENDIF
  IF 'buildcheck' EQ -2
     PUT -
$"ERROR READING LIGAND FILE:"$ -
INTO 'outmsg'
  ENDIF
  SHOW 'outmsg'
PUT -
$"many.mol"$ -
INTO 'filemsg'
SHOW 'filemsg'
  PUT $"Proceeding with final processing of ligands."$ INTO 'outmsg'
  SHOW 'outmsg'
  GOTO BREAK
ENDIF

PUT 'i' + 1 INTO 'i'
ENDWHILE
:BREAK

DOCK
  parameter clean final
  report -
   rmspose 0.500000 delpose 1.300000 write filename -
"1ets_example_mult"
  run
QUIT
:ABORT

END
```

The first thing to notice about this example is the initialization of four DICE variables near the top. Of these, `'buildcheck'` is set in the Impact code (as a result of the `build primary check` command), and `'strucseq'` is read by Glide to determine a sequential ligand number that it both uses in its internal bookkeeping and writes to output files. `NOTE:` the `'strucseq'` variable *must* be present, and incremented as in `PUT 'i' + 1 INTO 'strucseq'` above, in any Glide job that docks ligands from more than one input structure, or if a reference ligand (see below) is present. Its omission in such cases will cause the entire job to fail. `'startlig'` and `'endlig'` are set and used only within the input file itself, to control the loop over ligands. In particular, `PUT 0 INTO 'endlig'`, combined with the subsequent `WHILE` command, means loop until the end of the ligand structure file. By using different settings for these variables, it is possible to run Glide for different segments of a large multi-ligand database at different times (or at the same time on different machines), without physically splitting up the file containing the ligand structures. The script `para_glide`, in the `$SCHRODINGER/utilities` directory, is useful for running such "parallel" Glide jobs.

The first (setup) `DOCK` task is almost identical to that in the previous, single-ligand case. The order of the subtasks (`ligand` before `receptor` here, the opposite order above) is irrelevant, both because the two subtasks are independent and because neither actually results in any action until the `run` subtask. The larger values of `maxconf` and `maxkeep` in this case are the ones we recommend for multiple ligands with internal conformation generation.

Another difference in this task is the presence of the `external file` specification in the `report setup` subtask. This indicates a file to which Glide writes poses that pass all tests, in the order they are generated. Glide writes its final output (see `report write` below) after processing this file to find and sort the best `nreport` poses in the order requested. The `glide_sort` script, in the `$SCHRODINGER/utilities` directory, is also available for post-processing of this file according to different (user-selectable) criteria, and sorting in order of different scoring functions, including customizable combinations of various terms in GlideScore. Writing poses to an external file also serves as a *checkpointing* facility. If a job is interrupted in the middle, the data remain available in the external file for all ligands already docked. Note that The `external file` sorting mechanism is not compatible with "rigid docking" jobs such as the example in the previous section,[2], or with "Score in place" jobs (see below). For rigid docking jobs (or `confgen` jobs if the `external file` specification is omitted), the poses that pass are stored and sorted in program memory instead. For "Score in place," only the single input pose is treated, so saving, sorting, and structural reporting are not relevant.

This example also differs from the previous one by the presence of a *reference ligand*. This is useful in cases where one of the ligands to be docked is a known binder to the receptor, with a co-crystallized structure available. That is not actually the case here, but we specify a reference ligand anyway, just to illustrate the syntax. `ligand reference name lig` indicates that the structure just read into species `lig` is the reference structure: if the first ligand actually docked is the same molecule as this structure (as determined by `build primary check` below), the output will include RMS deviations of its docked pose(s) from this reference structure. `screen noscore` indicates that no actual docking calculations are to be done on this reference structure in this task; just its input coordinates are stored for subsequent RMS comparisons.

Like the first one, the subsequent `DOCK` tasks here are also very similar to those in the previous example. The differences are the increase in `maxref` to the number recommended for a multiple-ligand job; the presence of the

[2] Actually, `external file` would work with that specific example, because there is only one input ligand structure. But it doesn't work in general.

`confgen` subtask in the rough-scoring task, which invokes `Impact`'s internal conformation generator; and the keyword `flex` in the `minimize` subtask, which enables ligand torsional flexibility during the grid-energy minimization. The execution of the task is changed by `confgen`, however, in that for each ligand structure read in, Glide loops over the conformations it generates. The specifications appearing in this `confgen` subtask have the following meanings:

`name lig` Generate conformations for the indicated species.

`ecut` Reject any conformation whose internal energy (torsional and 1-4 vdW terms only) is more than the specified amount (in kcal/mol) higher than that of the best (lowest-energy) conformation generated.

Other than the implicit loops over conformations generated by `confgen`, the main differences in the Glide procedure between this example and the previous one come from the nature of the input (ligand) structure file and the `CREATE` tasks that read it, and more important, from the DICE loop itself, and other control structures.

`build primary check`

Before storing the structure (and other actions normally invoked by `build primary` in a `CREATE` task), check whether it is the same molecule as the one previously read. For this purpose, two structures are considered to be the same molecule if they contain the same atom types (to the extent that atom type is encoded in the file), with the same connectivity, listed in the same order. If they do, Impact does not need to repeat the atomtyping procedure, or to reset other parameters. (Note: if there were no reference ligand, this would be the first structure read into the ligand species, so `build primary check` and the subsequent parsing of `'buildcheck'` would not be needed here. They would still be needed inside the loop, as described below.)

The result of `build primary check` is encoded in the value of the DICE variable `'buildcheck'`. The possible values are:

`1` Structures are the same molecule

`2` Structures are different molecules

`-1` End of file (no "next structure" to read)

`-2` Error reading next structure

`IF 'buildcheck' LT 0`

If we hit end of file or error on reading the first ligand to be docked, we must exit the program.

The `PUT` and `SHOW` commands here are simply to provide informative output. Note that `SHOW` writes only to the "main output" file

(`1ets_example_mult.out` as specified in the `write file` command at the top), not to Standard Output (or the `.log` file to which it is redirected).

`GOTO ABORT`
: Jump to the label `:ABORT`, which is at the end of the command file.

`gotostruct 'startlig'`
: As noted above, `many.mol` is a multi-structure file in MDL's SD format. (Analogous syntax, with `read maestro file`, would be used to read such a file in Schrödinger's Maestro format.)[3] The keyword-value pair `gotostruct` *n* calls for reading from the *n*th structure in the file, where in this case *n* is the value of the DICE variable `'startlig'`, which we set to 1 at the top of this input file. Thus if we wanted to start at ligand 3001, the command at the top would be `PUT 3001 INTO 'startlig`.

`PUT 'startlig' INTO 'i'`
: Initialize the loop index.

`WHILE ('endlig' LT 1 OR 'i' LE 'endlig')`
: The loop control. If `'endlig'` is less than 1 (as it is set at the top), this is nominally an infinite loop. Fortunately, DICE provides a way of breaking out of such a loop, which we will do in case of end of file or unrecoverable error (see `GOTO BREAK` below). If `'endlig'` were 1 or greater, it would set a limit on the number of times through the loop (and thus the number of ligand structures to process), even if that meant exiting before end of file. Thus to run only through ligand 1000 (if there are that many), change the command at the top to `PUT 1000 INTO 'endlig'`.

`nextstruct`
: Read the next structure in the file.

`IF 'buildcheck' LT 0`
: This is the crucial control structure. We need to break out of the loop if we have encountered the end of the file or an error. The `PUT` and `SHOW` commands are as above (except for details of the messages), but the target of the `GOTO` is not.

`GOTO BREAK`
: Jump to the label `:BREAK`, which is outside the loop.

---

[3] For PDB format, Glide reads single-structure files, one per ligand (or input conformation, if `confgen` is not used). In this case, the Impact input file would have to include commands for storing the names of these files in a list, and the `CREATE` task in the loop would read the file whose name is the element of this list given by the loop index.

`PUT 'i' + 1 INTO 'i'`
Increment the loop index.

`ENDWHILE` End of the loop.

The final output of this job consists of the structure file `1ets_example_mult_pv.mae`, and the report file `1ets_example_mult.rept`, which follows. In the actual files on disk, all the columns are one one long row, to enable you to load them into a spreadsheet. They are printed here in separate sections for space reasons.

```
REPORT OF BEST 5 POSES

 The receptor and sorted ligand structures written to the file
 1ets_example_mult_pv.mae  for use in the Pose Viewer

Rank    Title     Lig# Conf# Pose# Score GScore E(Cvdw) Eintern Emodel
==== ============ ==== ===== ===== ===== ====== ======= ======= ======
   1 Lorazepam       5     2   112 -6.47  -6.47   -31.9     0.6  -45.3
   2 indomethacin    4     4    84 -6.24  -6.24   -35.0     8.5  -47.2
   3 Atropine        1     3    16 -5.42  -5.42   -38.8     2.1  -57.1
   4 Ibuprofen       3    24   151 -5.37  -5.37   -27.3     1.8  -42.2
   5 Diflucan        2   340    24 -3.61  -3.61   -34.4     4.9  -42.3

                  Ehbond Emetal Eclash E(Coul) E(vdW)  RMSD
                  ====== ====== ====== ======= ====== ======
                    -1.9    0.0    0.0   -2.5   -29.3   --
                    -1.9    0.0    0.0   -6.5   -28.5   --
                    -1.4    0.0    0.0   -9.6   -29.1 61.597
                    -1.5    0.0    0.0   -4.9   -22.4   --
                    -1.1    0.0    0.0   -5.3   -29.1   --
GlideScore (GScore) is the sum of a constant = -1.0, plus other
contributions including the following:

   EHbond:  Hydrogen-bonding term
   Emetal:  Metal-binding term
   Eclash:  Penalty for steric clashes

(GScore = 10000.0 indicates that a given ligand pose failed one
or more criteria for computing GScore.  Depending on which ones
it failed, the components of GScore may not be valid either.)

ECvdW is the non-bonded interaction energy (Coulomb plus
van der Waals) between the ligand and the receptor.
Emodel is a specific combination of GScore, ECvdW, and Eint,
which is the internal torsional energy of the ligand conformer.
```

As requested with `maxperlig 1`, this file contains information on one structure per ligand. For comparison of different ligands, the structures are sorted in order of increasing GlideScore (`GScore`), with the "best" ligand at the top. In choosing the best pose (or the best `maxperlig` poses) within the set of final structures for a single ligand, however, Glide uses the `Emodel` score rather than GlideScore. `Emodel` is a weighted average of the GlideScore function and the Coulomb+vdW interaction energy (ECvdW) for a given pose, and is better suited than GlideScore for comparing poses of a single ligand.

For each pose, the report file lists its rank in GlideScore order, the ligand "title" taken from the input structure file, and the ligand number in the order the ligands were read in. (This includes any skipped ligands. For instance, if ligand #5, Lorazepam, were not processed for some reason, but processing of other ligands continued after it, then progesterone would still be listed as ligand #6.) It also gives conformation and pose numbers according to Glide's internal ordering, which are useful for distinguishing different structures of the same ligand (when `maxperlig > 1`). The subsequent columns include GlideScore, Emodel, various components of these, and if a reference structure was specified and the first ligand (in the order they were read in) is the same molecule as the reference, the heavy-atom RMS deviation (in Angstroms) of poses of that ligand from the reference structure. (The RMSD here includes the effects of translation and rigid rotation of the ligand, not just conformational differences. The high RMSD value in this case occurs because the reference ligand in this case was the input structure of the first docked ligand, which in fact is not a cocrystallized ligand for this receptor.) For other molecules (or if there was no reference structure), `--` appears in the RMSD column. The "Score" column in the above table is the same as GlideScore because by default, Glide ranks poses according to this scoring function. By specifying `by energy` in the `report setup` command, or by using the `glide_sort` post-processing script with appropriate flags, the user may choose to sort on some other score such as ECvdW (`by energy`), or some custom combination of various terms in the table (`glide_sort`). The "Score" column will always contain the value of the function by which the poses are ranked. If the keyword-value pair `verbosity 2` (or greater) appears in a `parameter` subtask before (or in the same `DOCK` task as) the `report write` command, the report file shows the ligand center coordinates and Euler angles of each pose, instead of some of the score components.

GlideScore values of 10000.0 indicate that GlideScore was in fact not calculated for a given pose. This occurs when the pose fails one (or more) of the criteria specified in the `scoring` subtask.

3.5.5 Example 4: Scoring in Place

In addition to searching for the best conformation and pose of one or more ligands, Glide can also evaluate its scoring functions on an input structure. To request this *scoring in place* feature, use the keyword `singlep` (for "single-point" energy or scoring) in the `ligand` subtask of a `DOCK` task after the setup. If this appears in a loop, scoring in place will be done for each input structure read in the loop. Note in the following input file that the `DOCK` tasks for rough-score screening and energy minimization are combined into one; but no screening or minimization actually takes place. As noted above, the `external file` keywords cannot be used in the `report setup` subtask for such a job. Glide does not currently report an error if they are used (because they may occur in a separate `DOCK` task from the `singlep` keyword), but the job will not run correctly if they are present.

```
write file "1ets_single_inplace.out" -
      title "1ets_single_inplace" *

PUT 0 INTO 'buildcheck'
PUT 1 INTO 'startlig'
PUT 0 INTO 'endlig'
PUT -1 INTO 'strucseq'

DOCK
  smooth anneal 2
  ligand multiple maxat 100 maxrot 15 -
   ligvdwscale factor 1.000000 ccut 0.150000
  receptor rdiel readf -
"1ets_single_grid"
  screen readscreen -
"1ets_single_grid.save" -
   greedy readgreed -
"1ets_single_grid_greedy.save" -
   maxkeep 1000 scorecut 100.000000
  parameter setup save maxconf 1
  final glidescore
  report setup by glidescore nreport 500 -
   maxperlig 1 rmspose 0.500000 delpose 1.300000
  run
QUIT

PUT 0 INTO 'strucseq'
CREATE
  build primary name lig type auto read maestro file -
"1ets_single_inplace.mae" -
   tag LIG_ gotostruct 1
  build types name lig
QUIT

DOCK
  smooth anneal 2
  ligand name lig singlep
  screen noscore refine maxref 100
  parameter save
  final glidescore read -
"1ets_single_grid.csc"
  minimize itmax 100 dielco 2.000000
 scoring ecvdw -25.000000 hbfilt -0.700000 metalfilt 0.000000 -
 hbpenal 3.000000
  report collect -
   rmspose 0.500000 delpose 1.300000
```

```
  run
QUIT

DOCK
  parameter clean final
  report -
   rmspose 0.500000 delpose 1.300000 write filename -
"1ets_single_inplace"
  run
QUIT

END
```

The output of a score-in-place job is written to a `.scor` file, in this case `1ets_single_inplace.scor`. This file gives the components of GlideScore and ECvdW for each input ligand (in this case only one). There is no structural output file (like the `_pv.mae` files in previous examples), because the structure is the same as in the input file.

```
----------------------------------------------------------------------------
Lig # Title GScore  HBond Metal  Lipo  RotB Clash BuryP   ECvdW  ECoul   EvdW
    1       -11.40  -4.55  0.00 -6.58  0.73  0.00  0.00  -70.01 -19.98 -50.03
----------------------------------------------------------------------------


 GlideScore (GScore) is the sum of a constant = -1.0, plus the
 following contributions:

    HBond:  Hydrogen-bonding term
    Metal:  Metal-binding term
    Lipo:   Lipophilic contact term
    RotB:   Penalty for freezing rotatable bonds
    Clash:  Penalty for steric clashes
    BuryP:  Penalty for buried polar groups

 (GScore = 10000.0 indicates that a given ligand pose failed one
 or more criteria for computing GScore.  Depending on which ones
 it failed, the components of GScore may not be valid either.)

 ECvdW is the non-bonded interaction energy (Coulomb plus
 van der Waals) between the ligand and the receptor.
```

3.5.6 Example 5: Glide Constraints

Glide constraints are requirements that docked ligands have specific interactions with the receptor. During grid generation, you can define up to ten constraints in the receptor, each of which may be a polar hydrogen atom, hydrogen-bond acceptor, or metal ion (*atom-based constraint*); a hydrophobic region on and near the receptor surface (*hydrophobic constraint*); or the spherical region within a specified distance of a specified point (*positional constraint*). For atom-based constraints, if you specify a receptor atom that is part of a functional group, and has a structural symmetry with one or more other atoms of the same chemical type in the group, then Glide will automatically include the symmetry-related atoms as part of the same constraint specification, and will consider a ligand interaction with any one of them as satisfying the constraint.

During ligand docking, you can specify that ligand poses must have appropriate atoms in appropriate positions relative to up to four of these receptor constraint sites, in order to be considered for docking. The categories of ligand atoms that qualify to satisfy each constraint are specified by SMARTS patterns in a *feature file*, which allows both restriction within and flexibility beyond the atom types normally considered as participating in hydrogen bonding, metal ligation, etc. For each hydrophobic constraint that you choose to enforce, you can specify the minimum number of ligand hydrophobic heavy atoms (default 1) that must lie in the corresponding hydrophobic region around the receptor in order to satisfy the constraint.

Because Glide incorporates any constraint specifications in several of its hierarchical filters (and incurs little additional computational cost in doing so), using constraints can accelerate docking calculations. This occurs because large regions of pose space can be quickly eliminated (as well as entire ligands that don't have the right kind of atoms to satisfy the constraints), beyond what a given Glide filter would eliminate without the constraints. In addition, by eliminating "false positive" ligands or poses, constraints can improve enrichment factors in database screening. And by restricting the allowed binding modes, judiciously chosen constraints may also improve docking accuracy.

As the following two examples demonstrate, you must specify constraints in the `receptor` subtask of the initial `DOCK` task, in both the grid generation and ligand docking jobs. The grid generation job needs to know which receptor atoms or regions you want to require ligand atoms to interact with. In addition, because hydrophobic constraints are not associated with individual atoms, a grid generation job needs to read a file containing a description of the hydrophobic regions (a list of the grid cells included in each region) that define such constraints. The name of this file must be supplied explicitly in the main input file; the Maestro interface calls the file *base*.phob, where *base* is the "base name" specified with the `readf` and `writef` keywords. For a positional constraint, you must specify the Cartesian coordinates of a position, and the radius of the sphere around that position in which one or more ligand atoms must lie to satisfy the constraint. The grid generation job extracts or calculates information about the receptor atoms that define hydrogen-bond and metal constraints (such as their types and locations) that the docking job will use in enforcing the constraints, and writes the information to a file (default name *base*.cons), along with the grid cell lists it gets from the *base*.phob file for hydrophobic constraints, and those it calculates from the sphere centers and radii for positional constraints. The docking job needs to know that it must read the constraint definition file that the grid generation job wrote, and which of the constraints defined therein it must enforce.

```
DOCK
  smooth anneal 2
  receptor rdiel name recep -
   constraints ncons 4 nphobic 2 file "1kv2_grid.phob" -
    consatom 1065 -
    consatom 2531 -
   writef "1kv2_grid" writerdiel -
   protvdwscale factor 1.000000 ccut 0.250000 -
   box center read xcent 4.700036 ycent 15.307946 zcent 33.614067 -
   boxxrange 29.622122 boxyrange 29.622122 boxzrange 29.622122 -
   actxrange 29.622122 actyrange 29.622122 actzrange 29.622122
  screen greedy -
   box center read xcent 4.700036 ycent 15.307946 zcent 33.614067 -
   ligxrange 10.000000 ligyrange 10.000000 ligzrange 10.000000 -
   writescreen "1kv2_grid.save" -
   writegreed "1kv2_grid_greedy.save" -
   maxkeep 5000 scorecut 100.000000
  parameter clean
  final glidescore
  run
QUIT
```

In this grid generation job, we define four constraints (`ncons 4`) in the protein kinase P38 (Protein Data Bank entry 1KV2). Two of the constraints are hydrophobic (`nphobic 2`), and the hydrophobic regions of interest are in the file `1kv2_grid.phob`, which the Maestro interface wrote (based on a calculation of a hydrophobic surface for the protein, and user selection of desired grid cells) in setting up the job. In this case, the regions correspond to the locations of naphthalene and tert-butyl moieties of the cocrystallized ligand in the 1KV2 structure. The other two constraints (the number is not explicitly listed, but obviously equal to the difference between the `ncons` and `nphobic` values) are either hydrogen bonds or metal ions, in either case defined by single protein atoms (and symmetry-equivalent ones, if any). We list each of these atoms (`consatom`) by its atom index in the input structure. In this case, the atoms are the side-chain (carboxylate) oxygen(s) of residue `GLU 71` and the backbone (amide) hydrogen of `ASP 168`; the cocrystallized ligand in the 1KV2 structure makes hydrogen bonds to both of these atoms, though not all known active ligands do.

In a ligand docking job, you may specify up to four of the constraints defined in the previous gridgen job, for Glide to enforce when docking ligands. The listing of which constraints are eligible for enforcement, and the specification of how many of those eligible are required to be satisfied, are contained in the *feature file*, along with the specification for each listed constraint of SMARTS patterns that ligand atoms must match in order to satisfy that constraint.

In the excerpt shown below from a ligand docking job, the `receptor` subtask indicates that we want to apply constraints set up in a prior grid generation job. The feature file `1kv2_dock_1cons.feat` might list any number of the previously defined constraints (and SMARTS patterns to match ligand atoms that can satisfy them), but specify that only some smaller number of them is required to be satisfied. For instance, if it lists three constraints and specifies that one is required, then ligands and poses that satisfy any one of those three constraints may appear in the output. If the grid generation job defined ten constraints, then the feature file can in principle list all ten, but cannot specify a number greater than four as the satisfaction requirement. For a given set of grid files, different docking jobs will in general have different feature files associated with them.

The keywords `restcoef` and `restexp` give parameters of a *restraining potential* that Glide uses to enforce the constraints during grid-energy optimization. This potential is a Gaussian function of the distance r between a polar hydrogen and a hydrogen-bond acceptor, or a metal ion and its coordinating atom in the ligand, centered at the equilibrium distance for the given interaction:

$$V(r) = -A \exp\left[-b\,(r - r_0)^2\right]$$

where r_0 is the equilibrium distance, $1.85\mathring{A}$ for a hydrogen bond or $2.11\mathring{A}$ for a metal-ligand interaction. The default values for the coefficients A and b are those shown below for `restcoef` and `restexp`: $A = 30.0 kcal/mol$ and $b = 0.3\mathring{A}^{-2}$. These values of the parameters have yielded good results in our simulations, but we do not claim that they are the only reasonable values.

```
DOCK
...
  receptor rdiel readf -
"1kv2_grid" -
  constraints loosedock 2 featurefile -
"1kv2_dock_1cons.feat" -
    consname -
"1kv2_grid.cons" -
  restcoef 30.0 restexp 0.3
...
QUIT
```

3.5.7 Subtask Smooth

Request smoothing of energy functions used in constructing grids.

- `smooth [cwall` *val*`] [csoft` *val*`] [vsoft` *val*`]`
`[anneal [1|2]]`

`cwall, csoft`
: Smoothing parameters for Coulomb energy.

`vsoft`
: Smoothing parameter for Lennard-Jones energy.

`anneal`
: Controls minimization on smoothed and/or unsmoothed energy surface.

Both smoothing functions work by evaluating the standard energy functions for two atoms at an *effective distance* that is positive when the actual distance between the atoms is zero. For the Coulomb energy, the effective distance at an actual distance d is given by

```
ceff = sqrt[d * d + cwall * cwall * exp(  - (d * d) / csoft)],
```

and for the Lennard-Jones energy, by

```
veff = d + vwall * exp( - (d * d) / vsoft).
```

(Note that in each case, the *wall* parameter is the value of the effective radius at d = 0, and the *soft* parameter determines how rapidly the function reverts to its unsmoothed value as d increases, with a larger parameter giving a slower (or "softer") transition.)

Note that `vwall` is not user-specifiable. Instead, for the contribution of a given protein atom, Glide uses half of the Lennard-Jones σ parameter for that atom. The default values for the other parameters are `cwall` = 2.0 Å, and `csoft` = `vsoft` = 4.0 Å^2. All of the parameters must be positive numbers; if the user specifies any negative, all are ignored, a warning is issued, and smoothing is not performed. In addition, if the softness parameters are below certain lower bounds, the resulting smoothed potential will have a local maximum (for a repulsive potential) at some positive distance, and a spurious minimum rather than a maximum at zero distance. For Coulomb smoothing, the lower bound is `csoft` = `cwall` * `cwall`. For Lennard-Jones, since `vwall` varies with the protein atom type, we use a lower bound large enough to accommodate the largest $\sigma/2$ in paramstd.dat (3.358 Å for the Cs^+ ion, which gives a lower bound of `vsoft` = 2.075 Å^2). If the user specifies a softness lower than the applicable lower bound, a warning is issued and the parameter is reset to equal the lower bound.

With the smoothing functions, Glide offers the option of *annealing* during grid-energy minimization. This involves starting the minimization on the potential-energy surface defined by the smoothed functions, and gradually shifting to the unsmoothed functions. The advantage of this procedure is to allow exploration of more regions of ligand pose and conformational space early in the process (because the smoothed functions have lower barriers), while still ending at a minimum of the original grid potential rather than at a pose whose energy is made artificially low by smoothing. Specifying `smooth anneal 2` when calculating grids will result in both smoothed and unsmoothed functions being calculated (and saved to disk); the same specification in the task where minimization is done will result in annealing during minimization. `Smooth anneal 1` means calculate, save, and/or minimize on only the smoothed surface. To calculate or minimize on only the unsmoothed potentials, omit the `smooth` subtask entirely. We strongly recommend using `smooth anneal 2` in all cases.

3.5.8 Subtask Receptor

Specify receptor molecule(s) and active site.

- `receptor [writef` *writebase*`] [readf` *readbase*`] -`
  `[cdiel | rdiel |` nil`] -`
  `[writecdie | writerdie |` nil`] -`
  `[name` *spec* `[mole [`*mol* `| all]] -`
  `[constraints [ncons` *num_cons* `-`
  `[nphobic` *num_phob* `file` *fname*`] -`
  `[nposit` *num_posit* `(xpos` *val* `ypos` *val* `zpos` *val* `-`
  `rpos` *val* `constitle` *cons*`)` repeated *num_posit* times`] -`
  `(consatom` *num* `constitle` *cons*`) -`
  repeated (*num_cons* − *num_phob* − *num_posit*) times`] -`
  `[consname` *file*`] [restcoef` *val*`][restexp` *val*`] -`
  `[metalbind [charged | neutral | any]] -`
  `[featurefile` *fname* `[featverb` *num*`] | -`
  `nusecons` *num_ucons* `[nusephob` *num_uphob* `-`
  `(usephob` *num* `nfill` *num*`)` repeated *num_uphob* times`] -`
  `(usecons` *num*`)` repeated (*num_ucons* − *num_uphob*) times`] -`
  `[loosegrid` *num*`] [loosedock` *num*`] [finalonly]] -`
  `[bsize` *size*`] [nlev` *nlevels*`] -`
  `[(scut` *val*`)` repeated *nlevels*-1 times`] -`
  `[box center read xcent` *val* `ycent` *val* `zcent` *val* `-`
  `boxxr` *val* `boxyr` *val* `boxzr` *val* `-`
  `actxr` *val* `actyr` *val* `actzr` *val*`] -`
  `[active nsec` *num_sections* `-`
  `(fres` *num* `lres` *num*`)` repeated *num_sections* times `-`
  `[buffer` *val*`]] [readsurface` *file*`] [writesurface` *file*`]`

`writef`
`readf` Write/read energy grids (or *fields*) to/from disk files. `writef` writes adaptive grid structure information to *writebase*.grd, Coulomb potential (constant dielectric) to *writebase*_coul.fld, Coulomb potential (linear dielectric) to *writebase*_coul2.fld, and Lennard-Jones grids to *writebase*_vdw.fld. `readf` reads the files if they exist, and calculates the energy grids from scratch if they don't (and there is a receptor structure specified with the `name` keyword). At least one of `readf` and `writef` should always be specified. If both are specified, Impact reads whatever files are present, and calculates and writes those that aren't. (If *readbase* and *writebase* are different, Impact reads from the former and writes to the latter.) The files specified by `readf` should of course have previously been written as a result of a `writef` in a previous docking task.

`cdiel`
`rdiel`
: Specifies whether the Coulomb energy should be calculated assuming a constant dielectric (`cdiel`) or a dielectric linear in the interatomic distance (`rdiel`). If neither is specified, the default is to use the constant dielectric. If both `cdiel` and `rdiel` are specified, `rdiel` wins, i.e., the linear dielectric is used. We recommend `rdiel` (and `dielco` 2.0 in the `minimize` subtask), to account, however roughly, for solvent effects. Note that these keywords affect which grid file is read, not the original calculation and writing of the grids, which is controlled by `writecdie`/`writerdie`.

`writecdie`
`writerdie`
: Specifies whether Coulomb grids are written to disk for the constant (`writecdie`) or linear distance-dependent (`writerdie`) dielectric model. If neither is specified, both grids are written. (If both are specified, the one that comes last wins.) Because grid files are large and we recommend always using the linear dielectric, we also recommend using `writerdie` to save disk space.

`name`
`mole`
: Specifies the Impact species that includes the receptor molecule(s). If the species contains more than one molecule (apart from bound solvent), then the `mole` keyword is *required*, with either the name (*mol*) of a single molecule, or `all` to indicate all molecules in the species are included.

`constraints`
: Require ligand poses to make specified interactions with the receptor. As noted above (see Section 3.5.6 [Constraints (Docking)], page 97), the `constraints` keyword must appear in both grid generation and ligand docking jobs in order for constraints to be used. The appearance of the following keywords depends on the type of job.

  `ncons`
  : This keyword appears in grid generation jobs, and the value gives the total number of constraints (of all types combined) defined.

  `nphobic` *`num`* `file` *`fname`*
  : The value *num* gives the number of hydrophobic constraints defined in a grid generation job. The file *fname* contains lists of grid cells near the receptor that constitute the hydrophobic region for each such constraint.

`nposit`
`xpos`
`ypos`
`zpos`
`rpos` Specification of positional constraints, which are requirements that a ligand atom (whose desired chemical characteristics will be defined in the ligand docking job) occupy a specifed (generally small) region of space. The `nposit` value gives the number of such constraints, each of which is defined as a spherical region centered at the Cartesian coordinates given by (`xpos,ypos,zpos`), with radius `rpos`.

`consatom` For each atom-based (H-bond or metal) constraint defined in a grid generation job, this specification lists the index of the constraint atom (or one of a set of symmetry-equivalent atoms) in the input receptor structure file.

`constitle`
An ASCII label for each constraint. This is specified in the Glide input file for positional and atom-based constraints only. For hydrophobic constraints, Glide reads the title from the file listed with `nphobic`.

`consname` This may appear in either grid generation or ligand docking jobs. It specifies an alternative file name for writing or reading information about the receptor constraint atoms. The default is *writefbase*`.cons` or *readfbase*`.cons`, whichever is present in the same `receptor` subtask.

`restcoef`
`restexp` These may be specified in a ligand docking job. They are the depth (multiplicative coefficient, without the negative sign) and inverse square half-width (coefficient of the exponent) in a Gaussian potential function added to enforce the constraints during energy minimization. For the form of the potential, See Section 3.5.6 [Constraints (Docking)], page 97.

`featurefile`
Gives the name of a "feature" file, which specifies which constraints must be satisfied in a ligand docking job (including optional as well as required constraints, in one or more groups with a "number required" specified for each group). In addition to listing the constraints (by title and index in the

consname file that the grid generation job wrote), this file specifies what type of ligand atoms (those matching listed SMARTS patterns) will be accepted as matching each constraint.

`featverb` This number is a "verbosity" parameter used by the portions of Glide that read the feature file, and match ligand atoms against SMARTS patterns. The default, equivalent to `featverb 1`, prints very little information about the file and the matches, whereas `featverb 4` gives a complete listing of which constraints and patterns are listed in the file, and which patterns are matched by each ligand to be docked.

`loosegrid`

Increase the distance tolerance (by *num* Å) for considering grid cells to be appropriate locations for constraint-satisfying ligand atoms. Used in grid generation jobs only, not docking, and affects only atom-based (H-bond and metal) and positional constraints, not hydrophobic. (The qualifying grid cells for hydrophobic constraints are always considered to be those stored in the file associated with the `nphobic` keyword, no more and no less.) Default, or `loosegrid 0`, is to use the distance tolerances built into the algorithm for calculating the grid cells. Looser criteria may improve *pose recovery* (i.e., increase the likelihood of finding constraint-satisfying poses for active ligands), possibly at the cost of a decrease in computational speed.

`loosedock`

Increase the tolerances (by *num* Å) for distance matches used to determine constraint satisfaction during the rough-score stage of the Glide funnel. Used in ligand docking jobs only. Default, or `loosedock 0`, is to use the distance tolerances built into the constraint algorithm. Looser criteria may increase the likelihood of finding constraint-satisfying poses for active ligands, possibly at the cost of a decrease in computational speed.

`finalonly`

With this keyword, used in ligand docking jobs only, constraints are used only at the beginning of the docking run to filter out ligands that lack appropriate atoms to satisfy the constraints, and at the end to filter out final poses that do not satisfy them, not at any intermediate stages of the Glide funnel. The output poses from a `constraints finalonly` run, for each ligand that contains appropriate atoms, are the best (by `Emodel` score) constraint-satisfying poses of that ligand that would have emerged from an unconstrained docking job.

`metalbind [DEPRECATED]`
: This may appear in a ligand docking job. It specifies that any ligand atom that satisfies a constraint to bind a metal ion in the receptor must bear a nonzero formal charge (`charged`), must bear zero formal charge (`neutral`), or may be in any formal charge state (`any`). The default, and the recommended value, is `charged`.

`nusecons [DEPRECATED]`
: This and the following keywords may appear in a ligand docking job, to select constraints to enforce from among those defined in the `consname` file that the grid generation job wrote. The `nusecons` value gives the total number of constraints to enforce, of all types.

`nusephob [DEPRECATED]`
: This gives the total number of hydrophobic constraints to enforce.

`usephob [DEPRECATED]`
`nfill [DEPRECATED]`
: For each selected hydrophobic constraint, the `usephob` value gives its position in the `consname` file, and `nfill` the number of ligand hydrophobic heavy atoms that must be located in the corresponding hydrophobic region.

`usecons [DEPRECATED]`
: These values are the positions of the selected non-hydrophobic constraints in the `consname` file. Note that hydrophobic constraints are listed first in this file, so if there are two hydrophobic constraints, the “first” non-hydrophobic one is selected using `usecons 3`.

`bsize`
: The size of the finest grid spacing for the energy grids, in Angstroms. Default 0.4.

`nlev` Number of *levels* of the adaptive grid. At each successive level (farther from the receptor surface), the grid spacing is twice what it is at the previous level. Thus if the smallest grid spacing is *size*, then the largest is $2^{(nlevels-1)} * size$. Default *nlevels* = 4.

`scut` Distances from the receptor (the closest receptor surface point) at which the grid spacing changes. Thus

```
bsize 0.4  nlev 2  scut 1.0
```

means that the grid spacing is 0.4 Å for points closer than 1.0 Å from the receptor surface, and 0.8 Å farther away. If there is more than one `scut` value (i.e., if *nlevels* > 2), they must be given in *descending* order. The default (corresponding to `bsize 0.4 nlev 4`) is `scut 4.4 scut 2.8 scut 2.0`.

`box` Explicitly specify the rectangular box in which the energy grid is defined, rather than building it based on a specification of active site residues.

`center read`
Gives the three Cartesian coordinates of the center of the box, as the the numbers following `xcent`, `ycent`, and `zcent`. The keyword `read` is required here because another option is available with the `center` keyword in the `screen` subtask, and the same code is used to parse the `box` input in both subtasks.

`boxxr`
`boxyr`
`boxzr` The size of the grid box (in Angstroms) in the x, y, and z directions. That is, the x-coordinates of the grid points in the box range from approximately `xcent` − `boxxr`/2 to `xcent` + `boxxr`/2. This is approximate because extra space may be added to the ends of the box so that it contains a whole number of elementary cubes of the grid.

`actxr`
`actyr`
`actzr` Dimensions (Angstroms) of the box used to determine "active" residues whose surface is used in early rough-score filters. Surface points are calculated for all residues that have any atom in this box. In general this should be the same size as the grid box, but memory limitations in the surface-generation algorithm require a box no larger than 50 Å on a side.

`active` An alternative method of defining the dimensions of the grid and "active surface" boxes. Specifies which residues are to be considered the active site of the receptor. The grid box is computed using the largest and smallest x-, y-, and z-coordinates of atoms in these residues, and adding a distance in each direction (positive and negative) as specified with the `buffer` keyword. As when directly specifying `actxr`, etc., surface points are actually generated for all residues with any atom in the box, not just the ones specified here. The initial `active` residues are specified as *num_sections* ranges, each given by a `fres lres` pair. Each `fres` value must be greater than the previous `lres` (the first must be greater than zero), and each `lres` must be greater than or equal to the corresponding `fres` (with equality implying a range consisting of a single residue). The maximum value of *num_sections* is 100. (If you need more than that, consider filling in to combine several ranges into one.) If neither `active` nor `box` is present, then all residues of the receptor are considered to be in the active site, with a `buffer` of the default size, 11.0 Angstroms.

`nsec` Indicates that the active site residues are given by the following `fres` *num1* `lres` *num2* pairs, where each of the *num_sections* pairs indicates that all residues in the range *num1* through *num2*, inclusive, are part of the active site. (Note that such a “range” may consist of a single residue, as `fres 79 lres 79`.)

`buffer` Indicates that the box in which the grids are defined extends a distance *bufval* Angstroms beyond the minimal box that encloses the active site, in each of the positive and negative x, y, and z directions. Default is 11.0.

`readsurface`
`writesurface`
Read/write receptor surface points from/to the indicated file. The surface points are calculated from the positions and radii of receptor atoms in residues contained in the “active” box defined by either `actxr`, etc., or `active`, and are used in early filters in the rough-score screening step. The surface calculation is somewhat time-consuming, so it may be convenient to store the points for future use, particularly in runs where the energy grids are not being recalculated (which takes a much longer time) but the rough-score grids are (which is quite fast, so recalculating the surface can add significantly to it).

3.5.9 Subtask Ligand

Specify ligand molecule.

- `ligand keep`
- `ligand multiple maxat` *`nat`* `[maxrot` *`nbond`*`] -`

```
   [amideoff]
 • ligand name spec [mole mol] -
   [init [zero | rand [randopts] | read posespec] -
   [cminit [zero | box | lig | grid gridspec] -
   [reference] [noelec] [[stdrot | norot]] -
   [multiple maxat nat maxmol nmol] -
   [new]
```

`keep` Indicates that no new parameters or coordinates are to be read in for the ligand, but that there is still a ligand present. The docking calculation will not run correctly if there is no `ligand` subtask present, so `ligand keep` is required in invocations of the `DOCK` task that do not introduce a new ligand conformation, as in a pose refinement and energy minimization step after a rough-score screening task (possibly in a loop over externally generated conformers) for the same ligand. If the `keep` keyword appears in a `ligand` subtask, all other keywords in that subtask are ignored.

`multiple` The keyword `multiple` is used here for historical reasons. It should really be called `ligand size`, because it is necessary even in single-ligand jobs that contain a "setup" `DOCK` task that doesn't dock (or otherwise specify) any specific ligand. For such a single-ligand job, `maxat` and `maxrot` should give the number of atoms and rotatable bonds in that ligand. For multiple-ligand jobs, they give bounds on the size of ligands that will be considered, that is, input ligands with more atoms or rotatable bonds will be skipped. The defaults are `maxat 100 maxrot 35`, and the maximum allowed value for `maxat` is 200.

The `multiple` keyword must appear in the `ligand` subtask of the first `DOCK` task of an Impact input file.

`amideoff` In Glide standard precision (SP) and high throughput virtual screening (HTVS) jobs, the `amideoff` keyword indicates that amide bonds should not be considered rotatable. By default, they are rotatable.

In Glide extra precision (XP) jobs, the `amideoff` keyword instead applies a 3.5 kcal/mol penalty on cis-amide conformations and a maximum penalty of 6.0 kcal/mol for 90 degree twisted amide conformations, with interpolated penalties in between.

`name` The name of the species in which the ligand molecule is to be found.

`mole` The name of the ligand molecule within species *spec*. Note that Glide can only handle single molecules (as defined in the `create` task) as ligands, so if *spec* contains more than one molecule, `mole` *mol* is *required*.

`reference`

Specifies that the current ligand molecule (the one most recently read in to the specified species) is to be taken as the reference conformation for root-mean-square deviation (rmsd) calculations. Such calculations are only meaningful, and Glide only does them, for ligands that are the same molecule as the reference. Glide also issues a warning that rmsd calculations may not be meaningful for a multiple-ligand job, but the rmsds it does calculate should be correct. In general (and in jobs set up and/or launched from the Maestro user interface), no actual docking calculations are done in the `DOCK` task that specifies the reference ligand. It is of course possible to include the reference ligand in a subsequent `DOCK` task that actually does dock it.

`init`
`cminit`

Specify the initial pose of the ligand for energy minimization, if rough-score screening is not performed. The usual specification of these keywords (and the default) is `init zero cminit lig`. If rough-score screening is run, these keywords are ignored, because the initial poses for minimization are those that survive screening.

`init zero` Specifies that the ligand center should start at the origin of coordinates, unless displaced by `cminit`.

`init rand [cmrange` *val*`] [thetarange` *val*`] [phirange` *val*`] [psirange` *val*`] [seed` *num*`]`

Specifies a random starting pose. This is chosen in the ranges given with the keywords `cmrange`, `thetarange`, `phirange`, and `psirange`. That is, each Cartesian coordinate of the center position starts in the range (-`cmrange`/2) to (`cmrange`/2) Angstroms about the position specified by `cminit`; the Euler angle θ starts in the range 0 to (`thetarange`) degrees; ϕ starts in the range (-`phirange`/2) to (`phirange`/2), and similarly for ψ. `iseed` is a seed for the random number generator. The defaults are `cmrange 2.0 thetarange 30.0 phirange 60.0 psirange 60.0 iseed 137`.

`init read xcm` *val* `ycm` *val* `zcm` *val* `phi` *val* `theta` *val* `psi` *val*

Initializes the ligand to the specified pose (center coordinates in Ansgtroms, angles in degrees), again subject to modification by `cminit`.

`cminit zero`
: Specifies that the starting position of the ligand center should be at the origin, or unmodified from the position specified by `init`. Thus specifying `cminit zero` with `init zero` or `init rand` would indeed place the ligand at the origin of coordinates, or randomly in the specified range around it, which is unlikely to be useful. But `cminit zero` is the default with `init read`, in which case it leaves the ligand at the specified position.

`cminit lig`
: Starts the ligand at the position given in the input file. This is the default with `init zero` and `init rand`. In the latter case, the starting position is randomly displaced in the specified range about the input position.

`noelec`
: Turn off electrostatic interactions, by setting partial charges to zero for all atoms in the current ligand. This is reset for each ligand structure read in, so the `noelec` keyword must appear in the first `DOCK` subtask for each ligand, e.g. in the ligand loop. Note also that the final reported Coulomb energy for a ligand pose is a "scaled" energy that depends on formal charges as well as partial charges, and `noelec` does not zero the formal charges, so the output files (.rept and .mae) may report nonzero Coulomb energies even if `noelec` is set. But `noelec` does guarantee that no electrostatic interactions are included in the sampling and energy minimization steps, in which the final poses are produced.

`stdrot`
`norot`
: Control the starting orientation of the ligand. `stdrot` places the ligand in a standard orientation, with its *diameter* (the line segment connecting the two most widely separated ligand atoms) pointing along the z-axis. `norot` leaves the ligand in the orientation specified with the `init` keyword. With `init read ... cminit zero`, the ligand starts in the user-specified position and orientation, and the default is `norot` to leave it there. In all other cases, the default is `stdrot`. The Euler angles that define poses, in both phases of the docking calculation, are then defined relative to the standard orientation.

`new`
: Indicates that the current ligand molecule has a distinct structure (not just a different conformation) from the preceding one. This keyword is usually unnecessary, because the newness of the ligand is perceived automatically by `build primary check` in its `CREATE` task.

3.5.10 Subtask Parameter

Specify various parameters and flags.

- `parameter [verbosity num] [maxconf num] -`
  `[setup] [save] [clean]`

This subtask sets certain controls on the overall operation of the task.

`verbosity`
: Controls the amount of information printed to STDOUT and to the main output file. The default is `verbosity 1`, which should be sufficient for most users' purposes. Certain things are printed independent of the value of this parameter, including the summary (labeled `DOCKING RESULTS`) of the best-scoring poses (by various criteria) for each ligand and their scores. `verbosity 0` (or less, which is equivalent to 0), prints a bare minimum of additional information. Values higher than 2 or 3, and especially higher than 5, print information that's very unlikely to be useful to anyone other than developers and debuggers, and can result in extremely large output files. A given `verbosity` level remains in effect unless and until the `parameter` subtask of a subsequent `DOCK` task changes it.

`maxconf`
: The maximum number of ligand conformations to be processed in this job. This parameter sets the size of a dynamically allocated array, and attempting to read conformations beyond this number will result in an error.

`setup`
: Indicates that the current invocation of the `DOCK` task is only for the purpose of setting up arrays (including rough-score and energy grids) for use by subsequent invocations in the same Impact job (as in multi-conformation loops). Though there will in general be a `screen` subtask along with `parameter setup` to set parameters for the rough-score screening, no actual screening calculation on the ligand will actually be done at this point. (Nor will minimization, which there's no reason to specify at all in a task with `parameter setup`.)

`save`
`clean`
: Specify the disposition of various dynamically allocated arrays (including those that hold the rough-score grids, and the ligand and receptor coordinates copied from the main Impact arrays) at the end of the current invocation of the `DOCK` task. `save` means leave them in place for use by subsequent invocations of the task, `clean` means delete them, which means any subsequent invocations must build them again. If `setup` is specified, `save` is the default. (Indeed, `setup clean` doesn't make sense: set up the grids, don't use them, and then throw them away.) If neither `setup` nor `save` is specified, `clean` is the default. (But it doesn't

hurt to specify `save` or `clean`, where appropriate, even if it is the default.)

3.5.11 Subtask Confgen

Request internal generation of ligand conformers.

- `confgen -`
  `ecut` *val* `-`
  `[maxcore` *num*`] [corescale` *val*`] -`
  `[noringconf]`

This is the recommended method of incorporating ligand flexibility into Glide, especially in a multi-ligand job. As shown in the examples above, the command sequence in an Impact input file should be different depending on whether there is one input structure per ligand, with `confgen` specified, or multiple structures assumed to be externally generated conformers for each ligand. In the latter case, we recommend a loop over `screen` subtasks, to run the first stages of rough-score screening (through greedy score evaluation) on all of the conformers of a given ligand, before running pose refinement, grid-energy optimization, and final (`GlideScore`) scoring on all poses that pass the first stages for that ligand. With `confgen`, by contrast, the loop over the internally generated conformations is specified by a single `screen` subtask, so the subsequent steps should ensue immediately.

By default, `confgen` generates alternative ring conformations for five and six membered non-aromatic rings. To turn off this procedure, use the `noringconf` keyword. For six membered rings, the alternative chair conformation is generated if the equatorial–axial conformational change of the substituents is empirically not too energetically costly. The five membered rings currently treated are sugar rings and five membered rings with N and/or S atoms. The alternative sugar ring conformation generated from the input consists of the energetically preferred pseudorotation. Five membered rings with N or S atoms have a second ring conformation generated by rotation of the out-of-plane corner.

`ecut` This parameter is the energy cutoff used in the gas phase conformation generation. Conformations with an energy above ecut relative to the lowest energy conformation are not considered. Note that the energy scale here is with respect to the model torsion/1-4 vdW confgen potential and not a full force field potential.

`maxcore` The `maxcore` parameter allows the user to define a maximum number of core conformations to be generated. The default behavior is to use a functional form depending on the number of rotatable bonds. The maxcore parameter could be used to make a very approximate rough quick pass at docking. See Section 2 of the *Glide Technical Notes* for details.

`corescale`

`Corescale` is a fractional value to scale down the default number of core conformations kept. See Section 2 of the *Glide Technical Notes*.

`noringconf`

The `noringconf` keyword disables ring conformation generation.

3.5.12 Subtask Similarity

Request Glide similarity scoring.

Similarity scoring entails assigning a number to each ligand based on its similarity to one or more of a set of selected *active* ligands, and optionally (*weighted* or *calibrated* similarity) also its dissimilarity to a set of selected *inactive* or *decoy* ligands. Unlike most quantities calculated in Glide, similarity is a *ligand-based* rather than a *structure-based* property. That is, the similarity between two molecules depends only on the types and connectivity of the atoms in those molecules, and not on any details of their coordinates or conformations, or on anything to do with the receptor. Glide thus performs similarity scoring, if requested, just once per ligand. It therefore adds negligible overhead to a typical Glide database screening job, and may even speed it up because some ligands can be immediately rejected. Weight calibration adds a small amount of time to a grid generation job.

The similarity of one ligand (in the test set) to another (in the training set) is evaluated by comparing the set of all atom pairs in the test ligand to the set of all atom pairs in the training ligand. Within each ligand, each atom pair is characterized by the element types, bond orders, and formal charges of the two atoms, and the number of bonds in the shortest path connecting them. The similarity is normalized to a number between 0 (the two molecules have no atom pairs in common) and 1, in which case the molecules have all the same atoms with the same connectivities, and are thus either identical or stereoisomers of each other. For weighted similarity, each atom pair in the training set (actives) is assigned a weight factor, which is higher if the given pair appears more often in the actives and lower if it appears in the inactives.

To use similarity scoring, put `simil` subtasks in the grid generation (only for calibration in weighted similarity) and ligand docking tasks, following the meta-examples below. Glide will then adjust the Glidescore of each docked ligand pose by adding a term that depends on the maximum similarity of that ligand to any of the actives.

- `simil weight actives [maestro | sd] afile` *`fname`* `-`
  `inactives [maestro | sd] ifile` *`fname`* `-`
  `percent` *`val`* `wfile` *`fname`* `[allprint | noprint]`
- `simil actives [maestro | sd] afile` *`fname`* `-`
  `[wfile` *`fname`*`] -`
  `[penalty` *`val`*`] [lowsim` *`val`*`] -`
  `[highsim` *`val`*`] [reject` *`val`*`] [allprint | noprint]`

- `simil name` *`spec`*

The calibration step for weighted similarity is specified by the `weight` keyword. This step should be performed in a grid generation job, and all of the following keywords are *required*.

`actives [maestro | sd] afile` *`fname`*
: Specifies that the active ligands in the training set are in file *fname*, which may be in either Maestro or MDL SD format. Note that at least two active ligands are required for calibration.

`inactives [maestro | sd] ifile` *`fname`*
: Specifies the file containing the decoy ligands.

`percent` *`val`*
: Roughly specifies the percentage of the inactives to be included in weight calibration. Rather than using preselected ligands from the `inactives` file, each molecule in the file has *val* percent probability of being used in weight calibration. Thus, the number of ligands selected may not exactly match the user's `percent` input. Note that at least one decoy compound is required for calibration, and that a weight calibration job will exit if it has not read in at least two active ligand structures, and chosen at least one inactive. For best results, we recommend making the `inactives` file large enough, and the `percent` probability high enough, to use about 5 to 15 times as many decoys as actives. For instance, if the `actives` file contains 10 ligands and the `inactives` file contains 1000, use a `percent` value between 5.0 and 15.0. Weight calibration may produce a message stating that it did not converge (more likely the higher the ratio of inactives to actives), but this is not a problem: a valid weights file is produced in any case, and contains the "best" weights obtained with the given structures.

`wfile` *`fname`*
: Write the weights to the file *fname*. This will be a text file, with each line containing a symbolic representation of an atom pair, followed by the calibrated weight for that pair.

`allprint`
: The `allprint` keyword enables maximum printing of output from the similarity machinery including output of the similarity of each docked ligand to each probe molecule. Default printing outputs only the maximum simiarity of the docked ligand to any probe molecule.

`noprint`
: The keyword `noprint` disables printing of output from the similarity machinery.

To use similarity scoring in a ligand docking job, all that's required is the specification of an `actives` file. The `simil` subtask should appear in the first (setup) `DOCK` task of the job.

`actives [maestro | sd] afile` *`fname`*
: Adjust the Glidescore values for poses of each ligand according to the similarity of that ligand to those in file *fname*. This need not be the same file as was used for weight calibration in the previous grid generation job, even if the weights generated in that job are to be used.

`wfile` *`fname`*
: Use calibrated similarity, with weights taken from file *fname*.

`penalty` *`val`* `lowsim` *`val`* `highsim` *`val`*
: Parameters for adjusting Glidescores. If the maximum similarity between a given docked ligand and any ligand in the `actives` file is less than `lowsim`, add the full `penalty` value to the Glidescores of all docked poses of that ligand. If the maximum similarity is greater than `highsim`, do not adjust the Glidescores for that ligand. If the maximum similarity is between those two values, the Glidescore adjustment is determined by a linear ramp between the maximum `penalty` value and zero. Note that while `lowsim` must be less than or equal to `highsim`, there are no other restrictions on their values; in particular, they need not be between 0.0 and 1.0, even though all similarity scores will be in that interval. Choosing `lowsim` less than zero, for instance, simply means that the maximum `penalty` value will never be applied to any ligand. Also, `penalty` may be negative, in order to reward ligands that are *not* similar to any of the actives (to promote diversity, for instance). The defaults are `penalty 6.0 lowsim 0.3 highsim 0.7`.

`reject` *`val`*
: Skip any ligand whose maximum similarity to any active ligand is less than *val*. Must be between 0.0 (accept all ligands) and 1.0 (skip all ligands that are not identical to or stereoisomers of one of the actives). Default is `reject 0.0`.

The third form of the `simil` command, `simil name` *`spec`*, should appear in the `DOCK` task for each ligand. (The first for that ligand, with `ligand name` *`spec`* rather than `ligand keep`.) It simply indicates that similarity scoring is to be applied to species *spec* (the current ligand), using the `actives` file (and weights, if any) read in the initial (setup) `DOCK` task.

3.5.13 Subtask Screen

Request screening phase of docking calculation.

- `screen noscore -`
  `[refine [refstep` *num*`] [maxref` *num*`] [refgreedy]]`
- `screen [scbsize` *val*`] [skipb` *num*`] -`
  `[maxkeep` *num*`] [scorecut` *val*`] -`
  `[readscreen` *fname*`] [writescreen` *fname*`] -`
  `[box center -`
  `  [lig | read xcent` *val* `ycent` *val* `zcent` *val*`] -`
  `  [boxxr` *val* `boxyr` *val* `boxzr` *val*`] -`
  `  [ligxr` *val* `ligyr` *val* `ligzr` *val*`]] -`
  `[readcmsite` *fname*`] [writecmsite` *fname*`] -`
  `[greedy [fraction` *weight*`] [readgreed` *fname*`] -`
  `                                 [writegreed` *fname*`]] -`
  `[refine [refstep` *num*`] [maxref` *num*`] [refgreedy]]`

`noscore`
: Do not perform rough-score calculations or screening on the current ligand. This keyword is needed when the `refine` step must be performed after a loop (either in DICE or internally) has already done screening on multiple (internally or externally generated) conformations. It is probably not useful otherwise.

`scbsize`
: The grid spacing, in Angstroms, of the rough-score grid. Default is `scbsize 1.0`.

`skipb` *n*
: Use only every *n*'th grid point in each direction as a possible site for the ligand center. Thus `skipb 2`, the default uses one-eighth of all grid points.

`maxkeep`
: Maximum number of poses to pass to the grid energy calculation. Default is `maxkeep 1`, but it's generally not useful to leave it at that. In our tests, we have found that a few hundred poses, over multiple conformations, are usually enough to find one or more good docked poses, at least if *greedy scoring* and *pose refinement* are employed.

`scorecut`
: Rough-score cutoff for keeping poses. When accumulating poses to pass to the grid energy calculations (after they have passed all other screening tests), a given pose survives if its rough score is within `scorecut` of the best pose accumulated so far. Default is `scorecut 100.0`.

`readscreen`
`writescreen`
: Read/write the rough-score grids (and possibly other information: see `readcmsite` below) from/to the indicated file. The file specified in a `readscreen` should have been written as the result of a `writescreen` in a previous run with the same receptor.

`writecmsite`
: Write to disk information about possible grid sites for the ligand center, for those sites that pass an initial (ligand-independent)

filter. This is generally a much smaller set than the entire box where the rough-score grid is defined, so Glide calculates it once for a given receptor and store the list on disk for subsequent use with different ligands. If `writecmsite` is not specified, this information is appended to the file specified in `writescreen`. Different `box` specifications, or different `skipb` specifications, result in different lists of sites, so we provide the option of writing these to separate files, without repeating the much larger rough-score grids in the `writescreen` file, which are independent of `skipb`.

`box` Specifies the rectangular box where the rough-score function is defined (*enclosing box*), and/or narrower limits on the position of the ligand center (*bounding box*). Default for the enclosing box is that specified in the `receptor` subtask for the energy grids, either by the `active` and `buffer` specifications or by a `box` specification in that subtask. The `box center` and `boxxr` specifications are as in the `receptor` subtask, with the additional option `box center lig` to put the center of the box at the coordinates of the ligand center in the input file. If the input is a known co-crystallized complex, `box center lig` biases the calculation in favor of the known correct answer, and should not be used except for testing. The parameters `ligxr`, `ligyr`, and `ligzr` give the size of the search space for positions of the ligand center. That is, the ligand center may be placed at grid points with x-coordinates between approximately `xcent`$-$`ligxr`/2 and `xcent`$+$`ligxr`/2, and similarly for y and z. In general, the bounding box should be much smaller than the enclosing box, because grid points near the edges of the enclosing box will have many ligand atoms outside the box, and thus be rejected as possible ligand center positions. The Maestro user interface determines the size of the enclosing box (purple outline on the Maestro display) by adding to the user-specified size of the bounding box (green) a buffer big enough to fit ligands up to a user-specified size, when the ligand center is at the edges or corners of the bounding box. The limits on the ligand center position are incorporated in the grid file written by `writescreen` (or `writecmsite`), so `box ... ligxr ...` is unnecessary when reading existing grid files from disk `readscreen`.

`greedy` Specifies the greedy scoring algorithm, as described above. `fraction` *weight* specifies that the combination to use is *weight* times the score at the best surrounding grid point, plus (1 − *weight*) times the original score at the central point. The default is `fraction 0.33`, and acceptable values are between 0 and 1. `readgreed` and `writegreed` specify reading/writing the *greedy grid* (the linear combination at each point, not the best surrounding score) from/to the indicated file.

`refine` Specifies the *pose refinement* step of the screening algorithm. This involves moving each pose from its original central grid point to a 3 x 3 cube of surrounding grid points. Each point is either zero or `refstep` grid points away from the central one in each of the positive or negative x, y, and z directions, where `refstep` must be smaller than `skipb` (so as not to get to a position already tested for the ligand center), and the default is `refstep 1`. The algorithm evaluates the score of the pose centered at each of the 27 grid points (in the same orientation as the original), and chooses the best (lowest) score to pass to energy minimization. The refinement step improves the scores of poses that are close to favorable ones that were initially skipped because of the `skipb` specification, and thus often decreases the number of poses that need to be passed to energy minimization in order to assure that good ones are included. To decrease the number actually passed, specify `maxref` less than `maxkeep`. Since pose refinement and greedy scoring are both intended to find good scores that would otherwise be missed because of `skipb`, the default is for refinement to evaluate the 27 poses using the *original* (non-greedy) score, even if the rest of the screening process used the greedy score. The keyword `refgreed` specifies that refinement should use greedy scoring (if the greedy-score grid is available), but we have not found any advantage in doing this, and it runs the risk of increasing the rate of false positives.

3.5.14 Subtask Minimize

Request energy minimization phase of docking calculation.

- `minimize flex ftol` *val* `dielco` *val* `-`
  `[ maxhard` *val* `] [ maxsoft [val] [ sampling` *val* `] -`
  `[ highacc [ ncycle` *val* `] ]`

`flex` Indicates that ligand torsional angles are to be varied during minimization.

`ftol` Convergence criterion for the minimizer, expressed as a bound on the relative energy change at the last iteration. The default is `ftol 1.0e-4`.

`dielco` The dielectric constant, or coefficient of the interatomic distance in the distance-dependent dielectric function, to be used in calculating electrostatic energies. Thus if `rdiel` is specified in the `receptor` subtask, and `dielco 2.0` is specified here, the dielectric used is $2r$. The default is `dielco 1.0`, but we recommend (and the Maestro interface writes) `dielco 2.0`, along with `rdiel`, to weaken long-range electrostatic interactions.

`sampling` The value of this keyword controls the sampling of ligand torsions, performed after minimization and before final scoring. Lower values indicate more sampling. The default, `sampling -1`, does the most sampling, and `sampling 10` does no post-minimization sampling. In general, more sampling results in better-docked and better-scoring poses, at the cost of increased computation time.

`maxhard` The maximum number of minimization iterations on the hard Coulomb-vdW surface, default is 50.

`maxsoft` The maximum number of minimization iterations on the soft Coulomb-vdW surface, default is 100.

`highacc` This keyword activates Glide's *extra precision* mode, it directly corresponds to choosing "Extra Precision" in the Maestro Glide panel "Choose Docking Mode" pull-down selector.

`ncycle` *val*

This keyword is only available when `highacc` is also used, and sets the number of times the ligands are *recycled* through the docking process. This additional effort greatly improves Glide's ability to sample all the docking positions of the ligand in the receptor grid. The default value is 5.

3.5.15 Subtask Final

Specify final scoring function.

- `final [glidescore|noglidescore] [read` *fname*`]`

The `final` subtask specifies the scoring function to be used for final evaluation of the docking affinity of ligand poses. The recommended scoring function is Schrödinger's proprietary GlideScore (tm). `final glidescore` should appear in the setup `DOCK` task, and in cases where receptor information is to be read from disk, the keyword-value pair `read` *fname* should appear in the `DOCK` tasks that do the scoring, to indicate the file that contains receptor information needed for calculating GlideScore. In general, the name of this file will be *gridjob*`.csc`, where *gridjob* is the name of the job in which receptor grids were created.

3.5.16 Subtask Scoring

Filters and parameters for final scoring.

```
scoring ecvdw val hbfilt val metalfilt val -
 hbpenal val
```

The `scoring` subtask is useful for filtering out ligands, structures, or poses that might be assigned favorable GlideScore values, but are unacceptable for other reasons. The filters consist of maximum allowed values for the Coulomb plus van der Waals interaction energy calculated by grid interpolation (`ecvdw`), or the hydrogen-bonding (`hbfilt`) or metal-binding (`metalfilt`) terms in GlideScore. Poses that fail these filters are either skipped or assigned specific unfavorable GlideScore values such as 10000.0. Alternatively, the user may specify undemanding values (such as 0.0) for the filters in the Glide run, and impose more stringent filters in postprocessing, by running the `glide_sort` script, with the filter values among its arguments, on Glide's output structure files. This script allows not only filtering with a variety of criteria, but also re-sorting according to user-specified scoring criteria, without rerunning the Glide job.

The `hbpenal` parameter is not a filter, but rather the coefficient (default 3.0) of a term in GlideScore that penalizes poses in which potential hydrogen-bonding atoms are buried next to non-polar atoms in the ligand-receptor interface.

3.5.17 Subtask Report

Write final ligand structures and scores to disk, and/or copy coordinates back to top-level Impact arrays.

- `report setup [by glidescore | by energy] -`
  `[nreport` *num* `[cutoff` *val*`]] [norecep | recep | nil] -`
  `[external file` *fname*`] -`
  `[maxperlig` *num*`] rmspose` *val* `delpose` *val*
- `report collect rmspose` *val* `delpose` *val*
- `report rmspose` *val* `delpose` *val* `-`
  `write filename` *fname*
- `report keep [current | reference | best]`

The `report` subtasks specify how Glide is to select ligands and poses for output, and how to sort that output. In addition, the `keep` keyword specifies the ligand structure to copy internally, for use by subsequent (non-Glide) Impact tasks.

`setup` This version of the `report` subtask, with the following specifications, is required in the "setup" `DOCK` task, in order to allocate memory for the data to be saved and reported.

`by glidescore`
`by energy` Indicates whether the poses written to external files are to be those with the best `nreport` GlideScore or the best `nreport` grid energies (Coul + vdW). (`by score`, for the best `nreport` rough scores, is also available but not recommended.) The poses will be sorted in order of the selected scoring function.

`nreport` The maximum number of poses to be written to external files. The actual number written may be less than this either because fewer poses survive the rough-score or final scoring filters or because of the `cutoff` parameter.

`cutoff` Saves for output only those poses whose scores or energies are less than the best (lowest) plus the `cutoff` value.

`norecep`
`recep` Indicates whether the output structure file (in Maestro format) should include the receptor structure or not. The default is to include it (`recep`). If it is included, the file is suitable for on-screen analysis using the Glide *Pose Viewer*; otherwise (`norecep`), the file is suitable for use as ligand input in a subsequent Glide job. (Actually, files that do include the receptor may also be used in this way, simply by using the `gotostruct` keyword upon reading the file, to skip the receptor structure (which is always the first structure in the file).)

`external file`
Store qualifying poses from each ligand, as it is processed, in the specified file. The resulting file will in general be larger than the final output, as poses saved from one ligand may ultimately be displaced by better-scoring ones from subsequent ligands. But this method saves both CPU time and system memory, and also provides a "checkpoint" file of results so far, in case the job fails in the middle of the run. Unfortunately, external file storage does not work for "score in place" jobs, or if the `confgen` option (flexible docking of internally generated conformations) is not selected. We strongly recommend its use in all other cases.

`maxperlig`
Maximum number of poses to save for each distinct ligand molecule. `Maxperlig 1` is particularly appropriate for relatively rapid filtering of a large ligand database. The best-scoring ligands from such a run may then be used as input to a run with larger `maxperlig`, to get finer detail of binding modes, etc., of the top ligands.

`rmspose`
`delpose` Criteria for eliminating "duplicate" poses, i.e., those that are too similar for both to be worth saving.

Two poses are considered distinct if they satisfy *either* the RMS deviation or the maximum deviation criterion. The recommended values are `rmspose 0.5 delpose 1.3`. These must be specified in every `report` subtask.

`collect` Store the data for poses to be saved from the current ligand. This version of the `report` subtask typically appears in a loop over ligand (and/or conformer) structures. If `external file` was specified with `report setup`, the qualifying poses are saved to the external file; otherwise, their scores and identifiers, and information needed for reconstructing their structures, are stored in memory.

`write filename` *`fname`*
Write the saved poses, and a summary report, to disk, using *fname* as a base for the file names. The report will be written to *fname*.rept. If the receptor structure is included, it and the ligand pose structures will be written to *fname*_pv.mae (pv for Pose Viewer); if not, the ligand structures will be written to *fname*_lib.mae (a "library" of ligand structures for future use). If an "intermediate" `external file` was specified in the `report setup` subtask, Glide internally runs the `glide_sort` script (with filters as specified in the `scoring` subtask, and defaults for other arguments) on the intermediate file to get the final output. For postprocessing, the user can run `glide_sort` on either the intermediate file or the final output file.

`keep` Specifies which coordinates to copy back to the main Impact coordinate arrays, for subsequent Impact tasks.

`current` Do nothing. This maintains the Impact coordinate arrays as they were upon input to the current `DOCK` task.

`reference`
Copy the reference conformation (in its input pose) back to the Impact arrays.

`best` Copy the best pose (by GlideScore or grid energy, as specified with `report by`) back to the Impact arrays.

3.5.18 Subtask Run

Run docking calculation as specified in previous subtasks.

- `run`

Run the calculation. No keywords because they're all specified in the previous subtasks.

3.5.19 Results printed to Impact output

In addition to the structural output and summary reports described above (Maestro format structures in '`*.ext`' and either '`*lib.mae`' or '`*pv.mae`'; summary reports in either '`*.rept`' or '`*.scor`'), Glide reports results for each ligand it processes to the usual Impact output, namely "standard output" (typically redirected to file '*`jobname`*`.log`') and the *main output* file (typically '*`jobname`*`.out`') specified in the `write` command at the top of the Impact input file. For each ligand processed, this output includes information on the best pose found according to each of several scoring criteria.

```
DOCKING RESULTS FOR LIGAND        1 (Atropine)
Best Glidescore=-6.24 E=-26.53 Eint=5.56, pose 277, conf 2, lig 1; rmsd=66.161
Best Emodel=-57.10 E=-43.85 Eint=2.10 Glidescore=-5.42, pose 16, conf 3, lig 1; rmsc
Closest rmsd=61.572, pose 57, conf 3, lig 1; Glidescore=-2.48 E=-43.70 Eint=2.02
Lowest Efinal=-43.99 Eint=1.99 Glidescore=-2.23, pose 17, conf 3, lig 1; rmsd=61.596
```

In each of the above output lines, `E` or `Efinal` is the minimized, grid-interpolated Coulomb + vdW interaction energy between the receptor and the ligand in the particular pose; `Eint` is the internal (torsional) energy for the particular Glide-generated conformation of the ligand, and `Emodel` is the combination of `E` and GlideScore that Glide uses to rank poses of the same ligand. `Rmsd` is the heavy-atom RMS deviation between the particular pose and the reference ligand, and is reported only for the first ligand processed, and only if it is the same molecule as the reference.

In rigid docking runs, Glide groups together conformers of the same ligand that appear consecutively among its input structures. In such cases, the `DOCKING RESULTS` above are reported for the entire group, with an indication that all are conformers of one molecule.

```
DOCKING RESULTS FOR LIGANDS  57 --  58 (Confs of p38-pyrimidone0003)
Best Glidescore=10000.00 E=329.44, pose 1, conf 1, lig 57
Lowest Efinal=237.13 Glidescore=10000.00, from pose 9, conf 2, lig 58
Best Emodel=10000.00 E=237.13 Glidescore=10000.00 from pose 9, conf 2, lig 58
```

The values of 10000.00 in the above table indicate that Glidescore and Emodel were not evaluated for those poses, because they did not pass the filters specified in the `scoring` subtask. Note that `lig 57` and `lig 58`, and all ligand numbers reported in Glide output, refer to the position of the molecule in the user's input structure file. This correspondence is maintained not only for multiple conformers as above, but even if Glide cannot process some of the input structures. In other words, if the 56th structure in the input is skipped because it's too big, has unrecognized atoms, etc., the next structure will still be reported as ligand 57. Also, since this job did not generate ligand conformations internally, the designations `conf 1, lig 57` and `conf 2, lig 58` are actually redundant: the only conformations analyzed are those that were in the input, so `lig 57` *is* the first conformation of this molecule, and `lig 58` is the second.

In addition to the above output of "best" poses, Glide will print tables of poses processed from each ligand, after the rough-score and energy minimization steps, if the `verbosity` parameter is set higher than 1. Since this output can run to tens or hundreds of poses per ligand, we strongly recommend against setting `verbosity` that high in jobs with many ligands, except for testing or debugging purposes.

4 Advanced Input Scripts

In this chapter, we will discuss some advanced features of Impact input scripts (DICE scripts). You will find it is very powerful after you spend some time with it. You can manipulate internal data lists; you can use `if else endif` statements inside the input file; you can specify a `while endwhile` do loop to control a simulation; you can even `call` a previously written script *subroutine* to perform a common task, etc.

4.1 Background

As you have probably noticed already, at its core Impact is a program for processing a series of commands in a control file, the *input file*. These basic commands comprise a set of powerful tools for modeling complex chemical structures; the three levels of commands are the *task*, *subtask* and the "program" levels. The last level is independent of which task or subtask is presently being used, and consists of a set of data structures and programming constructs. At the program level it is possible to write programs defining the execution of Impact, as well as to access and modify internal Impact data structures using *lists*. For example, counters can be created and incremented, tasks and subtasks can be executed inside of looping constructs, and the internal state of Impact can be examined or modified.

The task level communicates to the program that a group of complex operations will be performed. Each task is invoked by giving the task name alone on a line of the input file. For example, for the `dynamics` task, which integrates the equations of motion for a chemical system, the word `dynamics` appears alone on a line. This causes the program to branch into the portion that performs a molecular dynamics simulation. The word `quit` (alone on a line) ends the current task and returns the execution pathway to the main controller. At this point the subsequent task is performed.

Inside each task a series of subtasks are performed. Here details are given about the particular pathways to follow or parameters to use in the context of the current task. For example, in the task `setmodel` (which specifies the features of the energy model to be used in simulations) the subtask `setpotential` specifies the types and weights to be used in the energy function. The subtask `mixture` takes a solute molecule and places it in a box of solvent molecules.

At the lowest level, programming constructs and data structures are manipulated in a task/subtask independent way. When these programming constructs are used, the commands appear by themselves on a command line. For example, in using Impact's conditional construct, an `if` block, a line such as '`if 'a' eq 'b' dynamics endif`' would not work, however, the following multiple line command is acceptable:

```
if 'a' eq 'b'
  dynamics     ! do the task dynamics if 'a' and 'b' are equal
```

```
        < some dynamics operations >
      quit
    endif
```

The existence of a programming language inside of Impact greatly increases both its ease of use and the ability to express complex computational experiments that might otherwise be all but impossible to perform.

The data structures available in Impact are *scalars* and *lists*, which correspond to variables and constants in typical programming languages. Lists are perhaps most similar to arrays of records, and may contain one number, or thousands. An Impact list is like a two dimensional array in containing rows and columns; the number of rows is called the *size* and the number of columns is called the *dimension* of the list. An *element* of a list is, for example, the value at row 1 and column 1.[1] Generally the size of a list is flexible and will grow as needed, whereas the dimension is fixed and is determined by how the list was first created. Arithmetic operations on lists normally require that both operands be of the same dimension or that one be scalar. When used as a logical expression, an empty list will be the same as a false expression. Conversely, a list with any elements in it is a true expression.The elements of lists can be referenced in a number of ways.

4.1.1 Lists

For the user of Impact, the primary means to manipulate data is using the data structures referred to here as *lists* or *tables*.[2] The names of lists are **always** placed within single quotes when used, and these names have maximum lengths of 30 characters. All characters supported by the computer are allowed with the exceptions of single quotes and underscores. Some valid names are '*Validname*', '*themotherofalllists*'' and '*abc123me&u*'. Note that underscores should not be used in list names since they are used to delimit columns of real numbers.

A list is a collection of related elements with a well defined structure, both in size and dimension. Some major types of list structures in Impact are atom, residue, molecule and species number; these types of structure are automatically recognized within Impact. Properties such as charge and surface area are frequently calculated in one of these types of list. Other types of list may also be used, for example lists to store properties with cartesian (x, y, z) components, or lists of position, force and velocity. Another type of list is a set of of statistics containing the three components sum, average and standard deviation.

There are two broad catagories of lists, *user defined* and *internal*. Most properties are shared by these two types. However, several internal lists

---

[1] A list with size 1 and dimension 1 would be the same as a scalar variable found in many computer languages.

[2] Lists and tables are equivalent.

are tied to the internal system state. Internal lists are "peep holes" into the major Impact data structures. These lists are created the first time they are referenced as a copy of the current state of the related Impact data structure.[3] These lists are are structured according to the information contained within them, since Impact is able to create the structure of the list from the information in the chemical system currently being used. For example, the list `surfacearea` is structured by atom.

Both internal (built-in) and user defined lists only "come into existence" the first time they are specified. Because internal lists are only copies of the internal data structures used by Impact, they stay fixed after the initial copy is made, even if subsequent Impact tasks modify the corresponding internal data structures. These lists are only "refreshed" with current data when used the first time. To subsequently update the lists with new data the old copies are first erased using the `reset` command, after which any subsequent use of the list will cause it to be updated with the current Impact data. For later updating, the `reset` command must be used again. Many of these built-in lists are useful for storing information from tasks for later retrieval. This is particularly useful if dynamics is being run on the same system many times. Then the average of the averages of individual runs can be obtained.

While internal lists may be used before being assigned values, they will sometimes be undefined until certain subtasks are executed. For example, the `bondlist` has a component that is the actual bond energy, but this assumes that the parameters have been defined by using the `setmodel` task. The list `Current.kinetic` contains the current kinetic energy but this requires that `dynamics` has been run. Other internal lists requiring that a task or subtask be performed before they may be used are the lists for surface area (`surfacearea`) and the rms deviation (`rms.dev.atom`), where the analysis task must be run and the appropriate subtasks performed before the lists are properly defined. The creation of these lists is done automatically, and they may be used after the subtasks are run. The cartesian coordinate list (`cord`) can be used at any point after the task `create` is performed. In general, the contents of the list will vary depending on when the list is used. For example, the values of `cord` change after a dynamics run. Remember the caveat that the value of internal lists are set as soon as they are used, but if the values need to be updated the command `reset` must be used to clear the old contents of the list. The next use of the list name will then cause the values of the list to be updated.

4.1.2 Internal Lists

The following tables show the internal ("built-in") lists that carry the current state of various Impact internal data structures.

[3] We emphasize that internal lists are user-accessible copies of the Impact data structures.

Global `Impact` built-in lists		
List name	List type	`Impact` tasks
`surfacearea`	atoms	`analysis`
`hydration`	atoms	
`bondrr`	residues	
`torsionrr`	residues	
`14elerr`	residues	
`vdwerr`	residues	
`hb1012rr`	residues	
`totalrr`	residues	
`anglerr`	residues	
`14ljerr`	residues	
`noerr`	residues	
`eelrr`	residues	
`hbelrr`	residues	
`rmsfluctuations`	atoms	`mdanalysis`
`avg.temp`	species	`dynamics`
`avg.kinetic`	species	
`avg.bond`	species	
`avg.angle`	species	
`avg.torsion`	species	
`avg.nonbonded`	species	
`avg.lj612`	species	
`avg.coulomb`	species	
`avg.hbond`	species	
`avg.lj14`	species	
`avg.coulomb14`	species	
`avg.potenergy`	species	
`avg.totalenergy`	species	
`avg.translation`	species	
`avg.rotation`	species	
`avg.virial`	species	
`avg.tail`	species	
`current.kinetic`	species	
`current.translation`	species	
`current.rotation`	species	
`current.temp`	species	

Global `Impact` built-in lists (continued)		
List name	List type	`Impact` tasks
`potenergy`	species	`minimize, montecarlo,` or `dynamics`
`current.bond`	species	
`current.angle`	species	
`current.phi`	species	
`current.nonbonded`	species	
`current.lj612`	species	
`current.coulomb`	species	
`current.hbond`	species	
`current.lj14`	species	
`current.torsion`	species	
`current.buffer`	species	
`current.tail`	species	
`current.energy`	species	

Global `Impact` built-in lists with subfields				
List name	List type	Subfields (names)		
`atoms`	atoms			
`residues`	residues			
`molecule`	molecules			
`species`	species			
`force`	atoms	`x`	`y`	`z`
`velocity`	atoms	`x`	`y`	`z`
`box`	dimensions	`x`	`y`	`z`
`charge`	atoms			
`bondlist`	bonds	`bdis` (distance)	`enrg` (energy)	
`anglelist`	angles	`bang` (angle)	`enrg` (energy)	
`torsionlist`	torsions	`btors` (torsion)	`enrg` (energy)	
`cord`	atoms	`x`	`y`	`z`
`intcord`	atoms	`bnd` (bond)	`ang` (angle)	`phi` (torsion)

4.1.3 Subsets of Lists

It is often desirable to select an element, or sets of elements from lists. There are several ways to do this.

4.1.3.1 Underscore notation

Lists with multiple dimensions may be referenced by appending an appropriate suffix to the list name, where the format is '*`listname_ref`*'. For cartesian components the suffixes are `_x`, `_y` and `_z`, and for statistical components `_sum`, `_avg` and `_stdev`. For instance, the x component of the force list named '`myforce`' would be named '`myforce_x`'. A collection of other prefixes is:

`_1` `_2` `_3`

```
_bdis _enrg
_bang
_btors
_bnd  _ang  _phi
```

Another use of the underscore is to modify the order of printing or calculations. There are a number of field modifiers supported, and the order field modifiers appear will dictate the order they will appear in the resulting list.

`'cord_x_y_z'`	same as 'cord'
`'cord_y_z_x'`	a 90 degree rotation
`'intcord_phi'`	only interested in the angle value
`'bondlist_enrg'`	only interested in bond energy
`'torsionlist_btors'`	only interested in torsion value

4.1.3.2 Lists as arrays

A range of list elements can be specified using square brackets. For instance, `'myforce_x[1:100]'` specifies the first 100 elements of the list of x component of force. A sublist may always be substituted for a list.

4.1.3.3 Colon notation

Subsets of lists can also be specified using *colon notation* and a number of list operations. Note that the properties defined using colon notation make up a *virtual* list when used with the *list selectors*, i.e., the `with` command. This is done by defining constraints (properties), each constraint building on the previous ones, until a collection of properties is specified that defines the structure of interest. With this structure you can then select a subset of elements from a list of interest.

In the following code fragment

```
species:spec:molecule:mol:
```

we specify a subset where the elements share the properties of (a) belonging to species *spec* and (b) belonging to molecule *mol*. In

```
residue:res:atom:atom:
```

the elements of the defined subset would belong to residue *res* and possess the atom name *atom*[1]. Any of these specifiers may be replaced by a range of names or numbers separated by a hyphen, or a group of comma-separated names or numbers. The wild card character '*' may be used to specify all names or numbers of a particular type, or it may also be used with any combination of symbols to create a name.

It is important to emphasize that the rightmost component of this structure specification determines the structural feature referenced. For instance,

```
species:1:residue:1:atom:1
```

refers to atom number one in residue number 1; whereas

```
species:1:residue:1
```

[1] The specifiers *spec*, *res* or *mol* are names or numbers.

refers to the entire first residue. Molecule is an optional specification. If the `species` or the `residue` specification is omitted then all species or all residues are implied. Here are some examples:

```
species:1                    ! species one
species:1:residue:1          ! the first residue in species one
species:Water                ! the species named Water
residue:1                    ! residue one
residue:1:atom:*             ! all atoms in residue one
residue:1-3,6:atom:*         ! all atoms in residues one through three and six
residue:1:atom:C*            ! all carbon atoms in residue one
residue:HYP*:atom:C*         ! all carbon atoms in all HYP residues
```

A constraint is one of the following:

- Any internal list that contains a valid structure (e.g., an atom, residue, molecule or species list).
- `species`:*ranges*:
- `molecules`:*ranges*:
- `residues`:*ranges*:
- `atoms`:*ranges*:

4.1.3.4 Hyphen notation

Ranges are a list of numbers separated by hyphen (inclusive) or commas *or* a list of strings with or without wild cards, the '`*`' character.

```
residues:1-4:atoms:CA,C,N:
molecules:1:atoms:1,3-5:
species:1:
residues:*:atoms:C*:
atoms:1-4:'myproperty'
```

Note that an attempt will be made to locate the specified structure throughout the whole system. For example, the query

```
atoms:1:
```

returns a list containing the first atom for *each residue* and not just the first atom of the entire system.

Once a structure is defined, a subset can be chosen where the elements share appropriate properties. In the following items the subsets are equivalent to lists. The list selector `with` is used here for selecting subsets from lists, and along with other selectors is described below.

- '`'surfacearea' with atoms:1-4:`' results in a subset of the list `surfacearea` corresponding to atoms 1 to 4.
- '`'force_x_y' with residues:1-3:atoms:*:`' results in a subset of the list `force` containing the x and y force components for all atoms in residues 1 to 3.
- '`'rmsfluctuations' with residues:4:atoms:h*:`' results in a subset of the list `rmsfluctuations` for all hydrogen atoms in residues 1 to 4.

Having selected the range of properties you wish to work with you can do operations on those properties. A large library of arithmetic and statistical functions is available.

4.1.4 List Creation

Lists are generally created using the command `put`; however, `create` has some uses that the other doesn't.

4.1.4.1 Put

The `put` statement is used to assign values to lists. In doing so the list is created if it didn't already exist.

```
put 'expression' into 'list'
```

4.1.4.2 Create

Create a new list.

4.1.5 List Selection

As noted above, the properties describing subsets of lists are built up using several notations, and subsets of lists are actually constructed using list constructors like `with`; this and other list functions are described here. The resultant subsets are often placed in new lists, which is the convention followed in these examples.

4.1.5.1 With

The function `with` returns those elements in one list that are found in both lists. Atoms, molecules, residues, and species are recognized by these functions. In the following example those elements in the `'charge'` list belonging to atoms with names beginning with the letters `'CA'` are selected.

```
put 'charge' with atoms:CA*: into 'result'
```

4.1.5.2 Withonly

The `withonly` function extracts those elements in the list whose atom, molecule, residue or species specification match the entire target specification. In the following example, only those bonds containing **both** CA* and N* atoms are extracted. In contrast the selector `with` returns all bonds with CA or N atoms.

```
put 'bondlist' withonly atoms:CA*,N*: into 'result'
```

4.1.5.3 Without

The `without` function returns those elements in the first list that do not have relations with the second list. This example extracts those elements from the torsional internal coordinate list that are not hydrogen atoms.

```
put 'intcord_phi' without atoms:h*: into 'result'
```

4.1.5.4 By

The `by` function returns a list that is the result of applying the previous function over a long list split up by its structures. `By` requires two lists. One of these is called the limit and must be of type residue, molecule or species, and the other is called the *range* and must be of type atom, residue or molecule. The result is a list the same length as the limit, with each element storing the result of applying the previous function over the range split up along the structures of the limit. The functions you can apply `by` to include: `abs`, `int`, `avg`, `stat`, `sum`, `sum2`, `ln`, `sin`, `cos`, `tab`, `asin`, `acos`, and `atan`. The following example results in a list of type residue with each element storing the sum of the atom charges for each residue. (In most cases this would be a of list of zeros, ones and minus ones.)

```
put sum 'charge' by 'residue' into 'result'
```

4.2 Operations on Data

A range of functions and list-selectors are available, including the standard arithmetic expressions and a set of functions defined solely for lists. A *list expression* is a list or any arithmetic or functional expression that results in a list, and a list-expression may always be substituted for a list. The arithmetic operators include exponentiation (`^`), multiplication (`*`), division (`/`), addition (`+`), subtraction (binary `-`) and negation (unary `-`). These may be applied to constants, such as '`2 * 2`', or used as *list operators*. Operations may be performed between lists with common structures, or between lists and scalars.

When operations occur between lists of different dimensions, the result of the operation inherits the dimensionality of the list of higher dimension. Consider the following examples in which *'myforces'* is a list of atomic forces having an atomic cartesian (x, y, z) structure, *'jscal'* is a user-defined list having a simple atomic structure, and *'const'* is a scalar sonstant.

```
'myforces_x' * 'jscal'
```

multiplies the corresponding elements of the x component of '*myforces*' and '*jscal*'.

4.2.1 General Operations

Arithmetic functions are applied to a list in one of three ways:

1. If one of the operators is a single element, the operation is done with the value of that element against all the values in the other list. (That means that you can multiply an entire list by a single constant.)
2. Some functions take only a single list and return a few elements of information about that list, such as the average value of the list, or its four (4) greatest values.
3. If you are applying a function between two lists and both lists have size greater than 1, that function will be applied to each element in the two

lists that correspond to each other. This means you can add the values of two lists in an element by element manner.

```
1 + 'mydata'                      ! every element gains 1
'mydata' + 'mydata'               ! <--- these are
2 * 'mydata'                      !      the same
'mydata' pow 0.33333              !  cube root
7 lowest 'mydata'                 ! sorted lowest 7 elements
avg 'mydata'                      ! the list average put in a new 1 element list
(sum 'mydata')/(length 'mydata') ! silly way to avg
('newdata'+'olddata')/2           ! result is a new list consisting of the
                                  !  average values of each of the list elements
```

```
'myforces_x' * 'const'
```

multiplies all the x components of *'myforces'* by the value of *'const'*. The command

```
'myforces' + 2.0
```

adds the value of 2.0 to all of the components (x, y, z) of *'myforces'*.

General Operators			
Operator	Function	Parameters	Units
+	Addition	2	
-	Subtraction	2	
*	Multiplication	2	
/	Division	2	
abs	Absolute value	1	
acos	Arc Cosine	1	radian
add	Addition	2	
asin	Arc Sine	1	radian
atan	Arc Tangent	1	radian
avg	Average	1	
avgb	Special case of by function	2	
by	Apply a 1 parameter function over a list of values (e.g. sum 'charge' by 'residues')		
cos	Cosine	1	radian
distance	Distance Function	2	cord units
div	Division	2	
grdist	Greatest Distance	2	atoms units
greatest	N Maximum values e.g., 3 greatest 'bondlist_bdis'	2	
index	Extracts an element from a list e.g. index 10 'charge' gets the 10th value from the charge list)	2	
int	Truncation	1	
length	Size of list	1	
lowest	N Minimum values	2	
ln	Natural Log	1	
^	Exponentiation	2	
exp	Exponentiation (base e)	1	
lstdist	Least Distance	2	atoms units
alldist	All distances	2	atoms units
hist	Histogram	2	
max	Maximum value	1	
min	Minumum value	1	
mul	Multiplication	2	
pow	Power function (base 10)	1	
rand	Random number	1	
runavg	Running Average	1	

General Operators			
Operator	Function	Parameters	Units
`sin`	Sine function	1	radian
`sizeof`	Size of list	1	
`sqrt`	Square root	1	
`sqr`	Square	1	
`stat`	Sum, Average, Standard Deviation	1	result is dimension 3
`std`	Standard deviation	1	
`sub`	Subtraction	2	
`sum`	Add all columns	1	
`sum2`	Add and square columns	1	
`sumby`	Special case of `by`		
`tan`	Tangent function	1	radian

Relational Operators	
Name of function	Example of usage
`and`	`if ('timer' gt 1) and (atoms:ca:)`
`eq`	`131 eq 23`
`ge`	`'charge' ge 0.2`
`gt`	`'bondlist_bdis' gt 1.2`
`le`	`'bondlist_bdis' le 1.1`
`lt`	`'anglelist_bang' lt 45`
`not`	`if not ( 'timer' gt 50 )`
`or`	`while ( 'counter' lt 100) or ( sum 'list' lt 1 )`
`xor`	`avg ( species:*:atoms:c*:)  xor avg ( species:*:ca:)`

4.2.2 Relational Operators

Relational operators may be used to perform list comparisons, and include `lt`, `le`, `eq`, `gt`, and `ge`. For example, the following relational expression could be used to select the forces greater than 0.05:

```
('myforces_x'^2 + 'myforces_y'^2 + 'myforces_z'^2)^0.5 gt 0.05
```

The boolean operators `and`, `or` and `not` may be used to combine relational expressions; in particular, a "not-equal" operation can be performed by using `not` to negate an `eq` comparison.

In addition to the standard mathematical operators, Impact provides many higher level operators that perform selection operations on lists. For instance, the `with` operator allows a constraint to be applied to a list. In this example, `with` is employed to restrict the list of surface area for each atom to those cases in which the charge on each atom in list `'qbyatom'` is greater than 0.2:

```
put 'surfbyatom' with ('qbyatom' gt 0.2)  into 'result'
```

Character and String Operators		
Operator	Function	Parameters
`char`	Integer to char conversion	1
`concat`	Append two strings	1

4.2.3 List Operators

Here the remaining list operations are fully described. These are really context-independent subtasks and are not expressions.

4.2.3.1 Restore

`Restore` copies the contents of a list to an internal list, from where it will be copied to one of the the internal data structures used in Impact (e.g., a common block). One such internal data structure is `charg`, another is `xyz`. For example, if some operations have been performed on a list of coordinates it may be desirable to have one of the standard tasks operate on these new coordinates. Note the required use of the square brackets as delimiters!

```
put 'cord' + [ 0.10 0.10 0.10 ] into 'cord' ! translate coordinate list
restore xyz 'cord'      ! put it back into the actual cartesian coordinates
dynamics                ! now run dynamics
```

4.2.3.2 Rand

The `rand` function returns a single random number in the range 0.0 to the first element of its parameter. A negative parameter resets the seed number.

4.2.3.3 Smooth

The `smooth` function returns a list that has less noisy data points. *Smooth* breaks up the input list into a series of short ranges and preserves for the final output those elements that are the mean value of the short ranges. The size of the range is determined by the first element of the first parameter, which should be an odd number such as 3, 5 or 7. Very large ranges will result in serious loss of information.

4.2.3.4 Histogram

The `hist` (histogram) function does a count frequency on a list (first parameter) using parameters in a second list. The first list can be any list with no more than 3 real columns of data. The second list must contain the minimum value of the histogram, the number of intervals and the width of each interval. This information can be stored in a list as in [0.0 100 0.25] or as a list of 3 elements each with 1 real field, e.g., ‘ `0.0 append 100 append 0.25`’. The result of this function is a list with the same number of real columns as the first argument containing the count of values in each interval plus an additional column containing the values of each interval (e.g., the above parameters would give 0.0, 0.25, 0.50, etc).

4.2.3.5 Distance

The `distance` function returns the distance between two coordinate sets. Coordinates are in x y z format. The coordinates for the current system are stored in the built-in parameter list named `'cord'`.

The `grdist` and `lstdist` functions return the greatest or least distance from every atom in the first parameter from every atom in the second parameter. The function `alldist` returns a list of all distances between the two input lists. This function should be used carefully since it creates lists of the size of $n \times m$ where n and m are the size of the atom lists used as parameters. The result is a bond list.

4.3 Advanced Scripts

Using the tools available in Impact, you can program simple tasks that allow one to:

- analyze data as it is being generated;
- automate simulations, look at results, modify input files and relieve resubmission drudgery;
- provide an easier method to plot and study Impact compatible data;
- analyze the result of past Impact runs stored in trajectory files;
- provide a mini programming language to allow simple algorithms not yet implemented in Impact to be tested with access to the Impact data bases for run time analysis.

4.3.1 Flow Control

Essential tools needed to control the flow of a program are provided.

4.3.1.1 While

The `while` statement is used to conditionally execute the contents of its body, repeating until the condition is false. While you can nest these loops, it is *very* important that you never use the `goto` statement to jump inside of one. The format of the `while` statement is

```
while expression
  body of while loop
endwhile
```

4.3.1.2 If/else/endif

In an `if` expression, the first expression following `if` is tested for its truth value. If true the *body* is executed. If an `else` is present then the *optional code* following `else` is executed when *expression* is false.

```
if expression
   body
else
   optional code
```

```
endif
```

If statements may also be nested, with one endif for every if. As in the case of the while statement it is illegal to jump into an if block using a goto.

4.3.1.3 Goto

Goto is provided but not recommend. The format of the goto statement is

```
:label                  ! note the colon
some code
goto label              ! loop to label
```

As noted, a goto may not cause a jump into the body of an if block or of a while block.[1] Use of a goto statement to jump out of an if or while block can cause stack overflows if done repeatedly.[2] A goto jump from within one if or while block into another if or while block will, of course, be fatal.

4.3.2 Subroutines

Call a subroutine and return. Call passes its optional parameters by the method of "pass by name"; this is a somewhat obscure method of passing parameters. "Pass by name" from the user's viewpoint is equivalent to "pass by reference". This means that any change in the value of the parameters within a subroutine will be passed back to the calling routine. Care must be taken to be sure that the main procedure does not extend into a subroutine. You should always follow the main procedure by the keyword end.

```
call alpha(100 'a' 'result')   ! call the subroutine
some more code
:alpha('a' 'b' 'c')            ! bind a, b, c to 100, 'a',, and 'result'
  definition body         ! perform calculations
put 'somevalue' into 'c'       ! return the result in variable 'result'
return
```

You may also append a file name after a call, this will cause the program to execute that subroutine within that file. Note that except for this special case all subroutines are searched for from the top of the current program in a first found, first executed manner.

```
call label [ parameters ] file fname
```

4.3.3 Spawn

Spawn starts a shell process at the operating system level and waits for the result.

```
spawn shell command UNIX shell command
spawn shell file executable file's name
```

---

[1] A block is all tasks up to the endwhile or endif.

[2] In a purely theoretical sense this is the only legitimate use for goto, and should properly be called break or exit.

4.3.4 Lists as Parameters

Numeric lists can be placed anywhere a number normally can be specified; if an operation requires a scalar value then the first element from the list's numeric field is used. Short character lists can also be used to hold filenames, which is especially useful when many files are being created and unique names are needed. Though we are getting ahead of ourselves by discussing specific tasks in the following example,[3] it does illustrate the use of different list operations and types of lists. Here we loop over the `run` subtask in `dynamics`[4]. While it would often only be desired to save the final state in a restart file, saving intermediate states assures that intermediate work has been saved if the job is terminated for any reason. A series of trajectory files might be saved in the same way.

`'i'`
: is a list that is used as if it were an integer variable.

`'filename'`
: is a list of characters that is modified in each stage of the dynamics run. Thus, unique trajectory files may be written for each phase.

`$protein$` and `$ps$`
: are string constants. Note the use of the dollar sign to delimit string constants.

[3] The example uses meta-variables that are explained in Chapter 2 [Setup System], page 13.

[4] The task `dynamics` is described in Section 3.2 [Dynamics], page 56

5 Trouble Shooting

This chapter describes some common problems with starting or running Impact. Naturally, we hope that you will never need to use this chapter. However, if you have problems using Impact, you may find useful advice here. You may also contact us using the information on the cover page.

5.1 Problems Getting Started

This section describes how to overcome some problems in starting up your Impact jobs. The next section describes problems that occur during job execution.

5.1.1 Environment variable `SCHRODINGER` not set.

Before running Impact, or any Schrödinger product, on any particular machine, you must set the environment variable `SCHRODINGER` to your Schrödinger installation directory. If this environment variable is not set correctly, you will be told directly:

```
unix% /usr/apps/schrodinger/impact -i dynamics_job.inp
ERROR: SCHRODINGER is undefined
unix%
```

Or if the program stops at automatic atom-typing for ligand molecules, it will prints out message like this:

```
%IMPACT-I (readhead): input file  23 has no header information.
%IMPACT-I (readhead): input file  23 has no header information.
 PARM read from file paramstd.dat
Environment variables MMSHARE_EXEC and OPLS_DIR not defined
Set OPLS_DIR so that ATOMTYPE can find data files
```

It is easy to fix this problem, first check whether `SCHRODINGER` is set or not, enter the command

```
 % echo $SCHRODINGER
```

If you see this environment variable is not set or set to a wrong directory, change it to a right directory. If you are running C shell (csh) or tcsh, type the command

```
 % setenv SCHRODINGER your Schrödinger installation directory
```

or if you are using bash, sh or ksh, type the command

```
 % export SCHRODINGER=your Schrödinger installation directory
```

5.1.2 Bad residue label

The current Impact program requires the user to separate a ligand molecule from the protein in the input PDB files. This means PDB files for proteins must contain only the regular amino acids and buried waters, but not a nonstandard residue name unless it has previously been defined. Here is an example of a PDB file containing a residue named NOA (NAPHTHYLOXY-ACETYL):

```
...............
...............
ATOM   1485  CD2 NOA I 201       4.098   9.733  20.948  0.50 20.67
ATOM   1486  CD1 NOA I 201       6.413  10.411  21.013  0.50 20.84
ATOM   1487  CE1 NOA I 201       6.706   9.320  21.850  0.50 21.17
ATOM   1488  CZ1 NOA I 201       5.694   8.437  22.228  0.50 20.95
ATOM   1489  CE2 NOA I 201       4.385   8.645  21.778  0.50 21.01
ATOM   1490  CZ3 NOA I 201       1.771   9.028  20.869  0.50 21.10
ATOM   1491  CE3 NOA I 201       2.786   9.926  20.504  0.50 20.98
ATOM   1492  CZ2 NOA I 201       3.379   7.740  22.165  0.50 21.13
ATOM   1493  CH2 NOA I 201       2.067   7.934  21.703  0.50 21.20
ATOM   1494  C   NOA I 201       4.312  13.086  17.860  0.50 18.24
ATOM   1493  CH2 NOA I 201       2.067   7.934  21.703  0.50 21.20
ATOM   1494  C   NOA I 201       4.312  13.086  17.860  0.50 18.24
ATOM   1495  O   NOA I 201       5.155  13.679  17.160  0.50 17.86
..............
..............
```

The program will stop because (we presume) there is no template file for residue NOA. The message printed out in the primary output file looks like this:

```
      *** BAD RESIDUE LABEL    NOA
%IMPACT-E (die): Fatal error at line   5
```

At present, the user has to separate the NOA molecule from the protein residues in the PDB file, and read it in through `type ligand`:

```
build primary name hiv type protein read file hiv.pdb
build primary name noa type ligand read file noa.pdb
```

5.2 Runtime Problems

This section documents some situations when an Impact job may terminate prematurely.

5.2.1 SHAKE problems

`SHAKE` is a commonly used algorithm for constraining bond lengths and (or) bond angles in protein or solvent molecules, such as water. It is especially useful for rigid water models such as SPC, TIP3P, and TIP4P. However, the algorithm is only useful for small perturbations from their equilibrium values. If the bond lengths are too far away from their equilibrium values, the algorithm will encounter problems with numerical instability:

```
%IMPACT-W (ishake): SHAKE was not accomplished within   1000 iterations
%IMPACT-W (ishake): SHAKE was not accomplished within   1000 iterations
%IMPACT-W (ishake): SHAKE was not accomplished within   1000 iterations
```

The problem is usually due to a too-large timestep in molecular dynamics, or the molecular structure is not well minimized. Thus, extremely large repulsion forces might appear in van der Waals interactions, which results in a large move in bond lengths. The way to avoid this problem is to check your structure first, make sure it is well defined and minimized to some extent, then try again. If it still fails, use smaller time steps.

5.2.2 FMM problems

If you specify `fmm` in `setmodel` task, the program will call the FMM method for calculating electrostatic interactions. Here is a common problem:

```
%IMPACT-W(FMM_load_bodies): particle out of box in FMM
%IMPACT-W(FMM_load_bodies): particle out of box in FMM
%IMPACT-W(FMM_load_bodies): particle out of box in FMM
%IMPACT-E(FMM_load_bodies) Too many particles out of box, check your timestep!
```

The problem usually appears when some particles move too much inside one `r-RESPA` big time step (or one `VERLET` time step). The box size, which is updated after every big time step in `r-RESPA`, might not be large enough to hold all the particles, thus some particles move out of the range of box size. Of course, the real underlying reason for this problem is similar to that in `SHAKE`, a too-large timestep in molecular dynamics, or an ill-defined molecular structure is used. Thus, the way to avoid this problem is similar to that in SHAKE, i.e., check your structure first, make sure it is well defined and minimized to some extent, then try again. If the problem still appears, use smaller time steps.

5.2.3 Atom overlap problems

The program may stop if two or more atoms overlap in space. Impact checks for atom overlaps in the very beginning when non-bonded lists are generated. Here is one example error message:

```
%IMPACT-I(code): found all bond parameters for system
%IMPACT-I(code): found all bend parameters for system
%IMPACT-I(code): found all tors parameters for system
  Moment of inertia tensor
          0.46449E+07     0.90790E+06     0.87475E+06
          0.90790E+06     0.45322E+07    -0.61956E+06
          0.87475E+06    -0.61956E+06     0.43931E+07
  Moment of inertia tensor after diagonalizing
          0.29204E+07     0.90495E-10     0.17211E-08
          0.90495E-10     0.50757E+07    -0.17493E-08
          0.17211E-08    -0.17493E-08     0.55741E+07
  Maximum distance along x,y,z-axis
          0.61017E+02     0.38485E+02     0.35377E+02
  Solutes are rotated 90 degree about y-axis
  Maximum distance along x,y,z-axis after the rotation
          0.35377E+02     0.38485E+02     0.61017E+02
 %IMPACT-I (trans): The system will be rotated to align the principal
                    axis with the largest eigenvalue along the diagonal
  Maximum distance along coordinate axis after the rotation
          0.46611E+02     0.44300E+02     0.45865E+02
%IMPACT-I (allocnb): Verlet list size =   261232
%IMPACT-I (allochb): Hydrogen bond list size =   206421
%IMPACT-E (die): At line  29
%IMPACT-E: TWO ATOMS HAVE THE SAME COORDINATES
```

The program stops because it finds that two or more atoms overlap. This may happen when missing H atoms generated by Impact sit on top of other

H atoms that already exist in a PDB file (usually those H atoms were generated by other programs, such as MacroModel or ChemEdit, etc.). Another possible cause of this problem is that some atoms' coordinates were not initialized to correct values, but are all zero. This is especially likely to happen in simulations with explicit solvent. The program needs to know the coordinates of solvent water molecules either by reading from a restart file or by reading from an old equilibrated water box (e.g., *spchoh.dat*, *tip4p.dat*). If a restart file is not used, no water atom coordinates will be assigned and FORTRAN code will initialize them all to zero. Thus they "overlap" in space. Here is an example of an incorrect input file:

```
!! Timings for testing protein/water system
write verbose 3 file test.out title test *

CREAT
  build primary name test type protein read file test.pdb
  read coordinates name test brookhaven file test.pdb
  build solvent name agua type spc nmol 10000 h2o
QUIT

SETMODEL
   setpotential
     mmechanics
   quit
   energy molcutoff name agua
   read parm file paramstd.dat noprint
!==>  solvent old file spchoh.dat bx 68 by 68 bz 68
   solute translate rotate diagonal
   enrg parm cutoff 9.0 -
     listupdate 20 diel 1.0 nodist print 1
   enrg periodic name test bx 68 by 68 bz 68
   enrg periodic name agua bx 68 by 68 bz 68
   enrg cons bond
QUIT

MINIMIZE
 input cntl mxcyc 1000
 steepest dx0 0.01 dxm 1.0
!==>  read restart box coordinates formatted file testh2o.min
 run
 write restart box coordinates formatted file testh2o.min
QUIT

END
```

The solution is to uncomment either of the two commented out (!==> ****) command lines.

5.2.4 Atomtyping problems

The automatic atomtyping code will assign atom types and parameters for virtually any kind of molecule or ion if the structure is well defined, i.e., if all missing H atoms are included and bond lengths are reasonable. If a structure is not well defined, i.e., if there are too many isolated atoms or too many atoms with bonds exceeding their maximum numbers, the atomtyping code will get confused. Here is an example of an output message:

```
%IMPACT-I(newres): Input template file is a PDB file
%IMPACT-I(newres): build template for this molecule

Warning: too many bonds for atom   H25 : nconn=2 max=1
Warning: too many bonds for atom   H26 : nconn=3 max=1
Warning: too many bonds for atom   H27 : nconn=3 max=1
Warning: atom   H30 is isolated
Warning: atom   H31 is isolated
Warning: atom   H32 is isolated
Warning: atom   H33 is isolated
Warning: too many bonds for atom   H37 : nconn=2 max=1
Warning: too many bonds for atom   H38 : nconn=2 max=1
Warning: too many bonds for atom   H40 : nconn=2 max=1
Warning: too many bonds for atom   H41 : nconn=2 max=1
Warning: atom   H42 is isolated
Warning: atom   H43 is isolated
Warning: atom   H44 is isolated
Error: Too many exceptions in connection table, check your molecule
```

Impact will try to adjust the connection table to resolve these issues, but will stop if too many problems are encountered. Such problems can occur when structures are used that have been converted from other programs, especially structures converted from 2D to 3D. A solution may be to use a program that has a builder, such as Maestro or ChemEdit, to rebuild the molecule.

Function Index

O

P

Q

R

S

T

U

V

W

X

Y

Z

Concept Index

E

F

G

H

I

R

S

T

U

V

W

Z

Table of Contents