



Aten v1.8

User Manual

Last Updated Thursday, 28 November 2013

Disclaimer

Aten comes with ABSOLUTELY NO WARRANTY.

This is free software, and you are welcome to redistribute it under certain conditions. For more details read the GPL at <http://www.gnu.org/copyleft/gpl.html>.

Aten is under continual development and tweaking. If you have a feature suggestion, a feature request, or have found a bug or an idiosyncrasy, please contact me at tris@projectaten.net

Aten uses Space Group Info © 1994-96 Ralf W. Grosse-Kunstleve, the Qt toolkit © Nokia, and the readline library.

Some general notes:

First off, if you find that Aten doesn't support the format you want, consider writing a filter to do so. And if you need help, please ask!

More so, if you find that Aten *does* support the format you want, but doesn't appear to write it out correctly, please let me know.

For forcefield expression files written with Aten, I recommend that you check that the correct atom types get assigned to your molecule(s). Blind trust that this has actually occurred should only be employed once you are confident that the forcefield types are consistently correct! Larger forcefields, e.g. OPLS-AA by Jorgensen *et al.*, do not have a complete set of type descriptions implemented in Aten. Once more, if you add some in, please send them to me so they can be included in a future revision of the code.

tris@projectaten.net

Acknowledgements

Aten is maintained entirely by T. Youngs. However, many people have contributed useful thoughts, ideas, bugfixes, and motivation to the project over its lifetime. These people are, in no particular order:

A. K. Croft and crew (Mac aficionado, supreme fault-finders)

S. Cromie (algebraic guidance)

A. Elena (bug-finder extraordinaire, ‘pathological case’-locator, Windows / Mac building and packaging)

D. Pashov (Windows / Mac building and packaging)

S. Norman (icon judgment)

N. C. Forero-Martinez and M. Smyth (bug finding, bug finding, and bug finding)

Table of Contents

1.	Introduction	1
1.1.	Overview.....	1
1.2.	Feature Matrix.....	1
1.3.	Supported File Formats.....	1
2.	Installation	3
2.1.	From Precompiled Packages (All Platforms)	3
2.2.	From Source (Linux/Mac OS X)	3
2.3.	From Source (Windows).....	6
3.	Frequently Encountered Problems	12
3.1.	Configuration Errors	12
3.2.	Compilation Errors	12
3.3.	Usage Errors	13
4.	Quickstart	15
4.1.	Terminology.....	15
4.2.	File Locations	16
4.3.	Referencing Aten	17
5.	Usage Examples	18
5.1.	Creating a Bulk Water Model (GUI)	18
5.2.	Building θ -Alumina from Basic Crystal Data	23
5.3.	Creating an NaCl/Water Two-Phase System (GUI).....	27
5.4.	Building Ice I _h from Crystal Information (GUI).....	33
5.5.	Exporting Coordinates in Bohr (Filters)	36
5.6.	Calculate Average Coordinates (CLI)	38
5.7.	Self-Contained Liquid Chloroform Builder (Script)	39
5.8.	Generating Images Without the GUI (CLI)	40

5.9.	Calculating a Torsion Energy Profile (CLI)	41
5.10.	Running a Script on Many Models (CLI/Batch).....	42
5.11.	Saving GAMESS-US Input with Options (CLI).....	43
5.12.	Printing all Bond Distances in a Model (CLI)	44
5.13.	Creating a Porous Silica Model.....	45
6.	Command Line Usage	46
6.1.	Switch Order	46
6.2.	Switches	46
6.3.	Batch Processing Modes	53
7.	The GUI.....	55
7.1.	Overview.....	55
7.2.	Mouse Control	56
7.3.	Keyboard Shortcuts.....	58
7.4.	The Main Toolbar	59
7.5.	The ToolBox	61
7.6.	Atom List Window	62
7.7.	Build Window.....	63
7.8.	Cell Definition Window.....	65
7.9.	Cell Transform Window	66
7.10.	Command Window	68
7.11.	Disorder Builder Wizard	71
7.12.	Forcefields Window	76
7.13.	Fragments Window	79
7.14.	Geometry Window	81
7.15.	Glyphs Window.....	82
7.16.	Grids Window	83

7.17.	Messages Window.....	86
7.18.	Pore Builder Window.....	87
7.19.	Position Window.....	89
7.20.	Select Window	92
7.21.	Trajectory Window	94
7.22.	Transform Window	95
7.23.	ZMatrix Window.....	98
8.	Command Language.....	99
8.1.	Command Language Overview	99
8.2.	Variable Types.....	105
8.3.	Custom Dialogs.....	147
9.	Command Reference	150
9.1.	Atom Commands	150
9.2.	Bond Commands.....	156
9.3.	Building Commands	160
9.4.	Cell Commands.....	168
9.5.	Charges Commands	173
9.6.	ColourScales Commands.....	175
9.7.	Disorder Commands	178
9.8.	Edit Commands.....	180
9.9.	Energy Commands.....	182
9.10.	Flow Commands	185
9.11.	Forcefield Commands	190
9.12.	Forces Commands	201
9.13.	Glyph Commands.....	202
9.14.	Grid Commands	208

9.15.	Image Commands.....	215
9.16.	Labeling Commands	217
9.17.	Math Commands	219
9.18.	Measuring Commands.....	222
9.19.	Messaging Commands	225
9.20.	Minimiser Commands.....	228
9.21.	Model Extras Commands.....	231
9.22.	Model Commands	233
9.23.	Pattern Commands	240
9.24.	Pores Commands.....	243
9.25.	Read / Write Commands	245
9.26.	Script Commands	255
9.27.	Selection Commands.....	256
9.28.	Site Commands	265
9.29.	String Commands.....	267
9.30.	System Commands.....	272
9.31.	Trajectory Commands	275
9.32.	Transform Commands.....	278
9.33.	View Commands	285
10.	Topics of Interest.....	289
10.1.	Colourscales	289
10.2.	Glyphs	290
10.3.	Patterns	291
10.4.	Partitioning Schemes.....	294
11.	Filters	295
11.2.	Trajectory Files	303

11.3.	Reading and Writing	306
12.	Forcefields and Typing.....	311
12.1.	Overview	311
12.2.	Supplied Forcefields.....	312
12.3.	Keyword Reference.....	315
12.4.	Rule-Based Forcefields	324
12.5.	Typing	326
12.6.	NETA Reference	332
13.	Functional Forms.....	338
13.1.	VDW Functional Forms	338
13.2.	Bond Functional Forms	Error! Bookmark not defined.
13.3.	Angle Functional Forms.....	Error! Bookmark not defined.
13.4.	Torsion Functional Forms	Error! Bookmark not defined.
14.	External Programs	342
14.1.	Movie Generation.....	342
14.2.	MOPAC.....	342
15.	Methods	343
15.1.	Custom Algorithms	343
15.2.	Literature Methods	348
16.	Enumerations	349
16.1.	Basis Shell Types	349
16.2.	Bond Types	349
16.3.	Bound Types	349
16.4.	Cell Types	350
16.5.	Colour Schemes.....	350
16.6.	Combination Rules.....	350

16.7.	Drawing Styles	351
16.8.	Energy Units.....	351
16.9.	Glyph Types	351
16.10.	Grid Styles.....	352
16.11.	Grid Types.....	352
16.12.	Label Types	352
16.13.	Monte Carlo Move Types	353
16.14.	Output Types	353
16.15.	Parse Options.....	354
16.16.	Read Success Integers	354
16.17.	ZMapping Types	354

1. Introduction

1.1. Overview

Aten will let you generate and edit coordinates for your simulations, and view any trajectories you might have generated. A set of tools in the GUI (and also accessible from the command line) enables you to change the geometry of bonds, angles, and torsions, translate atoms, create atoms, rotate groups of atoms, and cut, copy, and paste them around the place. All this can be done in the context of loading and saving in the format that you need - if the file format you need isn't currently a part of Aten, you yourself can write a filter to cover it.

Periodic systems are supported, be they crystals, liquids, gases, or a heady mixture. All editing functions that are possible for simple molecules apply to periodic systems as well. Moreover, given a basic unit cell a whole crystal or a larger supercell can be constructed. For any periodic system, a random ensemble of molecules can be added, allowing the facile creation of coordinates with which to begin your molecular dynamics simulations.

As well as coordinates, Aten has support for forcefields (in its own, plain-text format) and can automatically apply these forcefields to your system if correct type descriptions are present for the atom types in the forcefield. Then, in the same way as with coordinates, you may write out a forcefield description of your system in the format that you require it with a different filter. Please don't use Aten as a literal 'black box', though, and blindly write out forcefield files without checking them. While it will certainly make the process of generating your forcefield descriptions easier, the art of determining the correct types in the first place (and hence the correct forcefield terms) is not definite for larger forcefields that cover many atom types. Check the output – a cursory glance of the selected forcefield types is an excellent idea, and a good habit to get in to.

Aten is in continual development, so if you get stuck, find a bug, or have a suggestion to make, please go to the Support page of the website and visit the forums or send an email directly. Making Aten better depends to some degree on user feedback!

1.2. Feature Matrix

TODO

1.3. Supported File Formats

A list of formats currently supported by Aten follows as well as the file extensions and assigned filter IDs. Remember, adding support for other codes and formats is in your hands with Aten's filters (see Section 11).

Table 1-1 Supported Model Formats

Format	Extension(s)	Nickname	ID	R/W	Notes
Aten Keyword Format	*.akf	akf	1	RW	Plain-text format used by Aten
Cambridge Structural Database Service	*.dat *.fdat	csd	7	RO	
Chem3D Cartesian Coordinates	*.cc1 *.cc2	cc	34	RO	
Crystallographic Information File	*.cif	cif	8	RO	
DL_POLY Configuration	*CONFIG *REVCON	dlpoly	2	RW	
EPSR ATO Files	*.ato	ato	18	RW	
GAMESS-US Output (Log)	*.log	gamuslog	11	RO	
GAMESS-US Input File	*.inp	gamusinp	5	RW	Cartesian coordinates and Z-Matrices
Gaussian Input File	*.gjf	gjf	17	WO	Cartesian coordinates and Z-Matrices
Gromacs Configuration	*.gro	gro	14	RW	
Mopac Archive File	*.arc	arc	16	RO	
Mopac Control File	*.mop	mop	4	RW	Including periodic systems. Cartesian coordinates only.
MDL Molfile	*.mol	mol	10	RO	
MSI (Cerius2) Model File	*.msi	msi	12	RO	
Protein Databank (PDB)	*.pdb	pdb	13	RO	
Quantum Espresso	*.in	in	15	RW	
SIESTA Flexible Data Format	*.fdf	siesta	9	RW	
Tripos Sybyl Mol2	*.mol2	mol2	6	RW	
XMol XYZ	*.xyz	xyz	3	RW	

Table 1-2 Supported Trajectory Formats

Format	Extension(s)	Nickname	Notes
DL_POLY Formatted & Unformatted Trajectories	*HISu *HISf HISTORY	dlpoly	
GAMESS-US Trj File	*.trj	trj	QM atoms in MD and IRC runs
PDB Trajectory (Multiple PDB file)	*.pdb	pdb	
Siesta XV	*.XV	xv	
XYZ Trajectory (Multiple XYZ file)	*.xyz	xyz	

Table 1-3 Supported Grid Data Formats

Format	Extension(s)	Nickname	ID	Notes
Gaussian Cube	*.cube	cube	1	Also an importmodel filter
Probability density	*.pdens	pdens	2	Simple 3D volumetric data format
Surface	*.surf	surf	3	Simple 2D surface data format

Table 1-4 Supported Expression Data Formats

Format	Extension(s)	Nickname	ID	Read?	Write?	Notes
DL_POLY FIELD file	*.FIELD FIELD	dlpoly	1	no	yes	
Gromacs RTP file	*.rtp	rtp	--	no	yes	Preliminary version.
Gromacs TOP file	*.top	top	14	no	yes	

2. Installation

2.1. From Precompiled Packages (All Platforms)

The easiest way to get Aten is, of course, to download a precompiled installer for your system. Installers are available for Windows, Mac OS X, and many Linux distributions through the website (<http://www.projectaten.net>) and the OpenSuSE Build Service (<http://download.opensuse.org/repositories/home:/trisyousgs/>).

2.2. From Source (Linux/Mac OS X)

If you want to compile Aten by hand (e.g. if you're keeping your own copy of the source and updating it *via* subversion) then these guides may help. They are very general and (unavoidably) quickly go out of date, but they should provide some info to at least get started. The Windows method is entirely different and somewhat more complicated than for Linux and Mac OS X, but a working approach is documented here.

2.2.1. Obtaining the Source

From the Website (as an Archive)

Go to <http://www.projectaten.net> and grab the tar.gz of the version you want, and unpack it with:

```
bob@pc:~> tar -zxvf aten-1.7.tar.gz
```

A directory called `aten-nn.mm` will be created containing all the guts of the code, ready to build, where *nn.mm* is the current version number.

From GoogleCode (with subversion)

If you want to build Aten from source and maintain a local copy so you can update it quickly to make the most of bugfixes and new features, this is the best way of doing it. You'll need to have subversion (<http://subversion.tigris.org/>) installed, since the GoogleCode repository where Aten lives is a subversion-style repository. To get yourself a copy of the latest source, run the following command:

```
bob@pc:~> svn co http://aten.googlecode.com/svn/trunk ./aten-latest
```

Afterwards, you'll have a complete copy of the source in a directory called `aten-latest`.

2.2.2. Installing Necessary Pre-requisites

Aten has a fairly modest set of external dependencies, namely Qt4 and readline. Since you're building from source, all the development files related to these packages must also be installed. The C++ (g++) compiler and the automake/libtool packages are also a necessity.

Debian and Variants

On deb-based systems (e.g. Debian and Ubuntu) a command a bit like this should install all that you need:

```
bob@pc:~> sudo apt-get install autotools-dev libtool autoconf \
    automake g++ libreadline5-dev libqt4-gui libqt4-opengl \
    libqt4-core libqt4-dev
```

RPM-Based Systems

For most other Linux flavours the method of installing the necessary software really depends a lot on your personal preferences (or your system administrators). For instance, using zypper on an OpenSuSE distribution the command is (run as root) as follows:

```
bob@pc:~> zypper in readline-devel libqt4-devel libtool automake
```

Mac OS X

On Mac OS X you have two ways of installing Qt4. Nokia offer an installable disk image, providing Qt4 as a proper Framework (see <http://qt.nokia.com/downloads>). Alternatively, Qt4 is available on Fink (`qt4-x11`). If you're using the Fink installation of Qt4, then you'll also need `pkg-config` from the same source.

2.2.3. Configure

Make / Autotools

Configure the source with the following commands, run from the top level of the `aten-latest` or `aten-nn.mm` directory:

```
bob@pc:~> ./autogen.sh
```

This creates the necessary files needed to properly configure the build. If you unpacked the source from a tar.gz, it is not necessary to run `autogen.sh`.

Next, run the `configure` script to check for all programs and files that Aten depends on, and set up the build. If you plan on installing Aten ‘properly’ (i.e. install it on your machine so it is available to all users) the `configure` script will place all binaries and necessary files in `/usr/local` by default. This default location can be overridden by using the `--prefix=<path>` option. So, the plain command is:

```
bob@pc:~> ./configure
```

To set the installation location use, for example:

```
bob@pc:~> ./configure --prefix=/home/software
```

On Mac OS X it is necessary to specify which Qt4 installation you have installed (i.e. Framework or Fink):

```
bob@pc:~> ./configure --with-qt=framework
```

or

```
bob@pc:~> ./configure --with-qt=fink
```

With the Fink installation you may also need to direct `configure` to the correct Qt4 development binaries either by setting your `$PATH` to `/sw/lib/qt4-x11/bin:${PATH}` or by running:

```
bob@pc:~> ./configure --with-qt=fink --with-qtdir=<path>
```

where `<path>` points to the location of the Qt4 development binaries. All being well, no errors should be encountered by either of these two scripts. Check out the FAQ for some commonly-encountered problems. If you get find yourself up against unresolvable issues, please email me and I'll try to help.

CMake

An out-of-tree build is best. Make a directory somewhere, enter in to it, and run:

```
bob@pc:~> cmake /path/to/source/for/aten-nn.mm
```

For example:

```
bob@pc:~> cd ~
bob@pc:~> mkdir aten-build
bob@pc:~> cd aten-build
bob@pc:~> cmake ~/src/aten-1.7
```

2.2.4. Compile

Once successfully configured, build the source with:

```
bob@pc:~> make
```

This is probably a good time to make tea or brew coffee.

2.3. From Source (Windows)

Note that these instructions were written from the perspective of a Windows XP system. It's likely that the procedure for Vista, Windows 7 etc. will be similar. First step, download and install the following:

2.3.1. Install Visual Studio C++ 2010

Get the C++ version of Visual Studio from <http://www.microsoft.com/express/Windows/> – at the time of writing this was version 10.0, but earlier versions 8.0 and 9.0 are also fine). This will download a small web installer to your machine called `vc_web.exe`. Run this, and accept the license. You may choose not to install Silverlight since it is not necessary for Aten. Remember where the installation location is set to (by default it is `C:\Program Files\Microsoft Visual Studio 10.0\`) because you'll need this later on for the installation of Windows PowerShell. Chances are you'll need to restart your machine after the installation.

2.3.2. Install the Windows SDK

Get the Windows SDK from <http://msdn.microsoft.com/en-us/windows/bb980924> (or search for ‘Windows SDK’ on the internet). Note that the download is described as being for ‘Windows 7 and .NET’ but this is fine since it’s backwardly compatible with XP. Run the installer (`winsdk_web.exe`) and accept the license, and again take note of the installation directories (since they need to be provided in the PowerShell setup later on). The default installation options are fine, although you can uncheck the installation of ‘Samples’ since they are not required. Once installation has finished, the Windows Help Centre may pop up and ask where your local resources are. This can safely be canceled.

2.3.3. Install CMake

Download the latest CMake installer from <http://www.cmake.org/> (version 2.8.4 at the time of writing) and install it. Make sure you choose to add CMake to the PATH for all users when running the installation.

2.3.4. Install Readline

Go to <http://gnuwin32.sourceforge.net/packages/readline.htm> (or search for ‘Windows Readline’ on the internet) and download the ‘Complete package, except sources’ installer (around 2.3 Mb). If you choose to install somewhere other than the default location, you’ll need to tweak the PowerShell profile given in Step 7.

2.3.5. Download GLext Header

Go to <http://www.opengl.org/registry/api/glext.h> and save the page as ‘glext.h’ somewhere like your ‘My Documents’ directory. Again, if you choose somewhere other than this location, you’ll need to tweak the PowerShell profile given in Step 7.

2.3.6. Download and Unpack Qt4 Source

Go to <http://qt.nokia.com/downloads> and download the LGPL package for Windows (approx 322 Mb) and run the installer file. Again, you may choose where to unpack the files to (the default is ‘C:\Qt\2010.05’ for the release downloaded here) but if you change the default you’ll need to modify the relevant paths accordingly in Step 7. You may choose not to install the MinGW part of the package since we will be using Visual Studio for the compilation. There is no need to run Qt Creator once the installation is finished.

2.3.7. Install and Setup Windows PowerShell

Download PowerShell from <http://support.microsoft.com/kb/968930>. Once installed, run PowerShell (its Start Menu entry is typically placed inside the ‘Accessories’ folder) and you should be presented with a blue shell, starting inside your Documents and Settings folder (C:\Documents and Settings\Your Name). First thing is to set up your profile with the relevant paths set so we can find the Visual Studio, readline, and Qt4 files. Such settings are stored in a file which doesn’t yet exist, and which is referenced by the environment variable \$profile. You can type this into PowerShell and see exactly where it points to:

```
PS C:\Documents and Settings\Your Name> $profile
C:\Documents and Settings\Your Name\My
Documents\WindowsPowerShell\Microsoft.PowerShell_profile.ps1
```

We must first create the directory where this file is expected to be:

```
PS C:\Documents and Settings\Your Name> mkdir "My Documents/WindowsPowerShell"
Directory: C:\Documents and Settings\Your Name\My Documents

Mode          LastWriteTime    Length Name
----          -----        ----- 
d---  23/04/2011      11:05           WindowsPowerShell

PS C:\Documents and Settings\Your Name> ls "My Documents"

Directory: C:\Documents and Settings\Your Name\My Documents

Mode          LastWriteTime    Length Name
----          -----        ----- 
d---  23/04/2011      10:49           Downloads
d-r-- 10/07/2009      23:41           My Music
d-r-- 08/04/2010      16:35           My Pictures
d---  14/03/2011      08:47           My Received Files
d-r-- 26/06/2010      19:44           My Videos
```

```
d---- 23/04/2011 11:05 WindowsPowerShell
-a--- 23/04/2011 17:43 637740 glext.h
```

You can then create and edit the profile file directly with Notepad:

```
PS C:\Documents and Settings\Your Name> notepad $profile
```

In the empty file, paste the following into it and adjust any paths/versions as may be necessary.

```
# Setup Visual Studio and Windows SDK environment variables
if ( test-path -path $env:VS100COMNTOOLS )
{
    echo "Setting Visual Studio environment"
    $VCSTUDIO="C:\Program Files\Microsoft Visual Studio 10.0"
    $WINSKDK="C:\Program Files\Microsoft SDKs\Windows\v7.1"

    # System variables
    $env:VSINSTALLDIR = "$VCSTUDIO"
    $env:VCINSTALLDIR = "$VCSTUDIO\VC"
    $env:FrameworkDir = "C:\Windows\Microsoft.NET\Framework"
    $env:FrameworkVersion = "v2.0.50727"
    $env:FrameworkSDKDir = "$VCSTUDIO\SDK\v3.5"
    $env:DevEnvDir = "$VCSTUDIO\Common7\IDE"

    # Executable path
    $env:PATH += ";$VCSTUDIO\Common7\IDE"
    $env:PATH += ";$VCSTUDIO\VC\BIN"
    $env:PATH += ";$VCSTUDIO\Common7\Tools"
    $env:PATH += ";$VCSTUDIO\Common7\Tools\bin"
    $env:PATH += ";$VCSTUDIO\VC\PlatformSDK\bin"
    $env:PATH += ";$VCSTUDIO\SDK\v2.0\bin"
    $env:PATH += ";$VCSTUDIO\VC\VCPackages"
    $env:PATH += ";$WINSKDK\Bin"

    # Include directories
    $env:INCLUDE += ";$VCSTUDIO\VC\ATLMFC\INCLUDE"
    $env:INCLUDE += ";$VCSTUDIO\VC\INCLUDE"
    $env:INCLUDE += ";$VCSTUDIO\VC\PlatformSDK\include"
    $env:INCLUDE += ";$VCSTUDIO\SDK\v2.0\include"
    $env:INCLUDE += ";$WINSKDK\Include"

    # Libraries
    $env:LIB += ";$VCSTUDIO\VC\ATLMFC\LIB"
    $env:LIB += ";$VCSTUDIO\VC\LIB"
    $env:LIB += ";$VCSTUDIO\VC\PlatformSDK\lib"
    $env:LIB += ";$VCSTUDIO\SDK\v2.0\lib"
    $env:LIB += ";$WINSKDK\Lib"
    $env:LIBPATH += ";$VCSTUDIO\VC\ATLMFC\LIB"
}
else
{
    echo "No Visual Studio installed, or incorrect version string used?"
}

# Setup Qt environment variables
$env:QTDIR="C:\Qt\2010.05"
if ( test-path -path $env:QTDIR )
{
    $env:PATH += ";" + $env:QTDIR + "\qt\bin"
    $env:INCLUDE += ";" + $env:QTDIR + "\qt\include"
    $env:LIB += ";" + $env:QTDIR + "\qt\lib"
}
```

```

# Setup GnuWin32 environment variables
$env:INCLUDE += ";C:\Program Files\GnuWin32\include"
$env:LIB += ";C:\Program Files\GnuWin32\lib"
$env:PATH += ";C:\Program Files\GnuWin32\bin"

# Setup custom library and header environment variables
$env:INCLUDE += ";$HOME\My Documents"

```

By default, the running of scripts in PowerShell is (likely to be) disabled, meaning that the profile will not be loaded when PowerShell next starts up. To enable the execution of scripts, run the following command:

```
PS C:\Documents and Settings\Your Name> set-executionpolicy remotesigned
```

You need to answer Yes to the question which then pops up. Afterwards, close this shell by typing:

```
PS C:\Documents and Settings\Your Name> exit
```

...and then restart PowerShell. Your new profile should now be loaded, and you should see the ‘Setting local environment’ message which we added to it. All being well, there should be no error messages here – if there are, then its likely that there’s a mistake in one of the paths set in the profile.

2.3.8. Build Qt4

First stage is to configure Qt ready for building by running the following commands:

```

PS C:\Documents and Settings\Your Name> cd C:\Qt\2010.05\qt
PS C:\Qt\2010.05\qt> ./configure -release -opensource -platform win32-msvc2010
    -qt-libjpeg -qt-libmng -qt-libtiff -qt-libpng -qt-zlib -qt-gif -no-webkit
    -no-script

```

This will take a few minutes. Once complete, build Qt with:

```
PS C:\Qt\2010.05\qt> nmake
```

This will take a few more minutes (approximately one hour, depending on your system’s speed).

2.3.9. Download and Build Aten

Go to www.projectaten.net/download and get a zipped copy of the source. For this example the latest beta version is 1.7.1626, so the zipfile is called ‘aten-1.7.1626.zip’. Unpack the source somewhere – here it has been placed in the users’s home directory – and run cmake to generate the necessary makefiles:

```
PS C:\Documents and Settings\Your Name> cd aten-1.7.1626
PS C:\Documents and Settings\Your Name\aten-1.7.1626> cmake -G "NMake Makefiles"
```

Then, build Aten by running nmake:

```
PS C:\Documents and Settings\Your Name\aten-1.7.1626> nmake
```

Time to wait again, but hopefully after a little while you will see something resembling the following (the important part is the [100%] Built target Aten):

```
C:\Program Files\Microsoft Visual Studio 10.0\VC\INCLUDE\stdlib.h(433) :
see declaration of 'getenv'
C:\Documents and Settings\Your Name\My Documents\aten-1.7.1626\src\main.cpp(52) :
warning C4996: 'getenv': This function or variable may be unsafe. Consider using
_dupenv_s instead. To disable deprecation, use _CRT_SECURE_NO_WARNINGS. See
online help for details.
C:\Program Files\Microsoft Visual Studio 10.0\VC\INCLUDE\stdlib.h(433) :
see declaration of 'getenv'
Linking CXX executable bin\Aten.exe
[100%] Built target Aten
PS C:\Documents and Settings\Your Name\My Documents\aten-1.7.1626>
```

If this is what you see, then you've built Aten successfully. You'll need to point Aten to it's data directory by hand. From the command line, you can run the following:

```
PS aten-1.7.1626> .\bin\Aten.exe --atendata .\data
```

You can also set up a shortcut to the executable and set the option there.

2.3.10. Potential Readline Errors

There is the potential for the build to fail on the basis of odd readline errors:

```
C:\Program Files\GnuWin32\include\readline/keymaps.h(97) : see
declaration of 'rl_set_keymap'
C:\Program Files\GnuWin32\include\readline/readline.h(364) : error C2375:
'rl_get_keymap' : redefinition; different linkage
    C:\Program Files\GnuWin32\include\readline/keymaps.h(94) : see
declaration of 'rl_get_keymap'
NMAKE : fatal error U1077: 'C:\PROGRA~1\MICROS~1.0\VC\bin\cl.exe' : return code
'0x2'
Stop.
NMAKE : fatal error U1077: '"C:\Program Files\Microsoft Visual Studio
10.0\VC\BIN\nmake.exe"' : return code '0x2'
Stop.
NMAKE : fatal error U1077: '"C:\Program Files\Microsoft Visual Studio
10.0\VC\BIN\nmake.exe"' : return code '0x2'
Stop.
PS C:\Documents and Settings\Your Name\My Documents\aten-1.7.1626>
```

These errors arise because the functions `rl_*_*` are declared differently between the `readline.h` and `keymaps.h` header files, but can be fixed as follows. Since keymaps are not used, the offending lines can be commented out in `readline.h`. Locate the file (by default, it is installed in `C:\Program Files\GnuWin32\include\readline/readline.h`) and edit it with Notepad or something similar. Put a comment marker `/*` at the very beginning of line 356, and a comment end marker `*/` at the very end of line 364. Rerun `nmake` and everything should be fine.

2.3.11. Anything Else?

Instead of downloading the zipped source of Aten, you could download a Windows subversion client and maintain an up-to-date copy of the source on your machine – useful if you want to frequently get the latest updates. There are many subversion clients available, but Win32svn (<http://sourceforge.net/projects/win32svn/>) works well for me.

3.Frequently Encountered Problems

What follows is a sort of mini FAQ detailing some commonly-encountered problems and issues when compiling and executing Aten for the first time, and also some of the more persistent problems that various platforms exhibit.

3.1. Configuration Errors

[ALL] ./autogen.sh: line 35: libtoolize: command not found

You need to install the `libtool` package - it should be on the original CD of your linux distro or on one of the billions of repository mirrors around the world.

[OSX] Error: Possibly undefined macro: AC_DEFINE

When running `./autogen.sh`, `autoconf` sometimes fails with ‘`configure.ac:16: error: possibly undefined macro: AC_DEFINE`’. This is related to the version of `pkg-config` you have installed (e.g. version 0.15.1 gives this error, but version 0.21 does not) with Fink / MacPorts. Upgrade to the latest version. Incidentally, the line-number reported (16) is not the actual location of the error - `autoconf` reports this wrongly (the actual error occurs later on with the ‘`PKG_CHECK_MODULES(GTK28, ..., [AC_DEFINE...]`’ command).

[OSX] Warning: Underquoted definition of PKG_CHECK_MODULES

Running `./autogen.sh`, `aclocal` complains ‘`/sw/share/aclocal/pkg.m4:5: warning: underquoted definition of PKG_CHECK_MODULES`’ If `pkgconfig` is not installed this is likely to give rise to the said spurious error. Install `pkgconfig` from Fink / MacPorts to proceed.

[ALL] autogen.sh appears successful, but aclocal complains about ‘underquoted definition of AM PATH...’

These warnings should not have affected the generation of a working `./configure` script. So you may as well move on to the next step in the build.

3.2. Compilation Errors

[ALL] Error: 'uic: File generated with too recent version of Qt Designer (4.0 vs. 3.x.x)'

This error can occur when both the Qt3 and Qt4 development packages are installed. The `$PATH` is sometimes set so that the Qt3 binaries are found first, so the Meta-object compiler from Qt3 is called, and then the generated source includes the Qt4 headers. Running `which moc` will tell you which binary is being used - you can then run this binary with the `-v` option to check the version. If its version 3.x.x then you will need to reconfigure the build as follows.

Each of the three Qt4 utilities needed to compile Aten – `moc`, `rcc`, and `uic` – can be specified explicitly with three configure options; `--with-qtmoc`, `--with-qtuic`, and `--with-qtrcc`. Typically, the Qt3 binaries are located in `/usr/lib/qt3/bin` while the Qt4 binaries are in `/usr/bin`,

but this may not be the case on your particular system. For the sake of argument, let's say they are, then the configure command will be run as follows:

```
bob@pc:~/src/aten> ./configure --with-qtmoc=/usr/bin/moc --with-qtuic=/usr/bin/uic  
--with-qtrcc=/usr/bin/rcc
```

[ALL] Error: 'This file was generated using the moc from 3.X.X. It cannot be used with the include files from this version...'

Again, this is usually related to both Qt3 and Qt4 development tools being installed simultaneously. See the previous question and its solution above.

[OSX] Undefined reference to __stdoutp expected to be defined in /usr/lib/libSystem.B.dylib

Chances are you have an older operating system - Panther (10.3.9) is confirmed to show this error. Since the Universal Mac binaries are built on a machine with Snow Leopard (10.6) this error can only be avoided by upgrading your operating system or compiling Aten by hand.

3.3. Usage Errors

[LINUX] I saved an image from the GUI and it was completely black / corrupt. Why?

A common issue with (integrated) Intel graphics chips, but may affect others. There are some issues with the Qt code used to grab the current view as an image, but an alternative method is available and works. Try the prefs option 'aten.prefs.useframebuffer = TRUE;' to use the alternative method. If this remedies the problem, add the line to your preferences file. One caveat - although this appears to fix problems when saving images from the GUI, it does not (apparently) solve the problem when saving the image from the command line.

[ALL] Why is Aten's main view corrupt/black? I can't see anything!

This is a fairly common issue, and the exact causes are not known for sure. Certainly on some Linux machines the presence of an ATI graphics card seems to be related. Using the `radeon` or `fglrx` drivers supplied with the OS rather than the proprietary `ati` driver sometimes helps.

Forcing the main view widget to manually refresh itself usually solves the problem. To do this on the latest versions, activate **Settings→Manual Swap Buffers** from the main menu (on newer versions) or type the following into Aten's command window:

```
aten.prefs.manualswapbuffers = TRUE;
```

If this remedies the problem, add the line to your user preferences file (see Section 4.2.2).

[WINDOWS] Aten fails to start, giving this message:



This descriptive (!) message suggests that the installation process failed, but this is not the case. The real reason for Aten's failure is that your machine lacks the Visual Studio C++ 2008 runtime libraries. These can be freely downloaded from Microsoft's website:

<http://www.microsoft.com/downloads/en/details.aspx?familyid=a5c84275-3b97-4ab7-a40d-3802b2af5fc2&displaylang=en>

Allternatively, search the internet for 'Microsoft Visual C++ 2008 Redistributable Package SP1'. Note that it **must** be the Service Pack 1 version.

4.Quickstart

The following sections give an overview of the most important things to know about Aten. It's only a few pages long, and its well worth taking five minutes to read it.

4.1. Terminology

It is useful to know exactly what a few terms mean in Aten's world:

Model

A model is a single molecule, a snapshot of an ensemble of molecules in a liquid, a crystal's unit cell – basically, any collection of atoms and bonds, optionally including a unit cell definition and/or a number of glyphs (annotations or shapes).

Pattern

A system containing many molecules can be described (and manipulated) efficiently through the recognition that there are sets of molecules of the same type. A pattern describes one such set of similar molecules, and a model's pattern definition may contain many individual patterns. Many operations in Aten require that a pattern definition be present, and one is automatically created as and when necessary. See Section 10.3 for more information.

Filter

A filter is a set of commands (i.e. a program) which loads data in to or saves data from Aten. Aten depends on filters to be able to load and save models, forcefields, expressions, trajectories, and grid data. All filters are written in Aten's own scripting language (which is based syntactically on C) and are loaded in on startup. New filters can be added at will (by the user) to cater for specific formats. See Section 11 for more information.

Expression

While a forcefield is a collection of terms (bonds, angles, van der Waals terms, etc.) which describe (usually) a large number of molecular types and systems, an expression is a subset of terms specific to one model.

NETA

Aten is able to automatically assign forcefield atom types to atoms in a model through the use of type descriptions in the Nested English Typing of Atoms (NETA) language. This is a simple, readable system for describing the connectivity and environment of individual atoms. See Section 12.5 for more information.

Fragment

A fragment is a molecule or structure which can be used to quickly build up a new model, or modify an existing one. For example, cyclohexane, or a tertiary butyl group.

4.2. File Locations

4.2.1. Installed / Provided Files

Aten depends on several sets of files in order to function properly and, generally-speaking, knows where to look for them. Sometimes, however, you may need to tell Aten where these files are (e.g. if you have not installed Aten after compiling the source yourself). There are several ways of achieving this. When running Aten from the command-line, the `--atendata` switch can be used to specify the location of Aten's data files. For instance:

```
bob@pc:~> aten --atendata=/usr/software/aten/data
```

Alternatively, the environment variable `$ATENDATA` can be set. For example, in a bash-style shell:

```
bob@pc:~> export ATENDATA=/usr/software/aten/data
```

In both cases, Aten should be directed to the `data` directory; either it's installed location or the directory in the top level of the source distribution.

The structure of the data directory is as follows:

data/filters	Contains stock filters for import and exporting data
data/fragments	Fragment models for drawing / modifying models
data/ff	Forcefield files
data/fftesting	Forcefields that are incomplete or have not been tested
data/partitions	Partition data for the disorder builder
data/test	Various test files (models, grids etc.) (not installed)

4.2.2. User Files

Aten will search for additional filters, fragments, and forcefields in a specific location in the user's home directory. On Linux and Mac OS X systems this directory is called `.aten`, while on Windows the directory should be called simply `aten` (i.e. without the preceding dot). Within this directory exists (optionally) further directories named similarly to those in the `data` directory, in which user files of the relevant type should be located.

Finally, the two main preferences files are located in the user's `.aten` (or `aten`) directory. Both are optional. The first, `prefs.dat` (or, alternatively, `prefs.txt`) is written by Aten from the Prefs window. While this file may be modified by hand, changes will be lost if overwritten by Aten. The second file, `user.dat` (or, alternatively, `user.txt`) is maintained entirely by the user, and should be used to change or set up exotic situations and preferences. For instance, a specific forcefield could be loaded automatically on startup.

4.3. Referencing Aten

Aten has been published in the Journal of Computational Chemistry, and the article can be found online [here](#). The full reference is:

“Aten - An application for the creation, editing, and visualization of coordinates for glasses, liquids, crystals, and molecules”, T. G. A. Youngs, *J. Comp. Chem.* **31**, 639-648 (2010).

It is not a requirement to cite Aten in your work, but if you feel that Aten has been particularly useful then the credit would be nice.

5. Usage Examples

5.1. Creating a Bulk Water Model (GUI)

Bulk (periodic) systems are the staple diet of the molecular dynamicist, particularly systems that are isotropic. This example illustrates how to generate a simple bulk configuration of liquid water.

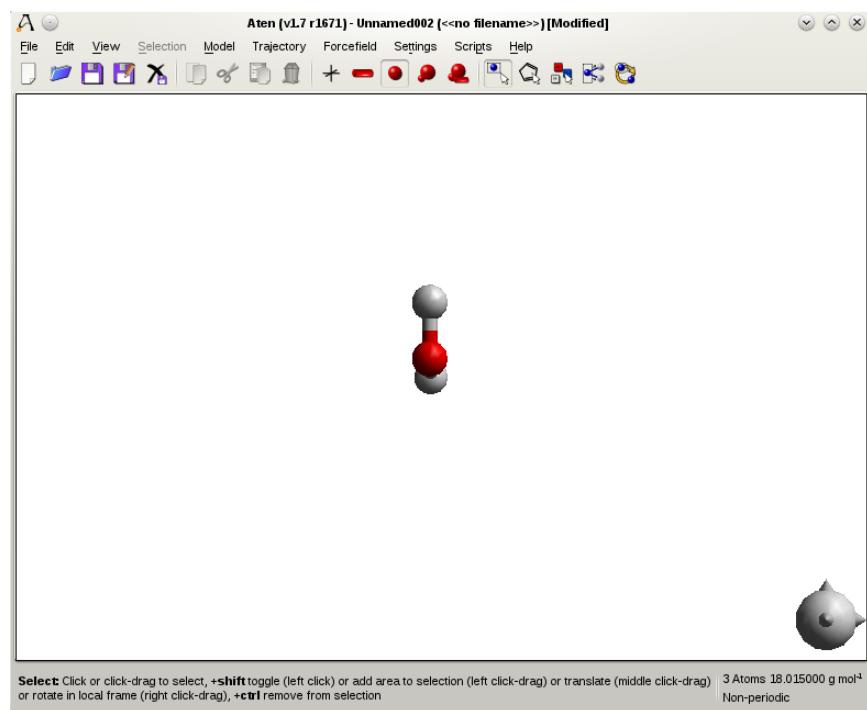
Make a Water Molecule

We need a water molecule. We *could* load one in, but its marginally more interesting to build one from scratch. So, we'll place an oxygen atom down somewhere and automatically add hydrogens to it.

Main Toolbar Create a new, empty model (if you don't have one already) with  In the **Build Window** change the active element to oxygen on the Edit page by clicking 

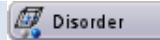
Select the draw single atoms tool  and click once somewhere in the empty model to draw an oxygen atom

Add hydrogens to the model by clicking  **AddH Model**

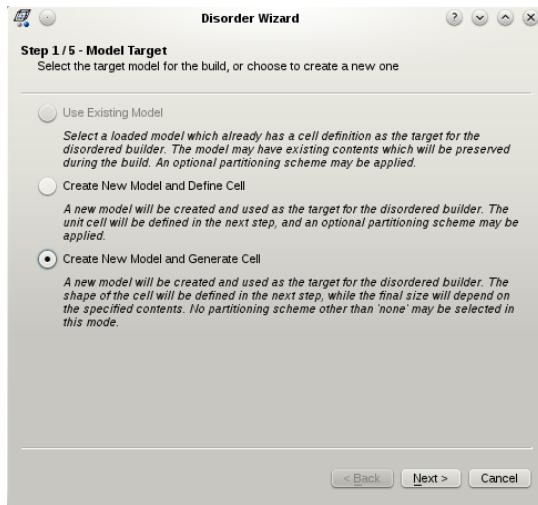


Run the Disorder Builder

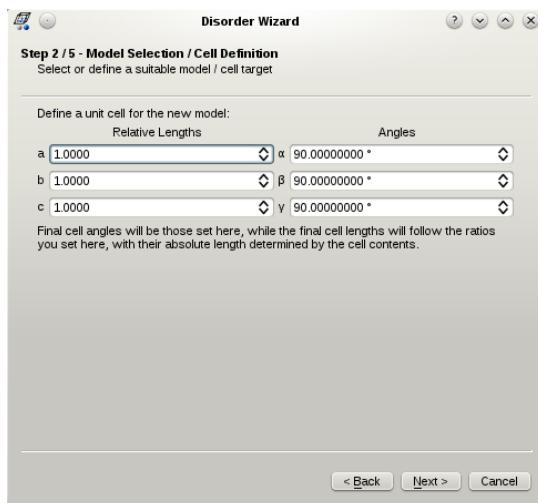
The Disordered Builder adds molecules into a new or existing model with a unit cell. The only requirement of the Disorder Builder is that all the models you wish to add must be loaded into Aten before the builder is run. Everything else is handled by a wizard which guides you through the process.



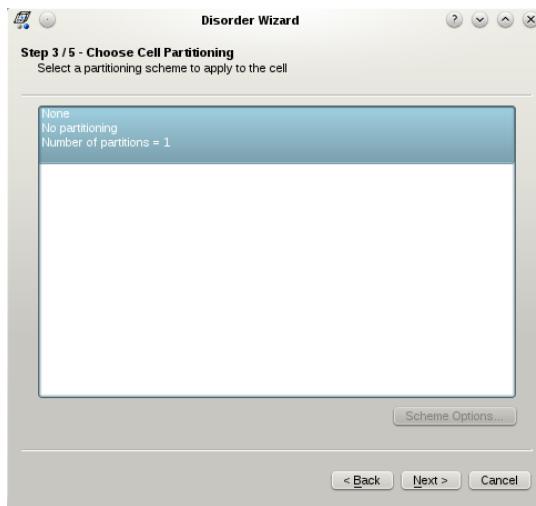
Run the Disorder Builder wizard



Step 1 is to select either an existing model (which has a unit cell) or choose to create a new one. Since the water molecule is the only one loaded in this example, the existing model option is grayed out in the graphic above. When creating a new model, there are two options for the generation of the unit cell – either you can specify the cell lengths and angles explicitly to get exactly the cell you want, or you can specify just the angles and *relative* lengths of the cell, which will then be enlarged automatically to contain whatever you choose to put inside it. For this example, we will do the latter, so select the last option and click **Next**.



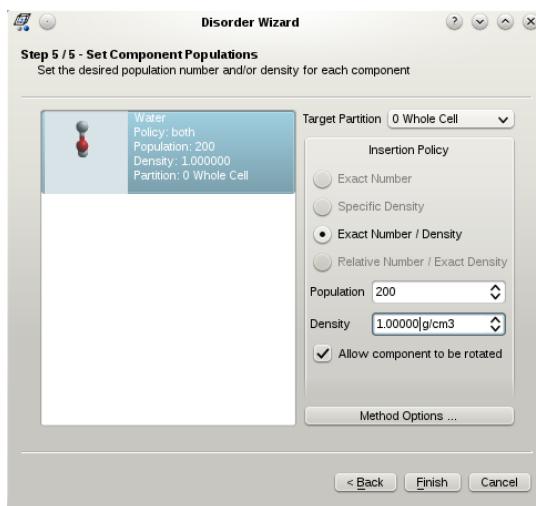
As mentioned, here we define exactly the angles of the cell, but any lengths we specify are relative lengths which will be scaled according to the number and density of the molecules we choose to put in the cell. We will leave them as they are for now, so we will end up with a cubic cell with some side length l .



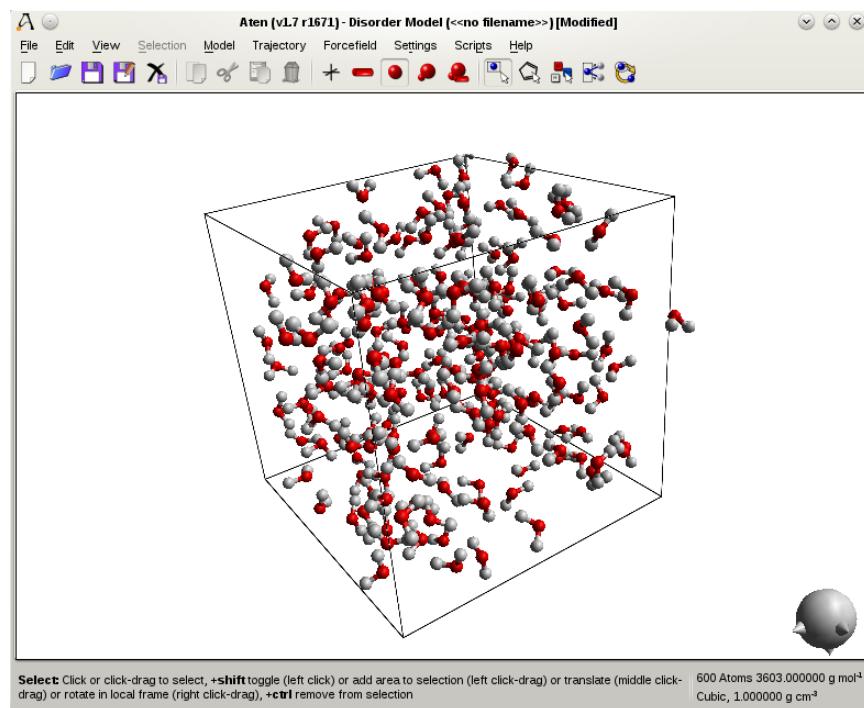
The Disorder Builder normally permits a partitioning scheme (see Section XXX) to be selected in order to partition molecules up into different regions in the cell. When generating a cell, however, this is not possible, so we must choose the basic cell option and click **Next**.



Next we must choose which models or *components* we wish to add into the new system. Here there is only one choice, the water model we created earlier, so select it and press **Next**.



Finally, the most important step of the wizard is to define a policy for the insertion of each model we selected in the last step. There are usually four options here; either a specific number of molecules or a specific density (more correctly a specific *partition* density), and exact number and density, or a relative number and exact density. Since our cell size is to be automatically determined, only one of those four options is available, **Exact Number / Density**, since both pieces of information are required in order to work out the final cell volume. So, choose the number of water molecules to add, say 200, and the density you want them to have in the cell. Press **Finish** and the system will then be built.



Script

When run from the command line the disorder builder always requires a model with a unit cell to be defined – only through the GUI is the model created automatically. This script defines a basic cubic cell in a new model before the disorder builder is run with the second argument set to FALSE (which indicates that the cell size is not fixed and should be adjusted to suit the defined molecule contents) and also minimises the system a little before saving out coordinates and a forcefield expression for DL_POLY.

```

newModel("water");
newAtom(O);
addHydrogen();
setupComponent("both", 1, 200, 1.0);

newModel("box");
cell(1.0,1.0,1.0,90,90,90);
disorder("None", FALSE);

loadFF("spce.ff");
mcMinimise(20);
sdMinimise(20);

saveExpression("dlpoly", "water.FIELD");

```

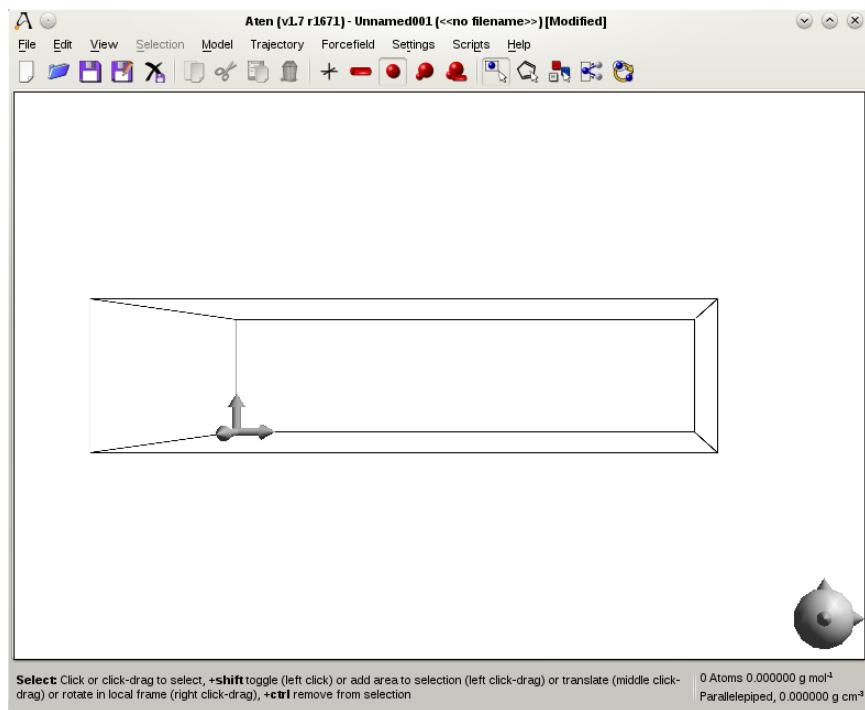
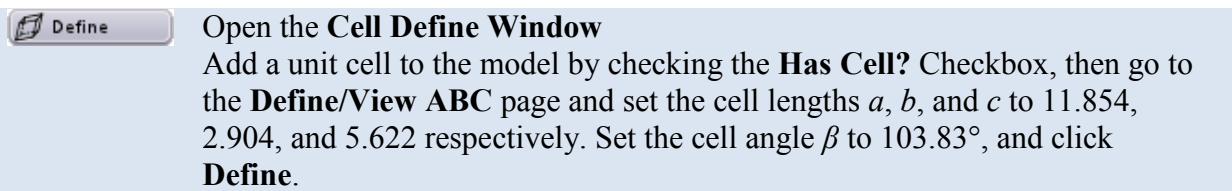
```
saveModel("dlpoly", "water.CONFIG");  
quit();
```

5.2. Building θ -Alumina from Basic Crystal Data

Crystal structures are useful for many things, provided you can find them in electronic format. When you can't and are left with the bare crystallographic data in a paper things are slightly more troublesome. This was exactly my experience when working with θ -alumina – the crystal information is readily available in a paper by Zhou and Snyder (*Acta. Cryst. B*, **47**, 617 (1991)), and here's how to make use of it.

Create the Unit Cell

First, we create the crystal cell in which to add our atoms. From the paper we see it is a monoclinic cell with $\beta = 103.83^\circ$, and side lengths $a = 11.854$, $b = 2.904$, and $c = 5.622 \text{ \AA}$. You may want to zoom out afterwards to get a proper look at the new cell.



Add Symmetry Unique Atoms

There are five symmetry-unique atoms to add into the cell, which will in turn be used to generate the remaining 35 symmetry-related atoms to give the complete unit cell. To add the atoms we will use the **Add Atom** method in the editing tools available in the **Build Window**. Atom positions in the paper are given in fractional coordinates - we will create the atoms using these coordinates which will be converted to their 'real' equivalents by Aten (according to the current cell) as we add them.

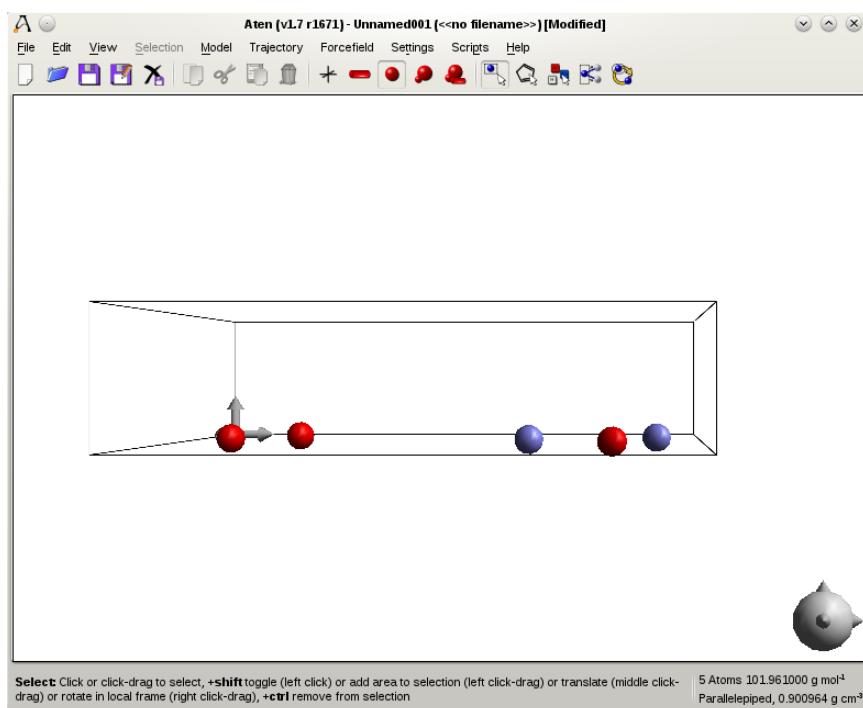


In the **Build Window** change the active element on the **Edit** page to aluminium by clicking the periodic table button and selecting it there. On the **Add Atom** panel in the **Tools** page make sure **Fractional Coordinates** is checked and then enter the following sets of coordinates, clicking **Add** after each set is entered:

$x = 0.9166, y = 0.0, z = 0.2073$
 $x = 0.6595, y = 0.0, z = 0.3165$

Go back to the the **Edit** page and change the active element to oxygen by clicking the button. Then, return to the **Tools** page and add three more atoms at the following fractional coordinates:

$x = 0.8272, y = 0.0, z = 0.4273$
 $x = 0.495, y = 0.0, z = 0.2526$
 $x = 0.1611, y = 0.0, z = 0.0984$

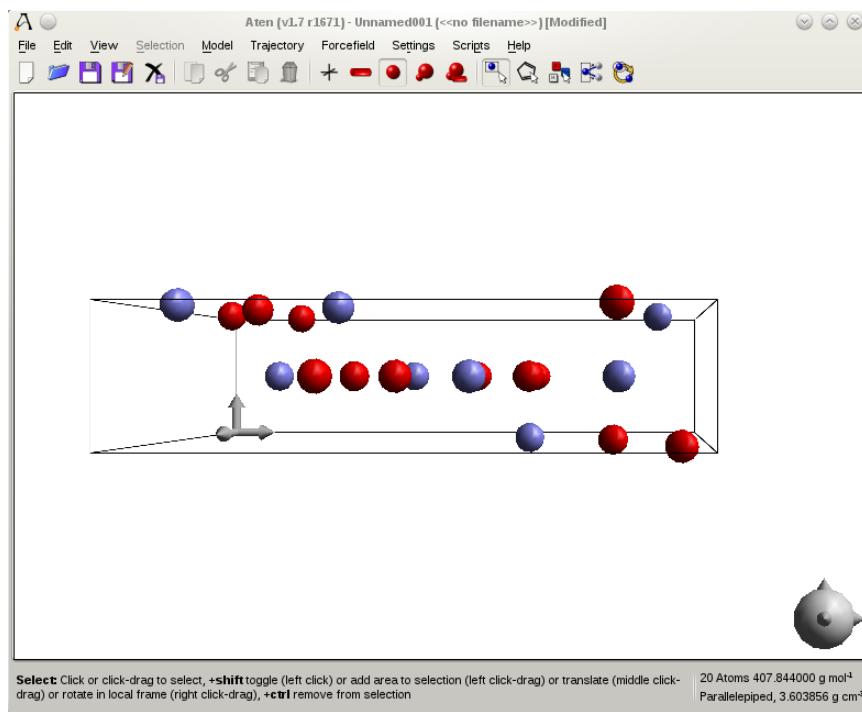


Assign the Spacegroup and Pack

To complete the model the spacegroup of the crystal must be set, so that generation of the symmetry-related atoms can be performed. This is all done on the **Cell Define** window. The spacegroup can be entered as either its numeric ID (as listed in the IUC Tables) or as the unformatted spacegroup name - θ -alumina belongs to spacegroup C2/m (number 12). Symmetry-related copies of the five atoms present in the cell can then be generated. Any overlapping atoms resulting from the application of the spacegroup's generators are automatically removed.



Open the **Cell Define Window** again, and on the **Spacegroup** page enter the spacegroup as “C2/m” or “12” and press **Set** to assign the spacegroup to the model. Then, generate symmetry equivalent atoms by pressing the **Pack** button



Replicate Cell and Recalculate Bonds

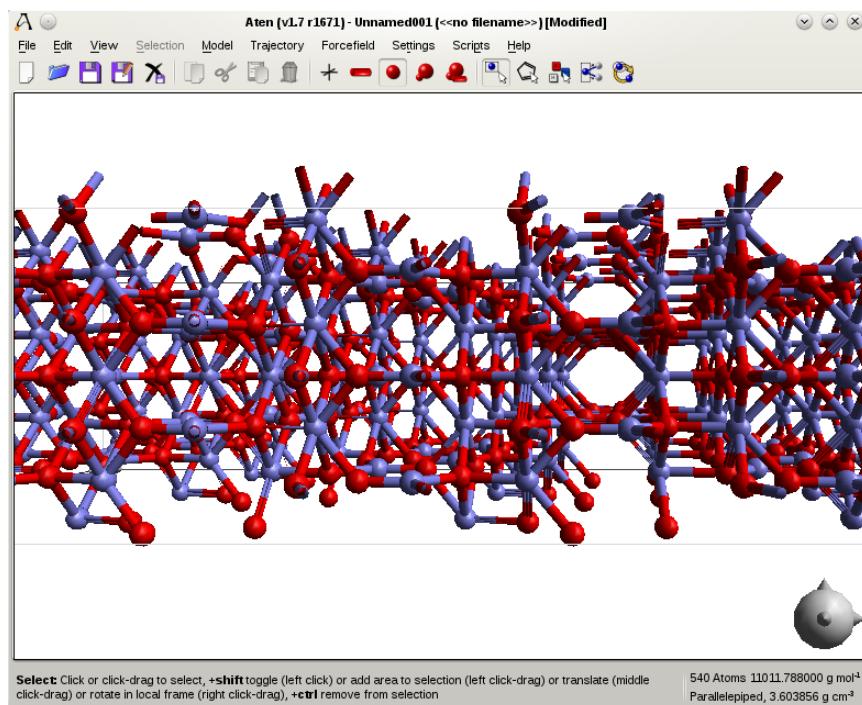
The basic crystal unit of θ -alumina isn't too interesting on its own, so let's create a chunk to properly see the structure. Replicate the unit cell by 3 or 4 units in each positive direction (or more if you're feeling adventurous) and then calculate the bonding in the model.



Open the **Cell Transform Window** and on the **Replicate** page set each positive replication direction to 3 or 4. Press the **Replicate** button to generate the new supercell.



Back in the **Build Window**, go to the **Edit** page and **Rebond** the model.



Script

```
newModel("alumina");
cell(11.854, 2.904, 5.622, 90, 103.83, 90);
newAtomFrac(Al,0.6595,0,0.3165);
newAtomFrac(Al,0.9166,0,0.2073);
newAtomFrac(O,0.8272,0,0.4272);
newAtomFrac(O,0.495,0,0.2526);
newAtomFrac(O,0.1611,0,0.0984);
spacegroup("C2/m");
pack();
replicate(0,0,0,3,3,3);
rebond();
```

5.3. Creating an NaCl/Water Two-Phase System (GUI)

It's often necessary to create a larger system from a simple (or complex) unit cell, for example to generate bulk supercells of crystals, surfaces etc. The method outlined below shows how to do this for a simple crystal, and then extends this system to create a solid liquid interface.

Create the Template Model

First off, we will create a basic FCC template model which has a unit cell of exactly 1 Å, with atoms at {0,0,0}, {0.5,0.5,0}, {0.0,0.5,0.5}, and {0.5,0.0,0.5}, representing the basic positions of atoms in a face-centred cubic lattice.

 **Define**

Open the Cell Define Window
Add a unit cell to the model by checking the **Has Cell?** checkbox, then go to the **Define/View ABC** page and set the cell lengths a , b , and c to 1.0, then click **Define**.

 **Build**

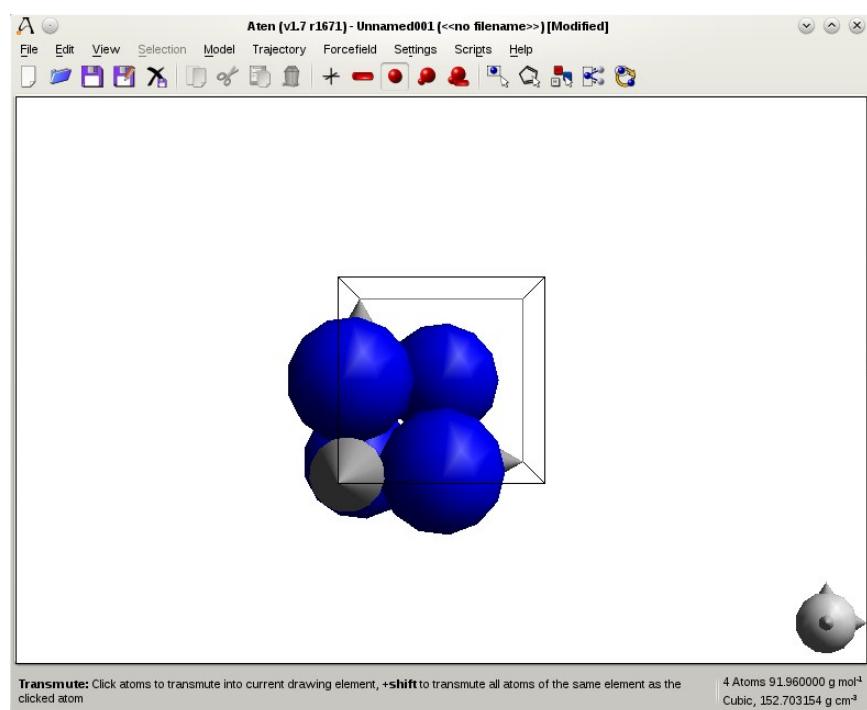
Add the following four atoms to the model at the coordinates specified.
Change the current element to Na with  **Pick** on the **Edit** page in the **Build Window**, and use the **Add Atom** panel in the **Tools** page to create the atoms.

$x = 0.0, y = 0.0, z = 0.0$

$x = 0.5, y = 0.5, z = 0.0$

$x = 0.0, y = 0.5, z = 0.5$

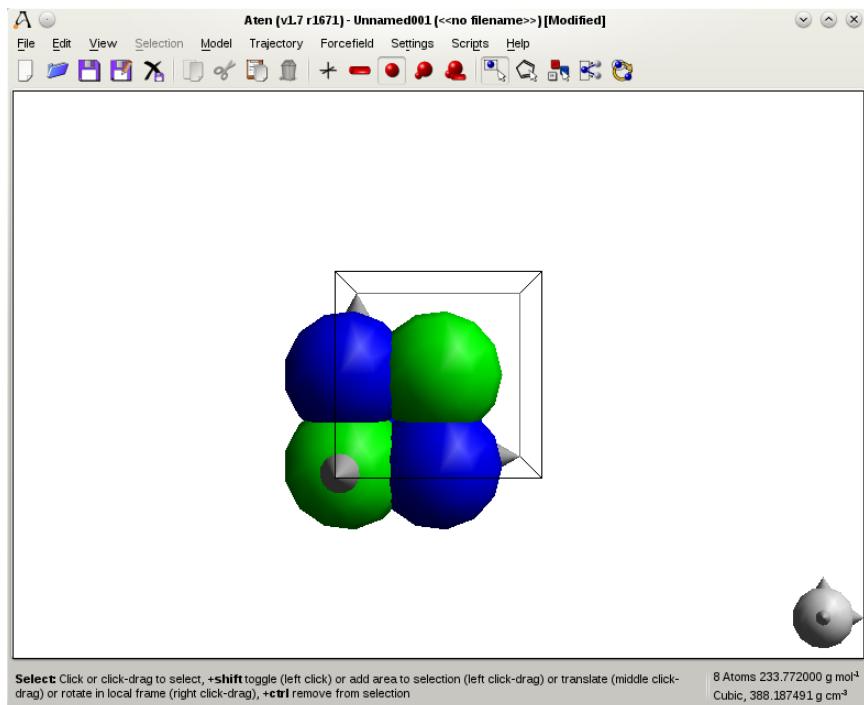
$x = 0.5, y = 0.0, z = 0.5$



Create Interpenetrating Secondary Lattice

We need a second FCC lattice on which the chlorine atoms will sit. We could add them all by hand, but there are easier ways.

Ctrl-A	Select all the atoms in the model with the Ctrl-A shortcut, or go to the Edit menu and choose Select All .
Ctrl-C	Copy the current atom selection with Ctrl-C , or Edit→Copy .
Ctrl-V	Paste the copied atoms with Ctrl-V or Edit→Paste .
Position	Open the Position Window and select the Flip page. Here we can mirror the coordinates of the current atom selection about its centre of geometry. So, mirror it once in any one of the x, y, or z directions (, , or). Do not deselect the atoms afterwards, since we still need to transmute them into chlorines.
Build	On the Edit page in the Build Window , set the current element to chlorine with Pick , and then click Transmute Sel .



Scale and Replicate Unit Cell

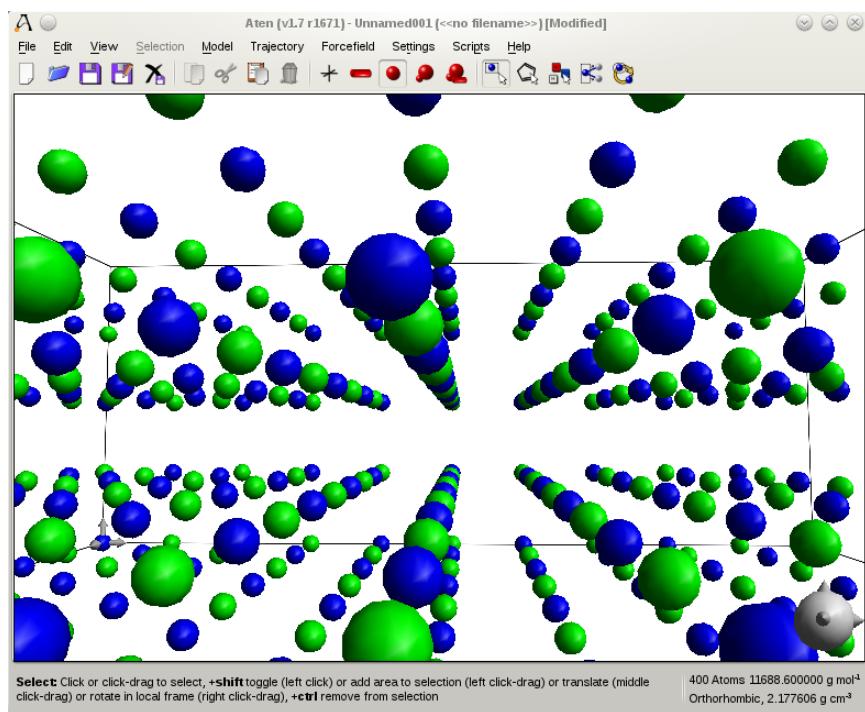
We wish to scale the cubic unit cell of the model (currently with side length $l = 1.0 \text{ \AA}$) to correspond to the unit cell of sodium chloride ($l = 5.628 \text{ \AA}$). The cell can be scaled by a different amount in each Cartesian axis, but since we want to end up with a cubic cell we must scale each axis by the same amount.

Once the model represents proper the basic sodium chloride unit cell we can replicate it to create a larger system. Aten can replicate a basic cell by any integer or non-integer amount along each of the three principal cell axes, but here we will stick to integer amounts. The Replicate method also allows specification of both negative and positive replication amounts for each direction. Note that the values given in the **Replicate** page represent the *total* size which we require, so input values (negative/positive) of {0,0,0} and {1,1,1} will result in an unchanged cell.

Transform	On the Cell Transform Window go to the Scale page and enter a scale factor
------------------	--

of 5.628 for each of x , y , and z . Press the **Scale** button to resize the unit cell.

Now, select the **Replicate** page and enter positive replication values of 5.0 for both x and z , and 2.0 for y , and press **Replicate** to create the supercell.



Create an Interface

It's a simple job to create an interface from the current system - all we need do is increase the cell dimension along the y direction (the cell's b length).



Open the **Cell Define Window**, and either change the central number of the **Matrix** page or the b value on the **Define/View ABC** page. If you replicated the y direction by 2.0 earlier, then the current value of b should be 11.238 Å. Change it to 22.5 to roughly double the volume of the cell and create an interface in the xz plane.

Make a Water Model

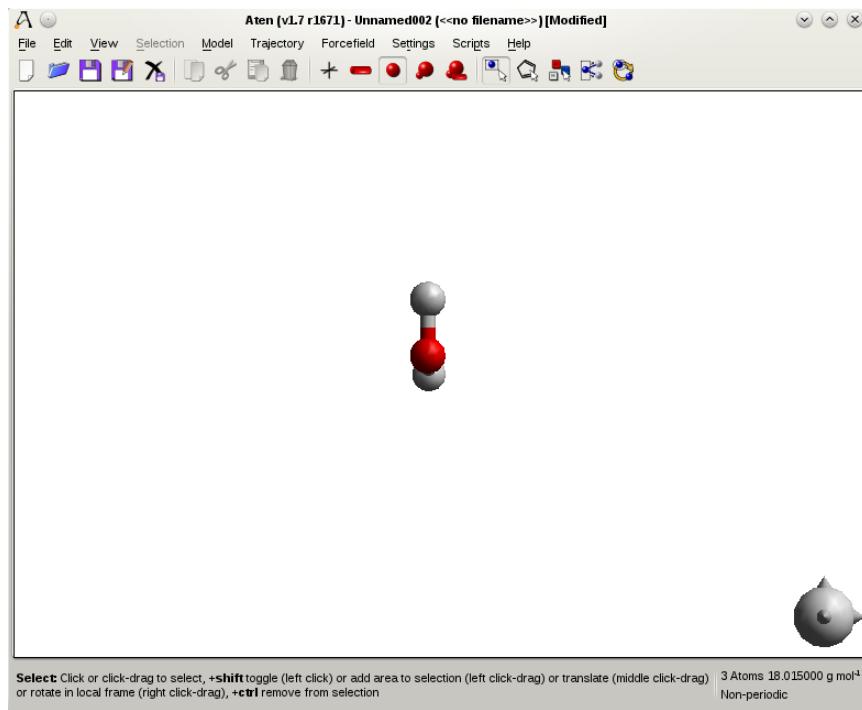
We will now create a water molecule in a separate model so we can add it in to the NaCl cell using the Disorder Builder.



Create a new, empty model (if you don't have one already) with . In the **Build Window** change the active element to oxygen on the **Edit** page by clicking

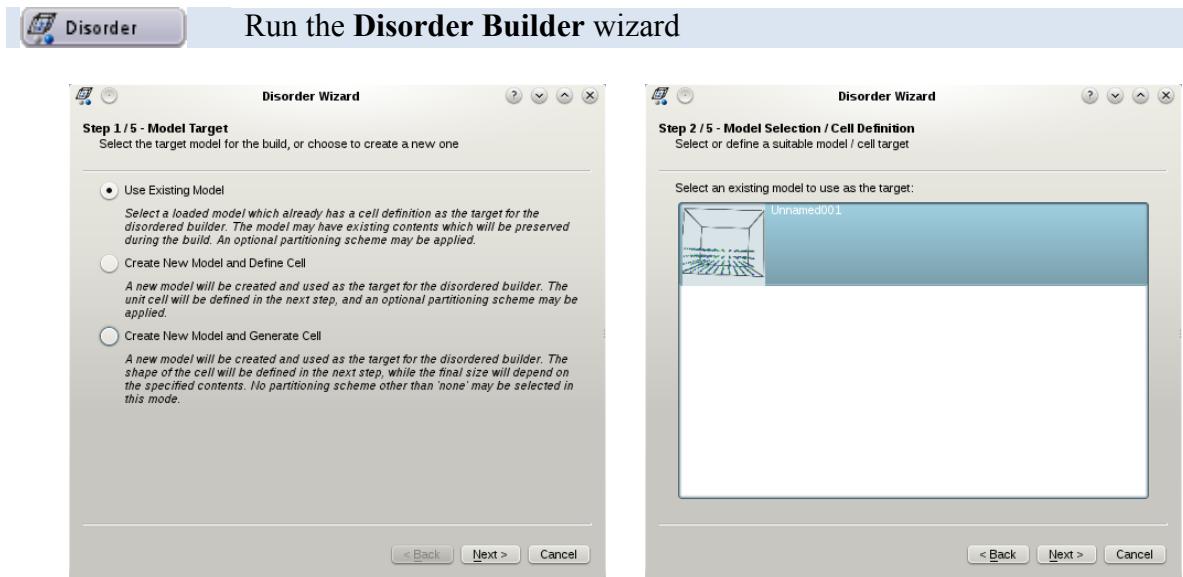
Select the draw single atoms tool **Atom** and click once somewhere in the empty model to create an oxygen

Add hydrogens to the model by clicking **AddH Model**

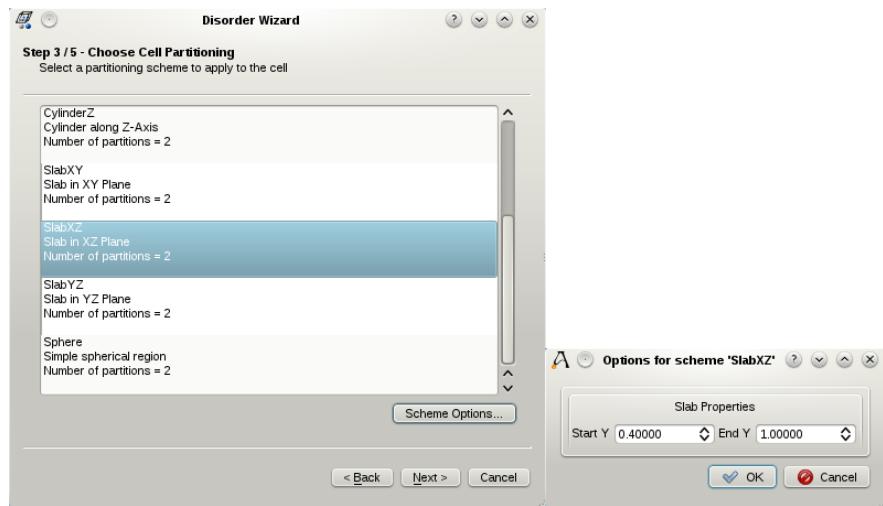


Add Water to the NaCl Cell

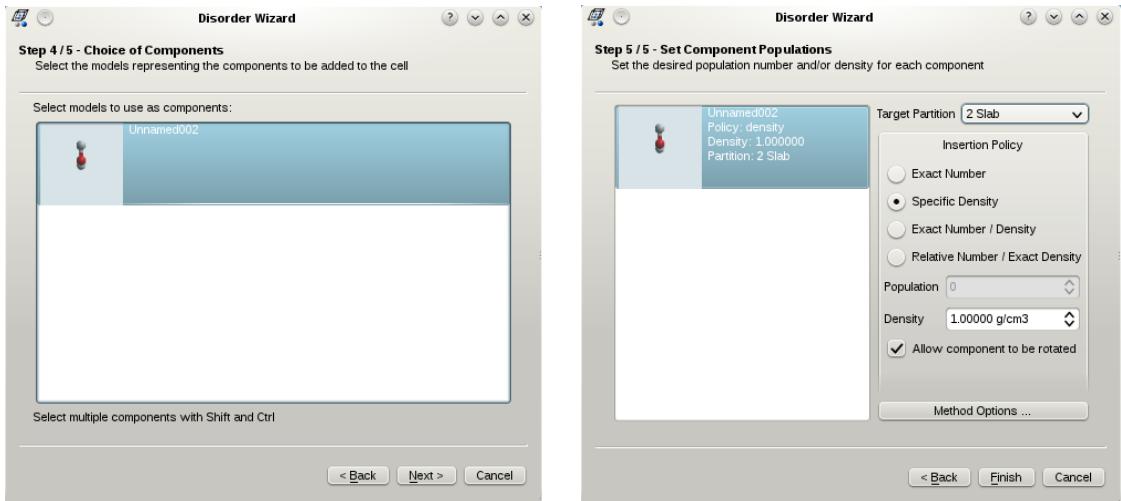
It's time to run the disorder builder on the system. We'll instruct the builder to add water in to the extended NaCl cell, but only into the part which is empty.



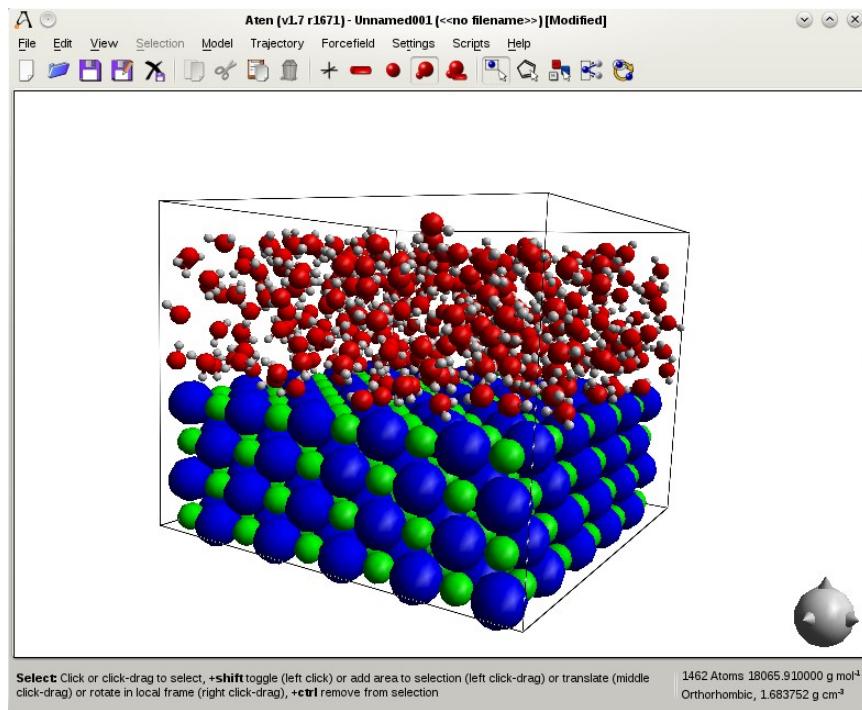
We wish to add in water to an existing system in this example, so choose the top option (**Use Existing Model**) and press **Next**. You then need to choose the target model for the builder, which is the extended NaCl system we just created. Select it and press **Next**.



Now we choose the partitioning scheme for the system. We could be lazy and just choose “None”, since the NaCl lattice should ‘reject’ any water molecule we attempt to add over it. However, here we will choose an appropriate partitioning scheme for the task, “SlabXZ”. This will allow us to restrict water molecules to a specific y -range in the unit cell. Select “SlabXZ” and press the **Scheme Options** button to bring up the options dialog for the scheme. There you will see the start and end values of y for the slab (in fractional cell coordinates). The initial minimum limit of 0.4 is, luckily, appropriate for the system, but the upper bound needs to be set to 1.0. Press **OK** when done and then **Next**.



Finally, model selection and preparation. Select the water molecule from the list and press **Next** to get to the component setup page. We must change the **Target Partition** of the water molecule to 2 (the slab we defined earlier), and then request that a **Specific Density** of molecules be inserted into this partition (the default of 1.0 g cm^{-3} is fine). Once this information has been entered, press **Finish** to start the build.



Script

The script below creates the water molecule as the first step to make life a little easier.

```

newModel("Water");
newAtom(O);
addHydrogen();
setupComponent("density", 2, 0, 1.0);

newModel("fcc");
cell(1,1,1,90,90,90);
newAtom(Na,0,0,0);
newAtom(Na,0.5,0.5,0);
newAtom(Na,0.5,0,0.5);
newAtom(Na,0,0.5,0.5);
selectAll();
copy();
paste();
flipX();
transmute(Cl);
scale(5.628,5.628,5.628);
replicate(0,0,0,5,2,5);
setCell("b",28.12);
disorder("SlabXZ,end=1.0");

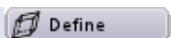
```

5.4. Building Ice I_h from Crystal Information (GUI)

Similar in spirit to the alumina example (Section 5.2), here we create a single ice I_h crystal from the crystal information, and then replicate it to form a larger supercell. The crystal information used below is from Leadbetter *et al.*, *J. Chem. Phys.*, **82**, 424 (1985), Table II (Structural parameters of ice at 5 K).

Create the Basic Unit Cell

First, we create the unit cell, which from the paper is orthorhombic with side lengths $a = 4.5019$, $b = 7.7978$, and $c = 7.328 \text{ \AA}$. There are five symmetry-unique atoms to add into the cell. This might seem odd given that this doesn't add up to a whole number of water molecules, but one of the water molecules lies with its oxygen on a mirror plane, and so only needs one hydrogen to be specified. Atom positions in the paper are given in fractional coordinates - we will create the atoms using these coordinates which will be converted by Aten into their real (cell) coordinates automatically.



Open the Cell Define Window

Add a unit cell to the model by checking the **Has Cell?** Checkbox, then go to the **Define/View ABC** page and set the cell lengths a , b , and c to 4.5019, 7.7978, and 7.328 respectively. Leave the cell angles all at 90° , and click **Define**.



In the **Build Window** change the active element on the **Edit** page to oxygen by clicking the button.

On the **Add Atom** panel in the **Tools** page make sure **Fractional Coordinates** is checked and then enter the following sets of coordinates, clicking **Add** after each set is entered:

$x = 0.0, y = 0.6648, z = 0.0631$

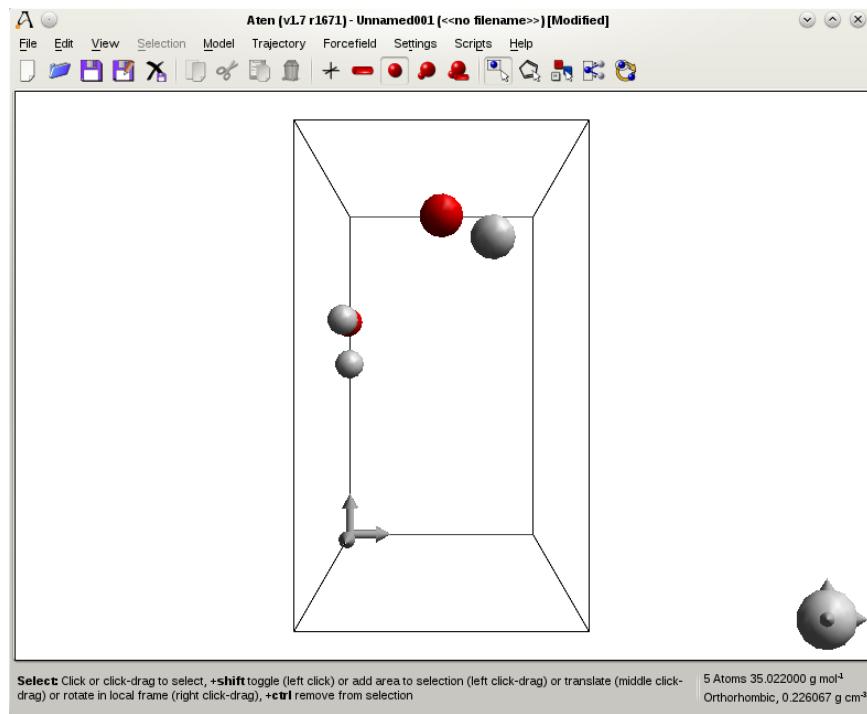
$x = 0.5, y = 0.8255, z = -0.0631$

Go back to the the **Edit** page and change the active element to hydrogen by clicking the button. Then, return to the **Tools** page and add three more atoms at the following fractional coordinates:

$x = 0.0, y = 0.6636, z = 0.1963$

$x = 0.0, y = 0.5363, z = 0.0183$

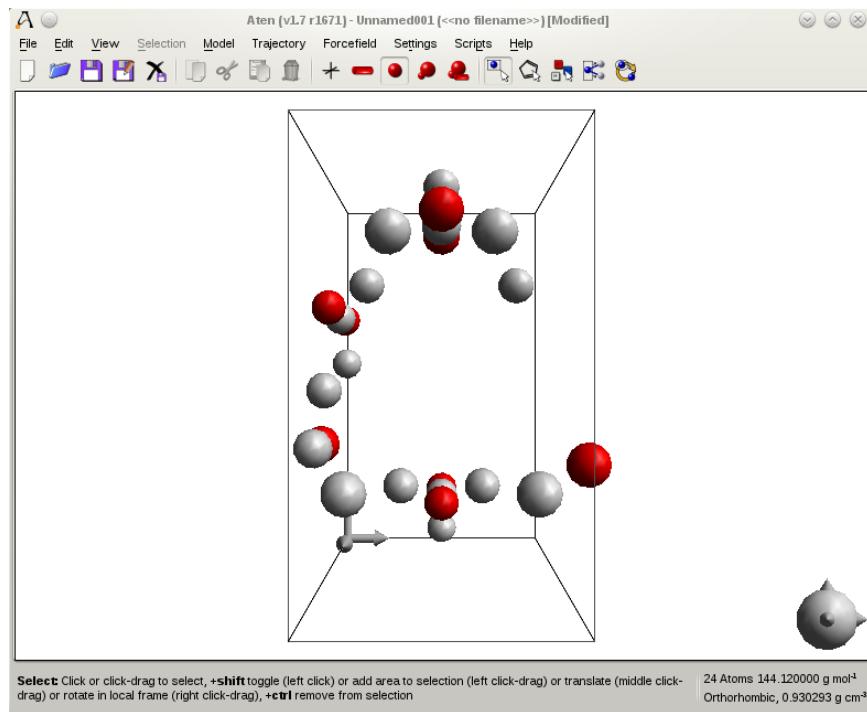
$x = 0.6766, y = -0.2252, z = -0.0183$



Set the Spacegroup and Pack

To complete the model the spacegroup of the crystal must be set so that generation of the symmetry-related atoms can be performed.

Open the **Cell Define Window** again, and on the **Spacegroup** page enter the spacegroup as “Cmc21” or “36” and press **Set** to assign the spacegroup to the model. Then, generate symmetry equivalent atoms by pressing the **Pack** button

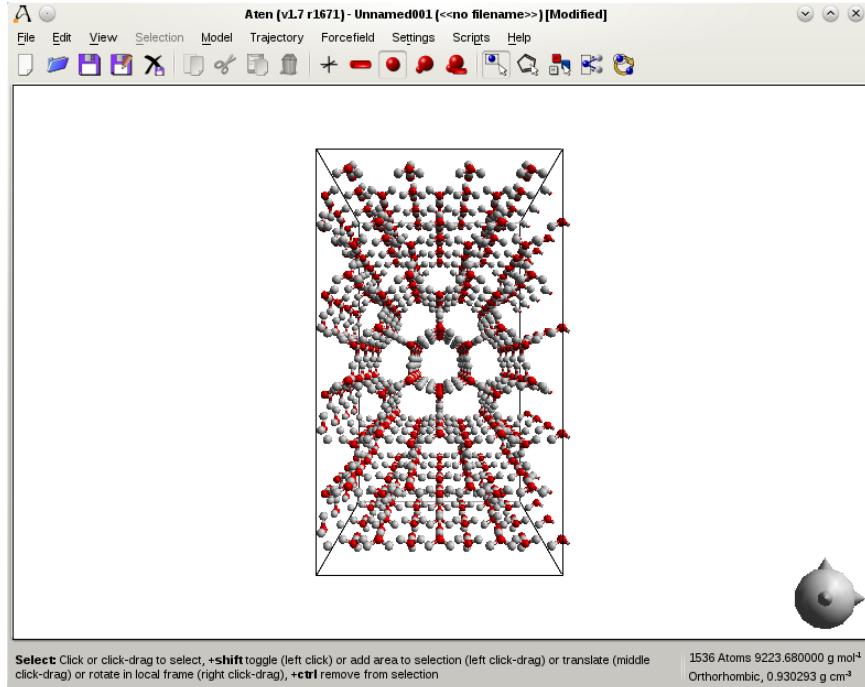


Replicate Cell and Calculate Bonding

The basic cell of ice I_h isn't particularly interesting by itself, so we will replicate the cell to create a larger supercell, and then calculate bonds in the new model so that the hexagonal structure is clear to see.

Transform On the **Cell Transform Window** go to the **Scale** page and enter a scale factor of 5.628 for each of *x*, *y*, and *z*. Press the **Scale** button to resize the unit cell.

Build Now, select the **Replicate** page and enter positive replication values of 5.0 for both *x* and *z*, and 2.0 for *y*, and press **Replicate** to create the supercell. Back in the **Build Window**, go to the **Edit** page and **Rebond** the model.



Script

```
newModel("ice");
cell(4.5019, 7.7978, 7.3280, 90, 90, 90);
newAtomFrac(O, 0, 0.6648, 0.0631);
newAtomFrac(O, 0.5, 0.8255, -0.0631);
newAtomFrac(H, 0, 0.6636, 0.1963);
newAtomFrac(H, 0, 0.5363, 0.0183);
newAtomFrac(H, 0.6766, -0.2252, -0.0183);
spacegroup("Cmc21");
pack();
replicate(0, 0, 0, 4, 4, 4);
rebond();
```

5.5. Exporting Coordinates in Bohr (Filters)

Aten works in units of Angstroms, but of course this does not suit every other computational code out there. Many physics code allow (or require) input in units of Bohr. The simplest way of generating coordinates in Bohr rather than Angstroms is to write a custom filter to output the data in the units that you want, converting to/from Angstroms on the fly. This example takes the existing xyz filter and creates a new ‘xyzbohr’ filter that does reads and writes in Bohr (note that Aten still works in Angstroms, however).

Firstly, copy the xyz filter that comes with Aten, probably in `/usr/share/aten/data/filters` on Linux/Mac, or `C:\Program Files\Aten N.MMM` on Windows, and copy this to your user filter directory. On Linux/Mac this is `~/.aten/filters`, while on Windows this will be a directory called `aten` in your user home in Documents and Settings. Rename your copy of the file to `xyzbohr`.

To make this new filter file read and write in units of Bohr we need to simply divide or multiply by 0.5292, depending on whether we’re writing or reading the coordinates. SO, begin by changing the `importmodel` section (around line 23) to read:

```
i = newAtom(e, rx*0.5292, ry*0.5292, rz*0.5292);
```

So this just converts the units of Bohr in the file to Angstroms on input, right? The `exportmodel` section must be modified as well (around line 38):

```
for (Atom i=m.atoms; i != 0; ++i) writeLineF("%-8s %12.6f %12.6f %12.6f\n", i.symbol, i.rx*0.5292, i.ry*0.5292, i.rz*0.5292, i.q);
```

The `importtrajectory` section can be modified in a similar way, or can be removed if it is not required.

As it stands, this new `xyzbohr` filter will ‘hide’ the original `xyz` filter since both the nicknames and IDs are the same, and so they must be changed in the `importmodel` and `exportmodel` headers (and the `importtrajectory` if it is still there). Simply changing all occurrences of ‘xyz’ to something like ‘xyzb’ is enough, as well as choosing an ID for the new filter which is not currently in use (something above 100 is safe).

The next time Aten is run the user filter directory will be searched and the new `xyzbohr` filter will be loaded ready for use, just as the normal stock of filters are.

The entire new filter file looks like this:

```
# Bohr XYZ coordinates files (for v1.2+)
# Created: 19/07/2011
# Last modified: 19/07/2011
# ChangeLog:

filter(type="importmodel", name="XMol XYZ Coordinates (in Bohr)", nickname="xyzb",
extension="xyzb", glob="*.xyzb", id=30)
{
    # Variable declaration
```

```

int natoms,n,m;
string e,title;
double rx,ry,rz,q;
Atom i;

# Read data
while (!eof())
{
    readLine(natoms);
    getLine(title);
    newModel(title);
    for (n=1; n<=natoms; ++n)
    {
        readLine(e,rx,ry,rz,q);
        i = newAtom(e, rx/0.529, ry/0.529, rz/0.529);
        i.q = q;
    }
    rebond();
    finaliseModel();
}
}

filter(type="exportmodel",name="XMol XYZ Coordinates (in Bohr)",
extension="xyzb", glob="*.xyzb", nickname="xyzb", id=30)
{
    # Variable declaration
    Model m = aten.frame;

    writeLine(m.natoms);
    writeLine(m.name);
    for (Atom i=m.atoms; i != 0; ++i) writeLineF("%-8s %12.6f %12.6f %12.6f
%12.6f\n",i.symbol,i.rx*0.529,i.ry*0.529,i.rz*0.529,i.q);
}

```

5.6. Calculate Average Coordinates (CLI)

This example takes the supplied water trajectory and determines the average coordinates of all the individual water molecules. Note that no accounting for the periodicity of the system is taken into account.

```
vector xyz[aten.model.natoms];
int nframes = aten.model.nFrames;
int natoms = aten.model.nAtoms;
printf("Number of frames is %i\n", nframes);
for (Model f = aten.model.frames; f; ++f)
{
    for (int n=1; n<=natoms; ++n) xyz[n] += f.atoms[n].r;
}
for (int n=1; n<=natoms; ++n) xyz[n] = xyz[n] / nframes;
Model f = aten.model.frames;
newModel("Average");
for (int n=1; n<= natoms; ++n)
{
    newAtom(f.atoms[n].z, xyz[n].x, xyz[n].y, xyz[n].z);
}
```

Copy and save this to a file named `avgeom.txt`.

To run the example:

```
aten --zmap singlealpha data/test/water66-spcfw.CONFIG
    -t data/test/water66-spcfw.HISu -s avgeom.txt
```

5.7. Self-Contained Liquid Chloroform Builder (Script)

This script (installed in the `$ATENDATA/scripts` directory) is an example of a completely self-contained script, depending on no external model or forcefield files. The chloroform model is built by hand, and the necessary forcefield is constructed in the script itself, before being used to create a box of liquid chloroform. Only the density of the liquid and the desired number of molecules are required in order to build the system, since the molecular volume is worked out from the density and the mass.

```
# Build chloroform system

# First, create chloroform model by transmuting a methane molecule.
# Since C-Cl distances will then be too short, we lengthen them afterwards
newModel("chloroform");
newAtom(C);
addHydrogen();
select(3,4,5);
transmute(Cl);
for (int n=3; n<6; ++n) setDistance(1,n,1.758);

# Set number of molecules and density required (g/cm3)
setupComponent("both", 1, 100, 1.483);

# Create a model with a basic unit cell - the disorder builder will adjust its
size as necessary
newModel("box");
cell(1.0,1.0,1.0,90,90,90);
disorder("None", FALSE);

# Construct new forcefield (using AMBER params)
newFF("chloroformff");
units("kj");
# Taken from files at http://www.pharmacy.manchester.ac.uk/bryce/amber
typeDef(1,"CZ","CZ",C,"-Cl(n=3),-H","Chloroform carbon");
typeDef(2,"HZ","HZ",H,"-&1","Chloroform hydrogen");
typeDef(3,"Cl","Cl",Cl,"-&1","Chloroform chlorine");
interDef("lj",1,-0.3847,4.184*0.1094, 1.9080*2.0/2.0^(1.0/6.0));
interDef("lj",2,0.2659,4.184*0.0157, 1.187*2.0/2.0^(1.0/6.0));
interDef("lj",3,0.0396,4.184*0.3250, 2.0*2.0/2.0^(1.0/6.0));
bondDef("harmonic","CZ","HZ",2845.12,1.1);
bondDef("harmonic","CZ","Cl",1944.7232,1.758);
angleDef("harmonic","HZ","CZ","Cl",318.8208,107.68);
angleDef("harmonic","Cl","CZ","Cl",650.1936,111.3);
finaliseFF();
ffModel();

# Minimise our new system a little
mcMinimise(100);

saveModel("dlpoly", "chloroform.CONFIG");
saveExpression("dlpoly", "chloroform.FIELD");
quit();
```

5.8. Generating Images Without the GUI (CLI)

Sometimes it's useful to quickly generate images for a system from the command line, either because the system is prepared from the command line, or because generating hundreds of images through the GUI is insanely repetitive. To quickly save a picture of a system in its 'standard' view orientation (i.s. as you would see it if you loaded it into the GUI) you can do the following:

```
aten data/test/cellulose.cif -c 'saveBitmap("png", "cellulose.png"); quit();'
```

Doing this will load the cellulose model, save an image of it, and then quit without ever starting the GUI. By default, the size of a saved image is 800x600, but optional arguments to the **saveBitmap** command allow this to be set explicitly.

5.9. Calculating a Torsion Energy Profile (CLI)

This example demonstrates a simple energy analysis procedure in which we load a model and perform a scan of a geometric parameter, calculating the energy at each step. The methanol model and OPLS-AA forcefield (both supplied with Aten) are used by the command, but of course can easily be substituted with your own choices. The command first sets the electrostatic calculation method to be the simple Coulomb sum (since the methanol model is non-periodic) and prior to the loop starting a text header is written. The loop iterates a variable *phi* over the range -180 to +180 degrees in steps of 5, and is used to set the torsion angle between atoms 2, 1, 4, and 6 (corresponding to one of the H–C–O–H torsions in the model). A line of output is written for each torsion angle considered, providing the total torsion, van der Waals, and electrostatic energy of the system at this geometry.

```
aten --ff oplsaa.ff data/test/methanol.inp -q
-c 'aten.prefs.elecMethod = "coulomb";
printf("Torsion Angle    E(Torsion)      E(VDW)      E(Coulomb)\n");
for (double phi = -180.0; phi <= 180.0; phi += 5) {
setTorsion(2,1,4,6,phi);
printf("%12.6f %12.6f %12.6f %12.6f\n", phi, aten.model.torsionEnergy(),
aten.model.vdwEnergy(), aten.model.elecEnergy()); } quit();'
```

Note that the use of the **-q** option means that only fundamental error messages and user output (through the **printf** statements in the command) is printed – all of Aten’s other working information is suppressed.

5.10. Running a Script on Many Models (CLI/Batch)

```
aten -c 'loadscript("test.script","ascript");' --keepnames *.gro --batch  
-c 'runscript("ascript");'
```

5.11. Saving GAMESS-US Input with Options (CLI)

Batch conversion or saving between molecule formats which contain purely coordinate and cell information is relatively trivial, but when saving to file formats which contain many user-adjustable parameters (e.g. Gaussian or GAMESS-US input) it may also be desirable to set these options explicitly on the command-line with Aten, rather than editing the files afterwards.

For export filters which contain GUI dialog definitions allowing the user to set these options it is possible to modify the values of all the controls defined in the dialog from the command-line, simply by specifying the control name and new value the control should take on. The following example loads the supplied water model (in xyz format), performs a MOPAC minimisation on the model (with **--batch** mode), and then saves it as a GAMESS-US input file. The **--export** switch takes just the nickname of the export filter at its most basic, but can also be supplied with a comma-separated list of control names and values to set before writing the file(s). In this case, we request that a DFT calculation be performed with the B3LYP functional, using the MCP-DZP basis set, and allowing for 200 steps in the geometry optimisation. Note that all the GUI controls in the GAMESS-US export filter have been given names which relate directly to the block and keyword which they relate to in the input file.

```
bob@pc:~> aten data/test/water.xyz --export
"gamusinp,contrl_dfttyp=B3LYP,basis_gbasis=MCP-DZP,statpt_nstep=200" --batch -c
'mopacminimise();'
```

Alternatively, one may specify the **--dialogs** switch to force the export filter's dialog box to be raised when running Aten from the command-line, allowing options to be set using the dialog (but without raising the full GUI).

5.12. Printing all Bond Distances in a Model (CLI)

Quickly getting at geometric data in a model file is often useful without having to go through the GUI, in order to quickly verify a geometry or get certain data into a file. The following command prints out all of the bond distances, including the indices of the atoms involved, from the command line:

```
bob@pc:~> aten data/test/methanol.inp -c 'for (Bond b = aten.model.bonds; b; ++b)  
printf("Distance %i-%i = %f\n", b.i.id, b.j.id, geometry(b.i,b.j)); quit();'
```

This will give you output along the following lines:

```
Autodetecting patterns for model 'Methanol'..  
New pattern 'OCH4' added - startatom 1, 1 mols, 6 atoms per mol.  
Pattern description completed (spans 6 atoms).  
Done.  
Augmenting bonds in pattern OCH4...  
Distance 1-2 = 1.080000  
Distance 1-3 = 1.080000  
Distance 1-4 = 1.080000  
Distance 1-5 = 1.080000  
Distance 4-6 = 1.079999
```

All very good, but what about the extra information printed by Aten (such as pattern detection, etc.)? This can be inhibited by adding the **-q** switch to the command.

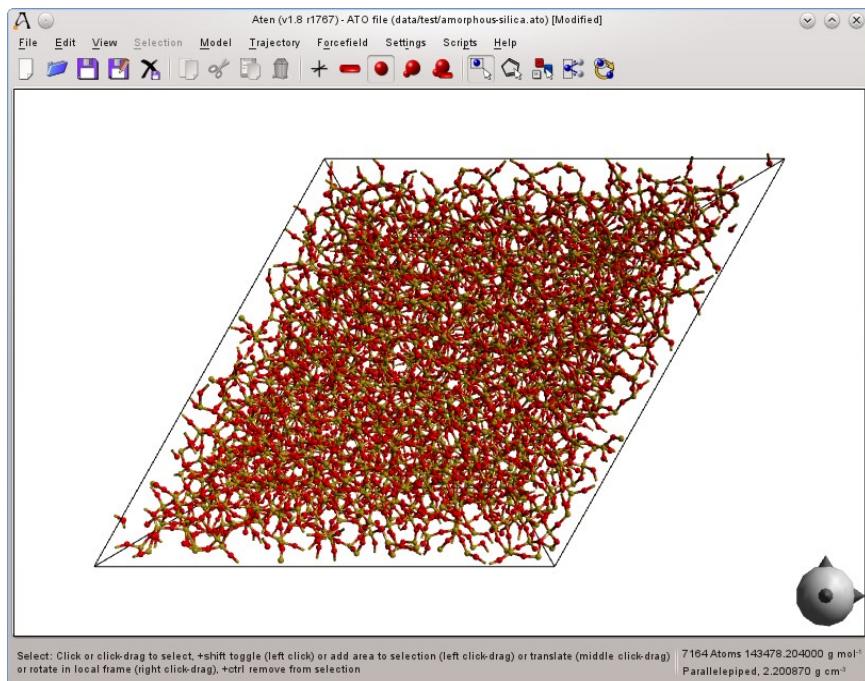
5.13. Creating a Porous Silica Model

This example takes an existing amorphous silica model (supplied with Aten) drills some cylindrical pores, hydroxylates the resulting internal surfaces, and fills the pore with a liquid.

Prepare the Template Model

Firstly, load the `amorphous-silica.ato` model file, and then calculate bonding within the model. The model has a monoclinic cell, just to make things interesting.

 **Build** Open the **Build Window** and click the **Rebond** button to recalculate bonding between the atoms in the model (since they are currently all unbound).



Drill Pores in the Model

6. Command Line Usage

From a shell, running Aten with no arguments will start up the code in GUI. If model files are provided on the command line, these will be loaded in so that, when the GUI starts, they may be hacked apart according to your desired tastes. The command-line is also a powerful way of editing without using the GUI at all. What follows is a description of the usage of command-line arguments, and a list of all recognised arguments.

6.1. Switch Order

Two important things to consider. Firstly, short options (e.g. ‘-b’, ‘-d’ etc.) may not be concatenated into one long specification of a short option (i.e. ‘-bd’) - they must be given separately as ‘-b -d’ or they will not be recognised. Secondly, the order of the given switches is important since their meaning is applied or acted out immediately. For example:

```
bob@pc:~> aten --nobond test1.xyz test2.xyz
```

will load the models ‘test1.xyz’ and ‘test2.xyz’, preventing recalculation of bonds between atoms in both. However:

```
bob@pc:~> aten test1.xyz --nobond test2.xyz
```

will only prevent recalculation of bonds for the second model. The reason for acting on switches and arguments in the order they are encountered on the command line is to allow for flexibility, especially when using Aten as a non-graphical processor.

Note: The position of debug switches or those affecting the verbosity of the program has no bearing on the timeliness of their effect – they are dealt with first by Aten regardless of where they appear in the program’s arguments.

6.2. Switches

A

--atendata <dir>

Tells Aten to use the specified *dir* as its data directory (where filters etc. are stored). This overrides the ATENDATA shell variable.

B

-b, --bohr

Specifies that the unit of length used in any models and grid data that follow is Bohr rather than Å, and should be converted to the latter.

--batch

Enter batch processing mode, modifying and saving models to their original filenames. See Section 6.3 for details and a list of other modes.

--bond

Force recalculation of bonding in loaded models, regardless of whether the filter used any of the rebond commands (see the list of bond-related commands in Section 9.2).

C

-c <commands>, --command <commands>

Provides a command or compound command to execute. Commands should be enclosed in single quotes (to prevent the shell from misquoting any character strings) and individual commands separated with semicolons. Commands provided in this way can be used to set up Aten in the way you want it from the command line, perform operations on model files before loading the GUI, or perform operations on model files without loading the GUI at all.

For example, to start the GUI with a new model named ‘cube’ that has a cubic cell of 30 Å side length:

```
bob@pc:~> aten -c 'newmodel("cube"); cell(30,30,30,90,90,90);'
```

Similarly, to load a model and make a duplicate copy of the atoms (pasted into the same model):

```
bob@pc:~> aten original.xyz -c 'selectall(); copy(); paste(10,10,10);'
```

In both cases the GUI initialises itself without being told, but this can be prevented with the **quit** command. Consider the last example – to save the newly-expanded model and quit without ever launching the GUI:

```
bob@pc:~> aten original.xyz -c 'selectall; copy; paste(10,10,10); savemodel("xyz", "pasted.xyz"); quit;'
```

Multiple sets of commands may be given:

```
bob@pc:~> aten source.xyz -c 'selectall; copy' target.xyz -c 'paste(10,10,10);'
```

Take care here, since the commands provided act on the current model, i.e. the one that was most recently loaded. Commands are available that select between the loaded models – see the list of model-related commands in Section 9.22.

--cachelimit <limit>

Sets the size limit for trajectory loading, in kilobytes. If an entire trajectory will fit into this cache, all frames in the trajectory are loaded immediately. If not, frames will be read from disk as and when required.

--centre

Force translation of non-periodic models centre-of-geometry to the origin, even if the **centre** command was not used in the corresponding filter.

D

-d [<type>], --debug [type]

Enables debugging of subroutine calls so that program execution can be traced, or enables extra debug output from specific types of routines (if *type* is given). Warning - this creates a lot of output, most of which is incomprehensible to people with their sanity still intact, but is useful to track the program up to the point of, say, a hideous crash. Valid *type* values are listed in Output Types in Section 16.14.

--dialogs

By default, filter and script user dialogs are not shown when performing functions on the command-line. This switch forces dialogs to be shown.

--double <name=value>

Creates a ‘floating’ variable *name* which is of type ‘double’ and that can be accessed from any subsequent script, command, or filter. Note that declarations of variables with the same name made in scripts, commands and filters will override any passed value names in order to avoid conflicts and breaking of existing filters and scripts. The intended use is to be able to pass values easily from the command-line into scripts or one-line commands.

For example, in a bash shell:

```
bob@pc:~> for num in 10.0 50.5 100.0; do aten --double d=$num -c 'printf("Value is %f\n", d); quit();' done
```

E

--export <nickname>

Enter export mode, where each model specified on the command line is loaded and saved in the format corresponding to the <nickname> specified. If specified in conjunction with **--batch**, batch export mode is entered instead, with commands run on models before being saved to the new format. See Section 6.3 for details and a list of other modes.

--exportmap <name=element, ...>

Manually map assigned atom typenames in an expression to the names defined here when expressions are written to a file. For example:

```
bob@pc:~> aten --ff spc.ff data/test/water.xyz --exportmap "OW=Ospc,H=Hspc" -c  
'saveexpression("dlpoly", "water.FIELD"); quit();'
```

writes the water forcefield with the `OW` and `HW` atomtype names mapped to `Ospc` and `Hspc` respectively.

--expression <file>

Loads the specified *file* as if it were an expression.

F

-f <nickname>, --format <nickname>

For any forthcoming model files provided as arguments on the command line, the specified model import filter is used to load them, regardless of their filename extension (or, indeed, actual format). Since Aten tends not to determine file formats by looking at their content, this is useful for when you know that file is in a particular format, but with an extension that doesn't help Aten recognise it as such.

--ff <file>

Loads the specified forcefield file, making it the current forcefield. If the desired forcefield is present in either Aten's installed `data/` directory or in your own `.aten/ff` directory (see Section 4.2), then just the filename need be given as Aten searches these locations by default.

--filter <file>

Load the specified *file* as if it were a filter file, installing any filters defined within it. Any filters already loaded that have the same 'nickname', 'id' etc. will be hidden by those loaded from *file*. See Section 11.1.3 for more information on overriding existing filters.

--fold

Force folding of atoms to within the boundaries of the unit cell (if one is present) in loaded models, even if the command `fold` was not used in the corresponding filter.

G

-g <file>, --grid <file>

Loads the specified grid data *file*, associating it to the current model, and making it the current grid. A model (even an empty one) must exist for a grid to be loaded.

H

-h, --help

Show the possible command-line switches and a short description of their meaning.

I

-i, --interactive

Starts Aten in interactive mode, where commands are typed and immediately executed. The GUI is not started by default, but may be invoked.

--int <name=value>

Creates a ‘floating’ integer variable *name*. See the **--double** switch for a full description.

K

-k, --keepview

Preserves the last stored view of models when the GUI starts, retaining any model rotations and camera transformations performed in scripts or on the command line (normally, the view is reset to display the entire model on startup).

--keepnames

If specified, for each model loaded the original atom names in the file will be preserved as a series of forcefield types generated within a new forcefield created specifically for the model. Elements are still determined from conversion of the atom names, and may still be mapped with the **--map** option. This option is useful for quickly creating a skeleton set of forcefield types from an existing model with type names, or to allow quick import and export of typed configurations without requiring the original forcefield file to be loaded.

Note that the **--keeptypes** and **--keepnames** switches are mutually exclusive.

--keeptypes

If specified, for each atom name converted to an element using a forcefield name match, the corresponding forcefield type will be assigned to the atom and fixed. Like the **--keepnames** switch, this is useful for preserving atom type data when importing certain models which do not store element information.

Note that the **--keeptypes** and **--keepnames** switches are mutually exclusive.

M

-m <name=element,...>, --map <name=element,...>

Manually map atom typenames occurring in model files to elements according to the rules defined here. For example:

```
bob@pc:~> aten --map 'CX=C,N_=P'
```

will result in atoms called CX being mapped to carbon, and atoms called N_ mapped to phosphorus (for whatever reason). These mappings are attempted prior to any z-mapping scheme defined in the filter, and so will take precedence over standard typename-to-element conversions.

-n

Create a new, empty model.

--nicknames

Print a list of all available import/output filter nicknames and quit.

--nobond

Prevent recalculation of bonding in loaded models, overriding filter directives. This basically means that, if a filter tries to run the **rebond** command, then specifying **--nobond** will prevent it.

--nocentre

Prevent translation of non-periodic models centre-of-geometry to the origin, overriding filter directives.

--nofold

Prevent initial folding of atoms to within the boundaries of the unit cell (if one is present) in loaded models, overriding the use of the **fold** command in the corresponding filters.

--nofragments

Prevent loading of fragments from both standard and user locations on startup.

--nofragmenticons

Prevent generation of fragment icons, used in the Fragment Library Window (see Section 7.13).

--noincludes

Prevent loading of global includes on startup.

--nolists

Prevent the use of OpenGL display lists for rendering. Simple vertex arrays will be used instead. Try this option out if rendering is corrupt or Aten crashed unexpectedly on startup. The rendering will be slower, but more compatible.

--nopack

Prevent generation of symmetry-equivalent atoms from spacegroup information in loaded models, overriding any occurrences of the **pack** command is used in the corresponding filter.

--nopartitions

Prevents loading of partitions on startup.

--noqtsettings

Don't read in any system-stored Qt settings on startup (such as window positions, toolbar visibilities etc.) using the defaults instead.

P

--pack

Force generation of symmetry-equivalent atoms from spacegroup information in loaded models, even if the **pack** command was not used in the corresponding filter.

--pipe

Read and execute commands from piped input on startup.

--process

Enter process mode, where commands are run on models but no changes are saved – instead, the GUI is started once all commands have been executed. See Section 6.3 for details and a list of other modes.

Q

-q, --quiet

Prevents nearly all text output from Aten, including error messages and the like, but **does** allow printing of user output via the **printf** command in scripts and commands passed with **--command**. Useful in order to print clean data to a file or standard output.

S

-s <file>, --script <file>

Specifies that the script file is to be loaded and run before moving on to the next command-line argument. A script file is just a plain text file that contains sequence of commands to be executed, written in the command language style (see Section 8.1).

--string <name=value>

Creates a ‘floating’ string variable *name*. See the **--double** switch for a full description.

T

-t <file>, --trajectory <file>

Associates a trajectory file with the last loaded / current model.

U

-u <nlevels>, --undolevels <nlevels>

Set the maximum number of undo levels per model, or -1 for unlimited (the default).

V

-v, --verbose

Switch on verbose reporting of program actions.

--vbo

Attempt to use OpenGL vertex buffer objects when rendering, for maximum performance.

Z

-z <maptype>, --zmap <maptype>

Override the names to elements z-mapping style defined in file filters. For a list of possible mapping types see ZMapping Types in Section 16.17.

6.3. Batch Processing Modes

Aten has several batch or ‘offline’ processing modes that do not need the GUI to be invoked. These permit calculations, analyses or processes to be performed on multiple models in one simple command. Most work by storing any commands or command sequences supplied with **--command** until all models are loaded, and then running the commands on each loaded model in sequence. The modes are as follows:

Batch Mode

Invoked by the **--batch** switch, this mode runs all commands provided on all models, once the last model has been loaded. The models are then saved in their original format to the same filename. Note that, if the an export filter does not exist for the original model file format, changes to that model will not be saved. It is advisable to work on a copy of the model files when using this command, or to use batch export mode to save to a different format in order to preserve the original files. The GUI is not automatically started in batch mode.

For example, to transmute all iron atoms into cobalt for a series of xyz files named ‘complex_001.xyz’, ‘complex_002.xyz’ etc.

```
bob@pc:~> aten --batch -c 'select(Fe); transmute(Co);' complex_*.xyz
```

Export Mode

Invoked by the **--export** switch, in export mode each model file specified on the command line is loaded and immediately saved in the format specified by the provided nickname, allowing multiple files to be converted to a different format at once. The GUI is not automatically started in export mode.

For instance, to convert three DL_POLY CONFIG files and an xyz into mol2 format:

```
bob@pc:~> aten --export mol2 bio1.CONFIG bio2.CONFIG watercell.CONFIG random.xyz
```

If specified in conjunction with the **--batch** switch, batch export mode is entered instead, and any supplied commands are executed on each loaded model file before it is saved. The original model files are not modified.

Batch Export Mode

Invoked by providing the **--batch** and **--export** switches together, batch export allows a series of commands to be run on a set of loaded models, the results of which are then saved in new files in the model format provided to the **--export** switch. The GUI is not automatically started in batch export mode.

Let's say that you have a directory full of xyz files that you wish to energy minimise with MOPAC2009 (see Section 14.2), centre at zero, and then save as input to GAMESS-US. This can be achieved with the following command:

```
bob@pc:~> aten --export gamusinp --batch -c 'mopacminimise(); selectall(); centre();' *.xyz
```

Various export options for the GAMESS-US filter (e.g. method type, basis set) can be set at the same time. See how to set filter options in Section 11.1.5, and Section 5.11 for an example.

Process Mode

Similar to the **--batch** switch, in that all commands supplied with **--command** are executed on each model, but in this case the results are **not** saved, and the GUI starts once processing is complete.

7.The GUI

7.1. Overview

Aten's main window is predominantly taken up with the rendering canvas where model(s) are displayed. Multiple models may be displayed simultaneously – the ‘current’ model (i.e. the one to which all editing / data operations are send) always has a black box drawn around it. A single toolbar sits above the canvas providing quick access to file, edit, and select actions. All other functionality is contained with various tool windows. These tool windows are accessed by the ToolBox widget, which by default is the only window displayed on startup (if it is not visible, you can raise and centre it on the main window from the menu item **Settings→Show Toolbox**). At the foot of the window is a status bar reflecting the content of the current model, listing the number of atoms and the number of selected atoms (bold value in parentheses, but only if there are selected atoms), the mass of the model, and the cell type and density (if the model is periodic).

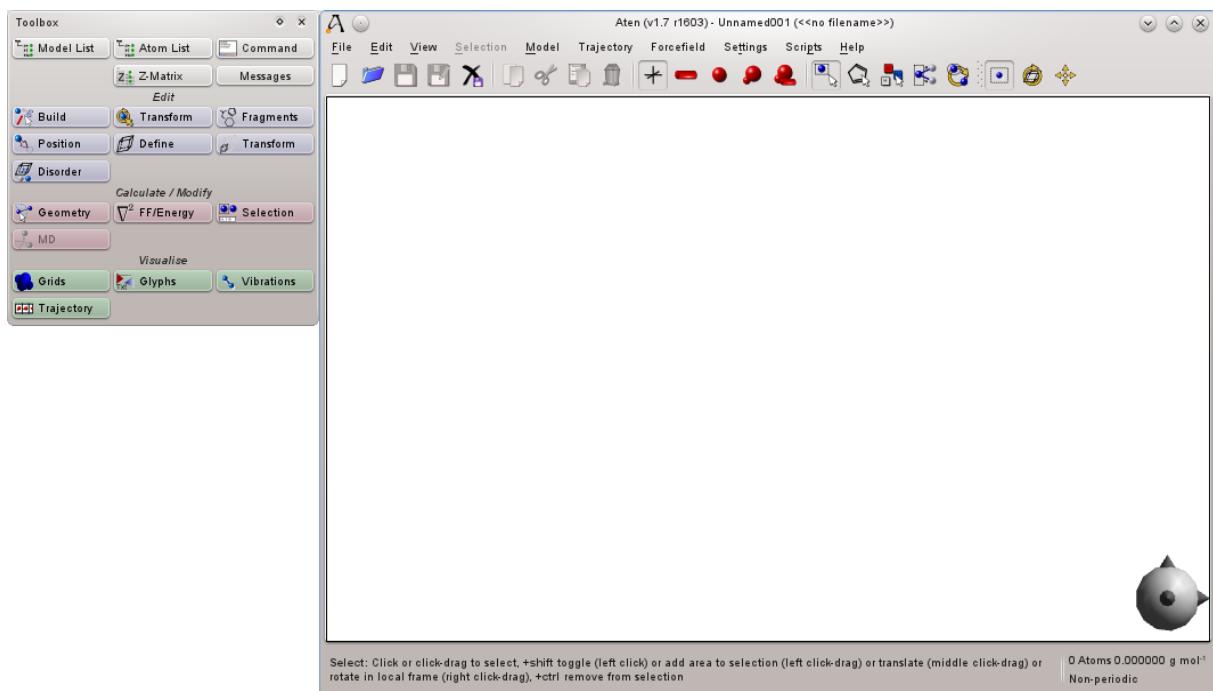


Figure 7-1 Aten’s main window and toolbox

The edges of the main window, around the canvas, are standard (Qt4) dock areas, in which the toolbox and any of the tool windows can be placed. Once you have the various subwindows set up how you like them, press **Settings→Store Default Window State** to remember these settings for when Aten next starts.

7.2. Mouse Control

Each of the mouse buttons has a different ‘style’ of action on the canvas, each of which can be set to the user’s taste in the preferences (menu item **Settings→Preferences** on Linux/Windows). In addition the **Shift**, **Ctrl**, and **Alt** keys modify or augment these default actions performed by the mouse. Standard settings out of the box are:

Table 7-1 Mouse Button Actions

Button	Key Modifier	Action
Left	None	Click on individual atoms to select exclusively
	Shift	Click-hold-drag to exclusively select all atoms within rectangular region Click on individual atoms to toggle selection state
	Ctrl	Click-hold-drag to inclusively select all atoms within rectangular region Click on individual atoms to toggle selection state
Right	None	Click-hold-drag to rotate camera around model
	Shift	Click on atom to show context (Selection) menu Click-hold-drag to rotate view around z-axis (perpendicular to plane of screen)
	Ctrl	Click-hold-drag to rotate selection in local (model) space
	Ctrl+Shift	Click-hold-drag to rotate selection around z-axis in local (model) space
Middle	None	Click-hold-drag to translate camera
	Ctrl	Click-hold-drag to translate selection in local (model) space

7.2.1. Manipulating the View

At its most basic Aten’s main view acts as a visualiser allowing models to be rotated, zoomed in and out, and drawn in various different styles. By default, the right mouse button is used to rotate the model in the plane of the screen (right-click and hold on an empty area of the canvas and move the mouse) and the mouse wheel zooms in and out. Note that right-clicking on an atom brings up the atom context menu (equivalent to main window’s **Selection** menu). The middle mouse button translates the model in the plane of the screen – again, click-hold and drag.

These rotation and translation operate only the position and orientation of the camera, with no modifications made to the actual coordinates of the model. The view can be reset at any time from the main menu (**View→Reset**) or by pressing **Ctrl-R**. Both the main menu (**View→Style**), the main toolbar, and the shortcuts **Ctrl-1** to **Ctrl-5** allow the drawing style of models to be changed between stick, tube, sphere, scaled sphere, and individual. The last option allows different view styles to be set for different atoms.

The **Ctrl** key changes the normal behaviour of the rotation and translation operations and forces them to be performed on the coordinates of the current atom selection instead of the camera. The centroid of rotation is the geometric centre of the selected atoms.

7.2.2. Atom Selection

Atom selection or picking is performed with the left mouse button by default – single-click on any atom to highlight (select) it. Single-clicks perform ‘exclusive’ selections; that is, all other atom(s) are deselected before the clicked atom is (re)selected. Clicking in an empty region of the canvas deselects all atoms. Clicking on an empty space in the canvas, holding, and dragging draws a rectangular selection region – releasing the mouse button then selects all atoms within this area. Again, this selection operation is exclusive. Inclusive selections (where already-selected atoms are not deselected) are performed by holding the **Shift** key while performing the above operations. Furthermore, single-clicking on a selected atom while holding **Shift** will deselect the atom.

7.3. Keyboard Shortcuts

Shortcut	Action	Description
Escape	Box Select	Cancel current mouse mode and return to basic box-select interaction mode
Ctrl-A	Select All	Select all atoms in the current model
Ctrl-C	Copy	Copy the current atom selection to the clipboard
Ctrl-Alt-C	Centre Selection	Centre the current atom selection at {0,0,0}
Ctrl-D	Deselect All	Deselect all atoms in the current model
Ctrl-Delete	Delete	Delete the current atom selection
Ctrl-F	Fold Atoms	Fold all atoms into the unit cell
Ctrl-Shift-F	Fold Molecules	Fold atoms into the unit cell, keeping molecules intact
Ctrl-H	Hide Selection	Hide all selected atoms
Ctrl-Shift-H	Show All Atoms	Unide any hidden atoms
Ctrl-I	Invert Selection	Toggle the selection state of all atoms
Ctrl-N	New	Create a new, empty model
Ctrl-O	Open	Load an existing model into Aten
Ctrl-P	Play/Pause	Play / pause current trajectory
Ctrl-R	Reset View	Reset the camera for the current model
Ctrl-S	Save	Save the current model under its original filename
Ctrl-V	Paste	Paste the contents of the clipboard to the current model
Ctrl-X	Cut	Copy the current selection to the clipboard and delete it
Ctrl-Y	Redo	Redo last undone operation
Ctrl-Z	Undo	Undo last operation
Ctrl-1	Stick	View models as sticks
Ctrl-2	Tube	View models as tubes
Ctrl-3	Sphere	View models as tubes and spheres
Ctrl-4	Scaled	View models as tubes and scaled spheres
Ctrl-5	Individual	View models according to atom's assigned styles
Ctrl--	Zoom Out	Zoom out
Ctrl-+	Zoom In	Zoom in
Ctrl-Alt-<	First Frame	Jump to first frame of current trajectory
Ctrl-Alt->	Last Frame	Jump to last frame of current trajectory
Ctrl-Right	Next Model	
F8	Detect H-Bonds	Toggle detection and display of hydrogen bonds
F10	Quick Command	Raise a dialog window allowing a quick command to be run

7.4. The Main Toolbar



The main toolbar provides quick access to model load / save, edit, and selection operations. Left to right these icons are:

Icon	Action	Shortcut	Description
	New	Ctrl-N	Create a new, empty model
	Open	Ctrl-O	Load an existing model into Aten
	Save	Ctrl-S	Save the current model under its original filename
	Save As		Save the current model under a different filename
	Close		Close the current model (prompting to save first if changes have been made)
	Copy	Ctrl-C	Copy the current atom selection to (Aten's internal) clipboard
	Cut	Ctrl-X	Copy the current atom selection to the clipboard and then delete it
	Paste	Ctrl-V	Paste the contents of the clipboard to the current model at the original coordinates. The pasted atoms then become the current selection. Note that pasted atoms are not translated to avoid overlap with existing atoms.
	Delete	Ctrl-Delete	Delete the current atom selection
	Stick	Ctrl-1	Atoms are not explicitly drawn unless they possess no bonds, bonds are drawn using simple lines
	Tube	Ctrl-2	Atoms and bonds are drawn as tubes
	Sphere	Ctrl-3	Atoms are drawn as uniformly-sized spheres, with bonds drawn as tubes
	Scaled	Ctrl-4	Atoms are drawn as spheres whose size depends on their atomic radii, bonds are drawn as tubes
	Individual	Ctrl-5	Each atom is drawn according to its own assigned style
	Box Select	Escape	Atoms may be (de)selected by clicking on them individually, or selected en masse with a click-hold-drag
	Molecule Select		Bound fragments are selected by clicking on a single atom within that fragment
	Element Select		Clicking on a single atom selects all atoms of the same element
	Expand Selection		The current selection is expanded by following bonds attached to any select atoms
	Invert Selection	Ctrl-I	Toggle the selection state of all atoms

7.4.1. The Mouse Toolbar

There is only one other toolbar in (newer versions of) Aten – the Mouse toolbar. For multi-button mice each button can be assigned an individual action (select, rotate model etc.). For those who use single-button rats, this toolbar changes the function of the first (or left) button between select / interact, rotate model, and translate model. Selecting these buttons overwrite the stored action for the left button in the Preferences.



Icon	Shortcut	Action
	F1	The left button selects and interacts with atoms
	F2	The left button rotates the view or selection
	F3	The left button translates the view or selection

7.5. The ToolBox

The ToolBox provides access to most of Aten's functionality, allowing various different subwindows to be shown (and hidden).

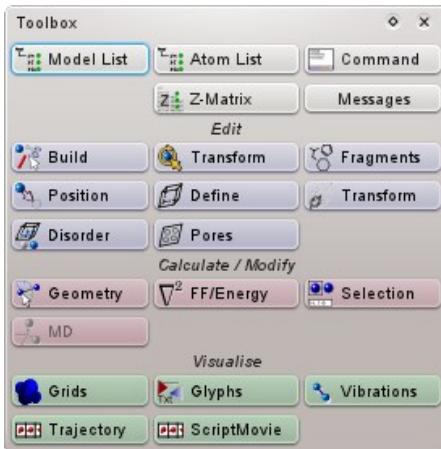


Figure 7-2 The ToolBox Window

Button	Window Function / Contents
Model List	List of models currently loaded
Atom List	List of atoms in the current model
Command	Runs single commands; Provides interactive shell; Load & run scripts
Z-Matrix	Edit Z-matrix for the current model
Build	Draw atoms, chains of atoms, and bonds; Rebond & clear bonds
Transform	Transform the current atom selection (e.g. flip, rotate, matrix multiply)
Fragments	Access to molecular fragments when in fragment drawing mode
Position	Position the current atom selection (e.g. shift, translate, centre)
Define	Define / create cell for the current model
Transform	Transform the cell for the current model
Disorder	Run the Disorder builder
Pores	Tools for pore drilling
Geometry	Measure distances and angles; Edit geometry of current atom selection
∇² FF/Energy	Load / edit forcefields; Perform geometry optimisation of models
Selection	Advanced atom selection tools
Grids	Grid manipulation and editing
Glyphs	Glyph manipulation and editing
Vibrations	Visualisation of molecular vibrations (if present)
Trajectory	Visualisation of associated trajectory (if present)

7.6. Atom List Window

The Atom List window provides a list of all atoms in the current model or frame, displaying elements, charges, and positions. If patterns have been defined for the model (see Section 10.3), the atoms in the list will be grouped according to their encompassing pattern, otherwise they are listed in one continuous run by ascending ID.



Figure 7-3 Atom List Window

Atom Selection

The selection of items in the atom list mirrors the selection in the current model - (de)selecting atoms in one will also (de)select them in the other. If patterns are defined, (de)selecting the pattern name in the list (de)selects all atoms making up the pattern.

Changing the Order of Atoms

The four buttons at the foot of the atom list allow the current selection of atoms to be moved up and down the list, useful for reordering atoms into a specific sequence. Selected atoms can be shifted up/down the list one place at a time, or all moved to the top or bottom of the list at once. In the latter case the order of the original selection is preserved.

7.7. Build Window

The primary function of the Build window is to allow for drawing, deletion, and transmuting of individual atoms and bonds using the mouse, along with drawing and automatic creation of bonds.

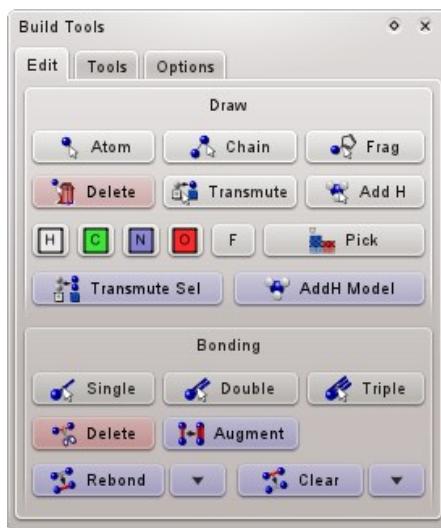


Figure 7-4 Build Window – Edit controls

The top half of the **Edit** page of the window provides tools to draw individual atoms, chains of atoms, and molecular fragments, allow the deletion and transmutation of atoms, and provide the ability to add hydrogen atoms automatically to atoms (or the whole model). Select the relevant tool, and simply click on atoms in the main view. The element of new atoms when drawing atoms or chains (as well as the transmutation target element) is determined by the currently-selected element button. The bottom half provides tools to draw individual bonds between atoms, or to calculate bonds automatically. Drawing a bond requires pairs of atoms to be clicked sequentially once the tool is activated.

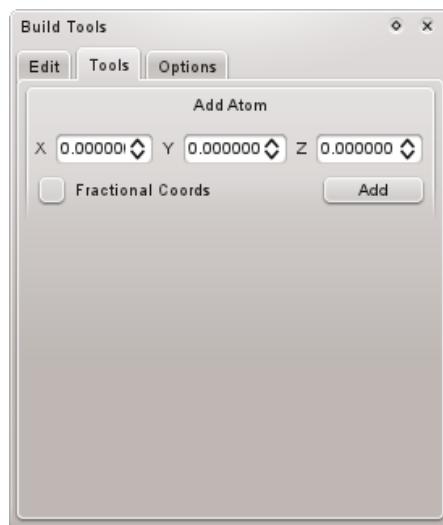


Figure 7-5 Build Window – Tools page

The **Add Atom** tool allows atoms to be created at specific positions in the model. Coordinates are entered and the atom created (with element defined by the current selected element on the Edit page) by pressing the Add button. If the **Fractional Coords** checkbox is ticked the coordinates are assumed to be fractional and are converted to cell coordinates as the atom is added. For example, setting coordinates to {0.5,0.5,0.5} will create an atom in the centre of the current unit cell.

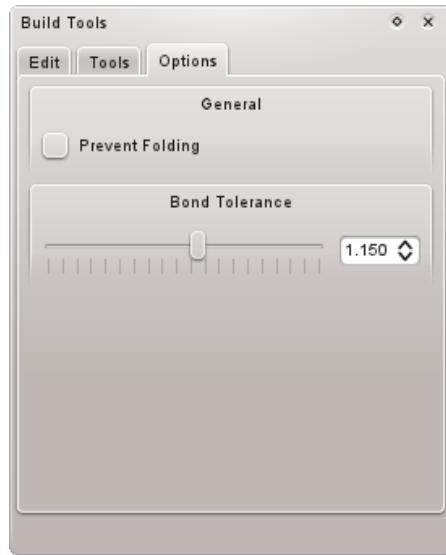


Figure 7-6 Build Window – Options page

Finally, the Options page allows various aspects of building to be adjusted.

When transforming atoms in a periodic system with the mouse (i.e. rotating or translating them) if any move outside the unit cell as a result of the transformation they are automatically folded back in to the confines of the cell. If the **Prevent Folding** checkbox is ticked, these folding operations will not occur.

All rebonding operations employ a tolerance or scaling factor in the calculation of distances between atoms, which can be adjusted with the Bond Tolerance slider. Larger values increase the maximum distance between atoms that will be recognised as a bond. Use restraint, however, as too large values will generate irregular bonding patterns.

7.8. Cell Definition Window

Allows the user to edit the unit cell specification of the current model, assign a crystallographic spacegroup, and perform spacegroup packing according to the spacegroup. Whether or not the current model possesses a unit cell is determined wholly by the **Has Cell** checkbox – if checked, the current model is periodic, if not, then it is an isolated collection of atoms.

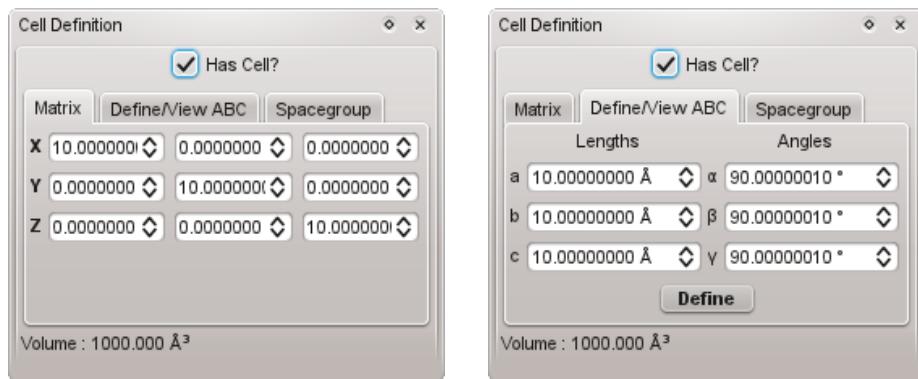


Figure 7-7 Cell Definition Window – Matrix and Define/View pages

If the model has a cell, its size and shape can be set in one of two ways. The primary method is through direct editing of the cell matrix on the **Matrix** page, which presents the individual components of the X (A), Y (B), and Z (C) axes. All values are in Angstroms, and any changes made here are immediately applied to the current model. Alternatively, the cell may be set by entering its lengths (in Angstroms) and angles (in degrees) on the **Define/View ABC** page and then pressing the **Define** button.

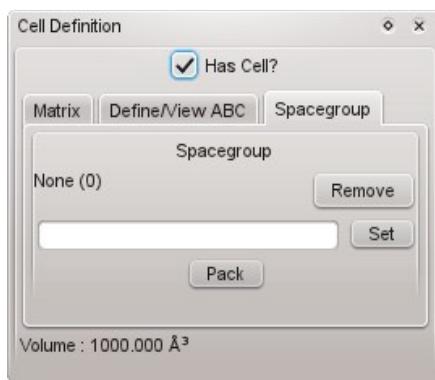


Figure 7-8 Cell Definition Window – Spacegroup page

The spacegroup assigned to the current model can be changed and removed on the **Spacegroup** page. On its own, an assigned spacegroup is just a number - no modifications to the atoms are made by changing the assigned spacegroup. Packing of atoms, however, is a different matter. Once a spacegroup is assigned, symmetry-equivalent atoms may be generated by use of the **Pack** button. This assumes that the current contents of the model represents the symmetry-unique set of atoms (i.e. the minimal generator set which, along with the spacegroup and unit cell, defines the entire crystal).

7.9. Cell Transform Window

Transformations of the unit cell and its contents can be made here, encompassing geometric scaling of the cell (and atoms contained within) and replication of the system in three dimensions.

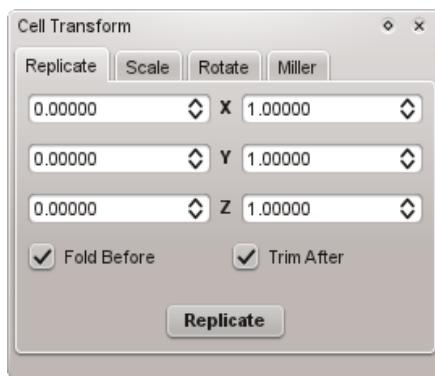


Figure 7-9 Cell Transform Window – Replicate page

The **Replicate** page allows the current cell to be replicated along its three principal axes in both positive and negative directions. The six inputs represent negative and positive replication values for each direction – most of the time it's probably only useful to consider the positive (right-most) replication directions. Note that the numbers define the *additional* cells that will be created in addition to the original one. So, if all numbers are left at zero the original cell will remain untouched. Entering a value of 1 for each positive direction will give a $2 \times 2 \times 2$ supercell of the original cell, and so on. The representative unit cells of replicated and partially replicated copies of the current cell are drawn onto the current model.

Atoms in the model are folded into the unit cell prior to replication, unless the **Fold Before** checkbox is unticked. Similarly, atoms that exist outside of the cell after replication are trimmed unless the **Trim After** checkbox is unchecked.

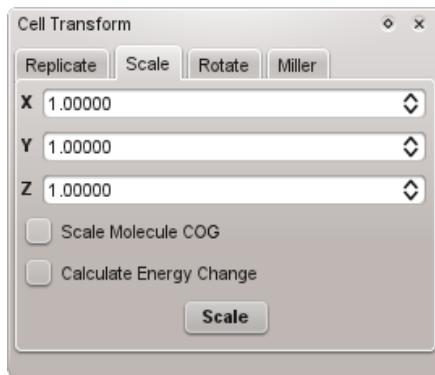


Figure 7-10 Cell Transform Window – Scale page

The **Scale** page allows the principal axes of the current unit cell to be arbitrarily scaled, along with the cell's contents. If a valid pattern description exists for the model, then the positions of individual molecules or bound fragments within the cell are scaled relative to their centres of geometry – all intramolecular distances within molecules remains the same as before the

scaling. If this is undesirable (or unintended) then performing the scaling with no pattern definition will scale the position of each atom separately.

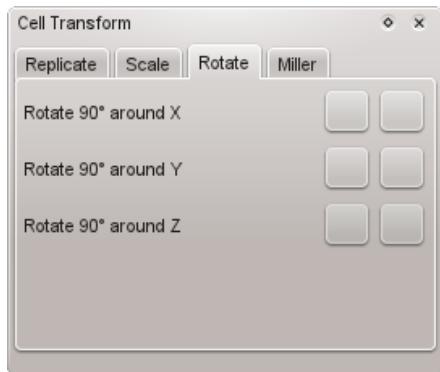


Figure 7-11 Cell Definition Window – Rotate page

Not currently implemented.

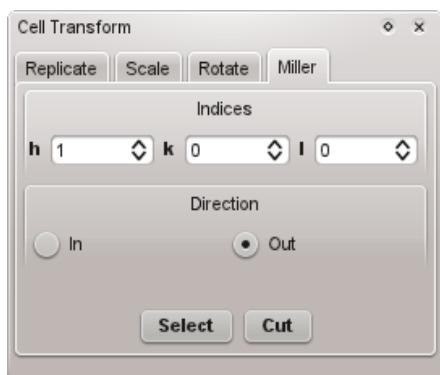


Figure 7-12 Cell Definition Window – Miller page

The **Miller** page allows the model to be cut so that various Miller surfaces are left behind. The hkl indices of the desired Miller plane should be entered in the three spin boxes, and the resulting plane(s) will be drawn onto the current model. The deletion of atoms can be done in one of two ways, removing either those atoms that are ‘inside’ or those atoms that are ‘outside’ of the defined Miller plane and its periodic or symmetric equivalent.

7.10. Command Window

The command window allows commands or sequences of commands to be run and stored for later use, allows management and execution of scripts, and provides access to searchable command help.



Figure 7-13 Command Window – Prompt page

Any command or compound command can be entered in the bottom editbox and will be executed immediately. This command will then be added to the list above, and can be single-clicked to return it to the editbox for tweaking or re-editing, or double-clicked to execute it again. The list of stored commands is saved when Aten exits, and loaded back in when restarted.

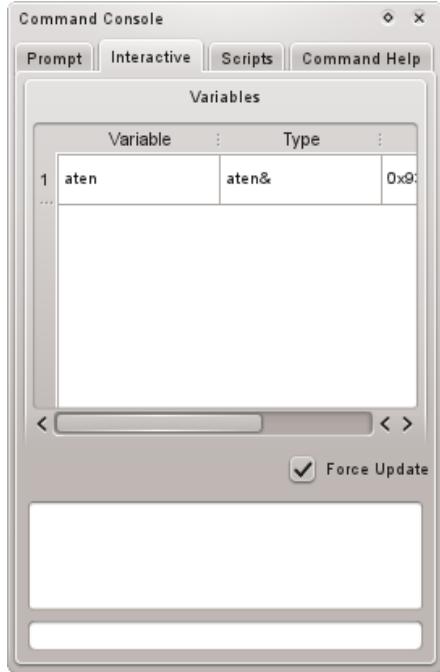


Figure 7-14 Command Window – Interactive page

For when single-line commands are too inflexible, the interactive page mimics a more ‘proper’ console-like environment. Here, variables can be defined in one instance and referred back to and manipulated in others, much like a normal shell. A list of variables currently defined in the local scope is shown in the uppermost part of the window.



Figure 7-15 Command Window – Scripts page

Scripts can be loaded in from here and executed at will by double-clicking individual scripts in the list or selecting multiple scripts and clicking the **Run Selected** button. Any script files loaded in this way are remembered when Aten exits and are loaded back in when restarted. All scripts can be reloaded from disk (if, for example, changes have been made to one or more files after they were loaded in) by clicking the **Reload All** button.

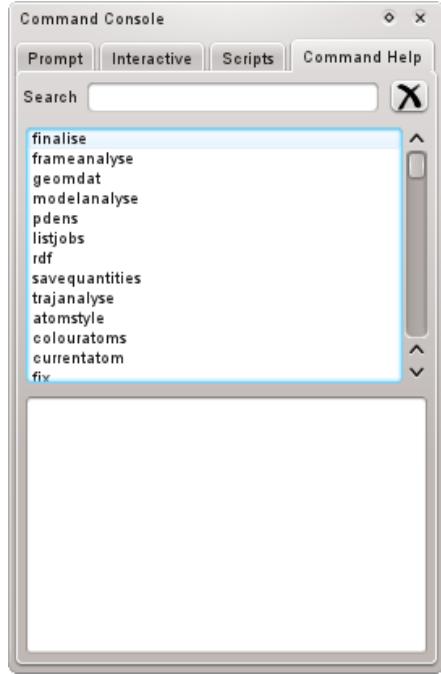


Figure 7-16 Command Window – Command Help page

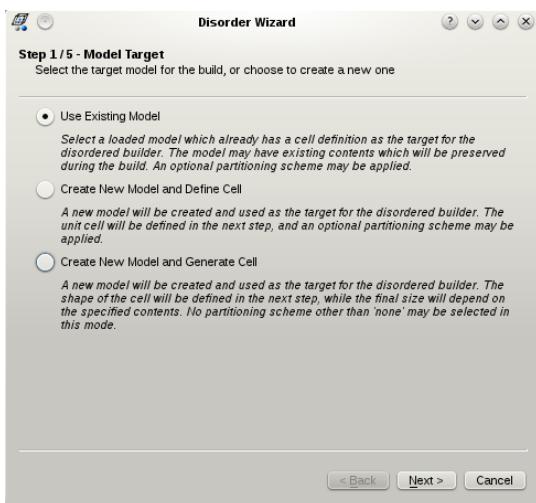
All of Aten's commands are listed in the panel in the upper half of the page, while the syntax and description of any command selected in the list is displayed in the lower half. The list can be searched by typing in a partial name in the **Search** box at the top.

7.11. Disorder Builder Wizard

The Disorder builder wizard generates a disordered system involving a collection of molecules. This can be a snapshot of a liquid suitable for use in a molecular dynamics program, a mixture of two or more species, an interface between two or more species, or something more exotic. The disordered builder requires that all molecules which are to be included in the system are currently loaded, and none must have its own cell defined. However, while all models which are to be ‘inserted into’ the final system may not possess a unit cell, a periodic model can be the target of the insertion, and may contain an existing collection of molecules / atoms.

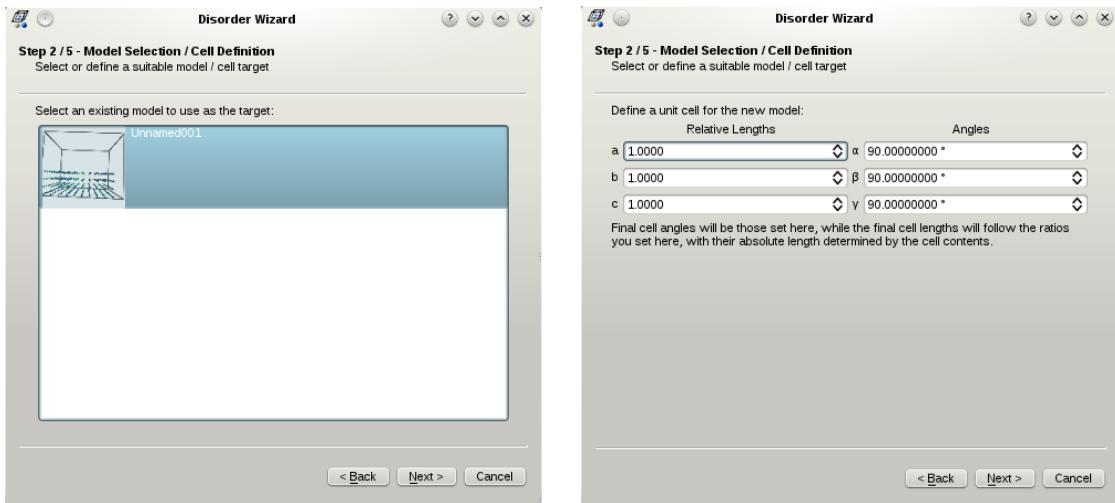
In order to begin the disordered wizard, click on the **Disorder** icon in the **Edit** section of the main toolbox (see Section 7.5). This will present the first of five steps designed to guide you through the building of your system.

7.11.1. Selection of Target Model



In the first wizard screen, the type of model target for the builder is selected. If a cell has already been defined in a loaded model choose **Use Existing Model**. The existing model target may already contain molecules, and these will be preserved at their original coordinates during the build. The remaining two choices create a new model for the builder to work on, but differ in how the unit cell is to be defined. **Create New Model and Define Cell** creates a new model for the system with a specific unit cell definition provided by the user, while **Create New Model and Generate Cell** creates a new model with a general cell definition which states only the angles and relative cell lengths required. In the latter case, the final size of the cell will be calculated by Aten based on the number and density of the requested molecules defined in subsequent steps. Note that this choice restricts the policy by which molecules are added into the system, requiring a population and density to be specified for every component. Furthermore, only the basic ‘unit cell’ partitioning scheme may be selected.

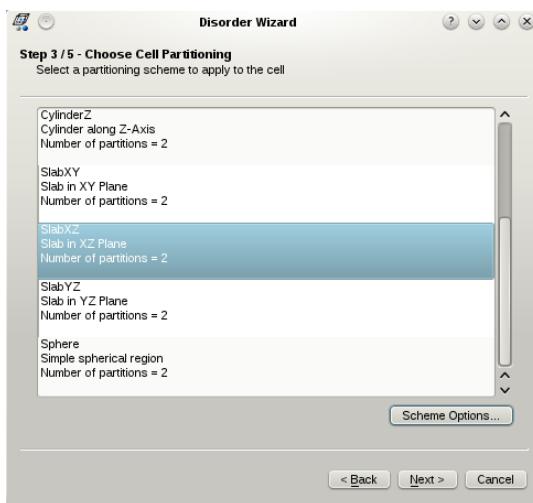
7.11.2. Selection / Definition of Target Model



The second step in the disordered builder either requires selection of an existing model, or definition of the (basic) unit cell, depending on the choice in Step 1:

If Use Existing Model was selected in the previous step, a list of currently loaded molecules which have a unit cell are shown, from which one should be selected (left-hand screenshot). If either **Create New Model and Define Cell** or **Create New Model and Generate Cell** was selected in Step 1, the unit cell must now be defined (right-hand screenshot). In the case of the former, the exact cell definition must be given, and will not be changed further by Aten. In the case of the latter, the required cell angles should be given as normal, but the relative lengths of the cell axes should be given rather than the absolute lengths. These lengths are subsequently adjusted by Aten based on the number and density of species defined later on. For instance, defining the relative cell lengths {A, B, C} to be {1.0, 1.0, 1.0} will result in all sides of the cell being identical, while setting them to {1.0, 2.0, 1.0} will give a final cell where the B length is twice as long as both A and C (which are the same).

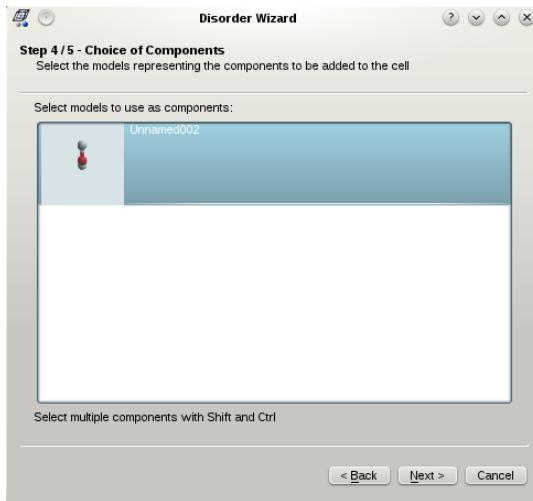
7.11.3. Choose Cell Partitioning



Aten allows each species to be added into a predefined region of the unit cell (unless the cell volume is to be generated, in which case only the basic ‘whole cell’ scheme is available). The various partitioning schemes available allow the cell to be split up into separate regions, for

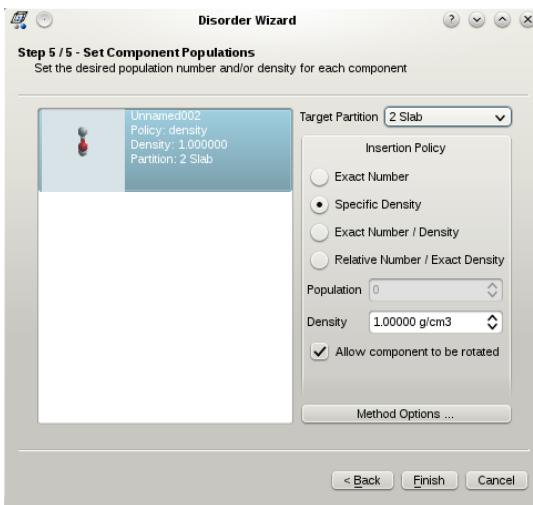
instance a spherical droplet in the middle of the cell. Many schemes have associated options which alter the size, shape, and position of the various partitions – select **Scheme Options...** to view and customise them. Partitions in the selected scheme are highlighted on the current unit cell as transparent areas.

7.11.4. Choice of Components



In step four the Disorder wizard provides a choice of components to be added into the system. Any loaded molecule which does not currently have a unit cell is displayed here, and any number of models may be selected for insertion.

7.11.5. Set Component Populations



Finally, the required populations and/or densities for each component model selected in the previous step must be set, as well as the target partition for each. There are four insertion policies available for each component:

Exact Number – The component will be added into the system/partition until the desired population has been reached. The density of the component in its partition is not important.

Specific Density – The component will be added into the system/partition until the desired density is reached. The population of the component is not important.

Exact Number / Density – The component will be added into the system/partition until both the specified population and density are reached. This is the only choice of policy if the cell size is to be determined by Aten (**Create New Model and Generate Cell** option in Step 1).

Relative Number / Exact Density – The component will be added into the system/partition until the specified density is reached. The total population of the component is variable but will depend on other components with the same policy selected. Over all such components, the relative population ratios between them will be maintained during the build. In this way, mixtures of specific mole ratio can be created, for example.

Note that, when specifying the density of a component, this refers to the density *within that components target partition*. This means that, when multiple components share the same partition (even in the basic unit cell scheme) the density refers to the total density of the partition, as calculated from the sum of contributions from each individual component. This simply means that the overall density should be defined when sharing partitions between components, rather than that required for each individual component.

The final step of the wizard also allows various parameters of the Disorder builder to be adjusted through the **Method Options** button. A brief description of these options follows:

TODO

The accuracy parameter determines what percentage error we will allow in actual vs requested densities

```
double accuracy = 0.01;  
  
// The maxfailures limit is the number of successive failed insertions we allow before we  
reduce the scale factor  
  
int maxFailures = 5;  
  
// The reductionfactor is the factor by which we multiply scale factors after reaching the  
limit of unsuccessful insertions  
  
double reductionFactor = 0.98;  
  
// Minimum scale factor to allow  
  
double minimumScaleFactor = 0.95;  
  
// Maximum scale factor to allow  
  
double maximumScaleFactor = 1.15;  
  
// Number of tweaks to attempt, per component, per cycle  
  
int nTweaks = 5;
```

```
// Maximum distance to translate molecule in tweak  
double deltaDistance = 0.3;  
  
// Maximum angle (each around X and Y) to rotate molecule in tweak  
double deltaAngle = 5.0;  
  
// Maxcycles and maxRecoveryCycles
```

7.12. Forcefields Window

The Forcefields is the place to go to load in and edit forcefields, perform atom typing on models, and calculate / minimise the energies of models. As well as being able to perform standard steepest descent and conjugate gradient minimisations, Aten also provides a molecular Monte Carlo minimiser, and the ability to run MOPAC directly from the GUI.

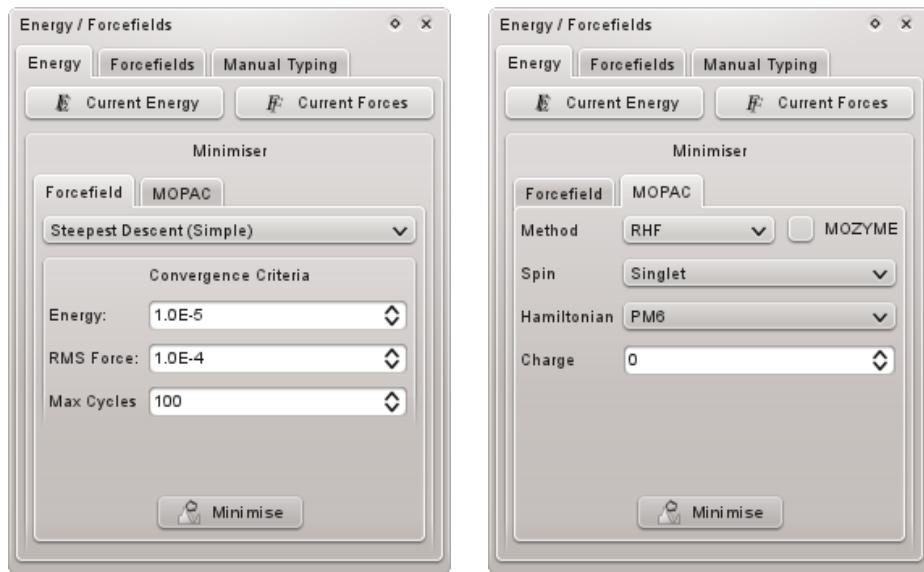


Figure 7-17 Forcefields Window – Energy minimisation controls

Energy Minimisation

The Energy page provides the means to perform geometry optimisations on loaded models using one of several methods. The forcefield selected in the Forcefields page is used to perform the minimisation, unless one has previously been explicitly associated to the model. The current energy and forces may also be calculated (akin to single-point energy calculations). The MOPAC minimiser requires that the locations of a valid MOPAC executable and temporary directory are defined – see Section 14.2 for more information. A job file is automatically written to disk and MOPAC executed when ‘Minimise’ is clicked, and the results loaded back in. Output of the program is buffered to the Messages window (see Section 7.17).

Forcefield Management

Forcefield files are managed through the **Forcefields** page. A list of currently-loaded forcefields is provided in the form of a drop-down list at the top; the selected item is the current default forcefield, and is used whenever one is required by a process but none has been linked to the target model. The forcefield selected in the list is the current forcefield, and the one used by all other actions on the page. Forcefields are loaded, unloaded, and edited with the buttons immediately underneath the list.

The Associate panel links the selected forcefield to one or more models and their patterns; the **Current Model** button links the current forcefield to the current model, while the **All Models** button links the current forcefield to all loaded models. The **Pattern in Current Model**

button brings up a dialog listing the patterns of the current model, from which one is selected to link the forcefield to. A forcefield associated to an individual pattern will be used in preference to the forcefield associated with its parent model (and, if none is assigned, the default forcefield).

Automatic atom typing can also be performed (or removed) from here. Finally, you may check that a full expression is available for the current model by pressing the **Create** button in the **Expression** panel at the very bottom. The nearby checkbox determines whether atomic charges should be assigned from atom type information, or whether the current charges (if any) should be left intact.



Figure 7-18 Forcefields Window – Forcefields page

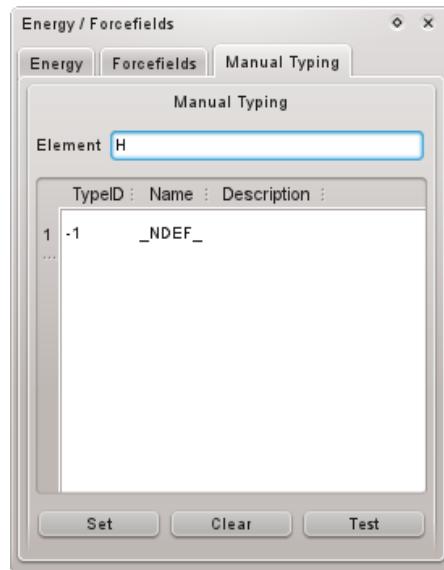


Figure 7-19 Forcefields Window – Manual typing page

Assigning Atom Types

Manual assignment of forcefield types can be performed in the **Manual Typing** page. The list gives all atom types in the current forcefield that are relevant to the element entered just

above the list. One can be selected from the list and be manually assigned (forced) onto the current atom selection with the **Set** button. Such assignments will not be overwritten by subsequent automatic typings. Manual typings can be removed with the **Clear** button, and the currently selected atomtype can be tested for suitability on the current selection of atoms with the **Test** button.

7.13. Fragments Window

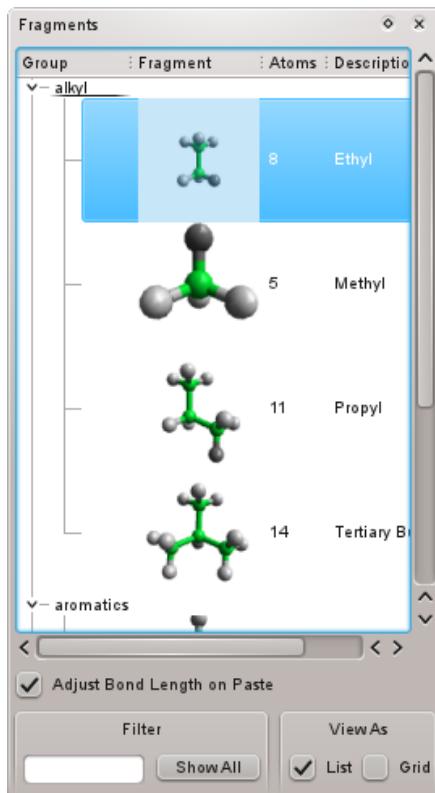


Figure 7-20 Fragments Window

When in fragment drawing mode, the **Fragments** window presents a list of all the available fragment models (in either **List** or **Grid** formats), and whichever is selected represents the current fragment to add in to the model. The current fragment is ‘attached’ to the mouse pointer when moving over the main canvas, and shows a preview of what will be the orientation and position of the fragment when a left-click is made.

If a fragment is drawn (attached) to an existing atom in a model, the resulting bond length is adjusted to match the two elements involved if the **Adjust Bond Length on Paste** checkbox is ticked.

Fragment Models

Fragment models are just normal models stored in specific places. The only real difference is that atoms with unknown element (integer ‘0’, or string ‘XX’) are slightly more useful than usual. These can act as anchor points for the fragment that do not correspond to the position of any ‘proper’ atom to allow for more control over the placement of structures - for instance, such an atom may be placed at the centre of a ring. It is important to note that all atoms with unknown element type are removed when the fragment is added to the current model.

Anchor Points

At any time, a single atom in the selected fragment represents the attachment point and appears directly under the mouse pointer. Any atom in the fragment can be selected as the

current anchor point, and may be cycled through by pressing **Alt**. If the atom acting as the anchor point has at least one bond to another atom then a reference vector is constructed allowing the fragment to be oriented along vectors dependent on the current geometry of existing atoms, for example.

Placing Fragments in Free Space

Single-clicking in free space (i.e. not over an existing atom) places a copy of the current fragment in the current orientation in the current model. Click-dragging allows the fragment to be freely rotated about the anchor point before the placement is final. On placement, the anchor atom is pasted too *unless* it is of unknown element (see above).

Attaching to Existing Atoms

When moving over an existing atom with an anchor point that has at least one bond to another atom, the fragment is attached at a suitable geometry, provided the valency of the atom allows (if not, the atom will be surrounded by a crossed-out box and the fragment will temporarily disappear). A single-click will place the fragment in the shown position and orientation, while click-dragging rotates the fragment about the newly-formed bond to allow fine adjustment before the final placement. If the anchor point has no bonds to other atoms, the fragment is placed in no specific orientation, and click-dragging freely rotates the molecule about the anchor point. In both cases, the anchor atom is *always* removed when being added to the model since the existing atom which was clicked on will replace it in terms of position.

If the existing atom in the model has one or more other atoms attached to it, holding **Shift** will orient the fragment along an existing bond vector. In this case, both the anchor atom and the atom at the other end of the bond are removed when the fragment is placed. Pressing **Ctrl** cycles over the bonds of the existing atom.

7.14. Geometry Window

Simple measurements of distances, angles, and torsions between picked atoms or the current selection of atoms is possible from the measure toolbar.

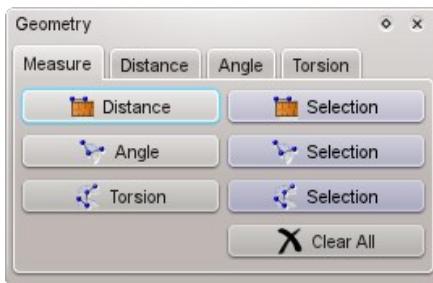


Figure 7-21 Geometry Window – Measure page

Distances, angles and torsions between atoms can be measured here – select the desired tool on the left-hand side and click the relevant number of atoms one-by-one in the main window. Alternatively, select a molecule or group of atoms for which you wish to measure all (bound) instances of a certain type of geometry, and press the corresponding button on the right-hand side. To clear all measurements from the current model, press the **Clear All** button.

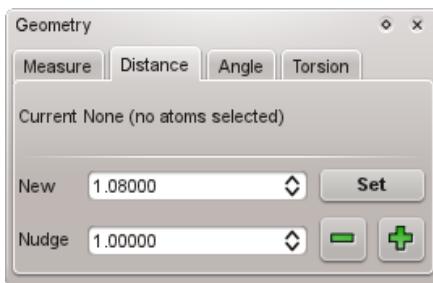


Figure 7-22 Geometry Window – Distance page (angle and torsion pages are similar)

The **Distance**, **Angle**, and **Torsion** pages of the Geometry window allow the specific geometry of a set of selected atoms to be set. Simply select, for example, two atoms from a model (note that they do not have to be connected by a bond) and set or adjust their distance from here. When choosing the **Set Bond Length**, **Set Bond Angle**, and **Set Torsion Angle** items on the atom context (or **Selection**) menu, the Geometry window is automatically presented on the correct page for the number of selected atoms.

7.15. Glyphs Window

TODO XXX

Glyphs Window

The glyphs window allows new glyphs to be created by hand, or existing glyphs in the model to be managed, edited, and deleted.

Figure 5.13. Glyphs Window



All glyphs in the current model are listed to the left. Selecting one will allow its properties to be edited on the right. The type of glyph can be changed also. Colours for individual glyph data (i.e. points) may be changed through the colour controls associated to each.

7.16. Grids Window

The **Grids** window provides management for grid data sets owned by the different models. All grid data sets held by the current model are displayed here. The appearance of individual grids may be changed, and axes / cutoffs changed. Grid data can also be cut, copied, and pasted to different models.

A ‘grid’ in Aten’s world can be more or less any kind of volumetric or surface data. For instance, orbital densities, 2D height maps, molecule probability densities etc. See the Grids section for more information.

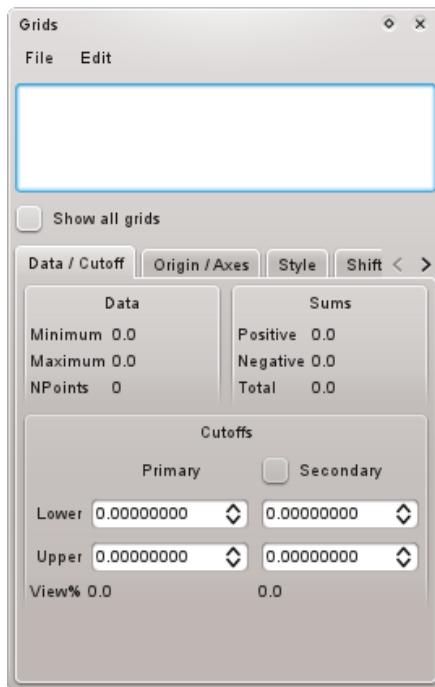


Figure 7-23 Grids Window – Data / Cutoff page

General Grid Management

Each grid in the list has associated with it a checkbox that determines whether or not it is currently visible. Below the list the minimum and maximum data values contained within the grid, and the current cutoffs (i.e. the values for which the isosurface is drawn at) for the primary and (optional) secondary surfaces to use when rendering the data. The **File** and **Edit** menus allow new grids to be loaded into the current model, and to cut, paste, and delete grids between models. Multiple grids may be selected at once in the listbox – in such a case any alterations made using the other controls are applied to all of the selected grids. In this way all cutoff values, for instance, may be set simultaneously to the same values for any number of loaded grids.

By default, only those grids associated to the current (active) model are visible, but all grids which exist within all models can be displayed with the **Show All Grids** checkbox.

The **Secondary** checkbox specified that a second isosurface should be created for the current grid. This is useful, for instance, to draw a second, transparent surface at a lower cutoff than

the primary, or to display an isosurface encompassing the negated cutoff range of the primary (e.g. to display both signs of the wavefunction in orbital data).

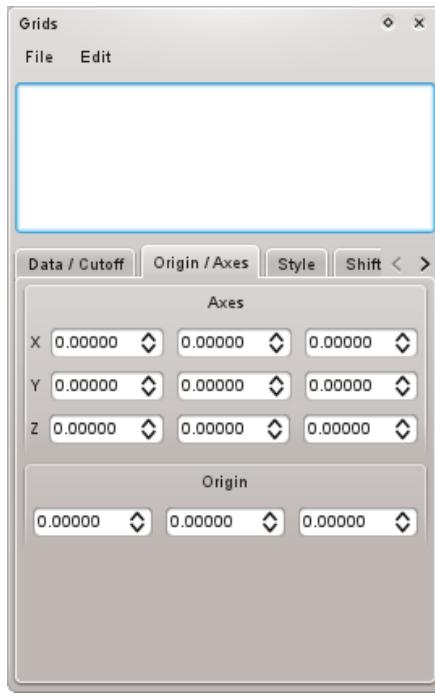


Figure 7-24 Grids Window – Origin / Axes page

The associated origin and axes to the currently selected grid can be modified here. In this way grid data may be arbitrarily flipped, stretched, and sheared, and its position in the local space of the model changed. All changes made are reflected immediately in the main view.

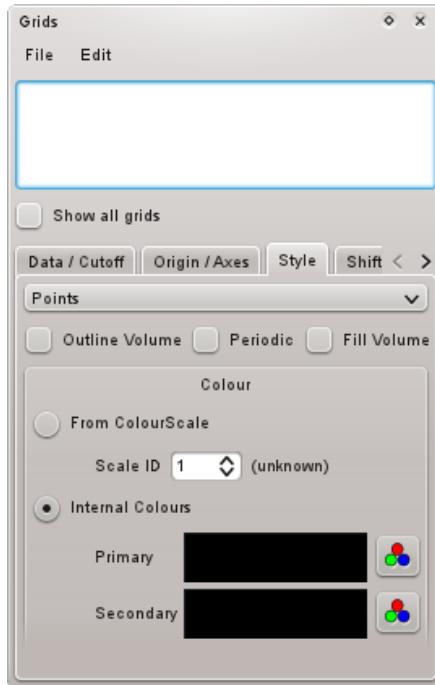


Figure 7-25 Grids Window – Style page

The general appearance of the selected grid can be modified in the Style page, with the main rendering style set with the combo box (see Section 16.10 for a list of possible styles).

Optionally, a bounding box around the grid data may be drawn by selecting the **Outline Volume** checkbox. The **Periodic** checkbox influences the way 3D surfaces are drawn close to the boundaries of the grid. If unchecked, a ‘gap’ will appear near to the edges of the grid volume since there is insufficient data with which to generate gradient information. If checked, data points on the opposite site of the grid will be used in this region (useful, for instance, in the case of orbital information calculated in periodic systems). For volumetric data, sometimes it is useful to see the filled volume rather than the enclosing surface – checking **Fill Volume** will do just that.

Colours for the primary and (if active) secondary surfaces can be set to simple (single) colours by selecting **Internal Colours** and choosing whatever colour is desired from the colour selection buttons. Should a more useful colouring scheme be desired than the surfaces may be rendered using a colour scale (see Section 10.1), in which case the isovalue (if volumetric) or height (if a 2D grid) determines the colour of the data point. To do so, select **From Colourscale**, and choose the ID of the colour scale to use (from the ten definable in the program).

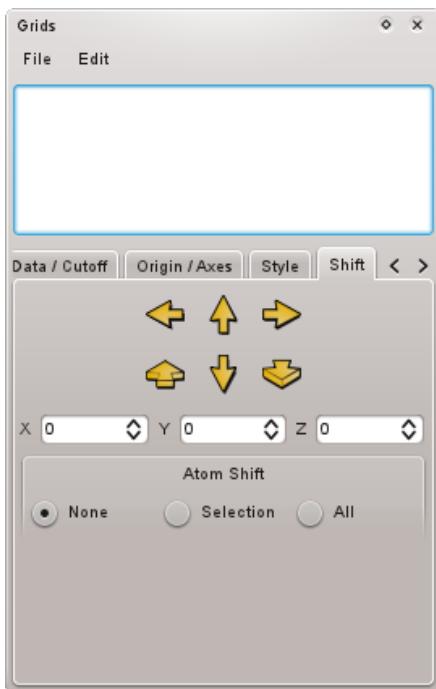


Figure 7-26 Grids Window – Shift page

The **Shift** page can be used to ‘translate’ the grid data points along each axis of the data set allowing, for instance, isosurface features of interest in a periodic grid to be displayed in a more central position. Note that no modification of the actual grid data is performed – the effect is entirely visual. The **Atom Shift** group allows translation of selected/all atoms in the parent model to be moved at the same time, so as to correlate them with the new positions of the grid data.

Orbitals Page

Not implemented yet.

7.17. Messages Window

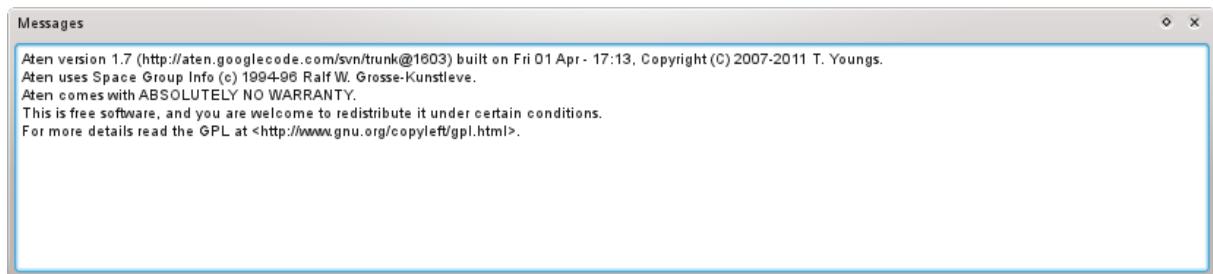


Figure 7-27 Messages Window

The messages window is simply that – a place for all of Aten’s text output to appear when the GUI is active. All error messages are printed to the messages window – if something doesn’t work as you expect it should (or, indeed, doesn’t work at all) there is usually a clue or, better, an explanation somewhere in the messages displayed here.

Sometimes messages will be printed which have the appearance of hyperlinks – these can be clicked on to perform various simple tasks on the model, relevant to the accompanying message.

7.18. Pore Builder Window

The **Pore Builder Window** offers some tools to make the construction of (regular) porous material systems a little easier, including when it comes to filling pores.

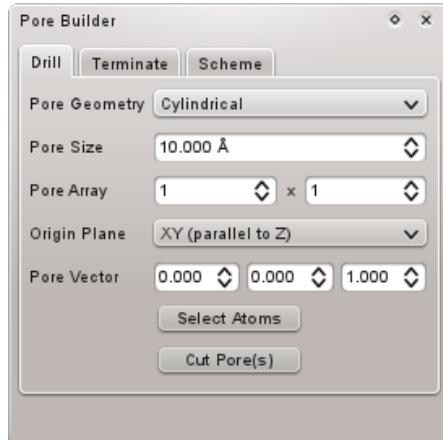


Figure 7-28 Pore Buider Window – Drill page

The **Drill** page provides tools to select and cut atoms from an array or pores of specified geometry in the current model. The **Origin Plane** determines the crystal face from which the pores originate, but from there the direction of drilling (the **Pore Vector**) may be set to anything although something coincident with the normal of the origin plane is probably sensible. The magnitude of this vector is unimportant, since it is normalised before use. Once all quantities are suitably defined, the atoms making up the pores may be simply selected (with **Select Atoms**) or deleted from the model (with **Cut Pore(s)**). The reason for providing both these actions is to allow for selection of atoms at the surfaces resulting from pore cutting, simply by increasing the **Pore Size** parameter a little.



Figure 7-29 Pore Builder Window – Terminate page

At present, the **Terminate** page provides a single button which terminates (adds H or OH to) any atoms in the current selection which have fewer than their normal quota of bonds as a result of pore drilling (or by other means). H or OH are added to atoms until the correct number of bonds per atom is restored. Only O and Si are considered at present.

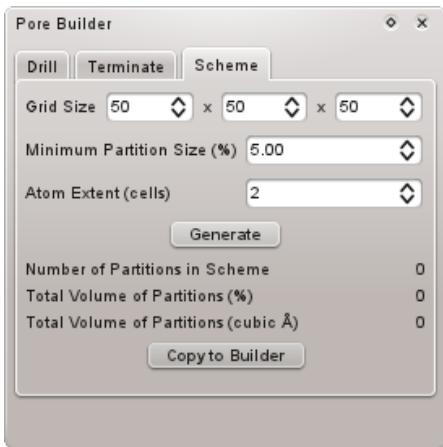


Figure 7-30 Pore Builder Window – Terminate page

The **Scheme** page permits a partitioning scheme (see Section 10.4) to be created from free space in the current model (not necessarily space created by pore drilling). The **Grid Size** determines how finely to divide up space in the current model – the defaults should be fine for most purposes, but exotic or very small shapes may require higher values. The minimum allowable size for a partition is given as a percentage of the total number of cells in the grid (**Minimum Partition Size**), below which any discovered regions of space are ignored. In order to correctly generate partitions it is often necessary to tweak the **Atom Extent** parameter. Any grid cell in which an atom exists is considered ‘full’ and not part of any partition, but in most cases this is not sufficient and more space must be excluded to account for the real space-filling of the atoms. The **Atom Extent** value defines a radius, in cells, within which additional cells around atoms will also be removed.

The **Generate** button will search for partitions in the model and overlay these partitions onto the main display, but will not copy the data to the Disorder builder. Once you are satisfied that the correct / relevant partitions have been discovered, the data can be made available in Disorder building by clicking **Copy to Builder**.

7.19. Position Window

Tools for the absolute positioning of atoms are available here. All work on the current selection of atoms in the current model.

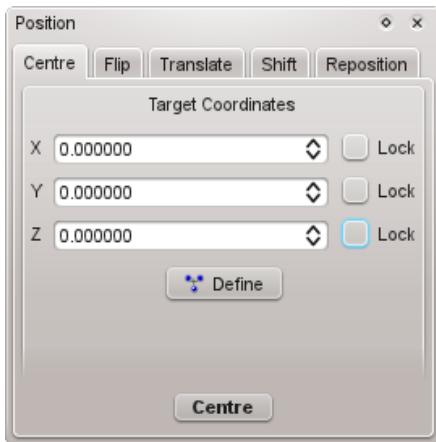


Figure 7-31 Position Window – Centre page

The **Centre** page allows the centre of geometry of the current selection to be positioned at absolute coordinates. The desired position is entered in the three input boxes, or can be defined from the geometric centre of a selection of atoms (prior to the positioning of a different set). Any (or all) of the Cartesian axes may be locked preventing coordinate adjustment along particular directions.



Figure 7-32 Position Window – Flip page

The **Flip** page mirrors the positions of atoms in the current selection through its centre of geometry in either the X, Y, or Z directions. Note that this tool currently works only along the Cartesian axes, and does not take into account the shape of any defined cell.

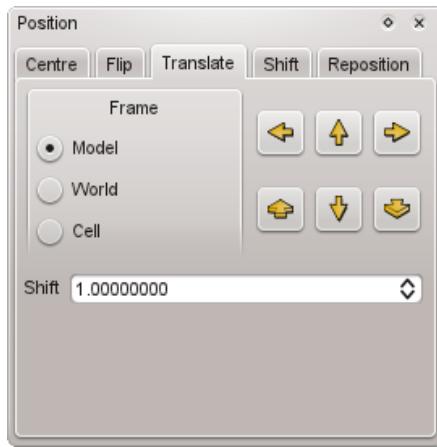


Figure 7-33 Position Window – Translate page

Translations of atoms within model (local), world (view) and cell frames of reference can be performed in the **Translate** page. The group of directional buttons move the selected atoms along the relevant axis within the selected frame of reference, and by the amount specified in the **Shift** control. For model and world reference frames the **Shift** control specifies the number of Angstroms moved along the axis in each step. For the cell reference frame it defines the fractional cell distance moved in each direction.

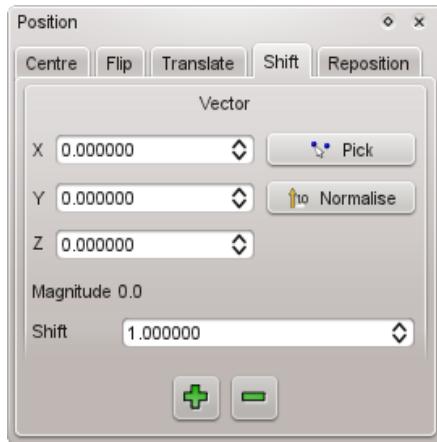


Figure 7-34 Position Window – Shift page

The vector along which to move the current selection is defined on the left hand side of the **Shift** page. Furthermore, the axis may be defined by **Picking** two atoms in the main window. The supplied vector does not need to be normalised, but thus may be performed through the **Normalise** button. The defined shift value dictates the multiple of the defined vector by which selected atoms are shifted.

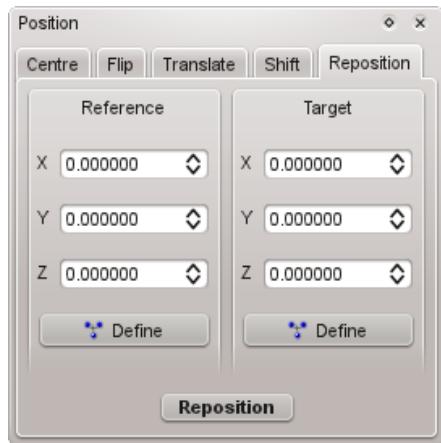


Figure 7-35 Position Window – Reposition page

The reposition page allows the centre of geometry of a selection of atoms to be moved from a reference coordinate (defined by the **Reference** panel) to a destination coordinate (defined by the **Target** panel). Either coordinate may be set from the centre of geometry of the current atom selection by pressing the relevant **Define** button.

7.20. Select Window

The select window provides a means to quickly access the capabilities of the **select** and **deSelect** commands from within the GUI. The total number of atoms selected along with their empirical formula is also displayed.



Figure 7-36 Select Window with ID / Element Tab open

Basic atom selection operations can be made using the **All**, **None**, **Invert**, and **Expand** buttons at the top of the window. The first three are self-explanatory, selecting all atoms, no atoms, and reversing the current selection respectively. The final button, **Expand**, looks at any bonds attached to any currently-selected atoms, and selects those atoms that are bound as well. Only the immediate bound neighbours of each atom are considered each time the expansion is made.



Figure 7-37 Select Window with NETA Tab open



Figure 7-38 Select Window with For Loop Tab open

Three more advanced selection methods are also available. In each case, selection or deselection of atoms based on the defined range(s) is made by the associated **Select** and **Deselect** buttons.

Selection by ID/Element Ranges

Ranges of atom IDs and/or elements are specified as ‘ $a-b$ ’ where a and b are either both atom IDs or both element symbols. In addition, the ‘+’ symbol can be used before (as in ‘ $+a$ ’) or after (as in ‘ $a+$ ’) an atom ID or element symbol to mean, respectively, ‘everything up to and including a ’ or ‘ a and everything after’. Multiple selection ranges may be given, separated by commas.

Select by NETA Description

The NETA language implemented in Aten also makes for a powerful tool when selection specific atoms in a model based on their chemical environment. The relevant NETA description for the target atoms should be inserted into the **Code** editbox, and the corresponding **Element** set.

Select by For Loop

More advanced still, here a short snippet of command-language code can be written. The code is automatically inserted into a function which is subsequently used in a for loop of the following construction:

```
int shouldSelect(Atom i)
{
    // ... code is inserted here
}

for (Atom i = aten.model.atoms; i; ++i)
{
    if (shouldSelect(i)) i.selected = TRUE;
}
```

Thus, the code should use the variable `i` to test for whatever criteria are necessary to cause the atom to be selected, and return `TRUE` (1) if it is to be selected, and `FALSE` (0) otherwise. In this mode, the **Deselect** button simply uses the reverse code:

```
int shouldDeselect(Atom i)
{
    // ... code is inserted here
}

for (Atom i = aten.model.atoms; i; ++i)
{
    if (shouldDeselect(i)) i.selected = FALSE;
}
```

7.21. Trajectory Window

If a trajectory is associated to the current model, the trajectory window allows to skip through frames and playback the trajectory.

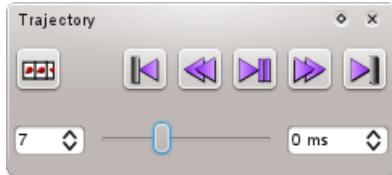


Figure 7-39 Trajectory Window

The button serves to act as a toggle between displaying the actual trajectory frames and the parent model which ‘owns’ those trajectory frames. Its state is linked to the **Trajectory**→**View Model** and **Trajectory**→**View Trajectory** menu items of the main window, and vice versa. The set of standard playback controls allow individual frames to be skipped, as well as going straight to the beginning or end of the trajectory. The current frame may also be set with either the spin control or slider at the foot of the window. The delay between swapping of frames is controlled by the slider on the right where the delay can be set in milliseconds. Note that the actual speed of playback will also depend on the size of the system being viewed and the raw power of your graphics card.

7.22. Transform Window

For a selection of atoms, rotational or matrix-based transformations can be applied through the Atom Transform window.

Rotation About Arbitrary Axis

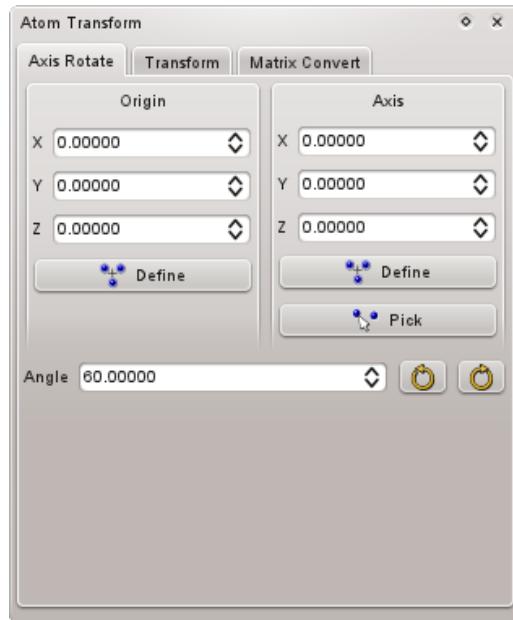


Figure 7-40 Transform Window – Axis Rotate page

The **Rotate** page allows an origin and a rotation axis about this origin to be defined about which to rotate atom selections. The origin and axis may be entered manually, can be determined from a current atom selection (**Define** buttons), or defined by the click-selection of two atoms (**Pick** button). Defining the rotation axis from the current selection will set the axis to the vector between the currently-defined origin and the centre of geometry of the current atom selection. Rotations of atom selections about this axis/origin combination are then made by defining the **Angle** of rotation (in degrees) and then applying the rotation either clockwise or anticlockwise.

Matrix Transformation

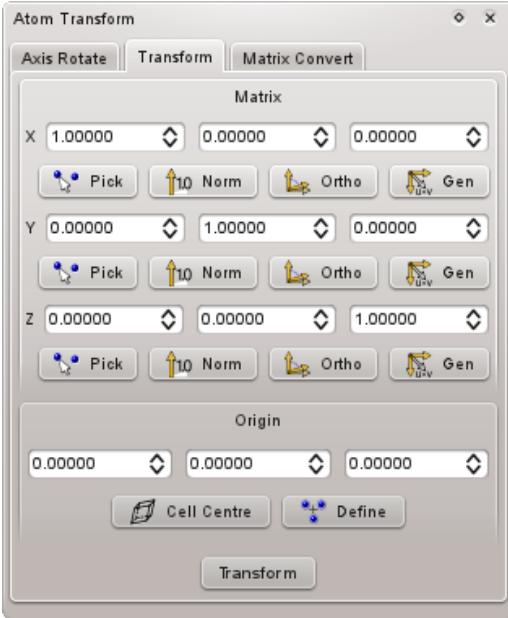


Figure 7-41 Transform Window – Transform page

From here a 3×3 transformation matrix can be applied to the current atom selection, and with a specific coordinate origin (not necessarily the centre of geometry of the selection). The **Pick** buttons allow selection of the various axes through click-selection of two atoms (not necessarily in the same model as the current atom selection), while the **Normalise** buttons will normalise each of the currently-defined axes to be of unit length. Finally, the **Orthogonalise** and **Generate** buttons allow particular axes to be orthogonalised relative to, or generated from, the other defined axes.

Matrix Conversion

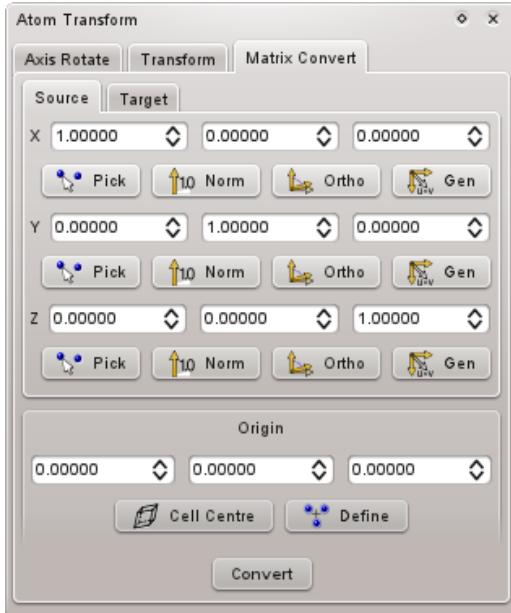


Figure 7-42 Transform Window – Matrix Convert page

It is possible to transform the orientation of a given set of atoms if a suitable pair of source and destination matrices are defined. The **Source** matrix defines what should be considered the current frame of reference for the current set of coordinates. Once the **Convert** button is pressed, a rotation matrix is generated such that, when applied to the **Source** matrix (and the current atom selection) it will be transformed into the **Target** matrix.

As for the matrix convert page, the **Pick**, **Normalise**, **Orthogonalise**, and **Generate** buttons allow each axis to be selected, normalised, orthogonalised relative to the others, and generated from the others respectively.

7.23. ZMatrix Window



The z-matrix window allows the z-matrix for the current model to be viewed and its variables edited. Any changes made are reflected immediately in the coordinates of the model.

Figure 5.29. ZMatrix Window



8. Command Language

8.1. Command Language Overview

The scripting language in Aten is based syntactically on C/C++, with echoes of useful Fortran features included for good measure. This means that if you're familiar with C/C++ then controlling Aten from the command line, or writing a script, command, or filter should be relatively straightforward. There are lots of C primers on the web, so the following sections provide only a brief summary of the style of the language.

8.1.1. General Input Style

Keywords, variables, and function names are case sensitive, so ‘for’ is different from ‘For’ (all internal commands in Aten are in lowercase). Individual commands must be separated by a semicolon ‘;’, and newlines are optional. For example:

```
int i; printf("Hello there.\n"); i = 5;
```

...and...

```
int i;
printf("Hello there.\n");
i = 5;
```

...are equivalent. Whitespace characters (spaces and tabs) are ignored and may be present in any amount in any position in the code.

Individual lines or parts of them may be commented out. The presence of either the hash symbol # or // in a line means ‘ignore rest of line’. Note that the /* ... */ commenting style is not supported.

8.1.2. Variables

There are several standard rules for variables within Aten:

- They must be declared before use
- They are strongly-typed, i.e. they hold a specific type of value (e.g. an integer, a character string etc.)
- They must begin with an alpha character ('a' to 'z'), but may otherwise consist of letters, numbers, and the underscore ('_')
- They may not be named the same as any existing function or internal keyword
- Since variables are strongly-typed, trying to set an integer variable from a string will not work since these are incompatible types. However, standard C-style string conversion functions are available - see String Commands in Section 9.29.

So, to initialise some variables, assign values to them, and print them out, we would do the following:

```
int i;
double result;
i = 10;
result = i*5.0;
printf("i multiplied by 5.0 = %f\n", result);
```

In addition to the standard `int` and `double` types, a `string` variable exists for the storage of arbitrary-length character strings, and do not need to have a length specified on their creation (they will adjust dynamically to suit the assigned contents). Literal character strings should be surrounded by double-quotes. A set of variable types exist that are able to contain references (not copies) of various objects within Aten, e.g. atoms, models, unit cells, etc. Variables of these types are declared in exactly the same way as normal variables (see Section 8.2 for a list of available types). A `vector` type is provided for convenience and stores a triplet of `double` values. There is no boolean type – use an `int` instead – but the built-in constants (see Section 8.1.4) `TRUE` and `FALSE` may be used in assignment, etc., and correspond to the integer values ‘1’ and ‘0’ respectively.

All variables will default to sensible values (e.g. empty string, numbers equal to zero) on initialisation. To create a variable with a specific value simply do the following:

```
int i=1,j=2,k=1001;
double myvar=0.0, angle = 90.0;
```

8.1.3. Arrays

Arrays of most variable types are allowed (some, for instance the `Aten` type, don't really make sense as an array). Arrays are requested by providing a constant size (or an expression) in square brackets after the name:

```
int values[100], i = 4;
double q[i];
```

Here, two arrays are created - an array of 100 integer numbers called `values`, and an array of four floating point numbers called `q`. Array indices always run from 1 to the size of the array, unlike C in which arrays run from 0 to N-1. Note that it is **not** possible to use a custom range of array indices, as is the case in Fortran.

Arrays can be initialised to a simple value on creation, setting all elements to the same value...

```
string s[4] = "Nothing";
```

...or each element to a different value using a list enclosed in curly brackets:

```
string s[4] = { "Nothing", "Nicht", "Nada", "Not a sausage" };
```

Also, all array elements can be set to the same value with a simple assignment:

```
int t[100];
t = 40;
```

8.1.4. Predefined Constants

Several predefined constants exist, and may not be overridden by variables of the same name. All predefined constants are defined using uppercase letters, so the lower case equivalents of the names *may* be used as variables, functions etc.

Table 8-1 Built-In Constants

Name	Type	Value
ANGBOHR	double	0.529177249
AVOGADRO	double	6.0221415E23
DEGRAD	double	57.295779578552
FALSE	int	0
NULL	int	0
PI	double	3.14159265358979323846
TRUE	int	1

In addition, all element symbols found in the input will be seen as their equivalent integer atomic number. So, instead of having to provide short strings containing the element name to, for example, the **transmute** command, simply the capitalised element name itself may be used. Thus...

```
transmute("Xe");
transmute(Xe);
transmute(54);
```

...are all entirely equivalent.

8.1.5. Blocks, Scope, and Variable Hiding

As with C, variable scope is employed in Aten meaning that a variable may be local to certain parts of the code / filter / script. In addition, two or more variables of the same name (but possibly different types) may peacefully co-exist in different scopes within the same code / filter / script. Consider this example:

```
int n = 4, i = 99;
printf("%i\n", i);
if (n == 4)
```

```
{
    int i = 0;
    printf("%i\n", i);
}
printf("%i\n", i);
```

...will generate the following output:

```
99
0
99
```

Why? Well, even though two variables called *i* have legitimately been declared, the second declaration is made within a different block, enclosed within a pair of curly braces. Each time a new block is opened it has access to all of the variables visible within the preceding scope, but these existing variable names may be re-used within the new block in new declarations, and does **not** affect the original variable. Hence, in the example given above, after the termination of the block the final printf statement again sees the original variable *i*, complete with its initial value of 99 intact. Note that if a variable is re-declared within a scoping block, there is no way to access the original variable until the scoping block has been closed. Blocks can be nested to any depth.

One exception to this rule is when user functions are declared – inside the function no variables declared outside the function are visible. Usually this is the safest practice, since it means that the rest of the code cannot be corrupted by a rogue function. However, on occasion it is useful or necessary to be able to use a function to change a variable which exists in the main program or enclosing scope. Here we introduce the concept of *global* variables – variables that are declared in either the main program scope or within a function, but are also accessible by any other functions defined within that scope. Such variables can be created by using the global prefix as follows:

```
global int x = 2;
void doubleVariable()
{
    x = 2 * x;
}

# Execute the function
doubleVariable();
printf("x is %i\n", x);
```

This will print “x is 4”. Without the *global* prefix on the declaration of the variable *x*, the code will not compile since, inside the function, the variable *x* is ‘hidden’ from this scope.

8.1.6. Functions

For functions that take arguments, the argument list should be provided as a comma-separated list in parentheses after the function name. For instance:

```
newAtom("C", 1.1, 0, 4.2);
```

Arguments may be constant values (as in this example), variables, arithmetic expressions, or any other function that returns the correct type. For functions that (optionally) do not take arguments, the empty parentheses may be omitted. A list of all functions, their arguments, and their return types is given in the command reference in Section 9.

All functions return a value, even if it is ‘no data’ (i.e. ‘void’ in C/C++). For instance, in the example above the **newAtom** command actually returns a reference to the atom it has just created, and this may be stored for later use:

```
Atom a;
a = newAtom("C", 1.0, 1.0, 1.0);
```

However, if the return value of any function is not required then it may simply be forgotten about, as in the example prior to the one above.

8.1.7. User Defined Functions

Custom functions may be defined and used in Aten, taking a list of variable arguments with (optional) default values. The syntax for defining a function is as follows:

```
type name ( arguments ) { commands }
```

`type` is one of the standard variable types and indicates the expected return value type for the function. If no return value is required (i.e. it is a subroutine rather than a function) then `type` should be replaced by the keyword `void`:

```
void name ( arguments ) { commands }
```

Once defined, functions are called in the same way as are built-in functions. The `name` of the function obeys the same conventions that apply to variable names, e.g. must begin with a letter, cannot be the name of an existing keyword, function, or variable. The `arguments` are specified in a similar manner to variable declarations. A comma-separated declaration list consisting of pairs of variable types and names should be provided, e.g.:

```
void testroutine ( int i, int j, double factor ) { ... }
```

Our new subroutine `testroutine` is defined to take three arguments; two integers, `i` and `j`, and a double `factor`. All three must be supplied when calling the function, e.g.

```
printf("Calling testroutine...\n");
int num = 2;
double d = 10.0;
testroutine(num, 4, d);
printf("Done.\n");
```

When defining the function/subroutine arguments, default values may be indicated, and permit the function to be called in the absence of one or more arguments. For instance, lets say that for our **testroutine**, the final argument *factor* is likely to be 10.0 on most occasions. We may then define a default value for this argument, such that if the function is called without it, this default value will be assumed:

```
void testroutine ( int i, int j, double factor = 10.0 ) { ... }

printf("Calling testroutine...\n");
int num = 2;
testroutine(num, 4);
testroutine(num, 4, 20.0);
printf("Done.\n");
```

Both methods of calling **testroutine** in this example are valid.

8.1.8. Return Values

For functions defined to return a specific type, at some point in the body of the function a suitable value should be returned. This is achieved with the **return** keyword. Consider this simple example which checks the sign of a numeric argument, returning 1 for a positive number and -1 for a negative number:

```
int checksign ( double num )
{
    if (num < 0) return -1;
    else return 1;
}
```

If an attempt is made to return a value whose type does not match the type of the function, an error will be raised. Note that, once a **return** statement is reached, the function is exited immediately. For functions that do not return values (i.e. those declared with **void**) then **return** simply exits from the function – no return value need, or should, be supplied.

8.1.9. Arithmetic Expressions and Operators

Arithmetic operates in the same way as in C, and utilises the same operator precedence etc. Similarly, comparison operators (less than, equal to etc.) are the same as those found in C.

8.2. Variable Types

8.2.1. Overview

For variables in Aten that are of reference type (i.e. the non-standard types) various sub-variables and functions may be accessed in the same way class members are utilised in C++. Each non-standard variable type is listed here along with the types and descriptions of their available subvariables / members.

In the same way that class members are accessed in C/C++, subvariables of reference types are accessed by use of the full stops between member names. For instance, the `Atom` type allows all the information of a given atom to be accessed. The following example illustrates the process of retrieving the third atom in the current model, finding its coordinates, and subsequently adjusting them:

```
Atom a = aten.model.atoms[3];
Vector v = a.r;
printf("Old coordinates are: %f %f %f\n", v.x, v.y, v.z);
v.x += 4.0;
a.r = v;
printf("New coordinates are: %f %f %f\n", a.rx, a.ry, a.rz);
```

Lots of paths are used here. Firstly, the global `Atenvvariable` is used to get access to the current model and grab a reference to its third atom (`aten.model.atoms [3]`). Then, the coordinates of this atom are placed in a vector `v` by requesting the `r` subvariable from the stored `Atom` reference. We then adjust the vector and store the new coordinates in the atom.

All members can be read, but not all can be written back to – these are read-only members for which setting a value makes no sense. Members which can be modified are indicated with mark in the ‘RW’ column in the following tables.

Function members act in the same as subvariable members except that they take one or more arguments enclosed in parentheses immediately following the member name, just as command functions do.

8.2.2. Aten Type

The ‘master’ class type, `Aten`, is there to provide access to various other structures such as the list of loaded models, preferences, element data etc. It is available at all times from any command, script, or filter, and is always called `Aten`. Note that it is not possible to declare new variables of type `Aten`.

Table 8-2 Aten Type Members

Member	Type	RW	Description
elements	Element []		Array of element data (masses, symbols, names, etc.)
frame	Model		The current model being displayed and the focus of editing, i.e. the current trajectory frame or, if no trajectory is

		associated to the current model then the current selected model is returned
model	Model	The current model selected (this is the parent model of a trajectory if a frame is currently being displayed)
models	Model []	Array of all loaded models currently available
nelements	int	Number of chemical elements defined in the elements array
nmodels	int	Number of models (parent models, i.e. not including trajectory frames) currently loaded
prefs	Prefs	Program preferences

8.2.3. Aten Type Member Functions

convertEnergy

Syntax:

```
double convertEnergy ( double value, string oldunits )
```

Convert the supplied energy *value* from *oldunits* to the current, internal unit of energy in use by Aten. See also Energy Units (Section 16.8) and the relevant `Prefs` accessor.

findElement

Syntax:

```
Element findElement ( string name )
```

Convert the *name* specified into an Element, according to the current ZMapping Type (Section 16.17).

8.2.4. Atom Type

The Atom type encompasses a single atom in a model or frame.

Table 8-3 Atom Type Members

Member	Type	RW	Description
bit	int	•	Temporary integer value, with associated bit-setting functions (see Section 8.2.5)
bonds	Bond []		List of bonds the atom is involved in
colour	double [4]	•	Custom colour of the atom (used when the Colouring Scheme, see Section 16.5, is set to ‘custom’)
data	string	•	Temporary character data stored on the atom (for use in filters etc.)
element	Element	•	Returns a pointer to the assigned element data of the atom
fixed	int	•	Whether the atom's position is fixed (1) or not (0)
f	Vector	•	Force vector
fracz	double	•	Position x-component in fractional cell coordinates
fracy	double	•	Position y-component in fractional cell coordinates
fracz	double	•	Position z-component in fractional cell coordinates

fx	double	• Force x-component
double	• Force y-component	
fz	double	• Force z-component
hidden	int	• Whether the atom is hidden (1) or visible (0)
id	int	Numerical ID of the atom within its parent model
mass	double	Atomic mass of the atom
name	string	Element name of the atom
q	double	• Atomic charge associated to the atom
r	Vector	• Position vector
rx	double	• Position x-component
ry	double	• Position y-component
rz	double	• Position z-component
selected	int	• Whether the atom is selected (1) or unselected (0)
style	string	• The current Drawing Style of the atom (see Section 16.7)
symbol	string	Element symbol of the atom
type	FFAtom	• Forcefield type of the atom
v	Vector	• Velocity vector
vx	double	• Velocity x-component
vy	double	• Velocity y-component
vz	double	• Velocity z-component
z	int	• Atomic number of the atom

8.2.5. Atom Type Functions

addBit

Syntax:

```
void addBit ( int bit )
```

Add (set) the specified bit for this Atom. If it is desired to set the Atom's bit to a certain value, use the accessor listed in Section 8.2.4.

copy

Syntax:

```
void copy ( Atom source )
```

Copy all information from the source Atom into this Atom, except for its id.

findBond

Syntax:

```
Bond findBond ( Atom i )
```

Return the Bond (if any) between this and the specified Atom *i*.

hasBit

Syntax:

```
int hasBit ( int bit )
```

Return whether this Atom has the specified *bit* set.

removeBit

Syntax:

```
void removeBit ( int bit )
```

Remove the specified *bit* from the Atom's bit variable.

8.2.6. BasisPrimitive Type

The BasisPrimitive type provides access to basis primitive coefficient and exponent information in a basis shell.

Table 8-4 BasisPrimitive Type Members

Member	Type	RW	Description
exponent	double	•	Exponent of the basis primitive
coefficients	double[]	•	Coefficients of the basis primitive

8.2.7. BasisPrimitive Type Functions

addCoefficient

Syntax:

```
void addCoefficient ( double coeff )
```

Add a new coefficient to the basis primitive.

8.2.8. BasisShell Type

The BasisShell type contains information about a basis shell centred on a specific atom in a model, allowing a full basis set specification to be associated with a system.

Table 8-5 BasisShell Type Members

Member	Type	RW	Description
atomId	int	•	Atom ID on which the basis shell is centred
nPrimitives	int		Number of primitives defined for the basis shell
primitives	BasisPrimitive[]		List of primitives defined for the basis shell
type	string	•	The type (shape) of the basis shell. See Basis Shell Types in Section 16.1 for a list.

8.2.9. BasisShell Type Functions

addPrimitive

Syntax:

```
BasisPrimitive addprimitive ( double exponent, double coeff1 = 0.0 ... )
```

Add a new primitive to the basis shell, with the specified exponent and (optional) coefficients given.

8.2.10. Bond Type

The `Bond` type represents a chemical connection of some defined order between two atoms.

Table 8-6 Bond Type Members

Member	Type	RW	Description
i	Atom		First atom involved in the bond
j	Atom		Second atom involved in the bond
order	double		Bond order
type	string	*	Bond Type – see Section 16.2 for a list

8.2.11. Bond Type Functions

partner

Syntax:

```
Atom partner ( Atom i )
```

Return the other atom (i.e. not *i*) involved in the bond

8.2.12. Bound Type

The `Bound` type is used by the pattern type, and defines a single bound interaction (e.g. a bond, angle, or torsion) between a group of atoms within the pattern. It differs from the `FFBound` type since no forcefield information is stored locally in the structure.

Table 8-7 Bound Type Members

Member	Type	RW	Description
data	double		Parameters describing the bound interaction
escale	double		Electrostatic 1-4 scaling factor (for torsion interactions)
form	string		Functional form of the bound interaction
id	int[]		Array of atom IDs involved in the interaction
termed	int		Array index of forcefield term in relevant list (ffangles, ffbands, or fftorsions) in local pattern

type	string	Returns the Bound Type (Section 16.3) of the interaction.
typenames	string[]	Array of typenames involved in the interaction
vscale	double	Short-range 1-4 scaling factor (for torsion interactions)

8.2.13. Bound Type Functions

parameter

Syntax:

```
double parameter ( string keyword )
```

Search for and return the named parameter of the bound interaction

8.2.14. ColourScale Type

The ColourScale type allows direct access to the points defined inside a colourscale.

Table 8-8 ColourScale Type Members

Member	Type	RW	Description
nPoints	int		Number of points contained in the colourscale
points	ColourScalePoint[]		Array of points in the colourscale

8.2.15. ColourScale Type Functions

addPoint

Syntax:

```
ColourScalePoint addPoint ( double value, double r, double g, double b,  
double a = 1.0 )
```

Add a new point to the colourscale at point *value* and RGB(A) colour specified, the components of which should have values ranging from 0.0 to 1.0 inclusive).

clear

Syntax:

```
void clear ()
```

Clear all points contained in the colourscale

colour

Syntax:

```
void colour ( double value, double &r, double &g, double &b, double &a )
```

Retrieve the colour components for the `value` specified. The RGBA values of the colour will be placed in the supplied variables.

8.2.16. ColourScalePoint Type

The `ColourScalePoint` is a simple type which allows definition of the point colour.

Table 8-9 ColourScalePoint Type Members

Member	Type	RW	Description
colour	double [4]	•	Colour associated to the point

8.2.17. Dialog Type

The `Dialog` is a parent window control used to contain one or more Widgets designed to allow setting of options etc.

Table 8-10 Dialog Type Members

Member	Type	RW	Description
title	string	•	Text displayed in the Dialog's titlebar
verticalFill	int	•	If TRUE then child widgets without specific coordinates will be created in a vertical stack. Otherwise child widgets will be added in a horizontal line

8.2.18. Dialog Type Functions

The `Dialog` is itself a `Widget` – for widget creation functions refer to Section 8.2.45 (the `Widget` variable).

asDouble

Syntax:

```
double asDouble ( string name )
```

Return the value of the named widget as a double.

asInteger

Syntax:

```
int asInteger ( string name )
```

Return the value of the named widget as an integer.

asString

Syntax:

```
string asString ( string name )
```

Return the value of the named widget as a string.

asVector

Syntax:

```
Vector asVector ( string name1, string name2, string name3 )
```

Return the values of the three named widgets as a Vector.

isInteger

Syntax:

```
int isInteger ( string name, int value )
```

Returns TRUE (1) if the named widget's current value matches that provided, or FALSE (0) otherwise.

isRange

Syntax:

```
int isRange ( string name, int minValue, int maxValue )
```

Returns TRUE (1) if the named widget's current value is within the range provided, or FALSE (0) otherwise.

isString

Syntax:

```
int isString ( string name, string value )
```

Returns TRUE (1) if the named widget's current string value matches that provided, or FALSE (0) otherwise.

show

Syntax:

```
int show ( )
```

Shows (executes) the Dialog, blocking all input to Aten until it is accepted (OK) or rejected (Cancel).

widget

Syntax:

```
Widget widget ( string name )
```

Search for and return the named Widget.

8.2.19. Eigenvector Type

The `Eigenvector` type stores the coefficients of a complete molecular orbital for a specific model.

Table 8-11 Eigenvector Type Members

Member	Type	RW	Description
eigenvalue	double	•	Associated eigenvalue of the eigenvector
name	string	•	Name of the eigenvector, e.g. orbital symmetry
occupancy	double	•	Associated occupancy of the eigenvector
size	int	•	Current size of the eigenvector array. Can be set to reinitialise the array.
vector	double[]	•	The eigenvector data array

8.2.20. Element Type

The `Element` type stores all information describing a given chemical element, the full array of which is stored in the `Aten` type.

Table 8-12 Element Type Members

Member	Type	RW	Description
ambient	double[4]	•	Ambient colour of the element
colour	double[4]	•	Returns the ambient colour of the element. Set this property to define both ambient (supplied values) and diffuse (0.75*supplied values) components simultaneously
diffuse	double[4]	•	Diffuse colour of the element
mass	double		Atomic mass of the element
name	string		Capitalised name of the element
radius	double	•	Atomic radius of the element. Affects the scaled sphere rendering style and bond calculation.
symbol	string		Atomic symbol of the element
z	int	•	Atomic number of the element

8.2.21. EnergyStore Type

The `EnergyStore` type stores the last set of energy values calculated for a model or frame.

Table 8-13 EnergyStore Type Members

Member	Type	RW	Description
angle	double		Total angle energy

bond	double	Total bond energy
electrostatic	double	Total electrostatic energy (from Coulomb or Ewald sums)
torsion	double	Total torsion energy
total	double	Total of all energy components
ureyBradley	double	Total Urey-Bradley energy
vdw	double	Total van der Waals energy

8.2.22. FFAtom Type

The `FFAtom` type stores parameter data relevant to a specific atom type in a forcefield.

Table 8-14 FFAtom Type Members

Member	Type	RW	Description
charge	double	•	Charge associated to the type
data	double[6]	•	Parameter data for short-range potential
dataKeyword	string[]		Keyword names of the associated parameters, up to 'nparams'
dataNames	string[]		Proper names of the associated parameters, up to 'nparams'
description	string	•	Text data describing the type
equivalent	string	•	Equivalent name for the type
form	string	•	Functional form of short-range potential
id	int		Internal ID of the atom type within its parent forcefield
name	string	•	Name of the type
neta	string		The original type description used to identify the type
nParams	int		The number of parameters used by the functional form of the interaction
ff	Forcefield		Parent forcefield containing the type
z	int	•	Element id (Z) corresponding to the target element of this type

8.2.23. FFAtom Type Functions

dataD

Syntax:

```
double dataD ( string varname )
```

Return the value of the defined data item `varname` as a double if it has been defined in a `data` block in its encompassing forcefield.

dataI

Syntax:

```
int dataI ( string varname )
```

Return the value of the defined data item *varname* as an integer if it has been defined in a **data** block in its encompassing forcefield.

dataS

Syntax:

```
string dataS ( string varname )
```

Return the value of the defined data item *varname* as a string if it has been defined in a **data** block in its encompassing forcefield.

parameter

Syntax:

```
double parameter ( string name )
```

Return the value of the parameter *name* in the associated vdW data. See VDW Functional Forms in Section 13.1 for a list of parameter names in the supported vdW functions.

8.2.24. FFBound Type

The **FFBound** type stores parameter data relevant to a specific bound interaction within a pattern. It differs from the **Bound** type in that no atom ID information is stored.

Table 8-15 FFBound Type Members

Member	Type	RW	Description
data	double[]	•	Parameter data for the potential
dataKeyword	string[]		Keyword names of the associated parameters, up to 'nparams'
dataNames	string[]		Proper names of the associated parameters, up to 'nparams'
eScale	double[]	•	For torsion-type interactions, the electrostatic scaling factor between atoms 1 and 4
form	string	•	Functional form of intramolecular potential - see the Section 13 for lists of allowable functional forms for each intramolecular interaction type
nAtoms	int		Number of atoms involved in the bound interaction
nParams	int		The number of parameters used by the functional form of the interaction
type	string		Actual type of the bound interaction (bond, angle, etc.)
typeNames	string	•	Names of the atom types the interaction is relevant to
vScale	double[]	•	For torsion-type interactions, the VDW scaling factor between atoms 1 and 4

8.2.25. FFBound Type Functions

parameter

Syntax:

```
double parameter ( string name )
```

Search for and return the value of the parameter *name* within the bound interaction

8.2.26. Forcefield Type

The Forcefield type stores a complete set of atom types and ffbound interaction data.

Table 8-16 Forcefield Type Members

Member	Type	RW	Description
energyGenerators	int[]	•	Array of integers flagging which generator data are ‘energetic’ and should be converted
filename	string	•	Filename if the forcefield was loaded from a file
name	string	•	Name of the forcefield
nAngles	integer		Number of angle terms defined in the forcefield
nAtomtypes	integer		Number of atomtypes defined in the forcefield
nBonds	integer		Number of bond terms defined in the forcefield
nImpropers	integer		Number of improper dihedral terms defined in the forcefield
nTorsions	integer		Number of torsion terms defined in the forcefield

8.2.27. Forcefield Type Functions

addAngle

Syntax:

```
FFBound addAngle ( string form, string type_i, string type_j, string type_k, double data1, ... )
```

Create a new angle definition in the forcefield. See the **angleDef** command for a full description.

addBond

Syntax:

```
FFBound addBond ( string form, string type_i, string type_j, double data1, ... )
```

Create a new bond definition in the forcefield. See the **bondDef** command for a full description.

addInter

Syntax:

```
FFBound addInter ( string form, int typeid, double charge, double  
data1, ... )
```

Create a new interatomic definition in the forcefield. See the **interDef** command for a full description.

addTorsion

Syntax:

```
FFBound addTorsion ( string form, string type_i, string type_j, string  
type_k, string type_l, double data1, ... )
```

Create a new torsion definition in the forcefield. See the **torsionDef** command for a full description.

addType

Syntax:

```
FFAtom addType ( int typeid, string name, string equiv, string|int  
element, string neta, string description = "" )
```

Create a new type definition in the forcefield. See the **typeDef** command for a full description.

finalise

Syntax:

```
void finalise ( )
```

Finalise the forcefield. See the **finaliseFF** command for a full description.

findAngle

Syntax:

```
ffbound findAngle ( string type_i, string type_j, string type_k )
```

Search for an existing angle definition in the forcefield between the type names supplied.

findBond

Syntax:

```
FFBound findBond ( string type_i, string type_j )
```

Search for an existing bond definition in the forcefield between the type names supplied.

findImproper

Syntax:

```
FFBound findImproper ( string type_i, string type_j, string type_k,  
string type_l )
```

Search for an existing improper torsion definition in the forcefield between the type names supplied.

findTorsion

Syntax:

```
FFBound findTorsion ( string type_i, string type_j, string type_k,  
string type_l )
```

Search for an existing torsion definition in the forcefield between the type names supplied.

findUreyBradley

Syntax:

```
FFBound findUreyBradley ( string type_i, string type_j, string type_k )
```

Search for an existing Urey-Bradley definition in the forcefield between the type names supplied.

8.2.28. Glyph Type

The `Glyph` type contains information describing a single glyph within a model.

Table 8-17 Glyph Type Members

Member	Type	RW	Description
data	<code>GlyphData</code>		Individual data for each vertex of the glyph
nData	<code>int</code>		How many data points (vertices) are associated with this glyph's type
rotated	<code>int</code>	•	Flag to indicate whether the glyph's rotation matrix has been modified (i.e. the glyph has been rotated). Setting this to '0' resets (and removes) any current rotation matrix (same as calling member function 'resetrotation').
rotation	<code>double[9]</code>	•	Rotation matrix for the glyph. Note that not all glyphs can be rotated (see the topic on Glyphs for more information).
selected	<code>int</code>	•	Whether the glyph is currently selected
solid	<code>int</code>	•	Specifies whether the glyph is drawn as a filled (solid) shape or in wireframe
text	<code>string</code>	•	Text data associated to the glyph. Not all glyphs use text
type	<code>string</code>	•	Style of the glyph – see Glyph Types (Section 16.9) for a list
visible	<code>int</code>	•	Flag indicating whether the glyph is currently visible

8.2.29. Glyph Type Functions

recolour

Syntax:

```
void recolour ( double r, double g, double b, double a = 1.0 )
```

Recolour all data vertices of the glyph to the specified RGB(A) value, each component of which should be in the range 0.0 to 1.0 inclusive.

resetRotation

Syntax:

```
void resetRotation ( )
```

Reset any rotation applied to the glyph

rotateX

Syntax:

```
void rotateX ( double angle )
```

Rotates the glyph by *angle* degrees about its x axis

rotateY

Syntax:

```
void rotateY ( double angle )
```

Rotates the glyph by *angle* degrees about its y axis

rotateZ

Syntax:

```
void rotateZ ( double angle )
```

Rotates the glyph by *angle* degrees about its z axis

8.2.30. GlyphData Type

The `GlyphData` type stores colour and position data for a single point in a `Glyph`.

Table 8-18 GlyphData Type Members

Member	Type	RW	Description
atom	Atom		• Atom (if any) from which positional data is retrieved
atomData	int		• Type of data to retrieve from specified atom (if any) - 0 = position, 1 = force, 2 = velocity

colour	double[4]	•	RGB colour of the vertex (components ranging from 0.0 to 1.0)
vector	Vector	•	Vertex coordinates

8.2.31. Grid Type

The `Grid` type stores all information for a single grid structure associated to a model.

Table 8-19 Grid Type Members

Member	Type	RW	Description
axes	UnitCell		Axes system for the grid, stored in a unit cell structure
colour	double[4]	•	The primary colour of the grid data
cutoff	double	•	The lower primary cutoff value
name	string	•	Name associated to the grid data
nx	int		Size of grid data in the x-dimension
ny	int		Size of grid data in the y-dimension
nz	int		Size of grid data in the z-dimension
origin	Vector	•	Coordinates of the origin (lower left-hand corner) of the grid data
outlineVolume	int	•	Flag specifying whether a bounding volume cuboid should be drawn for the grid
periodic	int	•	Flag specifying whether the grid is periodic (e.g. at the edges data from opposite side should be used to form grid)
secondaryColour	double[4]	•	The secondary colour of the grid data
secondaryCutoff	double	•	The lower secondary cutoff value
secondaryUpperCutoff	double	•	The upper secondary cutoff value
shiftX	int	•	Shift value (in points) for the grid along its x-axis
shiftY	int	•	Shift value (in points) for the grid along its y-axis
shiftZ	int	•	Shift value (in points) for the grid along its z-axis
upperCutoff	double	•	The upper primary cutoff value
visible	int	•	Flag specifying whether the grid is visible (i.e. should be drawn in the model)

8.2.32. Grid Type Functions

data

Syntax:

```
double data ( int i, int j, int k = -1 )
```

Return the value of the datapoint at grid ‘coordinates’ (i, j, k).

shift

Syntax:

```
void data ( int dx, int dy, int dz, bool shiftAtoms = FALSE )
```

Shift the data contained within the grid by the number of cells specified in each of the x, y, and z directions. If the *shiftAtoms* parameter is TRUE then all atoms within the parent model are shifted by the corresponding amount in real space.

8.2.33. Measurement Type

The Measurement type stores the atom indexes and current value of a single measurement currently displayed on a model.

Table 8-20 Measurement Type Members

Member	Type	RW	Description
atoms	Atom []		Array of atoms involved in the measurement
i	Atom		First atom involved in the measurement
j	Atom		Second atom involved in the measurement
k	Atom		Third atom involved in the measurement (if any)
l	Atom		Fourth atom involved in the measurement (if any)
literal	double		The literal value (e.g. distance, angle, or torsion angle) of the measurement using atom coordinates as-is without applying minimum image criteria in periodic systems
value	double		Value (e.g. distance, angle, or torsion angle) of the measurement

8.2.34. Model Type

The Model type contains all the information describing a single model within Aten, including bonds, grid data, measurements, etc..

Table 8-21 Model Type Members

Member	Type	RW	Description
angles	Measurement []		List of current angle measurements in the model
atoms	Atom []		Array of atoms in the model
bonds	Bond []		Array of bonds defined in the model
cell	UnitCell	•	The model's unit cell
componentDensity	double	•	Requested density of the model
componentPartition	int	•	The integer index of the partition which this model

		will be added to in the disordered builder
componentPolicy	string	• Insertion policy for this model
componentPopulation	int	• The number of times this model will be added to the specified partition in the disordered builder
distances	Measurement []	List of current distance measurements in the model
eigenvectors	Eigenvector []	List of current eigenvectors stored in the model
energy	EnergyStore command.types.energystore	The model's energy store, containing the total calculated energy and all associated contributions
ff	Forcefield	• Forcefield associated to the model (if any)
ffAngles	FFBound []	List of unique forcefield angle terms in the model
ffBonds	FFBound []	List of unique forcefield bond terms in the model
ffMass	double	Forcefield mass of the current model, which can differ from the actual mass if united-atom types have been assigned
ffTorsions	FFBound []	List of unique forcefield torsion terms in the model
ffTypes	FFAtom	Array of unique atom types used in the model
frame	Model	The current frame in the model's trajectory (if it has one)

frames	Model []	Array of trajectory frame pointers (only if the trajectory is cached)
glyphs	Glyph []	List of glyphs owned by the model
grids	Grid []	List of grids owned by the model
id	int	The index of the model in Aten's internal list of loaded models
mass	double	Mass of the current model
name	string	• Name of the model
nAngles	int	Number of angle measurements in the model
nAtoms	int	Number of atoms in the model
nBasisCartesians	int	Number of cartesian basis functions defined in stored basis shells
nBasisShells	int	Number of basis shells defined in the model
nBonds	int	Number of bonds in the model
nDistances	int	Number of distance measurements in the model
nFFAngles	int	Number of unique angle terms used in the model
nFFBonds	int	Number of unique bond terms used in the model
nFFTorsions	int	Number of unique torsion terms used in the model
nFFTypes	int	Number of unique atom types used in the model
nFrames	int	Number of frames in associated

nGlyphs	int	trajectory Number of glyphs owned by the model
nGrids	int	Number of grids owned by the model
nPatterns	int	Number of patterns defined for the model
nSelected	int	Number of atoms selected in the model
nTorsions	int	Number of torsion angle measurements in the model
nUnknown	int	Number of atoms in the model that are of unknown element
patterns	Pattern[]	Array of patterns currently defined for the model
selection	Atom[]	A list of atoms in representing the current atom selection of the model
torsions	Measurement[]	List of current torsion angle measurements in the model

8.2.35. Model Type Functions

atomWithBit

Syntax:

```
Atom atomWithBit ( int bit )
```

Returns the first Atom in the Model which has the specified value of *bit* (see the relevant accessor and commands in Sections 8.2.4 and 8.2.5 above).

copy

Syntax:

```
void copy ( Model source )
```

Copy all information from the source Model into this Model.

addHydrogen

Syntax:

```
void addHydrogen ( )
```

Hydrogen satisfy all atoms in the model. See the **addHydrogen** command for more details.

angleEnergy

Syntax:

```
double angleEnergy ( )
```

Calculates and returns the total angle energy for the current model.

augment

Syntax:

```
void augment ( )
```

Automatically detect and add multiple bonds in the system. See the **augment** command for more details.

bondEnergy

Syntax:

```
double bondEnergy ( )
```

Calculates and returns the total bond energy (including Urey-Bradley contributions) for the current model.

charge

Syntax:

```
void charge ( )
```

Assign charges to the model from the current forcefield. See the **chargeFF** command for more details.

clearBonds

Syntax:

```
void clearBonds ( )
```

Remove all bonds from the model. See the **clearBonds** command for more details.

clearCharges

Syntax:

```
void clearCharges ( )
```

Remove all charges from the model, setting them to zero. See the **clearCharges** command for more details.

clearSelectedBonds

Syntax:

```
void clearSelectedBonds ( )
```

Remove all bonds from the current atom selection. See the **clearSelectedBonds** command for more details.

copy

Syntax:

```
void copy ( )
```

Copy the current atom selection to the clipboard. See the **copy** command for more details.

cut

Syntax:

```
void cut ( )
```

Cut the current atom selection to the clipboard. See the **cut** command for more details.

delete

Syntax:

```
void delete ( )
```

Delete the current atom selection. See the **delete** command for more details.

elecEnergy

Syntax:

```
double elecEnergy ( )
```

Calculates and returns the total electrostatic energy for the current model, using the calculation method specified in the preferences.

expand

Syntax:

```
void expand ( )
```

Expand the current atom selection along bonds. See the **expand** command for more details.

finalise

Syntax:

```
void finalise ( )
```

Finalise the current model. See the **finaliseModel** command for more details.

interEnergy

Syntax:

```
double interEnergy ( )
```

Calculates and returns the total intermolecular (i.e. combined van der Waals and electrostatic) energy for the current model.

intraEnergy

Syntax:

```
double intraEnergy ( )
```

Calculates and returns the total intramolecular (i.e. combined bond, angle, and torsion) energy for the current model.

moveToEnd

Syntax:

```
void moveToEnd ( )
```

Move the current atom selection to the bottom (highest IDs) of the atom list. See the **moveToEnd** command for more details.

moveToStart

Syntax:

```
void moveToStart ( )
```

Move the current atom selection to the top (lowest IDs) of the atom list. See the **moveToStart** command for more details.

newAtom

Syntax:

```
Atomcommand.types.atom newAtom ( int|string el, double x = 0.0, double y = 0.0, double z = 0.0, double vx = 0.0, double vy = 0.0, double vz = 0.0, double fx = 0.0, double  = 0.0, double fz )
```

Create a new atom in the model. See the **newAtom** command for more details.

newAtomFrac

Syntax:

```
Atomcommand.types.atom newAtomFrac ( int|string el, double x, double y, double z, double vx = 0.0, double vy = 0.0, double vz = 0.0, double fx = 0.0, double fy = 0.0, double fz )
```

Create a new atom in the model, in fractional coordinates. See the **newAtomFrac** command for more details.

newBasisShell

Syntax:

```
BasisShellcommand.types.basisshell newBasisShell ( atom|int i, string type )
```

Create a new basis shell definition in the model, centred on the specified atom/id, and of the given shell type. See the **newBasisShell** command for more details.

newBond

Syntax:

```
Bondcommand.types.bond newBond ( atom|int i, atom|int j, string|int bondtype = "" )
```

Create a new bond between atoms in the model. See the **newBond** command for more details.

newBondId

Syntax:

```
Bondcommand.types.bond newBondId ( int id_i, int id_j, string|int bondtype = "" )
```

Create a new bond between atom IDs in the model. See the **Error! Reference source not found.** command for more details.

newEigenvector

Syntax:

```
Eigenvectorcommand.types.eigenvector newEigenvector ( int size = (auto) )
```

Create a new eigenvector in the model of the specified size. If the size is not specified, the vector length is set to match the number of cartesian basis functions stored with the current basis shell definitions in the model. See the **newEigenvector** command for more details.

newGlyph

Syntax:

```
Glyphcommand.types.glyph newGlyph ( string style, string options = "" )
```

Create a new glyph in the model. See the **newGlyph** command for more details.

newGrid

Syntax:

```
Gridcommand.types.grid newGrid ( string name )
```

Create a new grid in the model. See the **newGrid** command for more details.

paste

Syntax:

```
void paste ( )
```

Paste the current clipboard contents into the model. See the **paste** command for more details.

rebond

Syntax:

```
void rebond ( )
```

Calculate bonds in the model. See the **rebond** command for more details.

rebondPatterns

Syntax:

```
void rebondPatterns ( )
```

Calculate bonds within patterns in the model. See the **rebondPatterns** command for more details.

rebondSelection

Syntax:

```
void rebondSelection ( )
```

Calculate bonds in the current selection. See the **rebondSelection** command for more details.

redo

Syntax:

```
void redo ( )
```

Redo the last undone change in the model. See the **redo** command for more details.

reorder

Syntax:

```
void reorder ( )
```

Reorder atoms so bound atoms have adjacent IDs. See the **reorder** command for more details.

saveBitmap

Syntax:

```
void saveBitmap ( string format, string filename, int width = auto, int height = auto, int quality = 100 )
```

Save a bitmap image of the current model view. See the **saveBitmap** command for more details.

selectAll

Syntax:

```
void selectAll ( )
```

Select all atoms in the model. See the **selectAll** command for more details.

selectionAddHydrogen

Syntax:

```
void selectionAddHydrogen ( )
```

Hydrogen satisfy all atoms in the current selection. See the **selectionAddHydrogen** command for more details.

selectNone

Syntax:

```
void selectNone ( )
```

Deselect all atoms in the model. See the **selectNone** command for more details.

selectTree

Syntax:

```
int selectTree ( atom i, bond exclude = NULL )
```

Select all atoms from atom *i* reachable by following any number of chemical bonds. See the **selectNone** command for more details.

shiftDown

Syntax:

```
void shiftDown ( int n = 1 )
```

Shift the current atom selection down one (or more) places in the atom list (towards higher IDs). See the **shiftDown** command for more details.

shiftUp

Syntax:

```
void shiftUp ( int n = 1 )
```

Shift the current atom selection up one (or more) places in the atom list (towards lower IDs). See the **shiftUp** command for more details.

showAll

Syntax:

```
void showAll ( )
```

Unhides any hidden atoms in the model. See the **showAll** command for more details.

toAngstroms

Syntax:

```
void toAngstroms ( )
```

Converts cell specification and atomic coordinates in the model from (assumed units of) Bohr into Angstroms. No changes to associated trajectory frames or grid data is made.

torsionEnergy

Syntax:

```
double torsionEnergy ( )
```

Calculates and returns the total torsion energy (including improper terms) for the current model.

transmute

Syntax:

```
void transmute ( int|string el )
```

Transmute all selected atoms to the specified element. See the **transmute** command for more details.

undo

Syntax:

```
void undo ( )
```

Undo the last change made to the model. See the **undo** command for more details.

vdwEnergy

Syntax:

```
double vdwEnergy ( )
```

Calculates and returns the total van der Waals energy for the current model.

8.2.36. Monte Carlo Type

The `MonteCarlo` type stores various quantities which affect both the Monte Carlo minimiser and the disorder builder (see Section 15.1.6). Variables of this type cannot be created by the user – the sole instance of it exists as part of the `Aten` global variable.

Table 8-22 MonteCarlo Type Members

Member	Type	RW	Description
disorderAccuracy	double	•	Strictness of adherence to requested component populations and densities
disorderDeltaAngle	double	•	Maximum angle for molecular rotations
disorderDeltaDistance	double	•	Maximum distance for molecular translations
disorderMaxCycles	double	•	Maximum number of cycles to perform
disorderMaxFailures	double	•	Maximum failure rate per component before scale factor is reduced
disorderMaximumScaleFactor	double	•	Maximum scale factor to employ at start of

			build, and target scale factor for recovery
disorderMinimumScaleFactor	double	•	Minimum scale factor to allow in build
disorderNTweaks	int	•	Number of molecule tweaks to perform per disorder cycle
disorderRecoveryMaxCycles	int	•	Maximum number of recovery cycles to perform
disorderRecoveryMaxTweaks	int	•	Maximum number of tweaks to perform per cycle in recovery
disorderRecoveryThreshold	double	•	TODO
disorderReductionFactor	double	•	TODO
nCycles	int	•	Maximum number of Monte Carlo cycles to perform
temperature	double	•	Temperature for Monte Carlo ‘simulation’

8.2.37. MonteCarlo Type Functions

eAccept

Syntax:

```
double eAccept ( string moveType, double newValue = <none> )
```

Return the current acceptance energy threshold for the specified *moveType*, setting to the *newValue* if one is provided. See Section 16.13 for a list of move types.

maxStep

Syntax:

```
double maxStep ( string moveType, double newValue = <none> )
```

Return the current maximum step size for the specified *moveType* (if applicable), setting to the *newValue* if one is provided. See Section 16.13 for a list of move types.

moveAllowed

Syntax:

```
int moveAllowed ( string moveType, int newValue = <none> )
```

Return whether the specified *moveType* is allowed to take place, setting to the *newValue* if one is provided. See Section 16.13 for a list of move types.

nTrials

Syntax:

```
double nTrials ( string moveType, int newValue = <none> )
```

Return the current number of trials for each *moveType*, setting to the *newValue* if one is provided. See Section 16.13 for a list of move types.

8.2.38. Pattern Type

The `Pattern` type describes a single continuous collection of similar molecules/fragments within a model.

Table 8-23 Pattern Type Members

Member	Type	RW	Description
angles	Bound[]		Array of angle interactions in one molecule of the pattern
atoms	Atom[]		Array of atoms spanned by the pattern
bonds	Bound[]		Array of bond interactions in one molecule of the pattern
ff	Forcefield[]	•	Reference to the forcefield associated to the pattern (if any)
ffAngles	FFBound[]		List of unique forcefield angle terms in the pattern
ffBonds	FFBound[]		List of unique forcefield bond terms in the pattern
ffTorsions	FFBound[]		List of unique forcefield torsion terms in the pattern
ffTypes	FFAtom		Array of unique atom types used in the pattern
firstAtom	Atom[]		Reference to the first atom spanned by the pattern
firstAtomId	int		Atom ID of the first atom spanned by the pattern
fixed	int	•	Whether the coordinates of all atoms in the pattern are fixed in minimisation routines
lastAtom	Atom[]		Reference to the last atom spanned by the pattern
lastAtomId	int		Atom ID of the last atom spanned by the pattern
name	string	•	Name of the pattern
nAngles	int		Number of angles in one molecule of the pattern
nAtoms	int		Total number of atoms spanned by the pattern
nBonds	int		Number of bonds in one molecule of the pattern
nFFAngles	int		Number of unique angle terms used in the pattern
nFFBonds	int		Number of unique bond terms used in the pattern
nFFTorsions	int		Number of unique torsion terms used in the pattern
nFFTypes	int		Number of unique atom types used in the pattern
nMolAtoms	int		Number of atoms in one molecule of the pattern
nMols	int		Number of molecules (repeat units) in the pattern
nTorsions	int		Number of torsion interactions in one molecule of the pattern
torsions	Bound[]		Array of torsion interactions in one molecule of the pattern

8.2.39. Pattern Type Functions

atomsInRing

Syntax:

```
int atomsInRing ( int id_i, int id_j = -1 )
```

Return whether the supplied atom index (indices), given in local pattern atom numbering, is in a ring (the same ring)

cog

Syntax:

```
Vector cog ( int index )
```

Return calculated centre of geometry for the molecule index provided

com

Syntax:

```
Vector com ( int index )
```

Return calculated centre of mass for the molecule index provided

8.2.40. Prefs Type

The `Prefs` type contains all preferences and settings information detailing the inner bits and pieces of Aten. It exists as a single instance, owned by and available through the `Aten` master type.

Table 8-24 Prefs Type Members

Member	Type	RW	Description
angleLabelFormat	string		• The C-style format to use for the numerical value of angle labels
aromaticRingColour	double [4]		• Colour of aromatic ring circles
atomStyleRadius	double [4]		• The atom radii used for selection and rendering in the four Drawing Styles (see Section 16.7)
backCull	int		• Whether culling of backward-facing polygons should be performed
backGroundColour	double [4]		• Background colour of the main canvas on which models are drawn
bondStyleRadius	double [4]		• The bond radii used for selection and rendering in the four Drawing Styles (see Section 16.7)
bondTolerance	double		• Tolerance used in automatic calculation of bonds between atoms
cacheLimit	int		• The trajectory cache size (in kilobytes) - trajectory files calculated to have more than this amount of data will not be cached in memory
calculateIntra	int		• Controls whether intramolecular contributions to the energy/forces are calculated
calculateVdw	int		• Controls whether short-range van der

		Waals contributions to the energy/forces are calculated
chargeLabelFormat	string	• C-style format for atomic charge labels
clipFar	double	• The far clipping distance used when rendering
clipNear	double	• The near clipping distance used when rendering
colourScales	ColourScale[10]	• List of colourscales
colourScheme	string	• The current Colour Scheme used to colour atoms and bonds (see Section 16.5)
combinationRule	string	• Lennard-Jones parameter combination rule equations. See Combination Rules in Section 16.6 for a list
commonElements	string	• Comma-separated list of common elements that appear in the Select Element dialog
dashedAromatics	int	• Whether to render solid or dashed rings for aromatics
densityUnit	string	• The unit of density to used when displaying cell densities
depthCue	int	• Enables/disables depth cueing
depthFar	int	• The far fog distance used when rendering (if depth cueing is enabled)
depthNear	int	• The near fog distance used when rendering (if depth cueing is enabled)
distanceLabelFormat	string	• The C-style format to use for the numerical value of distance labels
elecCutoff	double	• The electrostatic cutoff distance
elecMethod	string	• The method of electrostatic energy/force calculation
encoderArgs	string	• Arguments to pass to the movie encoder. It should include the text strings FILES and OUTPUT will be replaced with a wildcard filelist and the output movie filename respectively.
encoderExe	string	• Encoder executable, including full path if necessary.
energyUnit	double	• Set the unit of energy to use
energyUpdate	int	• Update frequency for the energy in various methods
ewaldAlpha	double	• Convergence parameter in Ewald sum
ewaldKMax	int[3]	• Vector of Ewald reciprocal space vector limits (kmax)
ewaldPrecision	double	• Precision parameter to use when generating parameters in EwaldAuto
forceRhombohedral	int	• For spacegroups that are detected to have a hexagonal basis, force packing to use generators in a rhombohedral basis
frameCurrentView	int	• Whether to frame the current model with a box
frameWholeView	int	• Whether to frame the entire view with a

globeAxesColour	double [4]	• box Colour of axis pointers on the rotation globe
globeColour	double [4]	• Colour of the actual rotation globe
globeSize	int	• Size, in pixels, of the rotation globe in the lower-right-hand corner of the screen
glyphColour	double [4]	• Default colour of all created glyphs
hDistance	double	• Default H-X bond distance to use when adding hydrogen atoms
imageQuality	int	• The general rendering quality (i.e. number of triangles used to generate primitive objects) of the images, used if 'reusequality' is set to FALSE (otherwise the current 'quality' value is used).
keyAction	string [3]	• Current actions of the modifier keys Shift, Ctrl, and Alt
labelSize	int	• Font pointsize for label text
levelOfDetailStartZ	double	• Z-depth (in Angstroms) at which level of detail algorithm begins
levelOfDetailWidth	double	• Z-width (in Angstroms) of each level of detail strip
levelsOfDetail	int	• Number of levels of detail to employ
lineAliasing	int	• Enables/disables line aliasing
manualSwapBuffers	int	• Flag whether manual swapping of GL buffers is enabled
maxRings	int	• Maximum allowable number of rings to detect within any single pattern
maxRingSize	int	• Maximum size of ring to detect when atom typing
maxUndo	int	• Maximum number of undo levels remembered for each model (-1 = unlimited)
modelUpdate	int	• Update frequency for the current model in various methods
mopacExe	string	• Location of MOPAC executable (including full path)
mouseAction	string [4]	• Current actions of the Left, Middle, Right, and Wheel mouse buttons
mouseMoveFilter	int	• Sets the degree to which mouse move events are filtered, with 1 being no filtering. Use this to reduce update lag on sluggish systems.
multiSampling	int	• Enables/disables multisampling (hardware aliasing)
noQtSettings	int	• Flag controlling whether OS-stored Qt settings are loaded on startup
partitionGrid	int [3]	• Grid size to use for partitioning schemes
perspective	int	• Whether perspective view is enabled
perspectiveFov	double	• Field of vision angle to use for perspective rendering
polygonAliasing	int	• Enables/disables polygon aliasing
renderStyle	string	• The current model drawing style

quality	int	<ul style="list-style-type: none"> The general rendering quality (i.e. number of triangles used to generate primitive objects) of the main view. Higher values give rounder atoms but make rendering slower. See also the 'imagequality' setting.
replicateFold	int	<ul style="list-style-type: none"> Whether to fold atoms before cell replicate
replicateTrim	int	<ul style="list-style-type: none"> Whether to trim atoms after cell replicate
reuseQuality	int	<ul style="list-style-type: none"> Flag specifying whether to use the current rendering 'quality' value when saving images (FALSE) or the 'imagequality' value (TRUE).
selectionScale	double	<ul style="list-style-type: none"> Multiple of the standard atom radius to use for transparent selection spheres
shininess	int	<ul style="list-style-type: none"> The shininess of atoms (value must be between 0 and 127 inclusive)
specularColour	double [4]	<ul style="list-style-type: none"> Colour of all specular reflections
spotlight	int	<ul style="list-style-type: none"> Whether the spotlight is on or off
spotlightAmbient	double [4]	<ul style="list-style-type: none"> The ambient colour component of the spotlight
spotlightDiffuse	double [4]	<ul style="list-style-type: none"> The diffuse colour component of the spotlight
spotlightPosition	double [4]	<ul style="list-style-type: none"> Spotlight coordinates (in Å)
spotlightSpecular	double [4]	<ul style="list-style-type: none"> The specular colour component of the spotlight
stickNormalWidth	double	<ul style="list-style-type: none"> Line width of unselected stick atoms
stickSelectedWidth	double	<ul style="list-style-type: none"> Line width of selected stick atoms
tempDir	string	<ul style="list-style-type: none"> Temporary (working) directory for some operations (e.g. execution of MOPAC jobs)
textColour	double [4]	<ul style="list-style-type: none"> Colour of rendered text
transparencyBinWidth	double	<ul style="list-style-type: none"> Z-width of individual transparency bins
transparencyCorrect	int	<ul style="list-style-type: none"> Whether sort of transparent triangles is enabled. When enabled, any object which is to be drawn in a transparent colour is split into its component triangles and sorted over several bins (representing slices in the z direction). Rendering is a little slower with this option enabled, but corrects most transparency artefacts with sensible bin values.
transparencyNBins	int	<ul style="list-style-type: none"> The number of bins to use for sorting transparent triangles
transparencyBinStartZ	double	<ul style="list-style-type: none"> The z-depth at which binned transparency sorting begins. Before this value, all triangles are grouped (and rendered) together without sorting
transparentSelection	int	<ul style="list-style-type: none"> If set to 1 (TRUE) selected atoms are highlighted with transparent spheres,

		otherwise a simple wireframe style is used.
unitCellAxesColour	double[4]	• Colour of unit cell axis pointers
unitCellColour	double[4]	• Colour of unit cell
useFrameBuffer	int	• Whether to use the grabFrameBuffer() method of the main widget instead of the normal renderPixmap() method when saving bitmap images. If saving an image results in a completely black or corrupt bitmap, try setting this to TRUE.
useNiceText	int	• Whether QPainter (on/1/TRUE) or QGLWidget (off/0/FALSE) is used to render label text
vdwCut	double	• The VDW cutoff distance
vibrationColour	double[4]	• Colour of vibration vector arrows
viewRotationGlobe	int	• Whether to draw a rotation globe in the lower right-hand corner for each model
warn1056	int	• Many changes to the typing language syntax were introduced in revision 1056, and a warning message was implemented. This can be turned off by setting this variable to FALSE
wireSelectionColour	double[4]	• Colour of wire selection objects (if wireframe selection style is enabled)
zoomThrottle	double	• Zooming 'throttle' value used to calm down or increase the distance jumped when zooming with the mouse

8.2.41. Site Type

Table 6.38. Site Type Members

Member	Type	RW	Description

8.2.42. UnitCell Type

The `UnitCell` type contains a full unit cell specification for a model, including any associated spacegroup.

Table 8-25 UnitCell Type Members

Member	Type	RW	Description
a	double	•	Length of cell axis A
alpha	double	•	Angle between cell axes B and C
b	double	•	Length of cell axis B
beta	double	•	Angle between cell axes B and C
c	double	•	Length of cell axis C

ax	double	• x-component of cell axis A
ay	double	• y-component of cell axis A
az	double	• z-component of cell axis A
bx	double	• x-component of cell axis B
by	double	• y-component of cell axis B
bz	double	• z-component of cell axis B
centreX	double	x-coordinate at centre of defined cell
centreY	double	y-coordinate at centre of defined cell
centreZ	double	z-coordinate at centre of defined cell
cx	double	• x-component of cell axis C
cy	double	• y-component of cell axis C
cz	double	• z-component of cell axis C
density	double	Density of the current cell
gamma	double	• Angle between cell axes A and B
matrix	double[9]	Cell axis matrix containing all three cell vectors. For example, ax = matrix[1], ay = matrix[2], etc.)
sgId	int	• Integer ID of the current spacegroup
sgName	string	• Symbol of the current spacegroup
type	string	Type of the current unit cell (see Cell Types in Section 16.4)
volume	double	Volume of the cell in cubic Å

8.2.43. UnitCell Type Functions

copy

Syntax:

```
void copy ( UnitCell source )
```

Copy all information from the source `UnitCell` into this `UnitCell`.

fracToReal

Syntax:

```
Vector fracToReal ( double fracz, double fracy, double fracz )
```

Returns the real-space coordinates of the supplied fractional (cell) coordinates.

mim

Syntax:

```
Vector mim ( Atom i, Atom j )
```

Returns a vector containing the minimum image coordinates of the atom *i* with respect to the reference atom *j*. Note that the coordinates of both *i* and *j* are not modified in any way.

mimVector

Syntax:

```
Vector mimVector ( Atom i, Atom j )
```

Returns the minimum image vector from atom *i* to atom *j*. Note that the coordinates of both *i* and *j* are not modified in any way.

realToFrac

Syntax:

```
Vector realToFrac ( double x, double y, double z )
```

Returns the fractional coordinates of the supplied real-space coordinates.

translateAtom

Syntax:

```
Vector translateAtom ( Atom i, double dx, double dy, double dz )
```

Returns a vector corresponding to the original coordinates of Atom *i* translated by the specified fractional cell amounts in each cell axis direction. Note that the existing coordinates of *i* are not modified in any way.

8.2.44. Vector Type

The `Vector` type is a simple 3-vector class containing three `double` values.

Table 8-26 Vector Type Members

Member	Type	RW	Description
mag	double	•	Magnitude of the vector. Setting this value rescales the vector
x	double	•	The x component of the vector
y	double	•	The y component of the vector
z	double	•	The z component of the vector

8.2.45. Widget Type

The `Widget` is a control present in a GUI dialog created by the user, and provides functions to create further widgets (if the control supports doing so) and create dependency paths between itself and other widgets.

Table 8-27 Widget Type Members

Member	Type	RW	Description
enabled	int	•	Whether the widget is currently enabled (i.e. whether the user can interact with it)
verticalFill	int	•	If TRUE then child widgets without specific coordinates will be created in a vertical stack. Otherwise child widgets will be added in a horizontal line
visible	int	•	Whether the widget is visible in the GUI

8.2.46. Widget Type Functions

Most functions to create child widgets accept four optional integer arguments determining the position and size of the widget in the parent. These are:

- l* The coordinates of the left edge of the control
- t* The coordinates of the top edge of the control
- xw* The additional width to add to the control (default = 0)

xh The additional height to add to the control (default = 0)

If neither of the first two values are given (or both are set to -1) then the child widget's position is determined automatically according to the current fill policy (see the `fillVertical` property). Otherwise, the widget will be positioned absolutely according to the l and t values (note that overlapping widgets is possible, and no checks are made to this end).

addButton

Syntax:

```
Widget addButton ( string name, string label, int l = <auto>, int t = <auto>, int xw = 0, int xh = 0 )
```

Add a new button widget to this widget.

addCheck

Syntax:

```
Widget addCheck ( string name, string label, int state, int l = <auto>, int t = <auto>, int xw = 0, int xh = 0 )
```

Add a new checkbox widget to this widget.

addCombo

Syntax:

```
Widget addCombo ( string name, string label, string items, int l = <auto>, int t = <auto>, int xw = 0, int xh = 0 )
```

Add a new combo widget (drop-down list) to this widget.

addDoubleSpin

Syntax:

```
Widget addDoubleSpin ( string name, string label, double min, double max, double step, double value, int l = <auto>, int t = <auto>, int xw = 0, int xh = 0 )
```

Add a new doublespin widget to this widget.

addEdit

Syntax:

```
Widget addEdit ( string name, string label, string text, int l = <auto>, int t = <auto>, int xw = 0, int xh = 0 )
```

Add a new single-line edit box to this widget.

addFrame

Syntax:

```
Widget addFrame ( string name, int l = <auto>, int t = <auto>, int xw = 0, int xh = 0 )
```

Add a simple frame to this widget.

addGroup

Syntax:

```
Widget addGroup ( string name, string label, int l = <auto>, int t = <auto>, int xw = 0, int xh = 0 )
```

Add a new groupbox to this widget.

addIntegerSpin

Syntax:

```
Widget addIntegerSpin ( string name, string label, int min, int max, int step, int value, int l = <auto>, int t = <auto>, int xw = 0, int xh = 0 )
```

Add a new integer spin widget to this widget.

addLabel

Syntax:

```
Widget addLabel ( string name, string text, int l = <auto>, int t = <auto>, int xw = 0, int xh = 0 )
```

Add a simple label to this widget.

addPage

Syntax:

```
Widget addPage ( string name, string label )
```

Add a new page to this widget, which must be either a tab widget or a stack widget.

addRadioButton

Syntax:

```
Widget addRadioButton ( string name, string label, string group, int state, int l = <auto>, int t = <auto>, int xw = 0, int xh = 0 )
```

Add a new radiobutton to this widget.

addRadioGroup

Syntax:

```
Widget addRadioGroup ( string name )
```

Add a new (invisible) radiobutton group to the widget (but whichi is owned by the parent Dialog).

addSpacer

Syntax:

```
void addSpacer ( int expandH, int expandV, int l = <auto>, int t = <auto>, int xw = 0, int xh = 0 )
```

A

addStack

Syntax:

```
Widget addStack ( string name, int l = <auto>, int t = <auto>, int xw = 0, int xh = 0 )
```

Add a new widget stack to this widget. Individual tabs should be added with the addPage function, called from the new stack widget.

addTabs

Syntax:

```
Widget addTabs ( string name, int l = <auto>, int t = <auto>, int xw = 0, int xh = 0 )
```

Add a new tabbed widget to this widget. Individual tabs should be added with the addPage function, called from the new tab widget.

onDouble

Syntax:

```
void onDouble ( double minval, double maxval, string event, string Widget, string property, double|int|string value = <auto> ... )
```

Define an action to be performed when this widget's double value is within a certain range.

onInteger

Syntax:

```
void onInteger ( int minval, int maxval, string event, string Widget,  
string property, double|int|string value = <auto> ... )
```

Define an action to be performed when this widget's integer value is within a certain range.

onString

Syntax:

```
void onString ( string text, string event, string widget, string  
property, double|int|string value = <auto> )
```

Define an action to be performed when this widget takes on a specific string value.

setProperty

Syntax:

```
void setProperty ( string property, string|int|double value )
```

Set the named widget *property* to the given *value*.

8.3. Custom Dialogs

Occasionally it is useful to be able to provide more control over the actions of a script or filter. For instance, in the latter case it is often desirable to provide a set of options which control the exact data that is written to the file on export or, conversely, to define which data is read in on import. User interaction can be achieved by the use of custom dialogs which display any number of standard GUI controls in a dialog window, allowing such options to be set. These dialogs can be called at any point in the execution of a script or filter.

8.3.1. Default and Temporary Dialogs

Each filter, script, or user-defined function within has a *default dialog* associated with it, and can be retrieved through the defaultDialog XXX command. This dialog always exists, and when repeatedly shown will retain the state of the GUI controls from the last execution. Alternatively a new, temporary dialog can be created at any point in the code by calling the createDialog function. This dialog is then populated with widgets, executed, the necessary data taken from it, and then destroyed.

Default Dialog – Allows for a persistent options dialog in which the states of child widgets are retained between shows.

Temporary Dialogs – Usually offer a more simplistic set of widgets whose states are not remembered between shows (obviously, because the Dialog is destroyed after use)

Default dialogs are commonly used in export Filters in order to allow a complete set of options to be set which define many control variables, and which it is desirable to remember for the next time the user saves a file in the same format.

For example, we can set a string variable with one of several possible values using a combo box control in the GUI. When saving a file in a new format (or by selecting **Export Options** from the **File** menu) a dialog is shown in the GUI which contains all of the defined controls in the relevant filter, from which options can be set before the file is written. This would be written as follows in the filter file:

```
string runtype = option("Run Type", "combo", "check,run,restart,analyse", 2);
```

Note that when the filter code actually runs, the target variable `runtype` will be set with whatever value the control holds in the GUI (or the default if running from the command-line) – no questions are asked interactively at the points at which the `option` statements exist. So, the variable here will be guaranteed to take on one of the four possible values specified – either ‘check’, ‘run’, ‘restart’, or ‘analyse’.

Setting Options

All variables set from an `option` statement are guaranteed to take on at least the default value of the control when the filter or script executes. From the GUI it is possible to change the values of defined options, since an options dialog is presented once load/save filenames are selected and immediately *before* the filter actually executes. From the command line it is also possible to set options within the filter – whenever a filter nickname is provided, e.g. as for the first argument to the `savemodel` command, a list of variables and the value they should take can be provided as part of the string. Let us assume that there is a filter whose definition is as follows:

```
filter(type="exportmodel", extension="crd", nickname="crd")
{
    # Define some options...
    int n = option("Number", "intspin", 0, 10, 1, 2);
    string type = option("Style", "combo", "alpha,beta,gamma", 1);
    ...
}
```

Saving a file in this format (without setting any of the options) would be done with the following command:

```
savemodel("crd", "my_file.crd");
```

The two variables `n` and `type` can be set to different values in the following way:

```
savemodel("crd,n=6,type=beta", "my_file.crd");
```

Option Types

All available control types are listed in the following table. Note that *all* of the required data items must be given as arguments following the control type string in the `option` command.

Table 8-28 Option Types

Type	Required Data	Description
check	int state	A checkbox whose state is either off or on. Returns an integer value ('1' if the control is ticked, '0' otherwise). The initial state ('0' or '1') is supplied.
combo	string items int default	A combobox is a drop-down list of predefined items, supplied as a comma-separated list. The default item selection should be provided as an integer index. The text of the selected item is returned.
doublespin	double min double max double step double start	Control allowing a real number to be input, strictly within the min/max range, and with start value specified. Up/down arrows to the side of the control adjust the current value by the 'step' value.

		A real number is returned.
edit	string value	
intcombo	string items int default	Exactly the same as the ‘combo’ control above, but instead returns an integer value corresponding to the index of the selected entry.
intspin	int min int max int step int start	Control allowing an integer number to be input, strictly within the min/max range, and with start value specified. Up/down arrows to the side of the control adjust the current value by the 'step' value. An integer number is returned.
label	None.	A simple text label, with no associated value.
radio	string group int checked	A radio control is part of a set of checkable items, of which only one can be selected at any one point. The parent group does not need to be created beforehand - if the named group does not currently exist, it will be created. Note that the group widget itself is not visible, and so many contain any number of radio buttons spread over many tabs and pages.
radiogroup	None.	A radio group is a collection of radio buttons, of which only one can be selected at any one time. A radiogroup only provides the means to collect a set of radio buttons together, and it not itself visible. The index of the selected item is returned.
stringradiogroup	None.	Same as ‘radiogroup’, except that the text label of the selected item is returned.

Option Layout

The layout of specified GUI controls is done in as simplistic a manner as possible, while still offering reasonable control over the positioning of elements. If no layout options are specified, all defined controls will be added one after the other in a single row, left-to-right, possibly stretching further than the screen can handle. At the very least, the `newline` command should be used to force a control option to start a new row of controls in the GUI. All controls are added into a grid (Qt’s `QGridLayout`) so that controls always line up nicely. It is also possible to group controls together in tabbed widgets and group boxes.

All controls (except the plain label) are two ‘units’ wide on the grid, and this should be borne in mind when stretching a single control to be the same width as a set of controls on a different row.

9. Command Reference

9.1. Atom Commands

Define and set properties of atoms.

atomStyle

Syntax:

```
void atomStyle ( string style )
```

Sets the individual drawing *style* for the current atom selection, used when the global drawing style is `individual`.

For example:

```
atomStyle("tube");
```

sets the current atom selection to be drawn in the ‘tube’ style. See Section 16.7 for a list of all available styles

currentAtom

Syntax:

```
atom currentAtom ( )
```

```
atom currentAtom ( atom|int id )
```

Return a reference to the current atom. If a new atom/id is provided the current atom is set before being returned.

For example:

```
atom i = currentAtom(1);
```

makes the first atom in the current model the current atom, and returns a reference to it.

fix

Syntax:

```
void fix ( atom|int id = 0 )
```

Fix the positions of the current atom selection (or individual atom specified) so that they remain in the same position following various methods (e.g. minimisations).

For example:

```
fix();
```

fixes the positions of all selected atoms.

```
for (int n=1; n<=10; ++n) fix(n);
```

fixes the positions of the first 10 atoms in the current model.

free

Syntax:

```
void free ( atom|int id = 0 )
```

Free the positions of previously fixed atoms in the current selection (or the individual atom specified).

For example:

```
free(5);
```

allows the fifth atom in the current model to be moved again.

getAtom

Syntax:

```
atom getAtom ( atom|int id )
```

Return a reference to the atom specified.

For example:

```
atom i = getAtom(3);
```

returns a reference to the third atom in the current model.

hide

Syntax:

```
void hide ( atom|int id = 0 )
```

Hides the current selection of atoms (or the supplied atom) from view, meaning they cannot be selected by direct clicking/highlighting in the GUI. They are still subject to transformation if they are selected by other means.

For example:

```
select(H);  
hide();
```

selects and hides all hydrogen atoms in the current model.

setCharge

Syntax:

```
void setCharge ( double q )  
void setCharge ( double q, int id )
```

Set the atomic charge of the current (or specified) atom.

setCoords

Syntax:

```
void setCoords ( double x, double y, double z )  
void setCoords ( double x, double y, double z, int id )
```

Set the coordinates of the current (or specified) atom.

setElement

Syntax:

```
void setElement ( string|int element )  
void setElement ( string|int element, int id )
```

Set the element of the current (or specified) atom.

setForces

Syntax:

```
void setForces ( double fx, double fy, double fz )  
void setForces ( double fx, double fy, double fz, int id )
```

Set the forces of the current (or specified) atom.

setFx

Syntax:

```
void setFx ( double d )  
void setFx ( double d, int id )
```

Set the x force of the current (or specified) atom.

setFy

Syntax:

```
void setFy ( double d )  
void setFy ( double d, int id )
```

Set the y force of the current (or specified) atom.

setFz

Syntax:

```
void setFz ( double d )  
void setFz ( double d, int id )
```

Set the z force of the current (or specified) atom.

setRx

Syntax:

```
void setRx ( double d )  
void setRx ( double d, int id )
```

Set the x coordinate of the current (or specified) atom.

setRy

Syntax:

```
void setRy ( double d )  
void setRy ( double d, int id )
```

Set the y coordinate of the current (or specified) atom.

setRz

Syntax:

```
void setRz ( double d )  
void setRz ( double d, int id )
```

Set the z coordinate of the current (or specified) atom.

setVelocities

Syntax:

```
void setVelocities ( double vx, double vy, double vz )  
void setVelocities ( double vx, double vy, double vz, int id )
```

Set the velocity components of the current (or specified) atom.

setVx

Syntax:

```
void setVx ( double d )  
void setVx ( double d, int id )
```

Set the x velocity of the current (or specified) atom.

setVy

Syntax:

```
void setVy ( double d )  
void setVy ( double d, int id )
```

Set the y velocity of the current (or specified) atom.

setVz

Syntax:

```
void setVz ( double d )  
void setVz ( double d, int id )
```

Set the z velocity of the current (or specified) atom.

show

Syntax:

```
void show ( atom|int id = 0 )
```

Makes the current selection of atoms (or the supplied atom) visible again if they were previously hidden.

9.2. Bond Commands

Create bonds and perform automatic bonding operations.

augment

Syntax:

```
void augment ( )
```

Augments bonds in the current model, automatically determining multiple bonds based on the valency of atoms, and aromaticity based on double bonds in rings.

For example:

```
augment();
```

Augment method for a description of the rebonding algorithm implemented in Aten

bondTolerance

Syntax:

```
double bondTolerance ( )  
double bondTolerance ( double tol )
```

Adjust the bond calculation tolerance. It may often be necessary to tweak the bond tolerance in order to get Aten to recognise patterns within models correctly. The current or new bond tolerance is returned.

For example:

```
bondTolerance(1.20);
```

sets the bonding tolerance to 1.2.

```
double tol = bondTolerance();
```

retrieve the current bond tolerance. See Section 15.1.5 for a description of the rebonding algorithm implemented in Aten

clearBonds

Syntax:

```
void clearBonds ( )
```

Delete all bonds in the current model.

For example:

```
clearBonds();
```

clearSelectedBonds

Syntax:

```
void clearSelectedBonds ( )
```

Delete all bonds in the current atom selection.

For example:

```
clearSelectedBonds();
```

newBond

Syntax:

```
void newBond ( atom|int i, atom|int j, )
```

```
void newBond ( atom|int i, atom|int j, string|int bondtype )
```

Create a new bond in the model between the specified atoms. The optional *bondtype* argument specified the type of bond: e.g. single (default), double, or triple. Alternatively, an integer number representing the bond order may be given.

For example:

```
newBond(4, 5, "double");
```

creates a double bond between the fourth and fifth atoms in the model.

```
newBond(1, 2, 3);
```

creates a triple bond between the first and second atoms in the model.

```
Atom i,j;
i = newAtom("C", 0, 0, 0);
j = newAtom("H", 0, 1.08, 0);
newBond(i, j, "single");
```

creates a new single bond between two atoms, supplied as references.

rebond

Syntax:

```
void rebond ( )
```

Calculate bonding in the current model.

For example:

```
rebond();
```

rebondpatterns

Syntax:

```
void rebondPatterns ( )
```

Calculate bonding in the current model, but restrict bond creation to between atoms in individual molecules of defined patterns.

For example:

```
rebondPatterns();
```

This command is useful when molecules in a system are too close together to have the correct bonding detected. In such a case, bonds and any old patterns in the model may be cleared, new patterns created by hand, and then 'rebondpatterns' used to calculate bonds only between the atoms of individual molecules in the defined patterns in order to recreate the original molecules.

For example:

```
# Delete existing bonds in model
clearBonds();
# Delete any existing patterns in the model
clearPatterns();
# Add new pattern: 100 molecules of benzene (12 atoms), followed by...
newPattern("benzene", 100, 12);
# ...50 molecules of diethyl-ether (15 atoms)
newPattern("ether", 50, 15);
# Calculate bonds within individual benzene and ether molecules
rebondPatterns();
```

rebondSelection

Syntax:

```
void rebondSelection ( )
```

Calculate bonding between atoms in the current atom selection.

For example:

```
rebondSelection();
```

9.3. Building Commands

Tools to build molecules from scratch, or finalise unfinished models. When creating atoms using the commands listed below, if the coordinates of the new atom are not specified then it is placed at the current pen position. In addition, the reference frame of the pen position is represented as a set of three orthogonal vectors defining the pen's local coordinate system (set initially to the Cartesian axes) centred at an arbitrary origin (the pen position). Subsequent rotations operate on these coordinate axes. Think of it as a 3D version of the old-school turtle.

addHydrogen

Syntax:

```
void addHydrogen ( )  
void addHydrogen ( atom|int i )
```

Satisfy the valencies of all atoms in the current model by adding hydrogens to heavy atoms. If an integer id or atom reference is provided as the argument then the addition of hydrogen is restricted to the specified atom.

For example:

```
addHydrogen();
```

add hydrogens to all atoms in the current model.

```
addHydrogen(10);
```

add hydrogens to atom 10 only.

bohr

Syntax:

```
void bohr ( object x, ... )
```

Converts the specified object(s) data to Å, assuming that it is currently specified in Bohr.

For example:

```
Atom i = aten.model.atoms[2];  
bohr(i);
```

converts the coordinates of the supplied atom from Bohr to Å.

chain

Syntax:

```
Atom chain ( int|string el )
```

```
Atom chain ( int|string el, int|string bondtype )
```

```
Atom chain ( int|string el, double x, double y, double z )
```

```
Atom chain ( int|string el, double x, double y, double z, int|string bondtype )
```

Create a new atom of element *el* at the current pen position (or the specified coordinates) bound to the last drawn atom with a single bond (or of type *bondtype* if it was specified). The element can be provided as a character string containing the element symbol or element name instead of the integer atomic number. A reference to the new atom is returned.

For example:

```
atom i = chain("C");
```

places a carbon atom at the current pen coordinates, and creates a single bond with the last drawn atom.

```
atom i = chain(8, "double");
```

places an oxygen atom at the current pen coordinates, and creates a double bond with the last drawn atom.

```
atom i = chain("Cl", 4.0, 5.0, 6.0, "single");
```

creates a chlorine at coordinates { 4.0, 5.0, 6.0 }, joined by a single bond to the last drawn atom.

endChain

Syntax:

```
void endChain ( )
```

Ends the current chain (so that the next atom drawn with ‘**chain**’ will be unbound).

For example:

```
endChain();
```

growAtom

Syntax:

```
Atom growAtom ( Element el, Atom|int *i, string geometry =
    "tetrahedral", double distance = <auto> )
```

Grows a new atom of the specified element from the specified target atom *i*. The position of the new atom will conform to the required *geometry*, and will have the specified *distance* from the target atom. If the supplied *distance* is negative, then the atomic radii of the target and new atoms are used to calculate the new bond length. If the target atom already has enough (or too many) atoms to prevent a new position from being calculated, then no atom is added. If a new atom is successfully added, it is returned.

For example:

```
Atom i = newAtom(P);
for (int n=0; n<6; ++n) growAtom(F, i, "octahedral");
```

creates a PF₆ (anion), with a perfect octahedral arrangement of F atoms, and calculating the bond distance from the atomic radii of P and F atoms.

locate

Syntax:

```
void locate ( double x, double y, double z )
```

Sets the pen position to the coordinates specified (in Å).

For example:

```
locate(0.0, 0.0, 0.0);
```

moves the pen back to the coordinate origin.

move

Syntax:

```
void move ( double x, double y, double z )
```

Moves the pen position by the amounts specified (in Å).

For example:

```
move(1.0, 1.0, 0.0);
```

moves the pen one Å in both the positive x and y directions.

moveToEnd

Syntax:

```
void moveToEnd ( )
```

Move the current atom selection to the end of the list. The relative order of atoms in the selection is preserved.

For example:

```
select(H);  
moveToEnd();
```

moves all hydrogen atoms to the end of the atom list.

moveToStart

Syntax:

```
void moveToStart ( )
```

Move the current atom selection to the start of the list. The relative order of the atoms in the selection is preserved.

For example:

```
selectType(O, "nbonds=2,-H(n=2)");  
moveToStart();
```

moves all water molecules to the start of the atom list.

newAtom

Syntax:

```
void newAtom ( int|string el )  
  
void newAtom ( int|string el, double x, double y, double z )  
  
void newAtom ( int|string el, double x, double y, double z, double vx,  
double vy, double vz )  
  
void newAtom ( int|string el, double x, double y, double z, double vx,  
double vy, double vz, double fx, double fy, double fz )
```

Create a new atom of element *el* at the current pen position or, if provided, the specified coordinates (and optional velocities or velocities and forces). Either the integer atomic

number or the symbol/name of the element may be used to identify the desired element. A reference to the new atom is returned.

For example:

```
Atom i = newAtom("N");
```

places a nitrogen atom at the current pen coordinates.

```
Atom i = newAtom(18, 5.2, 0, 0);
```

places an argon atom at the coordinates { 5.2, 0.0, 0.0 }.

newAtomFrac

Syntax:

```
void newAtomFrac ( int|string el, double x, double y, double z )  
void newAtomFrac ( int|string el, double x, double y, double z, double  
vx, double vy, double vz )  
void newAtomFrac ( int|string el, double x, double y, double z, double  
vx, double vy, double vz, double fx, double fy, double fz )
```

Create a new atom of element *el* at the specified fractional coordinates (velocities and forces are optional). Either the integer atomic number or the symbol/name of the element may be used to identify the desired element. A reference to the new atom is returned.

For example:

```
Atom i = newAtomFrac("C", 0.5, 0.5, 0.5);
```

places a carbon atom at the centre of the model's cell.

reorder

Syntax:

```
void reorder ( )
```

Adjust the ordering of atoms in the current selection such that atoms in bound fragments/molecules have successive IDs. Useful to recover 'molecularity' in order to apply a suitable pattern description to the system.

For example:

```
reorder();
```

rotX

Syntax:

```
void rotX ( double angle )
```

Rotates the reference coordinate system about the x axis by *angle* degrees.

For example:

```
rotX(90.0);
```

rotates around the x axis by 90 degrees.

rotY

Syntax:

```
void rotY ( double angle )
```

Rotates the reference coordinate system about the y axis by *angle* degrees.

For example:

```
rotY(45.0);
```

rotates around the y axis by 45 degrees.

rotZ

Syntax:

```
void rotZ ( double angle )
```

Rotates the reference coordinate system about the z axis by *angle* degrees.

For example:

```
rotZ(109.5);
```

rotates around the z axis by 109.5 degrees.

selectionAddHydrogen

Syntax:

```
void selectionAddHydrogen ( )
```

Adds hydrogen atoms to the current atom selection only.

selectionGrowAtom

Syntax:

```
void selectionGrowAtom ( Element el, string geometry = "tetrahedral",
double distance = <auto> )
```

Grows a new atom on each of the currently selected atoms, conforming to the specified geometry and distance (if provided). See the **growAtom** command for more information.

shiftDown

Syntax:

```
void shiftDown ( )
```

```
void shiftDown ( int n )
```

Move the current atom selection one (or *n*) places down in the atom list (i.e. towards higher IDs).

For example:

```
shiftDown(4);
```

moves the current atom selection down four places.

shiftUp

Syntax:

```
void shiftUp ( )
```

```
void shiftUp ( int n )
```

Move the current atom selection one (or *n*) places up in the atom list (i.e. towards lower IDs).

For example:

```
shiftUp();
```

moves the current atom selection up one place.

transmute

Syntax:

```
void transmute ( int|string el )
```

Transmute the current atom selection to the specified element.

For example:

```
transmute("F");
```

changes all atoms in the current selection to fluorine.

```
transmute(Cl);
```

changes all atoms in the current selection to chlorine.

```
transmute(6);
```

changes all atoms in the current selection to carbon

9.4. Cell Commands

Create, remove, modify, pack, and replicate the model's unit cell.

adjustCell

Syntax:

```
void adjustCell ( string parameter, double value )
```

Adjust a single unit cell parameter (one of a, b, c, alpha, beta, gamma, or one of the matrix elements ax, ay, az, ..., cz) by the given *value*. This does not set the specified parameter to the given *value*; instead the supplied *value* is added to the existing value of the parameter.

For example:

```
adjustCell("alpha", 5.0);
```

increases the cell angle ‘alpha’ by 5 degrees.

```
adjustcell("c",-10.0);
```

decreases the cell length ‘c’ by 10 Å.

cell

Syntax:

```
void cell ( double a, double b, double c, double alpha, double beta,  
double gamma )
```

Set cell lengths and angles of current model. This command will modify an existing cell or add a new cell to a model currently without a unit cell specification.

For example:

```
cell(20.0, 10.0, 10.0, 90.0, 90.0, 90.0);
```

sets the model's cell to be orthorhombic with side lengths 20x10x10 Å.

cellAxes

Syntax:

```
void cellAxes ( double ax, double ay, double az, double bx, double by,  
double bz, double cx, double cy, double cz )
```

Set cell axes of current model. This command will modify an existing cell or add a new cell to a model currently without a unit cell specification.

For example:

```
cellAxes(15, 0, 0, 0, 15, 0, 0, 0, 15);
```

sets the model's cell to be cubic with side length 15 Å.

fold

Syntax:

```
void fold ( )
```

Fold all atoms so they are within the boundaries of the unit cell.

For example:

```
fold();
```

foldMolecules

Syntax:

```
void foldMolecules ( )
```

Fold all pattern molecules so they are unbroken across cell boundaries.

For example:

```
foldMolecules();
```

millerCut

Syntax:

```
void millerCut ( int h, int k, int l, bool inside = FALSE )
```

Remove all atoms from the unit cell that lay 'outside' the specified Miller plane (and its mirror, if it has one). If the final parameter is given as TRUE, then atoms 'inside' the bounding Miller plane(s) are selected.

For example:

```
millerCut(1,2,1,TRUE);
```

removes all atoms inside the two enclosing (121) planes.

noCell

Syntax:

```
void noCell ( )
```

Clears any cell description (removes periodic boundary conditions) from the current model.

For example:

```
noCell();
```

pack

Syntax:

```
void pack ( )
```

Perform spacegroup packing on the current model.

For example:

```
pack();
```

printCell

Syntax:

```
void printCell ( )
```

Prints the cell parameters of the current model.

For example:

```
printCell();
```

replicate

Syntax:

```
void replicate ( double negx, double negy, double negz, double posx,  
double posy, double posz )
```

Create a supercell of the current model, creating copies of the cell in each of the three cell axis directions. The number of cells to replicate in each positive and negative direction are specified as 'additional' cells beyond the original. So:

```
replicate(0, 0, 0, 0, 0, 0);
```

will do nothing at all to the model, while:

```
replicate(-5, 0, 0, 5, 0, 0);
```

will result in a supercell that consists of eleven copies of the original cell along the x-axis direction. Similarly,

```
replicate(0, 0, 0, 4, 4, 4);
replicate(-2, -2, -2, 2, 2, 2);
```

will both create a 5x5x5 arrangement of the original cell.

scale

Syntax:

```
void scale ( double x, double y, double z, bool calcenergy = FALSE )
```

Scale unit cell and its constituent atoms by the scale factors *x*, *y*, and *z*. The optional *calcenergy* parameter calculates the energy difference resulting from the scaling operation.

For example:

```
scale(1.0, 2.0, 1.0);
```

doubles the length of the y-axis of the system. x- and z-axes remain unchanged.

scaleMolecules

Syntax:

```
void scaleMolecules ( double x, double y, double z, bool calcenergy =
FALSE )
```

Scale unit cell and centres-of-geometry of molecules within it by the scale factors *x*, *y*, and *z*. Within individual molecules the relative distances between atoms stays the same, but the centres-of-geometry of other molecules do not. The optional *calcenergy* parameter calculates the energy difference resulting from the scaling operation.

For example:

```
scaleMolecules(0.5, 0.5, 0.5);
```

halves the lengths of all axes, scaling the positions of the molecules to reflect the new size.

setCell

Syntax:

```
void setCell ( string parameter, double value )
```

Set a single unit cell parameter (one of a, b, c, alpha, beta, gamma, or one of the matrix elements ax, ay, az, ..., cz) to the given value.

For example:

```
setCell("beta", 101.0);
```

sets the cell angle beta to 101 degrees.

```
setCell("a", 15.5);
```

sets the cell length a to 15.5 Å.

spacegroup

Syntax:

```
void spacegroup ( int|string sg )
```

Sets the spacegroup of the model, used for crystal packing.

For example:

```
spacegroup(12);
```

sets the model spacegroup to be C2/m (number 12).

```
spacegroup("P1/m");
```

sets the model spacegroup to P1/m.

9.5. Charges Commands

Assign partial charges to models, atoms, and patterns.

charge

Syntax:

```
double charge ( )  
double charge ( double q )
```

Assigns a charge of q to each selected atom in the current model, or returns the total charge of the current selection if no value is supplied.

For example:

```
charge(1.0);
```

gives each atom in the current model's selection a charge of 1.0.

chargeFF

Syntax:

```
void chargeFF ( )
```

Assigns charges to all atoms in the current model based on the forcefield associated to the model and the current types of the atoms.

For example:

```
chargeFF();
```

chargeFromModel

Syntax:

```
void chargeFromModel ( )
```

Copies charges of all atoms in the current model to the atoms of the current trajectory frame.

For example:

```
chargeFromModel();
```

chargePAtom

Syntax:

```
void chargePAtom ( int id, double q )
```

Assigns a charge of *q* to atom *id* in each molecule of the current pattern.

For example:

```
chargePAtom(3, 0.1);
```

assigns a charge of 0.1 to the third atom in each molecule of the current pattern.

chargeType

Syntax:

```
void chargeType ( string fftype, double q )
```

Assigns a charge of *q* to each atom that is of type *fftype* in the current model.

For example:

```
chargeType ("OW", -0.8);
```

gives a charge of -0.8 to every atom that has an assigned typename of OW.

clearCharges

Syntax:

```
void clearCharges ( )
```

Clears all charges in the current model, setting them to zero.

For example:

```
clearCharges();
```

9.6. ColourScales Commands

Define colourscales to colour objects. Each colourscale has a number of points defining colours at specific values - between adjacent points the colour is linearly interpolated. Points in a colourscale are numbered from 1 onwards. There are ten available colourscales, with IDs 1-10. Some of these have specific uses within Aten.

Colourscales for a discussion of colourscales and how they may be used within Aten.

addPoint

Syntax:

```
void addPoint ( int scaleid, double value, double r, double g, double b, double a = 1.0 )
```

Adds a new point to the end of the specified colourscale, at the point *value* and with the RGB[A] colour provided (each component of which should be in the range 0.0 to 1.0 inclusive).

For example:

```
addPoint(1, 15.0, 0.0, 1.0, 0.0);
```

adds a point to colourscale 1 at a value of 15.0 in a nasty green colour.

clearPoints

Syntax:

```
void clearPoints ( int scaleid )
```

Clears all points from the specified colourscale.

For example:

```
clearPoints(3);
```

clears all points from the third colourscale.

listScales

Syntax:

```
void listScales ( )
```

Lists the current types, colours, and ranges of the colourscales

For example:

```
listScales();
```

removePoint

Syntax:

```
void removePoint ( int scaleid, int pointid )
```

Remove a single point from the selected colourscale.

For example:

```
removePoint(1, 4);
```

deletes the fourth point from colourscale 1.

scaleName

Syntax:

```
string scaleName ( int scaleid )
```

```
string scaleName ( int scaleid, string newname )
```

Retrieves the name of the colourscale id provided, or sets the name if a new name is provided). The name is displayed next to the gradient bar (if drawn).

For example:

```
scaleName(1, "Orientation");
```

renames the first colourscale to ‘Orientation’.

scaleVisible

Syntax:

```
void scaleVisible ( int scaleid, bool visible )
```

Sets whether the gradient bar for the specified colourscale should be drawn in the main view. Default is ‘off’ for all colourscales.

For example:

```
scalevisible(9, "yes");
```

draws the gradient bar for the 9th colourscale in the main view.

setPoint

Syntax:

```
void setPoint ( int scaleid, int pointid, double value, double r,  
double g, double b, double a = 1.0 )
```

Sets the value and colour of an existing point in the specified colourscale.

For example:

```
setPoint(1, 2, -3.3, 1.0, 1.0, 1.0);
```

sets the second point on colourscale 1 to a value of -3.3 and white colour.

setPointColour

Syntax:

```
void setPointColour ( int scaleid, int pointid, double r, double g,  
double b, double a = 1.0 )
```

Sets the colour of an existing point in the specified colourscale.

For example:

```
setPointColour(5, 1, 0.0, 0.0, 1.0);
```

sets the first point on colourscale 5 to be coloured blue.

setPointValue

Syntax:

```
void setPointValue ( int scaleid, int pointid, double value )
```

Sets the value of an existing point in the specified colourscale.

For example:

```
setPointValue(4, 3, 0.1);
```

sets the third point of colourscale 4 to a value of 0.1.

9.7. Disorder Commands

Set up the disordered builder to create systems from individual components using Monte Carlo methods.

disorder

Syntax:

```
void disorder ( string scheme, bool fixedCell = TRUE )
```

Start the disordered builder on the current model (which must already possess a unit cell) employing the current *scheme* (which should correspond to the name of an available partitioning scheme). The size of the unit cell of the current model is, by default, not modified by the disorder builder, but this can be overridden by specifying *fixedCell* as FALSE. From the GUI, the equivalent of doing the latter is to specify the basic relative cell lengths and let the builder determine the final cell size.

For example:

```
disorder("CylinderX", FALSE);
```

runs the disorder builder using the ‘CylinderX’ partitioning scheme, and allows the cell size to change.

listComponents

Syntax:

```
void listComponents ( )
```

Prints a list of the currently requested populations, densities, and destination partitions for all models to be added during the disordered building process.

For example:

```
listComponents();
```

setupComponent

Syntax:

```
void setupComponent ( string policy, int partition = 1, int population = 0, double density = 0.0, bool rotate = TRUE )
```

Instead of setting up a model for insertion by the disorder builder by setting its relevant variables, the **setupComponent** command allows all variables for the current model to be set

simultaneously. The *policy* should correspond to one of “none”, “number”, “density”, “both”, or “relative”, and determines the final population of the model in the resulting disordered model. See Section 7.11.5 for more information on the various insertion policy types. The partition id refers to the partition number into which the model should be inserted, as defined by the partitioning scheme selected when using the **disorder** command.

For example:

```
setupComponent("both", 2, 0, 0.8);
```

sets up the current model to be added into partition number 2 (in the example given in the **disorder** command given above, for example, this would correspond to the cylindrical region) with a final density of 0.8 g cm^{-3} .

9.8. Edit Commands

Standard editing commands.

copy

Syntax:

```
void copy ( )
```

Copy the current atom selection to the clipboard, ready for pasting.

For example:

```
copy();
```

cut

Syntax:

```
void cut ( )
```

Cut the current atom selection to the clipboard.

For example:

```
cut();
```

delete

Syntax:

```
void delete ( )
```

Delete the current atom selection.

For example:

```
delete();
```

paste

Syntax:

```
void paste ( )
```

Paste the copied atom selection.

For example:

```
paste();
```

redo

Syntax:

```
void redo();
```

Redo the last ‘undone’ operation.

For example:

```
redo();
```

undo

Syntax:

```
void undo();
```

Undo the last operation.

For example:

```
undo();
```

9.9. Energy Commands

Calculate energies for models and trajectory frames. All printing commands refer to the last energy calculated for either the model or a trajectory frame.

elec

Syntax:

```
void elec ( string type = "none" )  
  
void elec ( "coulomb" )  
  
void elec ( "ewald" double alpha, int kx, int ky, int kz )  
  
void elec ( "ewaldauto" double precision )
```

Set the style of electrostatic energy calculation to use, either no electrostatics, coulombic (non-periodic) electrostatics, or Ewald-based electrostatics. For the latter, either the various parameters may be defined explicitly (when “ewald” is the chosen method) or may be estimated for the current system by using “ewaldauto”.

frameEnergy

Syntax:

```
double frameEnergy ( )
```

Calculate energy of the current frame of the trajectory associated with the current model.

For example:

```
double energy = frameEnergy();
```

modelEnergy

Syntax:

```
double modelEnergy ( )
```

Calculate the energy of the current model, which can then be printed out (in whole or by parts) by the other subcommands.

For example:

```
double e = modelEnergy();
```

printElec

Syntax:

```
void printElec ( )
```

Prints out the electrostatic energy decomposition matrix.

For example:

```
printElec();
```

printEwald

Syntax:

```
void printEwald ( )
```

Prints the components of the Ewald sum energy.

For example:

```
printEwald();
```

printInter

Syntax:

```
void printInter ( )
```

Prints out the total inter-pattern energy decomposition matrix.

For example:

```
printInter();
```

printIntra

Syntax:

```
void printIntra ( )
```

Prints out the total intramolecular energy decomposition matrix.

For example:

```
printIntra();
```

printEnergy

Syntax:

```
void printEnergy ( )
```

Prints the elements of the calculated energy in a list.

For example:

```
printEnergy();
```

printSummary

Syntax:

```
void printSummary ( )
```

Print out a one-line summary of the calculated energy.

For example:

```
printSummary();
```

printVdw

Syntax:

```
void printVdw ( )
```

Prints out the VDW energy decomposition matrix.

For example:

```
printVdw();
```

9.10. Flow Commands

Loop control and conditional statements.

do

Syntax:

```
do { commands } while { condition }
```

The do-while loop is cousin of the ‘for’ loop, except that there is no control variable. The termination of the loop depends on the *condition* which is tested at the end of every execution of the *commands*. If the condition evaluates to TRUE, the *commands* are executed again, and *condition* re-tested afterwards. If FALSE the loop ends.

For example:

```
int i = 1;
do { i = i * 2; printf("i = %d\n", i); } while (i < 100);
```

will print out the following:

```
i = 2
i = 4
i = 8
i = 16
i = 32
i = 64
i = 128
```

Note that the final value of *i* inside the loop is 128 (greater than 100) since the *condition* is only tested at the end of the execution of the *commands*. The while loop works in the same way, save that the *condition* is tested at the beginning of the loop, before *commands* are executed, rather than at the end.

for

Syntax:

```
for ( startvalue ; condition ; increment ) { commands }
```

Three separate components make up a ‘for’ loop. *startvalue* defines both the control variable (i.e. the variable that changes on each loop iteration) and optionally its starting value, the *condition* is tested on each loop iteration to see whether or not to continue with the loop, and finally the *increment* is an expression to modify the control variable after each iteration, setting it to a new value. If multiple *commands* are to make up the body of the loop (executed on each iteration) then they should be enclosed in curly brackets (as written in the syntax

above). If only a single command is executed on each iteration, the curly brackets may be omitted.

Some examples:

```
for (int i=1; i<=10; i = i + 1) printf("%i\n", i);
```

Loop over and print all integers between 1 and 10. A local variable *i* is declared and initialised all in one go in the *startvalue* part of the loop. The 'long' way of incrementing the integer variable (*i* = *i* + 1) is typically not used in C/C++, most people preferring to take advantage of the C's useful postfix and prefix operators, as in the next example).

```
for (n = 100; n>0; --n) printf("Counting down... %i\n", n);
```

Here, an existing variable *n* is decreased from 100 to 1, printing out all numbers along the way. Note the usage of the double-minus '--' operator (the prefix decrease operator) which decreases its associated variable, in this case *n*. For integers, to decrease means to reduce the value by 1. For other types the meaning may vary – for instance, with reference types the '--' operator means 'previous item in the list', since all such objects in Aten (e.g. atoms) are stored in lists containing many objects of the same type. This makes iterating over, say, all atoms in a given model pretty easy...

```
for (atom a = aten.model.atoms; a; ++a)
{
    printf("Atom id %i is element %s.\n", a.id, a.symbol);
}
```

In this example the variable *a* is declared and initialised to be a reference to the first atom in the current model. The *condition* part simply consists of the expression '*a*', which effectively tests the reference address currently stored in *a*. Since any positive number equates to TRUE (see below for the *if* test) the loop will continue until *a* contains no reference. Since most all reference objects in Aten are stored internally in linked lists, the prefix increment operator (++) changes the value of the variable to be the reference of the next item in the list, or 0 if there are no more items. In this way, the whole list of atoms can be traversed and neatly ended once the final item in the list has passed.

A variant of the *for* loop described above is the *for/in* loop – here, only a control variable and initial value are supplied, both of which must be of pointer types. The loop itself will increase the value of the variable (i.e. skip to the next item in the linked list) until a NULL pointer is found. For example:

```
select(H);
for (atom i = aten.model.selection; i; ++i)
{
    printf("Atom %i is selected\n", i.id);
}
for (atom ii in aten.model.selection)
{
    printf("Atom %i is selected\n", ii.id);
}
```

This will select all hydrogen atoms in the current model then loop over the atom selection twice, once with a for loop and once with a for/in loop, both of which are equivalent.

if

Syntax:

```
if ( condition ) { commands } [ else if { commands } ] [ else { commands }
```

The `if` statement permits sections of code to be executed based on the assessment of logical comparison of values. If the supplied `condition` evaluates to be `TRUE` then the following `commands` are executed, otherwise nothing happens. In the second form of the command, if the `condition` evaluates to be 'false' then the second set of `commands` are executed instead. If multiple `commands` are to be executed then they should be enclosed in curly brackets (as written in the syntax above). If only a single command is to be executed the curly brackets may be omitted.

Typically, comparisons are made between two variables, for example:

```
if ( var1 > var2 ) ...
```

checks for `var1` being greater in value than `var2`, executing the following commands if this turns out to be true. The comparison operator may be any one of the following symbols:

Table 9-1 Comparison Operators

Operator	Meaning
<code>==</code>	Equal to
<code>!=</code>	Not equal to
<code><></code>	Not equal to
<code>></code>	Greater than
<code><</code>	Less than
<code>>=</code>	Greater than or equal to
<code><=</code>	Less than or equal to

In truth, the `condition` part may be any expression, command, or amalgam of both, provided the end result of executing it is a single value. The type of the final result doesn't even matter, since conversion to a boolean is guaranteed. Deep down in the logic of Aten, integers are at the heart of it all, with zero or any negative number being `FALSE`, and any positive number meaning `TRUE`.

For example:

```
int i = 10;
if (i > 5) printf("Hooray!\n");
```

in this case 'Hooray!' *will* be printed, because `i` is greater than 5.

```
int i = 10, j = 20;
if (i > j) printf("Hooray!\n");
```

but in this case 'Hooray!' will *not* be printed, because `i` is not greater than `j`.

```
int i = 10, j = 20;
if (i > j) printf("Hooray!\n");
else { printf("Too small.\n"); i = j; }
```

Here, the test fails for the same reason, but since an `else` part was provided we still execute some commands (printing 'Too small.' and setting the variable `i` equal to `j`).

Since any positive number is TRUE, we can simply test the value of a variable.

```
int i = -55;
if (i) printf("Snoopy.\n");

Atom a = newAtom("H");
if (a) printf("New atom.\n");
```

In a similar way, a reference variable has a positive integer value at its heart, and so can also be tested in this way.

```
Atom a = newAtom("H");
double alpha = 100.0;
if ( (a) && (alpha < 50.0) ) printf("Alpha and atom are OK.\n");
else printf("No good!\n");
```

Two or more consecutive conditions can be tested in order to determine 'truth', in this case using the 'and' operator `&&`. Here, the value of the reference variable `a` and the value of `alpha` are both checked, and the text 'Alpha and atom are OK.' is only printed if both turn out to be TRUE.

```
if (time == 0) printf("There is no time.");
else if (time > 5) printf("There is more than enough time.");
else printf("There is only a little time.");
```

Multiple if tests can also be nested to create a sequence of tests. As soon as a condition is encountered that equates to 'true' the accompanying commands are executed and any subsequent 'else'd tests or commands are ignored.

return

Syntax:

return

return value

Used in (user-defined) functions, and returns control immediately back to the calling function. In the case of a `void` function, no return value must be specified. Similarly, for functions returning a value a valid value of that type must be given.

while

Syntax:

```
while ( condition ) { commands }
```

The while loop is another cousin of the `for` loop, and as with the `do-while` loop there is no control variable. The termination of the loop depends on the *condition* which is tested at the beginning of the loop, before execution of the *commands*. If TRUE, the *commands* are executed, but if FALSE the loop ends without executing the *commands* (and meaning that it is possible that the *commands* are *never* executed).

For example:

```
int i = 1024;
while (i > 100) { i = i / 2; printf("i = %d\n", i); }
```

will print out the following:

```
i = 512
i = 256
i = 128
i = 64
```

9.11. Forcefield Commands

Forcefield management and manual term creation.

angleDef

Syntax:

```
void angleDef ( string form, string type_i, string type_j, string  
    type_k, double data1 ... )
```

Add an angle definition to the current forcefield. *form* should correspond to one of the implemented angle functional forms, while the three *types* refer to either type or equivalent names of defined atom types. Up to ten data parameters may be supplied.

autoConversionUnit

Syntax:

```
void autoConversionUnit ( string unit = "" )
```

Sets the target energy unit for automatic conversion of energetic forcefield parameters when writing out expressions. Can only be used within a file filter definition. The 'unit' parameter should correspond to one of the energy units recognised by Aten (see energy units) or may be omitted to specify that no conversion of parameters from the current internal unit of energy should take place. Note that the conversion of energetic forcefield term parameters is performed only when accessing data through either the 'data' member or 'parameter' function of the forcefield atom, forcefield bound or bound variable types.

For example:

```
autoConversionUnit("kcal");
```

indicates that, no matter what the current internal unit of energy is, all energetic forcefield parameters, when accessed by the means listed above, will be automatically converted into units of kcal.

bondDef

Syntax:

```
void bondDef ( string form, string type_i, string type_j, double data1  
    ... )
```

Add a bond definition to the current forcefield. *form* should correspond to one of the implemented bond functional forms, while the two *types* refer to either type or equivalent names of defined atom types. Up to ten data parameters may be supplied.

clearExportMap

Syntax:

```
void clearExportMap ( )
```

Clear any manual export typemap definitions.

For example:

```
clearExportMap();
```

clearExpression

Syntax:

```
void clearExpression ( )
```

Removes any forcefield expression defined for the current model.

For example:

```
clearExpression();
```

clearMap

Syntax:

```
void clearMap ( )
```

Clear any manual typemap definitions.

For example:

```
clearMap();
```

clearTypes

Syntax:

```
void clearTypes ( )
```

Clear any previously-assigned atom types from the current model.

For example:

```
clearTypes();
```

createExpression

Syntax:

```
int createExpression ( bool noIntra = FALSE, bool noDummy = FALSE, bool assignCharges = TRUE )
```

Creates a suitable energy description for the current model. The optional flags control exactly what is in the created expression, or how it is created. The *noIntra* flag can be used to force the creation of an expression containing only atomtype (i.e. van der Waals) terms - in such a case, patterns will contain no definitions of intramolecular bonds, angles, and torsions whatsoever. The *noDummy* option indicates whether dummy intramolecular terms (of simple functional form and with all parameters set to zero) should be automatically added to the energy expression should no suitable terms be found in the associated forcefield(s). Finally, *assignCharges* specifies whether to assign charges to atoms from their assigned forcefield types (TRUE) or to leave atomic charges as they currently are (FALSE).

For example:

```
createExpression();
```

currentFF

Syntax:

```
void currentFF ( string|int|Forcefield ff )
```

Delete the specified forcefield (i.e. unload it) and remove all reference to it in all models.

deleteFF

Syntax:

```
void deleteFF ( string|int|Forcefield ff )
```

Delete the specified forcefield (i.e. unload it) and remove all reference to it in all models.

equivalents

Syntax:

```
void equivalents ( string name, string typeName(s) ... )
```

Define equivalent terms in the current forcefield. *name* is the new typename to which the list of quoted *typenames* are linked, for subsequent use in intramolecular term definitions. See the equivalents forcefield keyword for more information.

exportMap

Syntax:

```
void exportMap ( string maps )
```

Set up manual mappings that convert atomtype names when expression are exported. Works in the opposite way to the **map** command.

For example:

```
exportMap ("CT=Ctet, N3=N");
```

converts the atomtype names CT and N3 so that they appear as Ctet and N in any expression files written out.

ffModel

Syntax:

```
void ffModel ( )
```

Associates current forcefield to the current model.

For example:

```
ffModel();
```

ffPattern

Syntax:

```
void ffPattern ( string pattern )
```

```
void ffPattern ( int patternid )
```

```
void ffPattern ( Pattern p )
```

Associates current forcefield to the current pattern, or one specified by either a reference, integer ID in the current model, or a pattern pointer.

For example:

```
ffPattern();
```

associates the current forcefield to the current pattern.

```
ffPattern ("bulk");
```

associates the current forcefield to a pattern named bulk in the current model.

finaliseFF

Syntax:

```
void finaliseFF ( )
```

Perform necessary operations on the current forcefield once all data has been added. Must be called!

fixType

Syntax:

```
void fixType ( int typeId, Atom|int id = 0 )
```

Set the current atom selection, or the specified atom, to have the type id (in the current forcefield) specified. Types set in this manner will not be overwritten by the typing routines, allowing specific types to be applied above the normal rules. Note that the type's NETA description is not checked, and so any (even types not matching the base element) may be applied in this way.

For example:

```
typeDef(99, "NX", "NX", N, "-C(n=4)");
select(C);
fixType(99);
```

assigns newly-created type 99 (specific to nitrogen) to all carbons in the model.

freeType

Syntax:

```
void freeType ( Atom|int id = 0 )
```

For the current atom selection, or the specified atom, free any previously-fixed types

For example:

```
freeType(14);
```

frees any previously-set type on atom 14.

generateAngle

Syntax:

```
FFBound generateAngle ( Atom|int i, Atom|int j, Atom|int k)
```

Attempt to generate, from defined generator information in the current Forcefield, expression information for the angle between the specified atoms. The newly (or previously) generated term is returned.

For example:

```
newAtom(O);  
addHydrogen();  
FFBound generateAngle(2, 1, 3);
```

attempts to generate an angle term for the newly-created water molecule.

generateBond

Syntax:

```
FFBound generateBond ( Atom|int i, Atom|int j )
```

Attempt to generate, from defined generator information in the current Forcefield, expression information for the bond between the specified atoms. The newly (or previously) generated term is returned.

generateTorsion

Syntax:

```
FFBound generateTorsion ( Atom|int i, Atom|int j, Atom|int k, Atom|int l )
```

Attempt to generate, from defined generator information in the current Forcefield, expression information for the torsion between the specified atoms. The newly (or previously) generated term is returned.

generateVdw

Syntax:

```
FFAtom generateVdw ( Atom|int i )
```

Attempt to generate, from defined generator information in the current Forcefield, a van der Waals term for the specified atom. The newly (or previously) generated term is returned.

getCombinationRule

Syntax:

```
string getCombinationRule ( string form, string parameter )
```

Returns the combination rule in use for the specified parameter of the given functional form. The *form* and related *parameter* should correspond to those given in the VDW functional

forms table. A string corresponding to one of the available combination rule options is returned.

For example:

```
string cr = getCombinationRule("lj", "epsilon");
```

getFF

Syntax:

```
Forcefield getFF ( string name )
```

```
Forcefield getFF ( int id )
```

Select the named forcefield (or forcefield with the specified *id*) and make it current, returning a reference to it in the process.

For example:

```
Forcefield uff = getFF("uff");
```

makes the loaded forcefield named *uff* the current one, and stores a reference to it.

interDef

Syntax:

```
void interDef ( string form, int typeId, double charge, double data1  
... )
```

Add a new short-range data definition to a type in the current forcefield. *form* should correspond to one of the implemented VDW functional forms. Up to ten parameters for the VDW potential may be given.

loadFF

Syntax:

```
Forcefield loadFF ( string file, string name )
```

Load a forcefield from *file* and reference it by *name*. Becomes the current forcefield.

For example:

```
loadFF("/home/foo/complex.ff", "waterff");
```

loads a forcefield called *complex.ff* and names it *waterff*.

map

Syntax:

```
void map ( string map, ... )
```

Set up manual typename mappings for atom names that do not readily correspond to element symbols, forcefield types etc. All atoms that are subsequently created using *name* as the element are automatically converted to the corresponding element.

For example:

```
map ("CT1=C,CT2=C");
```

converts atoms with names CT1 and CT2 to carbon.

newFF

Syntax:

```
Forcefield newFF ( string name )
```

Create a new, empty forcefield with the given *name* and make it current. Returns a reference to the new forcefield.

For example:

```
Forcefield ff = newFF("testff");
```

printSetup

Syntax:

```
void printSetup ( )
```

Prints the current expression setup.

For example:

```
printSetup();
```

printType

Syntax:

```
void printType ( int id )
```

Prints the NETA description of type *id* in the current forcefield.

For example:

```
printType(99);
```

prints the NETA description of typ id 99.

recreateExpression

Syntax:

```
void recreateExpression ( bool noIntra = FALSE, bool noDummy = FALSE,  
bool assignCharges = TRUE )
```

Delete and recreate a suitable energy description for the current model. The optional *noIntra*, *noDummy*, and *assignCharges* flags control various aspects of parameter generation, as described in the *createExpression*.

For example:

```
recreateExpression();
```

saveExpression

Syntax:

```
int saveExpression ( string filter, string filename )
```

Export the forcefield expression for the current model in the format determined by the *filter* nickname, to the *filename* specified. Return value is 1 for successful write, or 0 otherwise.

For example:

```
saveExpression("dlpoly", "data.FIELD");
```

setCombinationRule

Syntax:

```
void setCombinationRule ( string form, string parameter, string rule )
```

Sets the combination rule to use for the specified parameter of the given functional form. The *form* and related *parameter* should correspond to those given in the VDW functional forms table, while *rule* should correspond to one of the available combination rule options.

For example:

```
setCombinationRule("lj", "sigma", "geometric");
```

torsionDef

Syntax:

```
void torsionDef ( string form, string type_i, string type_j, string  
    type_k, string type_l, double data1 ... )
```

Add a torsion definition to the current forcefield. *form* should correspond to one of the implemented torsion functional forms, while the four *types* refer to either type or equivalent names of defined atom types. Up to ten real-valued parameter values for the function may be provided.

typeDef

Syntax:

```
int typeDef ( int typeid, string name, string equiv, string|int  
    element, string neta, string description = "" )
```

Add a new atom type definition to the current forcefield, with the identifying *typeid* and called *name*, with the equivalent typename *equiv*. The basic element of the new type is given as *element*, and *neta* is the NETA definition of the type. An optional string describing the type in more detail can be given in *description*. The command returns '1' if the model was typed successfully or '0' otherwise.

For example:

```
typeDef(101, "Ctet", C, "nbonds=4", "Standard tetrahedral carbon");
```

creates a new simple type for a carbon atom with four bonds.

typeModel

Syntax:

```
int typeModel ( )
```

Perform atom typing on the current model. Returns 1 if atom typing was performed successfully or 0 otherwise.

For example:

```
int success = typeModel();
```

typeTest

Syntax:

```
int typeTest ( int typeId, Atom|int id )
```

Test the current forcefield's atomtype *typeId* on the atom specified, returning the type score of the match (zero indicating no match).

For example:

```
int score = typeTest(112, 10);
```

tests type id 112 on the tenth atom in the model.

units

Syntax:

```
void units ( string unit )
```

Sets the units in which energetic parameters are given for the current forcefield. For a list of available units see Section 16.8.

9.12. Forces Commands

Calculate forces for models and trajectory frames.

frameForces

Syntax:

```
void frameForces ( )
```

Calculate the atomic forces of the current frame of the trajectory associated with the current model.

For example:

```
frameForces();
```

modelForces

Syntax:

```
void modelForces ( )
```

Calculate the atomic forces of the current model.

For example:

```
modelForces();
```

printForces

Syntax:

```
void printForces ( )
```

Print out the forces of the current model.

For example:

```
printForces();
```

9.13. Glyph Commands

Add glyphs to atoms in the model.

autoEllipsoids

Syntax:

```
void autoEllipsoids ( )
```

Note: Experimental Feature!

Using the current atom selection, this command creates ellipsoid glyphs that cover (or represent) individual bound fragments within the selection. An ellipsoid glyph is added for each bound fragment within the selection, positioned at the geometric centre of the bound fragment, and scaled in an attempt to cover all atoms within the bound fragment. Such things are useful when wanting to represent molecules by simple geometric shapes, rather than by their fine-grained atomic positions.

For instance, given a box full of benzene molecules:

```
selectAll();  
autoEllipsoids();
```

will add on a flattened ellipsoid to each individual molecule. To do the same thing but using only the ring carbons to generate the ellipsoids:

```
select("C");  
autoEllipsoids();
```

Now the ellipsoids will cover the carbon atoms in each ring, leaving the hydrogens poking out.

autoPolyhedra

Syntax:

```
void autoPolyhedra ( string options = "" )
```

Note: Very Experimental Feature!

In a similar way to the **autoEllipsoids** command, **autoPolyhedra** adds triangle glyphs to the current selection in an attempt to enclose certain atoms within solid structures. There are two principal modes of operation. The first (the default) assumes that the current atom selection consists of individual atoms that should be enclosed in a polyhedron made up from triangles added between triplets of bound neighbours. The carbon atom at the centre of methane would make a good example. The alternative mode (requested with the ‘fragments’

option) assumes that atoms within individual bound fragments in the current selection should be used as the vertices to form an enclosed shell.

Possible *options* are:

Table 9-2 AutoPolyhedra Options

Option	Effect
<code>centres</code>	Assume that the current selection consists of individual atomic centres that should be enclosed (the default)
<code>fragments</code>	Use individual bound fragments instead of assuming individual centres
<code>nolink</code>	Do not link the coordinates of generated glyphs to coordinates of the atoms (the default is to link glyph coordinates to atoms)
<code>rcut=distance</code>	Specifies the maximum distance allowed between vertices of a triangle

glyphAtomF

Syntax:

```
void glyphAtomF ( int n )  
void glyphAtomF ( int n, Atom|int sourceAtom )
```

Set current (or specified) atom's forces as data *n* in the current glyph.

For example:

```
glyphAtomF(1);
```

links the current atoms forces to the first datum in the current glyph.

glyphAtomR

Syntax:

```
void glyphAtomR ( int n )  
void glyphAtomR ( int n, Atom|int sourceAtom )
```

Set current (or specified) atom's position as data *n* in the current glyph.

For example:

```
glyphAtomR(3, 55);
```

links the 55th atom's position to the third datum in the current glyph.

glyphAtomV

Syntax:

```
void glyphAtomV ( int n )  
void glyphAtomV ( int n, Atom|int sourceAtom )
```

Set current (or specified) atom's velocity as data *n* in the current glyph.

For example:

```
Atom i = newAtom("H");  
glyphAtomV(2,i);
```

links the velocity of new atom *i* to the second datum in the current glyph.

glyphAtomsF

Syntax:

```
void glyphAtomsF ( Atom|int sourceAtom ... )
```

Accepts one or more atoms, setting consecutive data in the current glyph to the forces of the atoms / atom IDs provided.

For example:

```
glyphAtomsF(1, 2, 3);
```

links the forces of atoms 1, 2, and 3 to the first three glyph data.

glyphAtomsR

Syntax:

```
void glyphAtomsR ( Atom|int sourceAtom ... )
```

Accepts one or more atoms, setting consecutive data in the current glyph to the positions of the atoms / atom IDs provided.

For example:

```
glyphAtomsR(3, 10);
```

links the positions of atoms 3 and 10 to the first two glyph data.

glyphAtomsV

Syntax:

```
void glyphAtomsV ( Atom|int sourceAtom ... )
```

Accepts one or more atoms, setting consecutive data in the current glyph to the velocities of the atoms / atom IDs provided.

For example:

```
glyphAtomsV(9, 11, 13);
```

links the velocities of atoms 9, 11, and 13 to first three glyph data.

glyphColour

Syntax:

```
void glyphColour ( int n, double r, double g, double b, double a = 1.0 )
```

Set the colour of vertex *n* for the current glyph to the RGB(A) colour provided (each component of which should be in the range 0.0 to 1.0 inclusive).

For example:

```
glyphColour(1, 1.0, 0.0, 0.0);
```

sets the colour of the first vertex in the current glyph to red.

glyphData

Syntax:

```
void glyphData ( int n, double r, double g, double b )
```

Set vector data *n* for the current glyph to the fixed values provided.

For example:

```
glyphData(1, 0.0, 5.0, 2.4);
```

sets the first positional data in the glyph to {0.0, 5.0, 2.4}.

glyphSolid

Syntax:

```
void glyphSolid ( bool isSolid )
```

Sets the drawing style of the current glyph to solid (true) or wireframe (false) (if the glyph style permits).

For example:

```
glyphSolid("true");
```

glyphText

Syntax:

```
void glyphText ( string text )
```

Set the text data in the current glyph. For text-style glyphs, this is a necessary piece of data.

For example:

```
glyphText ("Coordinate Origin");
```

newGlyph

Syntax:

```
Glyph newGlyph ( string style, string options = "" )
```

Create a new glyph of the specified style, and make it current. The colour of the glyph is set using the default glyph colour set in the global preferences. Valid glyph styles are listed in **glyph types**. Positional / size / scale vector data should be set afterwards with appropriate **glyphAtom*** and **glyphData*** commands.

One or more options may be given to the command. The list of possible *options* is:

Table 9-3 NewGlyph Options

Option	Effect
<code>solid</code>	Render the glyph in solid mode (if the glyph supports it). Same as calling ' <code>glyphSolid(TRUE)</code> ' after creation
<code>text=<i>string</i></code>	Set the character data associated to the glyph to <i>string</i> .
<code>wire</code>	Valid only for glyphs which display text Render the glyph in wireframe mode (if the glyph supports it). Same as calling ' <code>glyphSolid(FALSE)</code> ' after creation

For example:

```
newGlyph ("cube");
```

creates a new cube in the model.

```
newGlyph("text", "text=\\"I am some text\\\"");
```

creates a new text glyph in the model, reading “I am some text”. Note the need to escape the quotes surrounding the text.

```
newGlyph("tetrahedron", "wire");
```

creates a new wireframe tetrahedron in the model.

9.14. Grid Commands

Add gridded data to the current model.

Grid window for management of grids in the GUI.

addGridPoint

Syntax:

```
void addGridPoint ( int ix, int iy, int iz, double value )
```

Set a specific data point in the current grid.

For example:

```
addGridPoint(4, 1, 15, 4.123);
```

set the grid value at point { 4,1,15 } in the dataset to 4.123.

addNextGridPoint

Syntax:

```
void addNextGridPoint ( double value )
```

Add the next sequential grid point, starting at (1,1,1) and proceeding through dimensions as defined by the **gridLoopOrder** command (default is x→y→z, i.e. (1,1,1) is set first, then (2,1,1), (3,1,1) etc.).

For example:

```
addNextGridPoint(20.0);
```

sets the next grid point value to be 20.0.

finaliseGrid

Syntax:

```
void finaliseGrid ( )
```

Perform internal post-load operations on the grid. Must be called for every new grid, after all data has been read in.

For example:

```
finaliseGrid();
```

gridAlpha

Syntax:

```
double gridAlpha ( )  
double gridAlpha ( double newAlpha )
```

Set the alpha value (transparency of the current surface), with 0.0 being fully opaque and 1.0 being fully transparent (i.e. invisible), or simply return the current alpha value if no new value is provided. Note that this command sets the alpha values for both the primary and secondary surface colours.

For example:

```
gridAlpha(0.5);
```

gridAxes

Syntax:

```
void gridAxes ( double ax, double ay, double az, double bx, double by,  
double bz, double cx, double cy, double cz )
```

Set the axes of the current grid, specified as three vectors.

For example:

```
gridAxes(1, 0, 0, 0, 1, 0, 0, 0, 1);
```

sets a cubic system of axes for the current grid.

```
gridAxes(0.8, 0, 0, 0.1, 0.6, 0, 0, 0, 0.7);
```

sets a monoclinic system of axes for the current grid.

gridColour

Syntax:

```
void gridColour ( double r, double g, double b, double a = 1.0 )
```

Set the internal colour of the primary grid surface to the RGB(A) value (each component of which should be in the range 0.0 to 1.0 inclusive).

For example:

```
gridColour(1.0, 1.0, 0.0);
```

sets the primary surface colour to yellow.

gridColourSecondary

Syntax:

```
void gridColourSecondary ( double r, double g, double b, double a = 1.0 )
```

Set the internal colour of secondary grid surface to the RGB(A) value supplied (each component of which should be in the range 0.0 to 1.0 inclusive).

For example:

```
gridColourSecondary(0.9, 0.9, 0.9);
```

sets the secondary surface colour to off-white.

gridColourScale

Syntax:

```
void gridColourScale ( int id )
```

Set the colourscale to use for the current grid to the colourscale ID specified, which should be in the range 1-10. If '0' is given as the argument, the internal colour of the grid data is used. Linking a colourscale to a grid will result in the minimum and maximum ranges of the grid being recalculated to ensure all points in the grid are covered by the scale, whose range is adjusted if necessary.

For example:

```
gridColourScale(4);
```

colours the grid data according to colourscale 4.

```
gridColourScale(0);
```

uses the internal colour(s) specified for the grid.

gridCubic

Syntax:

```
void gridCubic ( double l )
```

Sets up a cubic system of axes for the current grid.

For example:

```
gridCubic(0.5);
```

sets up a cubic system of axes, each grid point 0.5 Å apart in all directions.

gridCutoff

Syntax:

```
void gridCutoff ( double lowercut, [double uppercut] )
```

Sets the lower and (if supplied) upper cutoff values for the current grid.

For example:

```
gridCutoff(0.002,0.005);
```

sets the lower grid cutoff for the current grid to 0.002, and the upper grid cutoff to 0.005.

gridCutoffSecondary

Syntax:

```
void gridCutoffSecondary ( double lowercut, [double uppercut] )
```

Sets the lower and (if supplied) upper secondary cutoff values for the current grid.

For example:

```
gridCutoffSecondary(0.0014);
```

sets the secondary lower grid cutoff for the current grid to 0.0014, leaving the upper secondary cutoff unchanged.

gridLoopOrder

Syntax:

```
void gridLoopOrder ( string order )
```

Set the grid loop order to use with addnextgridpoint, affecting in which order the dimensions of the data grid are filled. *order* should be given as a string of three characters, e.g. ‘xyz’ (equivalent to ‘123’), ‘yxz’ (equivalent to ‘213’), etc.

For example:

```
gridLoopOrder("zyx");
```

sets the loop order to the reverse of the default, so that the z-index is filled first.

gridOrigin

Syntax:

```
void gridOrigin ( double x, double y, double z )
```

Sets the origin of the grid data, in Å.

For example:

```
gridOrigin(0, 10, 0);
```

sets the grid origin to be offset 10 Å along the y-axis.

gridOrtho

Syntax:

```
void gridOrtho ( double a, double b, double c )
```

Sets up an orthorhombic system of axes for the grid data.

For example:

```
gridOrtho(0.5, 0.5, 0.8);
```

sets up a system of axes elongated in the z-axis.

gridStyle

Syntax:

```
void gridStyle ( string style )
```

Determines how the current grid data is drawn on-screen. Valid *styles* are listed in Section 16.10.

For example:

```
gridStyle("triangles");
```

draws the current grid as a triangle mesh.

gridUseZ

Syntax:

```
int gridUseZ ( )  
  
int gridUseZ ( bool usez )
```

For two-dimensional grid (i.e. surface) data this option controls whether the data value is used as the height (z) component of the surface, or if no data value is used and the surface is flat. If called with no arguments the current status of the option is returned (0 being off, and 1 being on).

For example:

```
gridUseZ("on");
```

gridVisible

Syntax:

```
int gridVisible ( )  
  
int gridVisible ( bool visible )
```

Set the visibility of the current grid (i.e. whether it is drawn on screen).

For example:

```
gridVisible(FALSE);
```

initGrid

Syntax:

```
void initGrid ( string type, int nx, int ny, int nz )
```

Initialises the current grid to be of the specified grid type and with the dimensions specified (if the *type* requires it). This must be called before any calls to addpoint or addnextgridpoint are issued.

For example:

```
initGrid("regularxyz", 64, 128, 64);
```

initialises the current grid to be a regularly-spaced and hold a total of (gets calculator...) 524,288 points.

loadGrid

Syntax:

```
Grid loadGrid ( string filename )
```

Load an existing grid from the specified file, and add it to the current model. If successfully loaded, a reference to the new grid is returned.

For example:

```
Grid density = loadGrid("density.pdens");
```

loads a grid called “density.pdens” and attaches it to current model.

newGrid

Syntax:

```
Grid newGrid ( string name )
```

Creates a new, empty grid with the provided 'name' in the current model, and returns a reference to it.

For example:

```
Grid g = newGrid("charlie");
```

creates a new grid called, for some reason, “charlie”.

9.15. Image Commands

Save bitmap and vector images of the current view. The GUI is not required in order to save images – using these commands from the command-line, for example, works just as well (models still have a current view, even without the GUI, and can be rotated, translated etc. just as if they were in the GUI).

saveBitmap

Syntax:

```
void saveBitmap ( string format, string filename )  
  
void saveBitmap ( string format, string filename, int width, int height )  
  
void saveBitmap ( string format, string filename, int width, int height, int quality )
```

Saves the current view as a bitmap image. Allowable values for *format* are:

Table 9-4 Bitmap Formats

Format	Description
bmp	Windows Bitmap
jpg	Joint Photographic Experts Group
png	Portable Network Graphics
ppm	Portable Pixmap
xbm	X11 Bitmap
xpm	X11Pixmap

If *width* and *height* are not specified the current dimensions of the view are used (800x600 if no GUI is present). The *quality* option determines the compression used on saved images, affecting, for example, the quality and size of jpegs and pngs, and should be an integer between 1 and 100 (with 100 being the best compression).

For example:

```
saveBitmap("bmp", "test.bmp");
```

saves the current view to a file test.bmp.

```
saveBitmap("png", "big.png", 5000, 5000, 10);
```

saves an enormous, highly uncompressed png.

saveMovie

Syntax:

```
void saveMovie ( string filename, string format, int width = -1 int  
height = -1 int quality = -1, int firstFrame = 1 int lastFrame =  
<last>, int interval = 1 )
```

Saves a movie of the trajectory associated to the current model. Note that a valid movie encoder must be installed (such as `mencoder`) and arguments must be set in the program preferences (see the `Prefs` variable). Aten will first save a series of png images of the width, height and quality specified to the temporary directory (also specified in the preferences) before passing them all to the provided encoder command. The encoder arguments should contain both the text ‘FILES’ and ‘OUTPUT’ – when it comes to running the command, Aten will substitute ‘FILES’ for a wildcard list of image files, and ‘OUTPUT’ for the target movie filename. Note that the format of the output movie is entirely guided by the options passed to the encoder command.

As with the `saveBitmap` command, if *width* and *height* are not specified the current dimensions of the view are used (800x600 if no GUI is present), and the *quality* option determines the compression used on saved images. The other arguments are self-explanatory - *firstframe* and *lastframe* give the range of trajectory frames which will be saved, while the *interval* value determines the ‘stride’ between frames (i.e. 1 for every frame, 2 for every other frame, 10 for every tenth frame etc.).

For example:

```
saveMovie("traj.mpg", 1024, 1024, -1, 50, 1000, 50);
```

saves a movie called “traj.mpg” with size 1024x1024, beginning at frame 50 and writing every 50th frame until frame 1000.

9.16. Labeling Commands

Add and remove atom labels.

clearLabels

Syntax:

```
void clearLabels ( )
```

Remove all atom labels in the current model.

For example:

```
clearLabels();
```

label

Syntax:

```
void label ( string type, Atom|int id = 0 )
```

Adds the specified label to each atom in the current selection, or alternatively just to the atom specified. Valid *types* are listed in Section 16.12.

For example:

```
label("element");
```

adds element labels to the current atom selection.

removeLabel

Syntax:

```
void removeLabel ( string type, Atom|int id = 0 )
```

Remove the specified label (if it exists) from each atom in the current selection, or alternatively just from the atom specified.

For example:

```
removeLabel("equiv");
```

removes the forcefield equivalent type label from each atom in the current selection.

removeLabels

Syntax:

```
void removeLabels ( Atom|int id = 0 )
```

Remove all labels from all atoms in the current selection, or alternatively just the atom specified.

9.17. Math Commands

Standard mathematical functions.

abs

Syntax:

```
double abs ( int|double num )
```

Returns the absolute (positively-signed) value of *num*.

cos

Syntax:

```
double cos ( double angle )
```

Returns the cosine of *angle* (which should be given in degrees).

dotProduct

Syntax:

```
double dotproduct ( vector u, vector v )
```

Calculate and return the dot product of the two vectors *u* and *v*.

exp

Syntax:

```
double exp ( double x )
```

Returns the exponential of *x*.

ln

Syntax:

```
double ln ( double x )
```

Returns the natural base-e logarithm of *x*.

log

Syntax:

```
double log ( double x )
```

Returns the base-10 logarithm of *x*.

nint

Syntax:

```
int nint ( double x )
```

Returns the nearest integer value to *x*.

normalise

Syntax:

```
double normalise ( vector v )
```

Normalises the vector *v*, returning the magnitude of the vector before the normalisation was performed.

For example:

```
vector v = { 1, 2, 3 };
double mag = normalise(v);
printf("Normalised vector is { %f, %f, %f }, mag was %f\n", v.x, v.y, v.z, mag);
```

prints the following:

```
Normalised vector is { 0.267261, 0.534522, 0.801784 }, mag was 3.741657
```

random

Syntax:

```
double random ( )
```

Return a random real number between 0.0 and 1.0 inclusive.

randomI

Syntax:

```
int randomI ( int max = RANDMAX )
```

Return a random integer between 0 and RANDMAX inclusive, or 0 and *max* if it is supplied.

sin

Syntax:

```
double sin ( double angle )
```

Returns the sine of *angle* (which should be given in degrees).

sqrt

Syntax:

```
double sqrt ( double x )
```

Returns the square root of *x*.

tan

Syntax:

```
double tan ( double angle )
```

Returns the tangent of *angle* (which should be given in degrees).

9.18. Measuring Commands

Make measurements of distances, angles, and torsion angles (dihedrals) in models. Note that there are two distinct sets of commands – those which 'measure' and those which calculate 'geometry'. The former create visible measurements within the model (which can then be viewed in the GUI), while the latter simply determine and return geometric values. Both sets take a variable number of arguments which determine whether a distance, angle, or torsion is measured/determined.

clearMeasurements

Syntax:

```
void clearMeasurements ( )
```

Clear all measurements in the current model.

For example:

```
clearMeasurements();
```

geometry

Syntax:

```
double geometry ( Atom|int i, Atom|int j, Atom|int k = 0, Atom|int l = 0 )
```

This command is a general measuring tool, able to measure distances, angles, and torsions in the model, depending on how many arguments are supplied. Note that, unlike the `measure` command, the resulting measurement is *not* added to the Model's internal list, and thus will not be displayed in the model.

For example:

```
double rij[50];
for (int i=1; i<=50; ++i) rij[i] = geometry(1,i);
```

calculates the distances between the first 50 atoms in the model and the first, regardless of whether they are bound or not

listMeasurements

Syntax:

```
void listMeasurements ( )
```

List all measurements in the current model.

For example:

```
listMeasurements();
```

prints out a list of measurements made so far.

measure

Syntax:

```
double measure ( Atom|int i, Atom|int j, Atom|int k = 0, Atom|int l = 0 )
```

This command is a general measuring tool, able to measure distances, angles, and torsions in the model, depending on how many arguments are supplied. Note that the resulting measurement is added to the Model's internal list, and will be displayed in the model. Also, note that measuring the same thing between the same atoms twice will remove the measurement from the Model.

For example:

```
double rij = measure(1, 2);
```

returns the distance between atoms 1 and 2.

```
double theta = measure(10, 20, 30);
```

returns the angle between atoms 10, 20, and 30.

```
double phi = measure(9, 8, 7, 6);
measure(9,8,7,6);
```

returns the torsion angle made between atoms 9, 8, 7, and 6, and then instantly removes it from the model by measuring it again.

measureSelected

Syntax:

```
void measureSelected ( int natoms )
```

This command is a general measuring tool to measure all of one particular type of interaction (i.e. bond distances, angles, or torsions) within the current atom selection. The single argument specifies the type of interaction to calculate by specifying the number of atoms

involved in the interaction – i.e. 2, 3, or 4 for bond distances, angles, and torsions respectively.

For example:

```
measureSelected(3);
```

calculates and displays all bond angles in the current atom selection.

9.19. Messaging Commands

Output messages from command lists / filters. All commands work like the C printf() command, and accept the same fundamental format specifiers. All output from these messaging commands is directed to either the GUI message box or stdout on the command line.

createDialog

Syntax:

```
Dialog createDialog ( string title = <none> )
```

Create a new, temporary Dialog with titlebar text *title* (if provided).

For example:

```
Dialog ui = createDialog("Choose a Number");
ui.addIntegerSpin("chooser", "Choice", 1, 100, 1, 25);
if (!ui.show()) error("Dialog Canceled. ");
else
{
    int n = ui.asInteger("chooser");
    printf("You chose %i\n", n);
}
```

defaultDialog

Syntax:

```
Dialog defaultDialog ( string title = <none> )
```

Returns the default Dialog structure for this filter / script / function, setting the window titlebar text to *title*, if it is provided.

For example:

```
Dialog ui = defaultDialog();
if (!ui.show()) error("Dialog Canceled. ");
```

error

Syntax:

```
void error ( string format, ... )
```

Print a message to screen and immediately exit the current command structure / filter.

For example:

```
int err=23;
error("Filter failed badly - error = %i.\n", err);
```

notifies the user that something bad probably happened and promptly exits.

printf

Syntax:

```
void printf ( string format, ... )
```

Standard printing command.

For example:

```
printf("Loading data...\n");
```

prints the string “Loading data...” to the screen.

```
printf("Number of atoms = %i\n", natoms);
```

prints the contents of the variable *natoms* to the screen.

showDefaultDialog

Syntax:

```
int showDefaultDialog ( string title = <none> )
```

Shows (executes) the default `Dialog` structure for this filter / script / function, setting the window titlebar text to *title*, if it is provided.

For example:

```
if (!showDefaultDialog()) error("Dialog Canceled. ");
```

verbose

Syntax:

```
void verbose ( string format, ... )
```

Prints a message, but only when verbose output is enabled (with the `-v` command-line switch).

For example:

```
verbose("Extra information for you.\n");
```

9.20. Minimiser Commands

Perform energy minimisation on models.

cgMinimise

Syntax:

```
void cgMinimise ( int maxsteps )
```

Geometry optimises the current model using the conjugate gradient method.

For example:

```
cgMinimise(20);
```

runs a conjugate gradient geometry optimisation for a maximum of 20 cycles.

Literature methods for details on the conjugate gradient method as it is implemented in Aten.

converge

Syntax:

```
void converge ( double econv, double fconv )
```

Sets the convergence criteria of the minimisation methods. Energy and force convergence values are given in the current working unit of energy in the program.

For example:

```
converge(1e-6, 1e-4);
```

sets the energy and RMS force convergence criteria to 1.0E-6 and 1.0E-4 respectively.

lineTol

Syntax:

```
void lineTol ( double tolerance )
```

Sets the tolerance of the line minimiser.

For example:

```
lineTol(1e-5);
```

sets the line tolerance to 1.0E-5.

mcMinimise

Syntax:

```
void mcMinimise ( int maxsteps )
```

Optimises the current model using a molecular Monte Carlo minimisation method.

For example:

```
mcMinimise(20);
```

runs a geometry optimisation for a maximum of 20 cycles.

Monte Carlo Minimiser method for details on the Monte Carlo minimisation method as it is implemented in Aten.

mopacMinimise

Syntax:

```
void mopacMinimise ( string options = "BFGS PM6 RHF SINGLET" )
```

Optimises the current model using the external MOPAC program (Copyright 2007, Stewart Computational Chemistry). Note that the program must be installed correctly as per the instructions provided with it, and the path to the MOPAC executable must be set in Aten's preferences, as well as a suitable temporary working directory. The optional argument allows a specific MOPAC command to be provided for the minimisation, but sensible defaults are used if this is not provided.

For example:

```
mopacMinimise();
```

minimises the current model with the default options listed above.

```
mopacMinimise("UHF TRIPLET PM6-DH2");
```

minimises the current model assuming a triplet state with the UHF method and the PM6-DH2 hamiltonian.

sdMinimise

Syntax:

```
void sdMinimise ( int maxsteps )
```

Optimises the current model using the Steepest Descent method.

For example:

```
sdMinimise(100);
```

minimises the current model for a maximum of 100 steps with a simple steepest descent minimiser.

9.21. Model Extras Commands

Store and manipulate molecular orbital data, vibrations, and z-matrix elements.

newBasisShell

Syntax:

```
basisshell newBasisShell ( Atom|int id, string type )
```

Adds a new basis shell definition to the current model, returning the generated structure. The atomic centre on which the basis function exists must be provided either as an integer or an atom pointer (from which the integer ID is extracted). The type of the basis shell should correspond to one listed in basis shell types (see Section 16.1).

For example:

```
BasisShell = newBasisShell(15, "D");
```

creates a new D-orbital shell centred on atom 15.

newEigenvector

Syntax:

```
Eigenvector newEigenvector ( int size = <auto> )
```

Adds a new, empty eigenvector to the current model. If the *size* argument is given the eigenvector array will contain this many elements. Otherwise, the size of the array is determined by the total number of cartesian basis functions implied by the current basis shell definitions of the model.

For example:

```
Eigenvector = newEigenvector(180);
```

creates a new eigenvector which will contain 180 coefficients.

newVibration

Syntax:

```
Vibration newVibration ( string name = <auto> )
```

Adds a new, empty vibration definition to the current model.

printZMatrix

Syntax:

```
void printZMatrix ( )
```

Prints a Z-matrix for the current model to the console, creating one first if necessary.

9.22. Model Commands

Model creation and management.

createAtoms

Syntax:

```
void createAtoms ( )
```

Can be run when importing trajectory frames. Creates enough atoms in the current trajectory frame to match the parent model.

For example:

```
createAtoms();
```

currentModel

Syntax:

```
Model currentModel ( )
```

```
Model currentModel ( int id )
```

```
Model currentModel ( string name )
```

```
Model currentmodel ( Model m )
```

Returns a reference to the current model (if no argument is given) or selects the supplied model and makes it the current model. The model may be selected either by name, by its integer position in the list of loaded models (i.e. 1 to N), or a model-type variable containing a valid model reference may be passed.

For example:

```
currentModel(4);
```

selects the fourth loaded model.

```
currentModel("Protein coordinates");
```

selects the model named “Protein coordinates” (provided it exists).

```
Model m1, m2;  
m1 = newModel("Test model 1");
```

```
m2 = newModel("Test model 2");
currentModel(m1);
```

creates two models, storing references to each, and then re-selects the first one and makes it the current target again.

deleteModel

Syntax:

```
void deleteModel ( int id )
void deleteModel ( string name )
void deleteModel ( Model m )
```

Deletes the current model (if no argument is given) or the supplied model.

finaliseModel

Syntax:

```
void finaliseModel ( )
```

Performs various internal tasks after a model has been fully created within a filter. Should be called after all operations on each created model have finished.

For example:

```
finaliseModel();
```

firstModel

Syntax:

```
model firstModel ( )
```

Makes the first loaded / created model the current model, and returns a reference to it.

For example:

```
firstModel();
```

getModel

Syntax:

```
Model getModel ( int id )
```

```
Model getModel ( string name )
```

```
Model getModel ( Model m )
```

Returns a reference to the requested model, but unlike **currentModel** does not make it the current model.

For example:

```
Model alpha = getModel("alpha2");
```

grabs a reference to the model named “alpha2”.

```
Model m = getModel(5);
```

gets a reference to the fifth loaded model.

info

Syntax:

```
void info ( )
```

Print out information on the current model and its atoms.

For example:

```
info();
```

lastModel

Syntax:

```
model lastModel ( )
```

Makes the last loaded / created model the current model, and returns a reference to it.

For example:

```
Model final = lastModel();
```

listModels

Syntax:

```
void listModels ( )
```

Lists all models currently available.

For example:

```
listModels();
```

loadModel

Syntax:

```
Model loadModel ( string filename )
```

```
Model loadModel ( string filename, string filter = <automatic> )
```

Load model(s) from the *filename* provided, autodetecting the format of the file. If the optional *filter* argument is present, then the file is forcibly loaded using the filter with the corresponding nickname. The last loaded model becomes the current model, to which a reference is returned.

For example:

```
Model m = loadModel("/home/foo/coords/test.xyz");
```

loads a model called `test.xyz`, returning a reference to it.

```
Model m = loadModel("/home/foo/coords/testfile", "xyz");
```

forces loading of the model `testfile` as an `xyz` file.

logInfo

Syntax:

```
void logInfo ( )
```

Prints out log information for the current model.

modelTemplate

Syntax:

```
void modelTemplate ( )
```

Can only be run when importing trajectory frames. Templates the atoms in the trajectory's parent model by creating an equal number of atoms in the target trajectory frame, and copying the element and style data. Positions, forces, and velocities are not copied from the parent model atoms.

For example:

```
modelTemplate();
```

newModel

Syntax:

```
model newModel ( string name )
```

Create a new model called *name* which becomes the current model, and return a reference to it.

For example:

```
newModel("emptymodel");
```

creates a new, empty model called `emptymodel` and makes it current.

```
Model c12 = newModel("dodecane");
```

creates a new, empty model called `dodecane`, makes it current, and stores a reference to it in the variable `c12`.

nextModel

Syntax:

```
Model nextModel ( )
```

Skips to the next loaded model, makes it current, and returns a reference to it.

For example:

```
Model next = nextModel();
```

parentModel

Syntax:

```
void parentModel ( )
```

Makes the parent model of the current trajectory frame the current model.

For example:

```
parentModel();
```

prevModel

Syntax:

```
Model prevModel( )
```

Skips to the previous loaded model, makes it current, and returns a reference to it.

For example:

```
model prev = prevmodel();
```

saveModel

Syntax:

```
int saveModel( string format, string filename )
```

Save the current model in the *format* given (which should correspond to a model export Filter nickname) to the *filename* specified. If the save was successful, an integer value of '1' is returned, otherwise '0'.

For example:

```
int success = saveModel("xyz", "/home/foo/newcoords/test.config");
```

saves the current model in xyz format to the filename given.

saveSelection

Syntax:

```
int saveSelection( string format, string filename )
```

Save the atom selection in the current model in the *format* given (which should correspond to a model export Filter nickname) to the *filename* specified. If the save was successful, an integer value of 1 is returned, otherwise 0. Unit cell information is also saved, if the model has any.

setName

Syntax:

```
void setName( string name )
```

Sets the name of the current model.

For example:

```
setName ("panther");
```

gives the current model the cool-sounding name of panther! Ahem.

showAll

Syntax:

```
void showAll ( )
```

Makes any previously-hidden atoms in the model visible again.

9.23. Pattern Commands

Automatically or manually create pattern descriptions for models.

clearPatterns

Syntax:

```
void clearPatterns ( )
```

Delete the pattern description of the current model. It's a good idea to run this command before adding a pattern definition by hand with calls to newpattern.

For example:

```
clearPatterns();
```

createPatterns

Syntax:

```
void createPatterns ( )
```

Automatically detect and create the pattern description for the current model.

For example:

```
createPatterns();
```

currentPattern

Syntax:

```
Pattern currentPattern ( )
```

```
Pattern currentPattern ( string name )
```

```
Pattern currentPattern ( int id )
```

Get the named pattern or pattern with given id (if either was specified), returning its reference and setting it to be the current pattern.

For example:

```
Pattern p = currentPattern("liquid");
```

sets the pattern named 'liquid' in the current model to be the current pattern, setting its reference in the variable *p*.

```
Pattern p = currentPattern();
```

returns a reference to the current pattern.

getPattern

Syntax:

```
Pattern getPattern ( string name )
```

```
Pattern getPattern ( int id )
```

Get the named pattern, or pattern with id specified, returning its reference.

For example:

```
Pattern p = getPattern("solid");
```

gets the pattern named "solid" in the current model, setting its reference in the variable *p*.

```
getPattern(3);
```

gets the third pattern in the current model.

listPatterns

Syntax:

```
void listPatterns ( )
```

List the patterns in the current model.

For example:

```
listPatterns();
```

newPattern

Syntax:

```
Pattern newPattern ( string name, int nMols, int atomsPerMol )
```

Add a new pattern node to the current model, spanning *nMols* molecules of *atomsPerMol* atoms each, and called *name*. A reference to the new pattern is returned.

For example:

```
Pattern p = newPattern("water", 100, 3);
```

creates a new pattern description of 100 molecules of 3 atoms each named “water” (i.e. 100 water molecules) in the current model, and returns its reference

9.24. Pores Commands

createScheme

Syntax:

```
int createScheme ( string name, int nx = 50, int ny = 50, int nz = 50,  
double minSizePcnt = 0.05, int atomExtent = 2, bool copyToBuilder =  
TRUE )
```

Create a partitioning scheme (called *name*) from empty space in the current model. An $nx \times ny \times nz$ grid is created and scheme partitions generated based on adjacent free cells in the grid once those containing atoms have been removed. Cells adjacent to those containing atoms are also removed, based on the *atomExtent*, which can be viewed as an integer defining the radius (in cells) of ‘spherical’ region around each atom. The defaults offer sensible values, but these may need to be tweaked when, for example, small pores are present. The minimum size (in grid cells) allowable for any discovered partition is governed by the *minSizePcnt* parameter multiplied by the total number of grid points. The final argument, *copyToBuilder*, determines whether the generated scheme is immediately copied to the Disorder builder, ready for use. The default is TRUE, but this can be set to FALSE if, for instance, you wish to adjust the other parameters in order to get the number and size of partitions you require first, before committing a scheme to the Disorder builder. The number of partitions found is returned.

For example:

```
createScheme("", 50, 50, 50, 10.0, );
```

drillPores

Syntax:

```
void drillPores ( string geometry, double sizeParameter, int nA, int  
nB, int originFace = 3, double vx = 0.0, double vy = 0.0, double vz =  
1.0 )
```

Drills an $nA \times nB$ array of pores of the specified *geometry* in the current model. The *originFace* of the pores defaults to the XY (AB) plane, but the YZ (BC) and XZ (AC) faces may be selected instead by specifying *originFace* as 1 or 2 respectively. Similarly, the vector along which the individual pores are drilled defaults to be normal to the z axis (0.0,0.0,1.0), but may be specified to any other vector suitable. The vector is normalised before use, so a vector of any magnitude may be specified (for example, the components of a cell axis vector, making drilling pores along crystal axes quite simple).

At present, the only implemented geometry is “cylindrical”.

For example:

```
drillPores("cylindrical", 5.0, 3, 3);
```

creates a 3x3 array of cylindrical pores of radius 5.0 Å along the z-axis (the default).

```
loadModel("data/test/amorphous-silica.ato");
drillPores("cylindrical", 5.0, 1, 3, 2, 25.0, 43.3013, 0.0);
```

creates a 1×3 array of cylindrical pores of radius 5.0 Å along the example amorphous silica model cell's B axis.

selectPores

Syntax:

```
int selectPores ( string geometry, double sizeParameter, int nA, int nB, int originFace = 3, double vx = 0.0, double vy = 0.0, double vz = 1.0 )
```

This acts as a complement to the **drillPores** command, operating in exactly the same way except that atoms which would be in the pores are not deleted from the model. Obviously, this can be used to assess the exact positions of pores before they are cut from the model. Furthermore, once the pores have been drilled proper, by increasing the size parameter and using **selectPores** atoms in the pore wall can be selected and then OH terminated with the **terminate** command. The number of atoms selected is returned. Note that any existing atom selection is cleared before the new pore atoms are selected.

For example:

```
drillPores("cylindrical", 5.0, 3, 3);
selectPores("cylindrical", 7.0, 3, 3);
```

As in the **drillPores** example, a 3×3 array of pores is drilled, and then all atoms in a 2 Å 'wall slice' are selected around the edge of each pore.

terminate

Syntax:

```
void terminate ( )
```

Adds hydrogen atoms and OH groups to atoms in the current selection, in order to completely satisfy the bonding requirements of the selected atoms. Only affects oxygen and silicon atoms at present.

For example:

```
terminate();
```

9.25. Read / Write Commands

Methods of reading and writing data from / to files in import and export filters. Many commands here use formatting strings to provide formatted input and output. All reading and writing commands here work on input or output files as defined internally by the program.

addReadOption

Syntax:

```
void addReadOption ( string option )
```

Controls aspects of file reading. See Section 16.15 for a list of possible *options*.

For example:

```
addReadOption("stripbrackets");
```

eof

Syntax:

```
int eof ( )
```

Returns 1 if at the end of the current input file, 0 otherwise.

For example:

```
string s;
while (!eof()) { getLine(s); printf("%s\n", s); }
```

Reads in and prints out all lines in the current source file.

filterFilename

Syntax:

```
string filterFilename ( )
```

Returns the name of the current input or output file (typically useful from within an import or export filter).

For example:

```
string filename = filterFilename();
```

Puts the current source/destination filename in the variable *filename*.

find

Syntax:

```
int find ( string searchString )  
  
int find ( string searchString, string lineVar )
```

Searches for the specified *searchString* in the input file, returning 0 if *searchString* is not found before the end of the file, and 1 if it is. The optional argument *lineVar* is a character variable in which the matching line (if any) is put. If the search string is not found the file position is returned to the place it was before the command was run.

For example:

```
int iresult = find("Final Energy");
```

searches for the string 'Final Energy' in the input file, placing the result of the search in the variable *iresult*.

```
string line;  
int n = find("Optimised Geometry:", line);
```

searches for the string "Optimised Geometry:" in the input file, placing the whole of the matching line from the input file in the variable *line*.

getLine

Syntax:

```
int getLine ( string destvar )
```

Read an entire line from the input file, and put it in the character variable provided. The line also becomes the current target for readnext. The command returns a Read Success integer (see Section 16.16).

For example:

```
string nextline;  
int n = getLine(nextline);
```

gets the next line from the file and places it in the variable *nextline*.

nextArg

Syntax:

```
int nextArg ( int i )
```

```
int nextArg ( double d )  
int nextArg ( string s )
```

Read the next whitespace-delimited chunk of text from the current file and place it in the variable supplied. Note that this command reads directly from the file and not the last line read with getline or readline (see the readnext command to read the next delimited argument from the last read line). The command returns TRUE (1) if an argument was successfully read, or FALSE (0) otherwise (e.g. if the end of the file was found).

The command will accept a variable of any ordinary type – one of int, double, or string – as its argument. Conversion between the text in the file and the target variable type is performed automatically.

For example:

```
int i;  
int success = nextArg(i);
```

peekChar

Syntax:

```
string peekChar ( )
```

Peeks the next character that will be read from the source file, and returns it as a string. The actual file position for reading is unaffected.

For example:

```
string char = peekChar();
```

peekCharI

Syntax:

```
int peekCharI ( )
```

Peeks the next character that will be read from the source file, and returns it as an integer value representing the ASCII character code (see <http://www.asciitable.com>, for example). The actual file position for reading is unaffected.

For example:

```
int char = peekCharI();
```

readChars

Syntax:

```
string readChars ( int nChars, bool skipEol = TRUE )
```

Reads and returns (as a string) a number of characters from the input file. If *skipEol* is true (the default) then the end-of-line markers ‘\n’ and ‘\r’ will be ignored and will not count towards *nChars* – this is of most use when reading formatted text files and you want to ignore the fact that data is presented on many lines rather than one. If *skipEol* is false then ‘\n’ and ‘\r’ will count towards the total number of characters. Used on formatted text files, this might give you unexpected results.

For example:

```
string text = readChars(80);
```

reads the next 80 characters from the input file and puts it into the variable *text*.

readDouble

Syntax:

```
double readDouble ( )
```

```
double readDouble ( int nBytes )
```

Read a floating point value (the size determined from the machines ‘double’ size) from an unformatted (binary) input file. Alternatively, if a valid number of bytes is specified and corresponds to the size of another ‘class’ of double (e.g. long double) on the machine this size is used instead.

For example:

```
double x = readDouble();
```

reads a floating point value into the variable *x*.

readDoubleArray

Syntax:

```
int readDoubleArray ( double d[], int n )
```

Read *n* consecutive integer values (whose individual size is determined from the result of calling ‘sizeof(double)’) from an unformatted (binary) input file, placing in the array *d* provided. The size of the array provided must be at least *n*. The command returns a Read Success integer (see Section 16.16).

For example:

```
double data[45];
int success = readDoubleArray(data, 45);
```

reads 45 double numbers into the array *data*.

readInt

Syntax:

```
int readInt ( )
int readInt ( int nBytes )
```

Read an integer value (the size determined from the result of calling ‘`sizeof(int)`’) from an unformatted (binary) input file. Alternatively, if a valid number of bytes is specified and corresponds to the size of another class of int (e.g. long int) on the machine this size is used instead.

For example:

```
int i = readInt();
```

reads an integer number into the variable *i*.

readIntArray

Syntax:

```
int readIntArray ( int i[], int n )
```

Read *n* consecutive integer values (whose individual size is determined from the result of calling ‘`sizeof(int)`’) from an unformatted (binary) input file, placing in the array *i* provided. The size of the array provided must be at least *n*. The command returns a Read Success integer (see Section 16.16).

For example:

```
int data[100]; int success = readintarray(data, 100);
```

reads 100 integer numbers into the array *data*.

readLine

Syntax:

```
int readLine ( int|double|string var, ... )
```

Read a line of delimited items from the input file, placing them into the list of variable(s) provided. Conversion of data from the file into the types of the destination variables is performed automatically. The number of items parsed successfully is returned.

For example:

```
double x,y,z;
int n = readLine(x,y,z);
```

reads a line from the file and places the first three delimited items on the line into the variables *x*, *y*, and *z*.

readLineF

Syntax:

```
int readLineF ( string format, ... )
```

Read a line of data from the input file and separate them into the list of variable(s) provided, and according to the format provided. The number of items parsed successfully is returned.

For example:

```
double x,y,z;
int n = readLineF("%8.6f %8.6f %8.6f",x,y,z);
```

reads a line from the file, assuming that the line contains three floating point values of 8 characters length, and separated by a space, into the three variables *x*, *y*, and *z*.

readNext

Syntax:

```
int readNext ( int i )
```

```
int readNext ( double d )
```

```
int readNext ( string s )
```

Read the next delimited argument from the last line read with either getline or readline into the variable supplied. The command returns either TRUE for success or FALSE (e.g. if the end of file was reached without reading any non-whitespace characters, or an error was encountered).

For example:

```
double d;
int n = readNext(d);
```

read the next delimited argument into the double variable *d*.

readVar

Syntax:

```
int readVar ( string source, int|double|string var, ... )
```

Parse the contents of the supplied string *source* into the supplied variables, assuming delimited data items. Delimited items read from *source* are converted automatically to the type inferred by the target variable. The number of data items parsed successfully is returned.

For example:

```
string data = "rubbish ignore Carbon green 1.0 2.5 5.3";
string element, discard; vector v;
int n = readVar(data,discard,discard,element,discard,v.x,v.y,v.z,discard);
printf("Element = %s, n = %i\n", element, n);
```

outputs

```
Element = Carbon, n = 7
```

The character string in the variable *data* is parsed, with delimited chunks placed into the supplied variables. Note the repeated use of the variable *discard*, used to get rid of unwanted data. Also, note that there are not enough items in *data* to satisfy the final occurrence of *discard*, and so the function returns a value of 7 (as opposed to the actual number of target variables supplied, 8).

readVarF

Syntax:

```
int readVarF ( string source, string format, int|double|string var, ... )
```

Parse the contents of the supplied string *source* according to the supplied *format* string, placing in the supplied variables. The number of format specifiers successfully parsed (or, to look at it another way, the number of the supplied variables that were assigned values) is returned.

For example:

```
string a, b, data = "abc def123456.0";
double d; int i, n;
n = readVarF(data,"%3s %3s%4i%4f%8*",a,b,i,d);
printf("a = %s, b = %s, d = %f, i = %i, n = %i\n", a, b, d, i, n);
```

outputs

```
a = abc, b = def, d = 56.000000, i = 1234, n = 4
```

The supplied format string contains a single space in between the two ‘%3s’ specifiers, and is significant since it corresponds to actual (discarded) space when processing the format. Furthermore, the last specifier ‘%8*’ (discard 8 characters) is not fulfilled by the *data* string, and so the number of arguments successfully parsed is 4, not 5.

removeReadOption

Syntax:

```
void removeReadOption ( string option )
```

Removes a previously-set read option. See Section 16.15 for a list of possible *options*.

For example:

```
removeReadOption("skipblanks");
```

rewind

Syntax:

```
void rewind ( )
```

Rewind the input file to the beginning of the file.

For example:

```
rewind();
```

skipChars

Syntax:

```
void skipChars ( int n )
```

Skips the next *n* characters in the input file.

For example:

```
skipChars(15);
```

discards the next 15 characters from the input file.

skipLine

Syntax:

```
void skipLine ( int n = 1 )
```

Skips the next line in the file, or the next *n* lines if a number supplied.

For example:

```
skipLine();
```

skips the next line in the file.

```
skipLine(5);
```

discards 5 lines from the file.

writeLine

Syntax:

```
void writeLine ( int|double|string var, ... )
```

Write a line to the current output file that consists of the whitespace delimited contents of the supplied arguments. The contents of the arguments are formatted according to their type and suitable internal defaults. A newline character is appended automatically to the end of the written line.

For example:

```
writeLine("Number of atoms =", aten.model.nAtoms);
```

writes a line indicating the number of atoms in the model to the current output file.

writeLineF

Syntax:

```
void writeLineF ( string format, ... )
```

Write a formatted line to the current output file, according to the supplied *format* and any supplied arguments. Usage is the same as for the printf command. Note that a newline character is *not* automatically appended to the end of the written line, and one should be written explicitly using the escape sequence ‘\n’.

For example:

```
writeLineF("%s = %8i\n", "Number of atoms", aten.model.nAtoms);
```

writes a line indicating the number of atoms in the model to the current output file, e.g.:

```
Number of atoms = 3
```

writeVar

Syntax:

```
void writeVar ( string dest, ... )
```

Write to the supplied string variable *dest* the whitespace delimited contents of the supplied arguments. The contents of the arguments are formatted according to their type and suitable internal defaults. A newline character is appended automatically to the end of the written line.

For example:

```
string s;
writeVar(s,"Number of atoms =", aten.model.nAtoms);
writeLine(s);
```

same result as the example for the **writeLine** command, except that the final string is written to a variable first, and then the file.

writeVarF

Syntax:

```
void writeVarF ( string dest, string format, ... )
```

Write to the supplied string variable *dest* the string resulting from the supplied *format* and any other supplied arguments. Apart from the mandatory first argument being the destination string variable, usage is otherwise the same as the printf command. Note that a newline character is *not* automatically appended to the end of the written line, and one should be written explicitly using the escape sequence ‘\n’.

For example:

```
string s;
writeVarF(s,"%s = %8i\n", "Number of atoms", aten.model.nAtoms);
writeLine(s);
```

same result as the example for the **writeLineF** command, except that the final string is written to a variable first, and then the file.

9.26. Script Commands

Commands to load and run scripts.

listScripts

Syntax:

```
void listScripts ( )
```

Lists all loaded scripts.

For example:

```
listScripts();
```

loadScript

Syntax:

```
void loadScript ( string filename )
```

```
void loadScript ( string filename, string nickname )
```

Loads a script from the *filename* specified, giving it the optional *nickname*.

For example:

```
loadScript("scripts/liquid-water.txt", "water");
```

loads the script from “scripts/liquid-water.txt” and gives it the nickname “water”.

runScript

Syntax:

```
void runScript ( string name )
```

Executes the specified script.

For example:

```
runScript("water");
```

executes the water script loaded in the previous example.

9.27. Selection Commands

Select atoms or groups of atoms within the current model.

deSelect

Syntax:

```
int deSelect ( Atom|int|string selection, ... )
```

Deselect atoms in the current model, returning the number of atoms deselected by the provided selection arguments. One or more arguments may be supplied, and each may be of the type int, atom, or string. In the case of the first two types, individual atoms (or those corresponding to the integer id) are deselected. On the other hand, strings may contain ranges of atom IDs and element symbols permitting atoms to be deselected in groups. Ranges are specified as ‘a–b’ where a and b are either both atom IDs or both element symbols. In addition, the ‘+’ symbol can be used before (‘+a’) or after (‘a+’) an atom ID or element symbol to mean either ‘everything up to and including this’ or ‘this and everything after’. Within a string argument, one or more selection ranges may be separated by commas.

For example:

```
deSelect(5);
```

deselects the 5th atom.

```
deSelect("1-10,N");
```

deselects the first ten atoms, and all nitrogen atoms.

```
int n = deSelect("Sc-Zn");
```

deselects the first transition series of elements, returning the number of atoms that were deselected in the process.

```
deSelect("C+");
```

deselects all elements carbon and above.

```
deSelect(1, 2, 5, "8+");
```

deselects the first, second, and fifth atoms, as well as the eighth atom and all that occur after it.

deSelectF

Syntax:

```
int deSelectF ( string format, ... )
```

Deselect atoms using the same syntax as the **deSelect** command, but constructing the string using a C-style **printf** approach. Useful when two integer numbers defining a range of atoms to deselect are stored in two local variables, for instance, or when the selection range must change dynamically in a loop.

For example:

```
int i = 10;
deSelectF("%i-%i", i, i+10);
```

deselects atom ids 10 to 20 inclusive.

deSelectFor

Syntax:

```
int deSelectFor ( string code )
```

Compiles and executes the *code* supplied within a loop which runs over all atoms. The control variable of the loop in which the *code* is inserted is of type **Atom** and is named *i*. The *code* provided should use this pointer to decide whether or not the atom in question should be selected or not, returning **TRUE** or **FALSE** in the process. A return path resulting in **FALSE** need not be specified, since this is the default if the code supplied does not return a value.

For example:

```
deSelectFor("if (i.z == 6) return TRUE;");
```

will deselect all carbon atoms in the current model.

deSelectType

Syntax:

```
int deSelectType ( string|int|Element el, string neta )
```

Deselect all atoms in the current model matching the element and NETA type description specified.

For example:

```
deSelectType(H, "-O(-C)");
```

will deselect all hydrogen atoms bound to oxygen atoms which are, in turn, bound to carbon atoms (i.e. all hydroxyl hydrogens).

expand

Syntax:

```
int expand ( )
```

Expand the current selection of atoms by selecting any atoms that are directly bound to an already-selected atom. The number of atoms added to the previous selection is returned.

For example:

```
expand();
```

invert

Syntax:

```
int invert ( )
```

Inverts the selection of all atoms in the current model. Returns the number of atoms selected.

For example:

```
invert();
```

select

Syntax:

```
int select ( Atom|int|string selection, ... )
```

Select atoms in the current model, keeping any previous selection of atoms. See the **deSelect** command for a full description of the syntax. The number of atoms added to the existing selection is returned.

For example:

```
select("+5");
```

selects the first five atoms.

```
int n = select("+5,H");
```

selects the first five atoms and all hydrogens, storing the number of new atoms selected in the variable *n*.

selectAll

Syntax:

```
int selectAll ( )
```

Select all atoms in the current model. The number of selected atoms is returned.

For example:

```
selectAll();
```

selectFFType

Syntax:

```
int selectFFType ( string ffType )
```

Select all atoms with forcefield type *ffType* in the current model. The number of atoms selected is returned.

For example:

```
selectFFType("CT");
```

selects all atoms that have been assigned the forcefield type ‘CT’.

selectF

Syntax:

```
int selectF ( string format, ... )
```

Selects atoms according to a string generated from a C-style **printf** call. See the **deSelectF** command for a full description.

selectFor

Syntax:

```
int selectFor ( string code )
```

Select atoms using the supplied *code*, which is inserted inside a loop over all atoms in the current model. See the **deSelectFor** command for a full description.

selectInsideCell

Syntax:

```
int selectInsideCell ( bool useCog = FALSE )
```

Select all atoms whose coordinates are currently inside the confines of the unit cell (if one exists). If *useCog* is TRUE whole molecules are selected if their centre of geometry is within the unit cell. The number of newly-selected atoms is returned.

For example:

```
int n = selectInsideCell();
```

selectionCog

Syntax:

```
Vector selectionCog ( )
```

Return the centre of geometry of the current atom selection.

For example:

```
Vector v = selectionCog();
printf("Centre of geometry of current selection is: %f %f %f\n", v.x, v.y, v.z);
```

calculates and prints the centre-of-geometry of the current selection.

selectionCom

Syntax:

```
Vector selectionCom ( )
```

Return the centre of mass of the current atom selection.

For example:

```
Vector v = selectionCom();
newAtom(Be, v.x, v.y, v.z);
```

calculates the centre-of-mass of the current selection and creates a beryllium atom at those coordinates.

selectLine

Syntax:

```
int selectLine ( double lx, double ly, double lz, double x, double y,
double z, double radius )
```

Selects all atoms that are within a distance *radius* from a line whose direction is $\{lx, ly, lz\}$ and which passes through the point $\{x, y, z\}$. The number of newly-selected atoms is returned.

For example:

```
selectLine(0,0,1,0,0,0,5.0);
```

selects all atoms within 5 Å of a line running through the origin along the Cartesian z direction.

selectMiller

Syntax:

```
int selectMiller ( int h, int k, int l, int inside = FALSE )
```

Select all atoms that are ‘outside’ the specified Miller plane (and its mirror, if it has one). If the final parameter is specified as TRUE then atoms *inside* the specified Miller plane (and its mirror) are selected.

For example:

```
selectMiller(1, 1, 1);
```

selects all atoms located beyond the (111) plane in the unit cell.

selectMolecule

Syntax:

```
int selectMolecule ( Atom|int target )
```

Select all atoms in the molecule / fragment to which the supplied *target* atom belongs.

For example:

```
selectMolecule(5);
```

selects the bound fragment in which atom number 5 exists.

selectNone

Syntax:

```
void selectNone ( )
```

Deselect all atoms in the current model.

For example:

```
selectNone();
```

selectOverlaps

Syntax:

```
int selectOverlaps ( double dist = 0.2 )
```

Select all atoms that are within a certain distance of another, or the default of 0.2 Å if no argument is provided. The number of selected overlapping atoms is returned.

For example:

```
int noverlaps = selectOverlaps("0.1");
```

selects all atoms that are less than 0.1 Å away from another.

selectOutsideCell

Syntax:

```
int selectOutsideCell ( bool useCog = FALSE )
```

Select all atoms whose coordinates are currently outside the confines of the unit cell (if one exists). If *useCog* is TRUE whole molecules are selected if their centre of geometry is outside the unit cell. The number of newly-selected atoms is returned.

For example:

```
int n = selectOutsideCell();
```

selectPattern

Syntax:

```
int selectPattern ( int id, string name, Pattern p )
```

Selects all atoms in the current (or named/specify) pattern. Returns the number of atoms added to the existing selection.

For example:

```
selectPattern(2);
```

select all atoms in the second pattern of the current model.

```
selectPattern("bubble");
```

select all atoms in the pattern “bubble” of the current model.

selectRadial

Syntax:

```
int selectRadial ( Atom|int id, double r )
```

Select all atoms within *r* Å of the supplied target atom (which is also selected). Returns the number of atoms added to the existing selection.

For example:

```
int nclose = selectRadial(10, 4.5);
```

selects all atoms within 4.5 Å of atom 10, and returns the number selected.

selectTree

Syntax:

```
int selectTree ( Atom|int i, bond exclude = NULL )
```

Select all atoms which are reachable by following chemical bonds, starting from (and including) atom *i*. If a bond to *exclude* is provided, then this connection will not be followed during the selection process. This is useful when one wishes to select a ‘headgroup’ fragment attached to an atom, without selecting the rest of the molecule. The number of atoms selected by the process is returned.

For example:

```
int nclose = selectTree(99);
```

selects all atoms reachable by following chemical bonds from (and including) atom 99.

selectType

Syntax:

```
int selectType ( int|string element, string neta )
```

Selects all atoms of the given *element* that also match the NETA description (see Section 12.5) given, allowing selections to be made based on the connectivity and local environment of atoms. The number of (previously unselected) atoms matched is returned.

For example:

```
int nch2 = selectType("C", "-H(n=2)");
```

selects all carbon atoms that are bound to two hydrogens.

9.28. Site Commands

Describe sites within molecules for use in analysis.

Note: These commands are outdated and may be removed completely in a future version.

getSite

Syntax:

```
void getSite ( string name )
```

Selects (makes current) the site referenced by *name*. If the site cannot be found an error is returned.

For example:

```
getSite("carb1");
```

makes “carb1” the current site.

listSites

Syntax:

```
void listSites ( )
```

Prints the list of sites defined for the current model.

For example:

```
listSites();
```

newSite

Syntax:

```
void newSite ( string name, string atomlist = "" )
```

Creates a new site *name* for the current model, based on the molecule of *pattern*, and placed at the geometric centre of the atom IDs given in *atomlist*. If no atoms are given, the centre of geometry of all atoms is used. The new site becomes the current site.

For example:

```
newSite("watercentre", "h2o");
```

adds a site called 'watercentre' based on the pattern called 'h2o' and located at the centre of geometry of all atoms.

```
newSite("oxy", "methanol", "5");
```

adds a site called 'oxy' based on the pattern called 'methanol' and located at the fifth atom in each molecule.

siteAxes

Syntax:

```
void siteAxes ( string x_atomlist, string y_atomlist )
```

Sets the local x (first set of atom IDs) and y (second set of atom IDs) axes for the current site. Each of the two axes is constructed by taking the vector from the site centre and the geometric centre of the list of atoms provided here. The y axis is orthogonalised with respect to the x axis and the z axis constructed from the cross product of the orthogonal x and y vectors.

For example:

```
siteAxes("1,2", "6");
```

sets the x axis definition of the current site to be the vector between the site centre and the average position of the first two atoms, and the y axis definition to be the vector between the site centre and the position of the sixth atom.

9.29. String Commands

Manipulation and conversion of string variables.

afterStr

Syntax:

```
string afterStr ( string source, string search, bool sourceOnFail =  
FALSE )
```

Return the part of the *source* string that comes after the first occurrence of the *search* string. If *source* doesn't contain any occurrences of *search* an empty string is returned, unless the flag *sourceOnFail* is set to TRUE in which case the original source string is returned instead.

For example:

```
string fullname = "BobBobbinson";  
string firstname = afterStr(name, "Bob");
```

sets the variable *firstname* to the value 'Bobbinson'.

```
string text1, text2;  
text1 = "No taxes on axes";  
text2 = afterStr(text1, "x");
```

results in *text2* having a value of 'es on axes'.

atof

Syntax:

```
double atof ( string text )
```

Converts the supplied *text* into its floating point (double) representation. Equivalent to the standard C routine 'atof'.

For example:

```
double x = atof("1.099d");
```

would set *x* to the value '1.099'.

atoi

Syntax:

```
int atoi ( string text )
```

Converts the supplied *text* into its integer representation. Equivalent to the standard C routine 'atoi'.

For example:

```
int i = atoi("000023");
```

would set *i* to the value '23'.

beforeStr

Syntax:

```
string beforeStr ( string source, string search, bool sourceOnFail =  
    FALSE )
```

Return the part of the *source* string that comes before the first occurrence of the *search* string. If *source* doesn't contain any occurrences of *search* an empty string is returned, unless the flag *sourceOnFail* is set to TRUE in which case the original source string is returned instead.

For example:

```
string source, target;  
source = "Engelbert";  
target = beforeStr(source, "e");
```

places the text "Eng" in the variable *target*.

```
string text1 = "No taxes on axes";  
string text2 = beforeStr(text1, " ax");
```

places the text "No taxes on" in the variable *text2*.

contains

Syntax:

```
int contains ( string source, string search )
```

Returns the number of times the *search* string is found in the *source* string. The function counts only non-overlapping occurrences of *search*.

For example:

```
string poem = "six sixes are thirty-six";
int count = contains(poem, "six");
```

sets *count* to ‘3’.

ftoa

Syntax:

```
string ftoa ( double d )
```

Converts the supplied double *d* into a string representation.

For example:

```
string num = ftoa(100.001);
```

would set *num* to the value ‘100.00’.

itoa

Syntax:

```
string itoa ( int i )
```

Converts the supplied integer *i* into a string representation.

For example:

```
string num = itoa(54);
```

would set *num* to the value “54”.

lowerCase

Syntax:

```
string lowerCase ( string source )
```

Returns the source string with all uppercase letters converted to lowercase.

replaceChars

Syntax:

```
string replaceChars ( string source, string searchChars, string
replaceChar )
```

Searches through the supplied *source* string, and replaces all occurrences of the individual characters given in the string *searchChars* with the character supplied in *replaceChar*, returning the new string.

For example:

```
string newstring = replaceChars("Zero 2599 these numbers", "123456789", "0");
```

replaces any numeric character with a zero.

replaceStr

Syntax:

```
string replaceStr ( string source, string searchStr, string replaceStr )
```

Replaces all occurrences of *searchStr* with *replaceStr* in the supplied *source* string, returning the result.

For example:

```
string fruity = replaceStr("I don't like apples.", "apples", "oranges");
```

would change your fondness towards apples.

removeStr

Syntax:

```
string removeStr ( string source, string searchStr )
```

Removes all occurrences of *searchStr* from the *source* string, returning the result.

For example:

```
string debt = removeStr("I owe you 2 million dollars.", "million ");
```

would reduce your outgoings considerably.

sprintf

Syntax:

```
string sprintf ( string dest, string format, ... )
```

Prints a formatted string to the supplied variable *dest*, and is an alias for the **writeVarF** command.

stripChars

Syntax:

```
string stripChars ( string source, string charList )
```

Strip the supplied character(s) from the *source* string, returning the result.

For example:

```
string abc = stripChars("Oodles of noodles", "o");
```

places the text “Odles f ndles” in the variable *abc*.

```
string abc = "Oodles of noodles";
abc = stripChars(abc, "aeiou");
```

strips all vowels from the input string, placing the text ‘Odls f ndl’ in the variable *abc*.

toa

Syntax:

```
string toa ( string format, ... )
```

Returns a string formatted according to the supplied *format*.

upperCase

Syntax:

```
string upperCase ( string source )
```

Returns the source string with all lowercase letters converted to uppercase.

9.30. System Commands

System commands for controlling debug output, instantiation of the GUI, and exiting from the program.

debug

Syntax:

```
void debug ( string type )
```

Toggles debug output from various parts of the code. A full list of valid types is given in output types.

For example:

```
debug ("parse");
```

getEnv

Syntax:

```
string getEnv ( string variable )
```

Retrieves the contents of the named environment variable so that transfer of useful quantities can be made to Aten through a shell.

For example:

```
string s = getEnv ("HOSTNAME");
```

gets the name of the host Aten is running on (although what you would then usefully do with it I don't know).

Better examples can be found in the resources section of the website.

getEnvF

Syntax:

```
double getEnvF ( string variable )
```

Retrieves the contents of the named environment variable, converting it to a floating-point (double) value in the process.

For example:

```
double d = getEnvF("num");
```

gets the shell variable *num* as a real number.

getEnvI

Syntax:

```
int getEnvI ( string variable )
```

Retrieves the contents of the named environment variable, converting it to an integer value in the process.

For example:

```
int i = getEnvI("count");
```

gets the shell variable *count* as an integer number.

gui

Syntax:

```
void gui ( )
```

Starts the GUI (e.g. from a script), if it isn't already running.

For example:

```
gui();
```

help

Syntax:

```
help command
```

Provide short help on the supplied *command*.

For example:

```
help cellaxes;
```

null

Syntax:

```
void null ( variable var, ... )
```

The null command accepts one or more pointer variables whose values are to be set to NULL (0).

searchCommands

Syntax:

```
void searchCommands ( string search )
```

Search all available commands for the (partial) command name specified.

seed

Syntax:

```
void seed ( int i )
```

Sets the random seed.

For example:

```
seed(3242638);
```

quit

Syntax:

```
void quit ( )
```

Quits out of the program.

For example:

```
quit();
```

9.31. Trajectory Commands

Open and associate trajectory files to models, and select frames to display/edit from the current trajectory.

addFrame

Syntax:

```
Model addFrame ( )
```

```
Model addFrame ( string title )
```

Append a new trajectory frame to the current model's trajectory. The reference to the new frame is returned.

For example:

```
Model m = addFrame("new config");
```

clearTrajectory

Syntax:

```
void clearTrajectory ( )
```

Clear any associated trajectory and frame data in the current model.

For example:

```
clearTrajectory();
```

firstFrame

Syntax:

```
void firstFrame ( )
```

Select the first frame from trajectory of current model.

For example:

```
firstFrame();
```

lastFrame

Syntax:

```
void lastFrame ( )
```

Select last frame in trajectory of current model.

For example:

```
lastFrame();
```

loadTrajectory

Syntax:

```
int loadTrajectory ( string filename )
```

Associate trajectory in *filename* with the current model. An integer value of '1' is returned if the association was successful, or '0' otherwise.

For example:

```
int success = loadTrajectory("/home/foo/md/water.HISf");
```

opens and associated the formatted DL_POLY trajectory file “water.HISf” with the current model.

nextFrame

Syntax:

```
void nextFrame ( )
```

Select next frame from trajectory of current model.

For example:

```
nextFrame();
```

prevFrame

Syntax:

```
void prevFrame ( )
```

Select the previous frame from the trajectory of the current model.

For example:

```
prevFrame();
```

seekFrame

Syntax:

```
void seekFrame ( int frameno )
```

Seeks to the frame number specified.

For example:

```
seekFrame(10);
```

seeks to the 10th frame of the current trajectory.

9.32. Transform Commands

Commands to transform the current selection of the model.

axisRotate

Syntax:

```
void axisRotate ( Atom|int i, Atom|int i, double theta )  
  
void axisRotate ( Atom|int i, Atom|int i, double theta, double ox,  
double oy, double oz )  
  
void axisRotate ( double x, double y, double z, double theta )  
  
void axisRotate ( double x, double y, double z, double theta, double  
ox, double oy, double oz )
```

Rotate the current selection by an angle *theta* (in degrees) about an axis defined either by the vector between two atom IDs or the vector components provided. If supplied, the rotation is performed using the coordinate origin specified by *ox*, *oy*, and *oz*, otherwise {0,0,0} is assumed.

For example:

```
axisRotate(4, 5, 90.0);
```

rotates the current selection 90 degrees about the axis formed by the vector between atom ids 4 and 5 (4→5).

```
axisRotate(0, 1, 0, -52.0);
```

rotates the current selection -52 degrees about the y-axis.

```
axisRotate(0, 1, 0, -52.0, 4.0, 4.0, 4.0);
```

rotates the current selection -52 degrees about the y-axis, but with the rotation centre at {4.0,4.0,4.0}.

centre

Syntax:

```
void centre ( double x, double y, double z, bool lockx = FALSE, bool  
locky = FALSE, bool lockz = FALSE )
```

Centre the current atom selection at the specified coordinates. The three optional arguments *lockx*, *locky*, and *lockz* specify one or more atomic coordinates that are to remain unchanged during the transformation.

For example:

```
centre(0.0, 0.0, 15.0);
```

centres the current selection at the coordinates (0 0 15).

flipX

Syntax:

```
void flipX ( )
```

Flip (negate) the x-coordinates of the current selection.

For example:

```
flipX();
```

flipY

Syntax:

```
void flipY ( )
```

Flip (negate) the y-coordinates of the current selection.

For example:

```
flipY();
```

flipZ

Syntax:

```
void flipZ ( )
```

Flip (negate) the z-coordinates of the current selection.

For example:

```
flipZ();
```

matrixConvert

Syntax:

```
void matrixConvert ( int i_sx, int j_sx, int i_sy, int j_sy, int i_sz,
int j_sz, int i_tx, int j_tx, int i_ty, int j_ty, int i_tz, int j_tz )

void matrixConvert ( int i_sx, int j_sx, int i_sy, int j_sy, int i_sz,
int j_sz, int i_tx, int j_tx, int i_ty, int j_ty, int i_tz, int j_tz,
double oy, double oz, double oz )

void matrixConvert ( double s_ax, double s_ay, double s_az, double
s_bx, double s_by, double s_bz, double s(cx, double s_cy, double s_cz,
double t_ax, double t_ay, double t_az, double t_bx, double t_by, double
t_bz, double t(cx, double t_cy, double t_cz, double oy, double oz,
double oz )
```

From a defined frame of reference (i.e. a set of axes defining the spatial orientation), rotate the current selection from this frame into the second frame, using the coordinate origin supplied or {0,0,0} by default. In the first form six pairs of atom IDs define each matrix, with the vectors taken to be $i \rightarrow j$ in all cases (normalised to 1.0), and specifying the x, y, and z axes in turn. In the second, the matrices are given as two sets of nine numbers that define the vectors of the axes.

When supplying atom IDs, the x axis is taken to be absolute, the y-axis is orthogonalised w.r.t. the x-axis, and the z-axis is taken as the cross product between the x and y axes. Note that providing the definition of the z axis is still important, however, since the vector cross product is adjusted (if necessary) to point along the same direction as this supplied z-axis. When supplying the complete matrices no orthogonalisation or normalisation of the axes is performed (permitting arbitrary scale and shear operations).

For example:

```
matrixConvert(1, 2, 1, 3, 1, 4, 10, 11, 12, 13, 14, 15);
```

defines a the current selection's frame of reference as (in terms of atom IDs) $X = (1 \rightarrow 2)$, $Y = (1 \rightarrow 3)$, and $Z = (1 \rightarrow 4)$, which will be rotated such that it corresponds to the new frame of reference (again defined by atom IDs) $X = (10 \rightarrow 11)$, $Y = (12 \rightarrow 13)$, and $Z = (14 \rightarrow 15)$.

```
matrixConvert(-0.7348, -0.0192, -0.678, 0.4979, 0.6635, -0.5584, 0.4606, -0.7479,
-0.47801, 1, 0, 0, 0, 1, 0, 0, 0, 1)
```

defines a the current selection's frame of reference as the vectors $X = \{-0.7348, -0.0192, -0.678\}$, $Y = \{0.4979, 0.6635, -0.5584\}$, and $Z = \{0.4606, -0.7479, -0.47801\}$, which will be rotated into the standard reference frame.

matrixTransform

Syntax:

```
void matrixTransform ( double ax, double ay, double az, double bx,  
double by, double bz, double cx, double cy, double cz )
```

```
void matrixTransform ( double ax, double ay, double az, double bx,  
double by, double bz, double cx, double cy, double cz, double ox,  
double oy, double oz )
```

Transform the current selection by applying the defined matrix to each coordinate, operating about the supplied origin (or {0,0,0} by default). No orthogonalisation or normalisation of the defined axes is performed.

For example:

```
matrixTransform(1, 0, 0, 0, 1, 0, 0, 0, 1);
```

does absolutely nothing (multiplying by the identity matrix).

```
matrixTransform(1, 0, 0, 0, 1, 0, 0, 0, -1);
```

mirrors the current selection in the xy plane.

```
matrixTransform(0.5, 0, 0, 0, 0.5, 0, 0, 0, -1);
```

scales the x and y-coordinates of all selected atoms by 0.5, leaving the z coordinates intact.

reorient

Syntax:

```
void reorient ( Atom|int i_sx, Atom|int j_sx, Atom|int i_sy, Atom|int  
j_sy, Atom|int i_sz, Atom|int j_sz, double t_ax, double t_ay, double  
t_az, double t_bx, double t_by, double t_bz, double t(cx, double t_cy,  
double t_cz )
```

```
void reorient ( Atom|int i_sx, Atom|int j_sx, Atom|int i_sy, Atom|int  
j_sy, Atom|int i_sz, Atom|int j_sz, double t_ax, double t_ay, double  
t_az, double t_bx, double t_by, double t_bz, double t(cx, double t_cy,  
double t_cz, double ox, double oy, double oz )
```

Operates in exactly the same manner as matrixtransform except that the source matrix is defined from atoms (or their IDs) and the destination matrix is provided as a matrix.

setAngle

Syntax:

```
void setAngle ( Atom|int i, Atom|int j, Atom|int k, double angle )
```

Adjusts the angle made between atoms $i-j-k$ so that it becomes the target `value`, moving the atom k and all its direct and indirectly-bound neighbours (except i and j). The coordinates of atom i and j remain unaffected.

This operation can only be performed when atoms j and k are not present in the same cyclic structure. The atoms i , j , and k do not have to be bound, so it is possible to move separate fragments relative to each other using this method.

For example:

```
setAngle(10,11,12,109.5);
```

sets the angle made between atoms 10, 11, and 12 to be 109.5°.

setDistance

Syntax:

```
void setDistance ( Atom|int i, Atom|int j, double dist )
```

Shifts the atom j and all its direct and indirectly-bound neighbours (except i) so that the distance between i and j is $dist$. The coordinates of atom i remain unaffected.

This operation can only be performed for atoms which are not present in the same cyclic structure, e.g. trying to set the distance of two atoms in a benzene ring is not allowed. However, note that the two atoms i and j do not have to be bound, so it is possible to move separate fragments further apart by this method.

For example:

```
setDistance(1,4,4.9);
```

sets the distance between atoms 1 and 4 to be 4.9 Å.

setTorsion

Syntax:

```
void setTorsion ( Atom|int i, Atom|int j, Atom|int k, Atom|int l,  
double angle )
```

Adjusts the atoms i and l (and all their attached neighbours) to give the torsion angle specified. The coordinates of atoms j and k remain unaffected.

This operation can only be performed for atoms which are not present in the same cyclic structure, e.g. trying to set the distance of two atoms in a benzene ring is not allowed.

translate

Syntax:

```
void translate ( double dx, double dy, double dz )
```

Translates the current selection by the specified vector.

For example:

```
translate(1, 0, 0);
```

moves the current selection 1 Å along the x axis.

translateAtom

Syntax:

```
void translateAtom ( double dx, double dy, double dz )
```

Translates the current atom by the specified vector.

For example:

```
translateAtom(1, 0, -1);
```

translates the current atom 1 Å along x and -1 Å along z.

translateCell

Syntax:

```
void translateCell ( double fracX, double fracY, double fracZ )
```

Translates the current selection by the fractional cell vectors specified. The model must have a unit cell for this command to work.

For example:

```
translateCell(0, 0.5, 0);
```

translates the current selection by an amount equivalent to half of the current cell's B vector.

translateWorld

Syntax:

```
void translateWorld ( double dx, double dy, double dz )
```

Translates the current selection by the Å amounts specified, with the XY plane parallel to the monitor screen.

For example:

```
translateWorld(0, 0, 10);
```

translates the current selection 10 Å ‘into’ the screen.

mirror

Syntax:

```
void mirror ( string axis )
```

Mirror the current selection in the specified plane about its geometric centre.

For example:

```
mirror("y");
```

mirrors the current selection about the y axis.

9.33. View Commands

Commands to change the current model's view.

axisRotateView

Syntax:

```
void axisRotateView ( double ax, double ay, double az, double angle )
```

Rotate the current view *angle* degrees about an axis defined between the supplied point {ax,ay,az} and the origin.

getView

Syntax:

```
void getView ( )
```

Outputs the rotation matrix elements and position vector of the camera for the current model. The list of numbers may be passed directly to the **setView** command to re-create the view exactly.

For example:

```
getView();
```

orthographic

Syntax:

```
void orthographic ( )
```

Set the view for all models to be an orthographic projection.

For example:

```
orthographic();
```

perspective

Syntax:

```
void perspective ( )
```

Set the view for all models to be a perspective projection.

For example:

```
perspective();
```

resetView

Syntax:

```
void resetView ( )
```

Resets the view rotation and zoom for the current model.

For example:

```
resetView();
```

rotateView

Syntax:

```
void rotateView ( double rotx, double roty )
```

Rotates the current view by *rotx* degrees around the x axis and *roty* degrees around the y axis.

For example:

```
rotateView(10.0, 0.0);
```

setView

Syntax:

```
void setView ( double ax, double ay, double az, double bx, double by,  
double bz, double , double cy, double cz, double x, double y, double  
z )
```

Sets the rotation matrix and position vector of the camera for the current model. The output of **getView** can be passed to **setView** to recreate an existing camera rotation and position.

For example:

```
setView(1, 0, 0, 0, 1, 0, -1, 0, 0.0, 0.0, -10.0);
```

sets a view with the z axis pointing up, and the y axis normal to the screen (i.e. rotated 90 degrees around the x axis)

speedTest

Syntax:

```
void speedTest ( int nRenders = 100 )
```

Performs a quick speed test based on rendering of the current model and general view.

For example:

```
speedTest();
```

spins the current model for the default of 100 rendering passes.

```
speedTest(2000);
```

spins the current model for 2000 rendering passes.

translateView

Syntax:

```
void translateView ( double x, double y, double z )
```

Translates the camera viewing the current model.

For example:

```
translateView(0.0, 0.0, 1.0);
```

viewAlong

Syntax:

```
void viewAlong ( double x, double y, double z )
```

Sets the current view so that it is along the specified vector.

For example:

```
viewAlong(0, 0, -1);
```

view the current model along the negative z-axis.

viewAlongCell

Syntax:

```
void viewAlongCell ( double x, double y, double z )
```

Sets the current view so that is along the specified cell vector.

For example:

```
viewAlongCell(1, 0, 0);
```

view the current model along the cell's x axis.

zoomView

Syntax:

```
void zoomView ( double dz )
```

Zooms the view by the specified amount.

For example:

```
zoomView(10.0);
```

moves the camera 10 Å forwards along the z-direction.

```
zoomView(-5);
```

moves the camera 5 Å backwards along the z-direction.

10. Topics of Interest

10.1. Colourscales

TODO

10.2. Glyphs

A ‘glyph’ in Aten is any of a series of primitives (or shapes, or objects) that can be rendered in addition to the components that make up a standard model (i.e. atoms, bonds, and unit cell). These can be used to illustrate points of interest in a system, illustrate some vector quantity, or draw whole new objects from scratch. Glyphs can be drawn at fixed positions within a model, or can have their vertices linked to existing atoms, enabling them to be moved in conjunction with the model’s atoms. Some glyphs can also be rotated about their current position.

A glyph requires from one to four vertices to be set, depending on the type. A different colour may be assigned to each vertex enabling, for example, each corner of a triangle to possess a different colour. The available glyph types and the roles of the four possible coordinates are:

Table 10-1 Glyph Types

Glyph	Rot?	r1	r2	r3	r4
arrow	No	Tail coords	Head coords		
cube	Yes	Centroid	XYZ scaling factors		
ellipsoid					
ellipsoidxyz					
line	No	Start coords	End coords		
quad	No	Vertex 1	Vertex 2	Vertex 3	Vertex 4
sphere	Yes	Centroid	XYZ scaling factors		
svector	No				
tetrahedron	Yes	Vertex 1	Vertex 2	Vertex 3	Vertex 4
text	No	Mid-left anchor of text			
text3d	No	Mid-left anchor of text			
triangle	No	Vertex 1	Vertex 2	Vertex 3	
tubarrow	No	Tail coords	Head coords		
vector	No	Centroid	Pointing vector		

10.3. Patterns

Patterns in Aten represent collections of molecules of the same type, and are used primarily in the forcefield engine where they allow the construction of compact descriptions of systems where many similar molecules exist (e.g. liquids). A set of patterns describing the contents of a model is often referred to as a *pattern description*. Such a description is generated automatically as and when required by Aten, and so in most cases should not need to be defined manually by the user. When Aten fails to find a suitable pattern for a model this is often an indication that a bond is missing (perhaps the bond tolerance is too low?) or the atoms present in the system are not in exactly the order you think they are. When Aten fails to find a proper pattern definition for a system, some operations will not be performed (e.g. folding molecules back into the cell with **Model→Fold Molecules**). In these cases it is best to take a careful look at the ordering of atoms and their bonds within a system to try and fix the cause of the problem, but a *default pattern* can be enforced if absolutely necessary through the main menu's **Forcefield→Add Default Pattern** option. This description adds in a simple pattern that encompasses all atoms in the model, and therefore always succeeds, at the expense of inefficiency.

A collection of atoms can live quite happily on its own in Aten, and can be moved around, rotated, deleted and added to at will. However, if you want to calculate the energy or forces of a collection of atoms (or employ methods that use such quantities) then a description of the interactions between the atoms is required. Creating a suitable expression is the process of taking a system of atoms and generating a prescription for calculating the energy and/or forces arising from these interactions from any standard classical forcefield available.

Patterns describe systems in terms of their constituent molecular units. For an N-component system (a single, isolated molecule is a 1-component system) there are N unique molecule types which are represented as, ideally, a set of N patterns. Forcefield sub-expressions can then be created for each pattern and applied to each molecule within it, allowing the expression for the entire system to be compact and efficient. Each pattern contains a list of intramolecular terms (bonds, angles etc.), atom types, and van der Waals parameters for a single molecule that can then be used to calculate the energy and forces of M copies of that molecule.

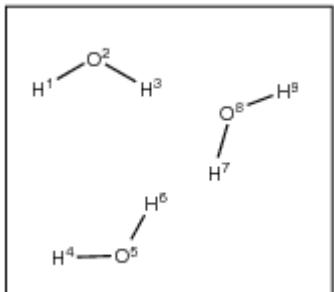
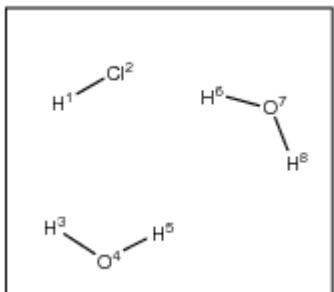
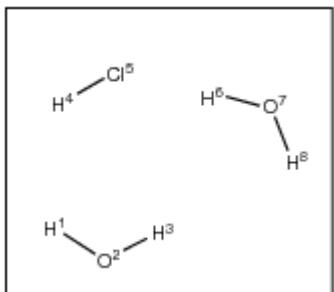
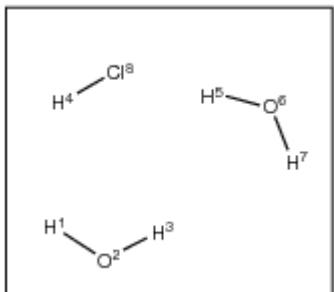
10.3.1. Determination of Patterns

From the atoms and the connectivity between them Aten will automatically determine patterns for systems of arbitrary complexity. The ordering of atoms is important, however, and the atoms belonging to a single molecule must not be interspersed by those from others. In other words, the atoms belonging to one instance of a molecule must be listed consecutively in the model. There are also many ways to represent the same system of atoms, all of which (from the point of view of the program) are equivalent and will produce the same total energies and atomic forces.

Consider the following examples:

Table 10-2 Pattern Examples

System	Atom Ordering	Automatic Pattern
--------	---------------	-------------------

1		H1 O2 H3 H4 O5 H6 H7 O8 H9	HOH (3)
2		H1 Cl2 H3 O4 H5 H6 O7 H8	HCl (1) HOH (2)
3		H1 O2 H3 H4 Cl5 H6 O7 H8	HOH (1) HCl (1) HOH (1)
4		H1 O2 H3 H4 H5 O6 H7 Cl8	HOH (1) H (1) HHO (1) Cl (1)

In 1) the three water molecules are identical with respect to the ordering of the atoms, so our description consists of a single pattern describing one HOH moiety. Obvious, huh? The two-component system illustrated in 2) has all molecules of the same type one after the other, giving a simple two-term pattern. However, in 3) the two water molecules are separated (in terms of their order) by the HCl, and so a three-term pattern results. In 4) there are, from the point of view of the program, three distinct molecules, since the ordering of atoms in the two water molecules is different, and so three terms are again necessary in the pattern description.

There are likely to be many possible pattern descriptions for systems, some of which may be useful to employ, and some of which may not be. Take the well-ordered system 1) – four different ways to describe the system are:

HOH (3)

HOH (1) HOH (1) HOH (1)

HOH (2) HOH (1)

HOH (1) HOH (2)

All are equivalent and will give the same energies / forces. Sometimes it is useful to treat individual molecules as separate patterns in their own right since it allows for calculation of interaction energies with the rest of the molecules of the system.

10.3.2. Pattern Granularity

Patterns work on the basis of bound fragments/molecules, and a molecule cannot be split up smaller than this – for instance in the examples above water cannot be represented by ‘ H_2O (1)’ since this would ‘neglect’ a bond. However, there is nothing to stop a pattern ‘crossing’ individual molecules. Consider again the example 1) above. Three further (reiterating the point, equivalent) ways of writing the pattern description are:

HOHHOH (1) HOH (1)

HOH (1) HOHHOH (1)

HOHHOHHOH (1)

Here, we have encompassed individual molecular entities into supermolecular groups, and as long as there are no bonds ‘poking out’ of the pattern, this is perfectly acceptable. Although this coarse-graining is a rather counter-intuitive way of forming patterns, it nevertheless allows them to be created for awkward systems such as that in 4) above. We may write the following valid patterns for this arrangement of atoms:

HOHHHOHCl (1)

HOH (1) HHOHCl (1)

HOHHHO (1) HCl (1)

Note that, when automatically creating patterns, if Aten stumbles across a situation that confuses it, the default pattern of one supermolecule will be assumed, i.e. X(1) where ‘X’ is all atoms in the model. This will work fine in subsequent energy calculations etc., but will result in rather inefficient performance.

10.4. Partitioning Schemes

TODO

11. Filters

Filters determine the model/trajectory, grid/surface, and forcefield expression formats that Aten can read and write. They are essentially small programs written in Aten's internal command language (based syntactically on C/C++) and are stored as plain text files. These files are parsed and 'compiled' when Aten starts up. This has several advantages:

- Users may add support for their own particular formats at will.
- No recompilation of Aten is necessary when adding new filters or adjusting old ones
- Potentially any file format can be supported, even binary formats

With this flexibility, of course, come some modest disadvantages:

- Speed - the C-style code contained within filters is, strictly speaking, interpreted, it is by no means as fast as properly compiled code
- File formats that need particularly awkward operations requiring a more 'complete' C language may be difficult to implement

These two points aside, though, filters make Aten a powerful and flexible tool, adaptable to conform to many different program/code input and output formats.

As mentioned, the programming language used by filters is essentially a subset of C, implemented with the same syntax and style (see the command language overview in Chapter 9 for a description), but includes several hundred custom commands to control Aten in order to build up atoms in models, access data etc. So, if you already know C or C++, writing a filter should be a breeze. If you don't, it's not too difficult to pick up, and there are plenty of filters already written to use as worked examples.

When a filter is called in order to write out data, no references to any of the current (i.e. displayed or selected) data are sent directly to the filter itself. Instead, this must be probed by using the `Aten` master reference available to all scripts, commands and filters. Within `Aten` the currently displayed model may be deduced, as well as the current frame (if a trajectory is associated). In most cases for model export filters, the path '`aten.frame`' should be used to determine the model data that should be written.

11.1.1. Filter Contents

A filter is a plain text file containing one or more C-style programs that permit the input or output of data in a specific format. For example, a purely model-oriented filter file may contain two filters, one to read in files of the specified format, and one to write the data out again. Each individual filter is given a short nickname, a shell-style glob, and possibly several other bits of data that allow files to be recognised (if the file extensions defined for it are not enough).

Different filters that recognise the same file type may be provided if necessary, each performing a slightly different set of import or export commands (if it is not convenient to do so within a single filter), and all will appear in the drop-down list of filters in file dialogs within the program. Note that in batch, command-line, or scripting mode, filters are either selected automatically based on the filename, extension, or contents, or picked by matching

only the associated nickname. In the former case, the first filter that matches the extension is used.

11.1.2. Filter Locations

A basic stock of filters is provided with Aten and installed with the program - several default locations are searched for these filters on startup. Alternatively, if Aten fails to find these filters, or you wish to point it to a suitable directory by hand, either the `$ATENDATA` environment variable may be set to the relevant path (on Windows, this variable is set by the installer) or the `--atendata` command-line option may be used to provide the path.

Additional filters may be placed in a user's `.aten` directory, e.g. `~bob/.aten/filters/`.

11.1.3. Overriding Existing Filters

Filters that possess the same ID or nickname as other filters of the same type may be loaded simultaneously, with the last to be loaded taking preference over the other. Thus, an importmodel filter nicknamed `xyz` from Aten's installed filter stock will be overridden by one of the same nickname present in a user's `.aten/filters` (or `aten/filters` on Windows) directory. Similarly, both these filters will be overridden by one of the same nickname loaded by hand from the command line (with the `--filter` switch). Note that this only holds true for filters referenced by nickname or determined automatically by Aten when loading data - from the GUI all filters are available in the file dialogs.

11.1.4. Filter Definitions

Filter definitions are made in a filter file in a similar way to declaring a user subroutine or function (see Section 8.1.7). The `filter` keyword marks the start of a filter definition, and contains a list of properties in parentheses that define the subsequent filter, its name, and how to recognise the files (from their filenames and/or contents) that it is designed for. The definition of the filter to import XYZ-style model data is as follows:

```
filter(type="importmodel", name="XMol XYZ Coordinates", nickname="xyz",
extension="xyz", glob="*.xyz", id=3)
{
    commands
    ...
}
```

The comma-separated list of properties defines the type of filter (`'type="importmodel"`) and how to recognise files of that type (e.g., `'extension="xyz"`), amongst other things.

The full list of possible properties is as follows:

Table 11-1 Filter Definition Keyword Summary

Property	Description
exact	Comma-separated list of filenames that are of this type
extension	Comma-separated list of filename extensions that indicate files of this type
glob	Shell-style glob to use in file dialogs in order to filter out files of the described type
id	Numerical ID of the filter to enable partnering of import/export filters for files of the same type
name	Descriptive name for the filter, shown in file dialogs etc.
nickname	Short name used by commands in order to identify specific filters
search	Provides a string to search for in the file. If the string is found, the file is identified as being readable by this filter type. The number of lines searched is governed by the within property
type	Defines the kind of filter that is described (i.e. if it loads/saves, acts on models/grid data etc.) so that Aten knows when to use it. <i>Must always be defined!</i>
within	Specifies the number of lines to search for any supplied search strings
zmap	Determines which zmapping style to employ when converting atom names from the file

exact

Syntax:

```
exact="name1, name2, ..."
```

Occasionally (and annoyingly) files have no extension at all, instead having short, fixed names, which must be checked for literally when probing files. This command defines one or more explicit filenames that identify files targeted by this filter. Multiple names may be given, separated by commas or whitespace. Exact filename matching is case-insensitive.

For example:

```
exact="coords"
```

associates any file called ‘coords’ to this filter.

```
exact="results, output"
```

associates any files called ‘results’ or ‘output’ to this filter.

extension

Syntax:

```
extension="extension1,extension2..."
```

Sets the filename extension(s) that identify files to be read / written by this filter. When files are being probed for their type, in the first instance the filename is examined and the extension (everything after the last '.') is compared to those defined in filter sections by this command. Multiple file extensions may be given, separated by commas or whitespace. File extension matching is case-insensitive.

For example:

```
extension="xyz"
```

means that files with extension ‘.xyz’ will be recognised by this filter.

```
extension="xyz,abc,foo"
```

means that files with extensions ‘.xyz’, ‘.abc’, and ‘.foo’ will be recognised by this filter.

glob

Syntax:

```
glob="*|*.ext"
```

Sets the file dialog filter extension to use in the GUI, and should be provided as a shell-style glob.

For example:

```
glob="*.doc"
```

filters any file matching ‘*.doc’ in the relevant GUI file selector dialogs.

id

Syntax:

```
id=n
```

When separate import and export filters for a given file type have been provided it is prudent to associate the pair together so that Aten knows how to save the data that has just been loaded in. Each filter has a user-definable integer ID associated with it that can be used to link import and export filters together. For example, if a model import filter has an ID of 7, and a model export filter also has this ID, then it will be assumed that the two are linked, and that a model saved with export filter 7 can be subsequently loaded with import filter 7. If the ID for a filter is not set it defaults to -1, and it is assumed that no partner exists and the file cannot be directly saved back into this format.

For example:

```
id=13
```

See the list of supported formats in Table 1-1 to Table 1-4 to find which ids are currently in use.

name

Syntax:

```
name="long name of filter"
```

Sets the long name of the filter, to be used as the filetype description of files identified by the filter. This name will appear in the file type lists of file dialogs in the GUI, and also in the program output when reading / writing files of the type.

For example:

```
name="SuperHartree Coordinates File"
```

nickname

Syntax:

```
nickname="short name of filter"
```

Sets a nickname for the filter, which allows it to be identified easily in the command language and, importantly, from the command line. It should be a short name or mnemonic that easily identifies the filter. No checking is made to see if a filter using the supplied nickname already exists.

For example:

```
nickname="shart"
```

sets the nickname of the filter to ‘shart’.

```
nickname="zyx"
```

sets the nickname of the filter to ‘zyx’.

search

Syntax:

```
search="string to search"
```

Occasionally, checking the contents of the file is the easiest way to determining its type, and is probably of most use for the output of codes where the choice of filename for the results is entirely user-defined. For example, most codes print out a whole load of blurb and references at the very beginning, and usually searching for the program name within this region is enough to identify it. For files that are only easily identifiable from their contents and not their filename, plain text searches within files can be made to attempt to identify them. Individual strings can be given to the `search` keyword, and may be specified multiple times. The default is to search the first 10 lines of the file for one or more of the defined search strings, but this can be changed with the `within` property.

For example:

```
search="UberCode Version 20.0"
```

matches the filter to any file containing the string ‘UberCode Version 20.0’ within its first 10 lines (the default).

```
search="SIESTA"
```

searches the first 10 lines of the file for the string ‘SIESTA’.

```
search=""GAMESS VERSION = 11 APR 2008 (R1)"
```

attempts to identify output from a specific version of GAMESS-US.

type

Syntax:

```
type="filtertype"
```

The 'type' keyword must be provided an all filter definitions - an error will be raised if it is not. It specifies which class of data the filter targets (e.g. models, grid data etc.) and whether it is an import or export filter. A given filter may only have one `type` specified, for which the possible values are:

Table 11-2 Filter Types

Type	Description
<code>exportexpression</code>	Describes how to export forcefield descriptions (expressions) for models
<code>exportgrid</code>	Describes how to export grid-style data
<code>exportmodel</code>	Describes how to write out model data
<code>exporttrajectory</code>	Filter suitable for the export of trajectory data
<code>importexpression</code>	Describes how to load in forcefield-style expressions
<code>importgrid</code>	Describes how to read gridded volumetric or surface data from files. Any grids created in these sections must have the <code>finaliseGrid</code> command called on them, otherwise they will not be registered

<code>importmodel</code>	properly within the program. Describes how to import model data, including atoms, cell and spacegroup data, bonds, glyphs etc. Any models created in ‘importmodel’ filters must have the <code>finaliseModel</code> command called on them, otherwise they will not be registered properly within the program.
<code>importtrajectory</code>	Read frames from trajectory files. See the section on trajectories (Section 11.2) for additional information on how trajectories are handled within Aten.

For example:

```
type="importgrid"
type="exportmodel"
```

within

Syntax:

`within=n`

Defines the maximum number of lines at the beginning of the file that will be `searched` for string definitions (default is 10).

For example:

```
within=50
```

specifies that the first 50 lines should be searched for identifying strings.

zmap

Syntax:

`zmap="zmaptype"`

By default, it is assumed that the commands which create new atoms will be given a proper element symbol from which to determine the atomic number. Case is unimportant, so `na`, `Na`, and `NA` will all be interpreted as atomic number 11 (sodium). Where element symbols are not used in the model file, there are several alternative options that tell these commands how to convert the element data they are passed into atomic numbers. For example, the `ff` style is particularly useful when loading in coordinate files which contain forcefield atom type names that do not always correspond trivially to element information (e.g. `DL_POLY` configurations).

For example:

```
zmap="numeric"
```

indicates that atomic numbers are provided in place of element names and no conversion should be performed. See Table 16-17 for a list of available z-mapping methods.

11.1.5. Filter Options

When writing data, in many cases all the information that the filter wants to write is contained within the current model, for example when outputting simple file formats such as xyz or Aten's own akf format. In other cases there may be additional data for which it would be nice to have some control over, and which lies beyond atoms and bonds. The best example is probably the input formats for nearly all *ab initio* codes which contain (as well as the atomic coordinates) statements and additional data necessary to control the running of the code itself. It is not a problem to write out static lines of control commands from the output filter, but it would of course also be nice to be able to tailor this output from within the GUI (or from the command-line). This can be achieved by assigning values to variables in the filter through the use of Custom Dialogs (see Section 8.3).

11.2. Trajectory Files

Trajectory files usually mean one of two things – either a sequence of frames from a molecular dynamics simulation trajectory, or a sequence of configurations from a geometry optimisation of some kind. Either way, both boil down to the same thing from Aten's perspective, that is a set of models in a sequence. In terms of displaying such a set of models, either they may be loaded as individual models (i.e. having a separate tab in the GUI) or the sequence of models may be associated to a single 'parent' model. Most commonly, the latter is the preferred method (especially when large numbers of models are present in the trajectory).

There are two related ways to get this data into Aten. From the perspective of molecular dynamics simulations, the parent model or configuration and the trajectory frames are stored in separate files. In this case, the model can be loaded first, and then the trajectory file attached or associated to this model afterwards. From the perspective of geometry optimisations, for example, the parent configuration (i.e. the starting point of the optimisation) and the sequence of coordinates are most often stored in the same output file. In this case, the importmodel filter can detect the presence of the additional trajectory frames and manually attach them to the parent model. The following sections explain the details of how both methods work.

11.2.1. Trajectories in Separate Files

DL_POLY, being a molecular dynamics code, stores its configuration and trajectory data in separate files. Filters are supplied with Aten that read in DL_POLY trajectories, and so this example will revolve around those filters.

Necessity for a Master Configuration?

Thus far, it has been implicitly stated that the 'master' configuration and the trajectory files come necessarily as a pair - the master configuration is read in, and then the trajectory associated to it. This implies that the trajectory file is somehow tied to the master configuration - perhaps the trajectory file does not contain information such as the number of atoms or element data, and so a 'reference' configuration is necessary? Of course, this may not always be the case, and it is possible that some trajectory formats will store all the necessary information needed in order to fully generate the trajectory configurations. DL_POLY trajectories do, in fact, contain all the necessary data, but even so the master configuration is still used as a template for the trajectory data, and is used to check that the correct number of atoms are present etc. It comes down to a matter of preference as to whether the master configuration should be demanded, or whether the trajectory can itself be associated to any (even an empty) model. Remember, `importtrajectory` filters always attach the trajectory data to an existing model.

ImportTrajectory Filters

An `importtrajectory` filter is written in a slightly different way to other filter types. Since trajectory files may contain header data which is written once at the beginning of the file, preceding the frame data, there are potentially two separate sets of data to read from trajectory files - header data and (individual) frame data. So, rather than putting the code to read this data directly in the main body of the filter, two functions should be defined instead,

one for reading the header (which must be called `readHeader`), and one for reading an individual frame (which must be called `readFrame`). Note that, if a given trajectory format does not contain a header, the corresponding function may be left empty, but must still be defined and should return a value of ‘1’. Both functions take no arguments, and must return an integer. A template for an `importtrajectory` filter is thus:

```
filter(type="importtrajectory", name="Example Filter Template")
{
    int readHeader()
    {
        // Code to read header data goes here
    }

    int readFrame()
    {
        // Code to read frame data goes here
    }
}
```

The functions must take responsibility for informing Aten when the desired data cannot be read. Both should return a value of ‘1’ if the data was read successfully, and should return ‘0’ if an error is encountered.

Header Data

When opening a trajectory file with an `importtrajectory` filter, the first thing Aten does is attempt to read any header information from the file by calling the `readHeader` function defined in the filter. Since a trajectory file may not contain a header, and consists simply of individual frames back to back, in these situations the `readHeader` function defined in the filter should not read any data. The filter definition then becomes simply:

```
filter(type='importtrajectory', name="Example Filter Template")
{
    int readHeader()
    {
        // No header, so just return
        return 1;
    }

    int readFrame()
    {
        // Code to read frame data goes here
    }
}
```

Here, the `readHeader` function always succeeds, so Aten always thinks it has successfully read a header.

Frame Data

If `readHeader` is successful, Aten proceeds to read the first frame (by calling the `readFrame` function) in order to get an idea of the size of an individual frame, and hence the total number of frames in the trajectory. Of course, this assumes that all frames take up the same number of bytes in the file, and may not always be the case, especially for plain-text trajectory files. Thus, the frame estimate output by Aten should not necessarily be taken as gospel.

Unless an error is encountered when reading the test frame (i.e. `readFrame` returns '0' or `FALSE`) the trajectory file is then rewound to the end of the header section (start of the frame data). One of two things then happens. Since trajectory files are typically enormous (hundreds or thousands of megabytes) then it is unwise to try and load the whole trajectory into memory at once. Aten knows this, and from the estimated frame size also knows roughly how big the whole trajectory is. If the total trajectory file size is greater than an internally-defined limit (the "trajectory cache size") then only a single frame is stored at any one point. If the total size is smaller then this limit, the whole trajectory is cached in memory. Both have their advantages and disadvantages, as listed in the following sections.

Uncached Frames

If the trajectory is too big to be stored in memory, Aten only holds a single frame in memory at any one time. This means that:

- Memory use is minimised since only a single frame is loaded at any time
- Performance is slower – moving between frames means data must be read from disk
- Edits are forgotten – changes (both atomic and stylistic) made to the loaded frame are forgotten when a different frame is read

Aten tries to minimise the seek time between frames by storing file offsets of frames it has already read in. However, since trajectory frames can be different sizes Aten never tries to 'jump' ahead in the file based on the size of a single frame. Skipping immediately to the final frame in the trajectory will, thus, read all frames along the way and store file offsets for all frames. Then, seeking to any individual frame is a much quicker process.

Although style and editing changes are forgotten between frames, the overall camera view of the model is linked to that of the master configuration and so is retained between frames. If the trajectory cannot be cached and you require changes (edits or styles) to be made to each frame (e.g. for the purposes of making a movie of the trajectory) then a script is the way to go (load frame, apply edits, save image, etc.).

Cached Frames

If the trajectory is small enough to be stored in memory, Aten reads in all frames at once. This means that:

- Memory use is increased
- Performance is optimal - speed of moving between frames is fast because all frames are in memory
- Edits are retained - edits can be made to individual frames and will be remembered on moving to a different frame

The size of the cache can be adjusted either from the command line with the `--cachelimit` switch or by setting the `cachelimit` member of the `Prefs` variable within Aten. No check is made of the new cache limit with respect to the memory available on the machine on which Aten is running, so use with care.

In the current versions of Aten, the total trajectory size is determined from the size of the frame on disk, whereas it would be more appropriate to use the size of the frame in memory. This will change in a future release.

11.3. Reading and Writing

Formatted output in Aten is based largely on string formatting in C, so if you're familiar with C then this should be a breeze. If you're a Soldier of Fortran, then the principles are very similar. If you're familiar with neither, then now's the time to learn.

11.3.1. Formatted Output

Formatted output corresponds to output to either the screen or to files, and is used in the following commands:

Table 11-3 Formatted Output Commands

Command	Function
error	Write a message to the screen and immediately terminate execution of the current script / filter / command structure
printf	Write a message to the screen
verbose	Write a message to the screen, provided verbose output mode is on
writelinef	Write a formatted line to the current output file
writevarf	Write a formatted string to a variable (equivalent to the C 'sprintf' command)

Basic Strings

Formatting a string for output, as mentioned elsewhere on numerous occasions, works the same as for C. The C **printf** command (equivalent to the command of the same name in Aten) takes a character string as its first argument, and at its simplest, this is all that is required:

```
printf("Hello");
```

This prints ‘Hello’ to the screen (minus the quotes). Importantly, however, a newline character is *not* printed, meaning that the next thing you try and **printf** will appear on the same line. For instance:

```
printf("Hello");
printf("There.");
```

would output:

```
HelloThere.
```

The end of a character constant in the printf command does not implicitly mean ‘and this is the end of the line’ - you must indicate the end of the line yourself by placing ‘\n’ at the point where you wish the line to end. So:

```
printf("Hello\n");
printf("There.");
```

would output:

```
Hello
There.
```

Newlines (\n) are an example of [escaped characters](#) - the backslash ‘\’ indicates that the following character, in this case ‘n’, is not to be treated as a normal ‘n’, but instead will take on its alternative meaning, in this case a newline character. There are one or two other escaped characters recognised - see [Escaped Characters](#) for a list. Note that the newline token can appear anywhere in the string, and any number of times. So:

```
printf("Hello\nThere\n.");
```

would output:

```
Hello
There
.
```

11.3.2. Printing Data

Being able to print simple text strings is good, but not nearly enough. The first argument to the 'printf' command must always be a character string, but any number of additional arguments may be provided. Now, these additional arguments may be number constants, other character strings, variables, etc., and may be output in the resulting string by referencing them with ‘specifiers’ placed within the first example. One example of a specifier is %i which is shorthand for saying ‘an integer value’ – if used within the character string provided to printf, the command will expect an integer constant or variable to be provided as an additional argument. For example:

```
printf("This number is %i.\n", 10);
```

will print

```
This number is 10.
```

Similarly,

```
int value = 1234;
printf("Constant is %i, variable is %i.\n", 10, value);
```

will print

```
Constant is 10, variable is 1234.
```

There are other specifiers suitable for different types of data – see Section 11.3.4. The way data is presented by the specifier in the final output can also be controlled (e.g. for numerical arguments the number of decimal places, presence of exponentiation, etc., can be defined).

11.3.3. Formatted Input

Formatted input corresponds to input from either files or string variables, and is used in the following commands:

Table 11-4 Formatted input commands

Command	Function
<code>readLineF</code>	Read a formatted line from the current input file
<code>readVarF</code>	Read a formatted string from a variable

Note that the meaning of the formatting string changes slightly here - in essence, the type and formats of the specifiers are used to break up the supplied string into separate arguments, which are then placed in the provided corresponding variable arguments. When reading in string data, note that blank characters are significant and will be retained. To strip trailing blank characters (spaces and tabs) when reading a fixed-length string in a format, supply the length as a negative number.

11.3.4. Specifiers

The list of allowable variable specified corresponds more or less exactly to that found in C, with some small omissions and minor inclusions. For a full list see the reference page at cplusplus.com or cppreference.com. The list of printf features that are not (currently) supported in Aten are as follows:

- The pointer specifier `%p` is not supported. To print out reference addresses, use `%li`
- The single-character specifier `%c` is not supported
- Output of long doubles by prefixing a specifier with `L` (e.g. `%Le`) is not supported

11.3.5. Extra Specifiers Within Aten

As well as the mostly complete standard set of specifiers provided by C, Aten also includes some other useful specifiers that may be used in formatted input and output.

Table 11-5 Extra read/write specifiers

Specifier	Meaning
<code>%*</code>	Relevant to formatted input only. Discard the next item, regardless of its type. A

<code>%r</code>	corresponding variable argument need not be provided Read characters (starting from the next delimited argument) until the end of the input line is encountered (i.e. ‘rest-of-line’ specifier). A corresponding string variable should be provided
-----------------	--

11.3.6. Escaped Characters

Table 11-6 Escaped characters in format strings

Escape Sequence	Meaning
\n	Print newline (next character will appear on the next line)
\r	Carriage return
\t	Tab character

11.3.7. Delimited Reading and Writing

Formatting strings (or ‘format specifiers’) can be used to specify the layout of data items on a line when reading or writing data, but if the data are separated by whitespace characters such as spaces or tabs (or, alternatively, commas), such delimited data can be read in more easily. In such cases, it is not necessary to know beforehand the number of characters taken up by each item on the line, since the delimiters separate adjacent data items. A simplified method for reading and writing can be employed in these cases.

Commands providing delimited reading and writing are:

Table 11-7 Delimited read/write commands

Command	Function
<code>readLine</code>	Read delimited items from a source file, placing into the variables provided
<code>readNext</code>	Read the next delimited item from a source file, placing into the variable provided
<code>readVar</code>	Read delimited items from a source variable, placing into the variables provided
<code>writeLine</code>	Write the supplied items to a single line in the output file, separating them with whitespace
<code>writeVar</code>	Write the supplied items to a supplied string variable, separating them with whitespace

Note that all are called the same as their formatted counterparts, but minus the ‘f’ at the end of the name.

Delimited Data Example

Consider this example datafile:

Na	0.0	1.0	0.0
C1	1.0	0.0	0.0
Na	0.0	-1.0	0.0

Since the data items (element type and coordinates) are separated by whitespace, we need only provide the target variables to the relevant command - a formatting string, as is demanded by the **printf** command, is not required. Using the **readLine** command, the following code will parse this data correctly:

```
double x,y,z;
string el;
while (!eof()) { readline(el,x,y,z); newatom(el,x,y,z); }
```

The variables `el`, `x`, `y`, and `z` will, at any one time, contain the element type and coordinates from one line of the file. In an analogous manner, the data may be written out again with the corresponding **writeln** command:

```
for (atom i = aten.model.atoms; i; ++i) writeln(i.symbol,i.rx,i.ry,i.rz);
```

Each line will have the individual data items separated by a single space.

The **readNext** command reads in a single delimited item from a source file, preserving the remainder of the input line for subsequent operations. If there is no data left on the current line, a new line is read and the first delimited item is returned. The example above might be written in a slightly clunkier form as:

```
double x,y,z;
string el;
while (!eof())
{
    readnext(el);
    readnext(x);
    readnext(y);
    readnext(z);
    newatom(el,x,y,z);
}
```

For all delimited reading operations, items of data read from the line are converted automatically into the type of the destination variable. So, the atom coordinates read in above, which are put into double-type variables, could equally well be put into string variables. Standard C routines are used to convert data items in this way, and only some conversions make sense. For instance, attempting to read an item which is a proper character string (such as element symbol/name data) into a double or integer variable does not make sense. No error message will be raised, and the variables will likely be set to a value of zero (or whatever passes for 'zero' in the context of the type).

For all delimited writing operations, a suitable standard format specifier is chosen with which to write out the data.

11.3.8. Unformatted Reading and Writing

TODO

12. Forcefields and Typing

12.1. Overview

If you're doing anything interesting with a model or a molecule, a suitable forcefield description of the system is a must. A forcefield contains lists of parameters that describe the interactions between atoms, for example bonds, angles, and van der Waals interactions. More specifically, a forcefield contains parameters to describe many such interactions in many different types of molecule or chemical environment. An 'expression', referred to throughout the manual, should be thought of as the subset of terms from a given forcefield necessary to describe all the interactions within a model.

Aten has its own free format for forcefield files, described in the following sections. Once loaded in, the energy and forces in Models can then be calculated, and allows energy minimisation etc. More so, once a set of forcefield parameters has been read in and used to describe a model, this expression can be written out using a custom format ready for input into something else.

12.1.1. File Format

The basic forcefield file format is designed to be as readable as possible by both machine and user. Lines are free format, meaning that any number of tabs, commas, and spaces may separate items. Text items should be enclosed with either double or single quotes if they themselves contain these delimiters (in particular, this applies to NETA descriptions). There is no terminating symbol at the end of a line, c.f. the command language where a ';' typically ends every command.

The majority of data is contained within blocks in the file. Blocks begin with a specific keyword (e.g. `inter`), contain one or more lines of definitions, and are terminated by an 'end' keyword. Forcefield files may contain many blocks of the same type, permitting terms of the same type but with differing functional forms to be defined easily.

More advanced forcefields may contain [generator sections and functions](#) that enable them to either generate all their parameters on the fly from a basic set of data, or fill in missing terms that are not defined in any blocks.

12.1.2. Example - SPC Water

The format is keyword-based and as simple as possible. Most input is enclosed within blocks, beginning with a keyword and terminated after several lines of data with an 'end' statement. As an example of the overall structure of a forcefield, consider the simple point charge (SPC) forcefield for water as is provided with Aten:

```
name "SPC Water"
units kj

types
1      HW      H      "nbonds=1"
2      OW      O      "-H,-H"
```

```

end

inter lj
1      HW      0.41    0.0     0.0
2      OW     -0.82   0.650   3.166
end

bonds constraint
HW      OW      4184.0  1.000
end

angles bondconstraint
HW      OW      HW      4184.0  1.62398
end

```

After giving the forcefield a name and defining the energy units used for the parameters, the two types of atom described by the forcefield (`OW` and `HW`) are listed. A unique id, type name, base element, and type description are provided, providing Aten with all it needs to be able to recognise and type these atoms within a model. For each of these types the van der Waals data are provided in the Lennard-Jones style (`inter lj`) – again, the type id and type name are specified, followed by the atomic charge and the epsilon and sigma values. Note here that there are default combination rules set for each functional form - see [VDW functional forms](#) for a list. Finally, the single bond and angle within the molecule are defined. The type names involved in the interactions are given, followed by the necessary parameters for the functional form specified ('constraint' for the bond, and 'bondconstraint' for the angle). And that's it. Detailed explanations of each section follow. A more complete test forcefield supplied with Aten can be found in `data/ff/test.ff`.

12.2. Supplied Forcefields

A handful of forcefields ready-formatted for import into Aten are provided with the code and are listed here. It should be a relatively straightforward process to convert others, unless the functional forms used are not yet implemented (but sure, if you ask then I will add them). If you export an expression from Aten, *please* check the parameters in the file are what you actually want. Aten is designed to ease the pain of setting up a simulation in this manner, but is *not* intended as a black box.

12.2.1. Canongia-Lopes & Padua Ionic Liquids (cldp-il.ff)

All-atom ionic liquids forcefield of Canongia Lopes *et al.* covering various cation/anion combinations.

References

- J. N. Canongia Lopes, A. A. H. Padua, *J. Phys. Chem. B*, 110 (39), 19586-19592 (2006)
- J. N. Canongia Lopes, A. A. H. Padua, *J. Phys. Chem. B*, 108 (43), 16893-16898 (2004)
- J. N. Canongia Lopes, J. Deschamps, A. A. H. Padua, *J. Phys. Chem. B*, 108 (30), 11250 (2004)
- J. N. Canongia Lopes, J. Deschamps, A. A. H. Padua, *J. Phys. Chem. B*, 108 (6), 2038-2047 (2004)

12.2.2. Youngs, Kohanoff, & Del Pópolo [dmim]Cl (dmimcl-fm.ff)

Force-matched model for the ionic liquid dimethylimidazolium chloride only. Integer charges on ions.

References

- T. G. A. Youngs, J. Kohanoff, and M. G. Del Pópolo, *J. Phys. Chem. B*, 110 (11), 5697-5707 (2006)

12.2.3. Youngs & Hardacre [dmim]Cl (dmimcl-fm2.ff)

Second force-matched model for the ionic liquid dimethylimidazolium chloride only. Non-integer charges on ions.

References

- T. G. A. Youngs and C. Hardacre, *ChemPhysChem*, 9 (11), 1548-1558 (2008)

12.2.4. Jorgensen et al. OPLS-AA (oplsaa.ff)

Original OPLS-AA forcefield of Jorgensen et al. Thanks to W. Jorgensen for supplying the parameter data.

References

- W. L. Jorgensen, D. S. Maxwell, and J. Tirado-Rives, *J. Am. Chem. Soc.* 118, 11225-11236 (1996).
- W. L. Jorgensen and N. A. McDonald, *Theochem* 424, 145-155 (1998).
- W. L. Jorgensen and N. A. McDonald, *J. Phys. Chem. B* 102, 8049-8059 (1998).
- R. C. Rizzo and W. L. Jorgensen, *J. Am. Chem. Soc.* 121, 4827-4836 (1999).
- M. L. Price, D. Ostrovsky, and W. L. Jorgensen, *J. Comp. Chem.* 22 (13), 1340-1352 (2001).
- E. K. Watkins and W. L. Jorgensen, *J. Phys. Chem. A* 105, 4118-4125 (2001).

Note: NETA definitions have been written for a large number of types in the forcefield, but not all.

12.2.5. Berensen et al. Simple Point Charge Water (spc.ff)

Rigid, simple point charge model for water

References

- H. J. C. Berendsen, J. P. M. Postma, W. F. van Gunsteren and J. Hermans, in *Intermolecular Forces*, B. Pullman (ed.), Reidel, Dordrecht, 1981, p331.

12.2.6. Berensen et al. Extended Simple Point Charge Water (spce.ff)

Simple point charge model for water, modified to reproduce molecular dipole in the liquid phase.

References

- J. C. Berendsen, J. R. Grigera and T. P. Straatsma, *J. Phys. Chem.* 91, 6269-6271 (1987)

12.2.7. Rappe *et al.* Universal Forcefield (uff.ff)

Universal forcefield for the whole periodic table by Rappe et al.

References

- A. K. Rappe, C. J. Casewit, K. S. Colwell, W. A. Goddard III, and W. M. Skiff, *J. Am. Chem. Soc.* 114, 10024-10039 (1992)

Notes:

- Generated terms should be checked by hand if forcefield expressions are exported.
- Detection of some atomtypes, namely transition metals, is imperfect.
- Warning: Generation of terms (especially angles) needs to undergo proper testing! If you wish to help, please contact me.

12.2.8. Mayo, Olafson & Goddard II's Generic Forcefield (testing/dreiding.ff)

Universal forcefield for the whole periodic table.

References

- S.L. Mayo, B.D. Olafson, and W.A. Goddard III, *J. Phys. Chem.* 94, 8897-8909 (1990).

Notes:

- dreiding.ff currently lives in the testing/ directory since it is a rule-based forcefield and is currently being rewritten.
- Generated terms should be checked by hand if forcefield expressions are exported.
- Detection of some atomtypes, namely transition metals, is imperfect.
- Warning: Generation of terms (especially angles) needs to undergo proper testing! If you wish to help, please contact me.

12.2.9. General Amber Forcefield (testing/gaff.ff)

General Amber forcefield containing precalculated terms for most intramolecular terms, and generator section for any that are missing.

References

- o J. Wang, R. M. Wolf, J. W. Caldwell, P. A. Kollman, and D. A. Case, *J. Comp. Chem.* 25, 1157-1174 (2004)

Notes:

- o gaff.ff currently lives in the testing/ directory since it's rule-based part has not been implemented yet.

12.2.10. Liu, Wu & Wang's United-Atom Ionic Liquids Forcefield (lww-il.ff)

United atom ionic liquids forcefield for a handful of cations and anions.

References

- o X. Zhang, F. Huo, Z. Liu, W. Wang, W. Shi, and E. J. Maginn, *J. Phys. Chem. B* 113, 7591-7598 (2009)
- o Z. Liu, X. Wu, and W. Wang, *Phys. Chem. Chem. Phys.* 8, 1096-1104 (2006)

12.3. Keyword Reference

12.3.1. General Keywords

General keywords are simple keywords that take single items of data as arguments.

name

Syntax:

```
name "name of forcefield"
```

Sets the name of the forcefield as it appears in the program.

For example:

```
name "Test Forcefield"
```

sets the name of the forcefield to "Test Forcefield".

units

Syntax:

```
units energyunit
```

Specifies the units of energy used for energetic forcefield parameters. Any energetic parameters specified in the forcefield are converted from the units specified here into the internal units of energy once loading of the forcefield has completed.

For example:

```
units kcal
```

indicates that any energetic values supplied in the forcefield are in kilocalories per mole.

convert

Syntax:

```
convert name ...
```

Only relevant if a **data** block exists, the **convert** keyword takes a list of parameter names defined in the data block(s), and marks them as being (or containing) a unit of energy. When Aten converts between energy units, these marked parameters will be converted also.

For example:

```
convert Dij e_sep
```

indicates that the defined data variables `Dij` and `e_sep` are energy-based and should be converted when necessary.

12.3.2. Block Keywords

All lists of terms, types, and extraneous data are specified in the form of blocks. Each keyword in this section marks the start of block, and must at some point be terminated by an **end** keyword. Blocks contain one or more lines of data, the contents of which is detailed in each section. In addition each keyword may take one or more (optional or required) values.

angles

Syntax:

```
angles form
```

```
typename_i typename_j typename_k data1 [data2 ...]
```

```
end
```

Definitions of intramolecular angle terms are given in **angles** blocks. Multiple **angles** blocks may exist, each defining a subset of terms, and each possessing a different functional form, specified by the *form* argument (see Section **Error! Reference source not found.** for a list of valid forms).

The three *typenames* identify the particular angle to which the parameters are relevant. Note that typenames given for an angle *i-j-k* will also match an angle *k-j-i*. Data parameters should be provided in the required order for the specified *form*.

For example:

```
angles harmonic
HT CT HT 80.0 109.4
end
```

provides parameters for an H-C-H angle using the harmonic potential.

bonds

Syntax:

```
bonds form
typename_i typename_j data1 [data2...]
end
```

Definitions of intramolecular bond terms are given in **bonds** blocks. Multiple **bonds** blocks may exist, each defining a subset of terms, and each possessing a different functional form, specified by the *form* argument (see Section **Error! Reference source not found.** for a list of valid forms)

The two *typenames* identify the particular bond to which the parameters are relevant. Note that typenames given for a bond *i-j* will also match a bond *j-i*. Data parameters should be provided in the required order for the specified *form*.

For example:

```
bonds constraint
HT CT 4000.0 1.06
end
```

provides parameters for an H-C bond using the constraint potential.

data

Syntax:

```
data "type name, ..."
typeid typename data1 [data2 ...]
end
```

The **data** block defines additional generic data for each atom type. The additional data can be accessed through the `datad`, `datai`, and `datas` members of the `FFAtom` type.

The quoted argument supplied to the block defines the data types and names to be expected for each specified atom type, and furthermore strictly defines the order in which they must be given. Any of the standard simple variable types int, double, and string may be used.

Following the identifying *typeid* and *typename* data items are given one after the other and in the order they are declared in the block keyword argument.

For example:

```
data "string dogtype, int bridge, int likespasta, double numtoes"
1    OW      "redsetter"   1      0      9
2    HW      "dalmation"   1      1      64
end
```

This defines a quartet of extra data (albeit random, odd data...) for each of the specified atom types.

For forcefields which rely on functions to generate the necessary function data, the **data** block should be used to define additional data for each atom type. For example, the GAFF forcefield is able to generate extra intramolecular terms if the relevant definitions are not already defined in the forcefield, and the UFF and DREIDING forcefields contain no pre-defined intramolecular terms whatsoever.

defines

Syntax:

defines

```
definename "NETA string"
```

end

The **defines** block makes it possible to state commonly-used or lengthy chunks of NETA that do not belong to any specific atom type, and which can then be reused multiple times in many different atom type descriptions. Each definition in the block is given an identifying unique name which allows it to be referenced with the ‘\$’ symbol.

The NETA descriptions provided for each definition *must* be valid, or the forcefield will not be loaded. In subsequent NETA definitions for atom types the definitions may be inserted by stating \$definename. For example:

```
defines
water_oxygen.."-O(nh=2,nbonds=2)"
end

...
types
1      HW2      H      "nbonds=1,$water_oxygen"      "Water hydrogen"
end
```

equivalents

Syntax:

```
equivalents
```

```
alias typename [ typename ... ]
```

```
end
```

In forcefields, the most detailed information provided is typically the short-range intermolecular and charge parameters, with different values (or at least different charges) given to each individual atom type. Usually, intramolecular terms are more general and don't depend so much on the exact atom type. For example, given a tetrahedral carbon CT and three different aliphatic hydrogens H1, H2, and H3, the bond between the carbon and any of the three hydrogen types will likely be the same with respect to the force constant, equilibrium distance etc.

So, the forcefield will have to contain three intramolecular bond definitions covering CT-H1, CT-H2, and CT-H3, each having the same parameters, right? Not necessarily. While this is perfectly acceptable, for large forcefields the number of redundant terms may become quite large, and even for small forcefields with only a handful of terms, adding in duplicate data might irk the more obsessive amongst us. On these occasions, atomtype equivalents can be defined, which effectively link a set of atomtypes to a single identifying name that can then be used in any intramolecular parameter definitions.

In the waffle above, aliasing the three hydrogens H1, H2, and H3 to a single typename H1 can be done as follows:

```
equivalents
H1      H2    H3
end
```

Note that the aliased name does not have to be an atomtype name already defined in a **types** section.

function

Syntax:

```
function
```

```
type functionname ( argument list )
```

```
{
```

```
}
```

```
end
```

The **function** block contains all the function definitions relevant to rule-based forcefields (Section 1.1). The function(s) should be written in the standard command language style (Section 8.1.1).

For example:

```
function
int generatebond(ffbound data, atom i, atom j)
{
    # Calculate bond potential parameters between supplied atoms i and j
    data.form = "morse";
    return 1;
}
end
```

defines the function to be used when generation of a bond term is required. Only functions with certain names will be recognised and used properly by Aten. See the functions section (12.4.1) in rule-based forcefields for more information and a list of valid function declarations that may be made.

generator

Syntax:

```
generator "type name, ..."
```

The **generator** block is defunct as of code revision 1267. Use the **data** block instead.

inter

Syntax:

```
inter form
typeid typename charge data1[data2 ...]
end
```

Intermolecular van der Waals parameters and the charge associated with each atom type belong in the **inter** section. There may be multiple **inter** sections within the same forcefield file, but parameters for an individual atomtype may be defined only once.

The **inter** keyword begins a block of intermolecular parameter definitions, and the single argument *form* should specify the functional form of the intermolecular interactions contained within. *typeid* and *typename* refer to a single type defined in a **types** section, *charge* is the atomic charge of this atomtype, and then follows the data describing the interaction. The order of the values given should correspond to the order of parameters expected for the specified functional form (see Section 13.1 for a list of valid forms and their parameters).

For example, the Lennard-Jones potential takes two parameters – ‘epsilon’ and ‘sigma’, in that order. For a chloride atomtype with ID 24, if ‘epsilon’ = 0.5, ‘sigma’ equals 3.0, and the charge on the atomtype is -1 e, the corresponding entry in the **inter** block will be:

```
24    C1    -1.0    0.5    3.0
```

Some functional forms have default values for some parameters used in the functional form, and need not be specified (if there are any, these are shown in Section 13.1). For this reason, it is important not to add any unnecessary extra data to the entries in the ‘inter’ block, since this may overwrite a default parameter that results in literal chaos.

torsions

Syntax:

```
torsions form [escale vscale]  
typename_i typename_j typename_k typename_l data1 [data2...]  
end
```

Definitions of intramolecular torsion terms are given in **torsions** blocks. Multiple **torsions** blocks may exist, each defining a subset of terms, and each possessing a different functional form, specified by the *form* argument (see Section **Error! Reference source not found.** for a list of valid forms). For torsions the electrostatic and VDW 1-4 interactions (i.e. those between atoms *i* and *l* in a torsion *i-j-k-l*) are scaled by some factor between 0.0 and 1.0. The optional *escale* and *vscale* arguments specify these scaling factors – if they are not provided, they both default to 0.5.

The four *typenames* identify the particular torsion to which the parameters are relevant. Note that typenames given for a torsion *i-j-k-l* will also match a torsion *l-k-j-i*. Data parameters should be provided in the required order for the specified form.

For example:

```
torsions cos  
HT  CT  OC  HO  3.0  5.0  0.0  
end
```

provides parameters for an H-C-O-H torsion using the cosine potential.

```
torsions cos3 0.8333333 0.25  
CT  CT  CT  O1  1.0  -2.0  0.0  
CT  CT  CT  O2  0.5  -1.4  1.0  
end
```

defines two C-C-C-O torsions of the triple cosine form, and with custom scale factors.

types

Syntax:

types

```
typeid typename element NETA [description]
```

```
end
```

The core of the forcefield file is the **types** section, listing the ids, names, and elements of the different atom types present in the forcefield, as well as a description telling Aten how to recognise them.

The *typeid* is an integer number used to identify the type. It should be positive, and must be unique amongst all those defined in a single forcefield. *typename* is the actual name of the atom type (OW, C1, N_ar etc.), and is referred to in the other sections of the forcefield file, and *element* is the type's element symbol as found in the periodic table (O, C, N, etc.). The string *NETA* defines how Aten should recognise this particular type (in quotes if necessary), optionally followed by a short text *description* of the type (which appears in lists within the program to help identify particular types). Atom types may be defined over multiple **types** blocks within the same file if necessary, but while more than one **types** block may exist, but all type IDs must be unique over all such blocks.

For example:

```
types
35 CT C "nbonds=4" "Simple tetrahedral carbon"
end
```

describes a bog-standard tetrahedral carbon called CT, and assigns it an ID of 35.

uatypes

Syntax:

uatypes

```
typeid typename element mass NETA [description]
```

```
end
```

The **uatypes** section contains exactly the same information as the **types** block except that a mass must also be provided. In the **types** block it is assumed that the character element of the type also implicitly defines the mass (as would be expected). In the case of united-atom forcefields, this is not necessarily the case. Thus, the **uatypes** block allows a mass to be associated in order to account for the light atoms subsumed into the heavy atom's mass. This information can be accessed through the *mass* member of the `FFAtom` variable type.

For example:

```
uatypes
10 CH2 C 14.0265 "nbonds=2, nh=0" "United atom methylene carbon"
```

```
end
```

describes a united-atom methylene carbon, with mass of 14.0265 (C+2H).

ureybradleys

Syntax:

```
ureybradleys form
```

```
XXXX
```

```
end
```

12.3.3. Wildcards

In any of the typenames given in the specification of intramolecular interactions, a wildcard character '*' may be used to 'finish off' any or all of the typenames (or replace individual typenames entirely). In doing so, a single definition is able to match more than one set of typenames.

For example:

```
bonds harmonic
CT      H*      4184.0      1.06
end
```

will describe bonds between CT and any other atom beginning with an H.

Using a * on its own will match any typename in that position. As an extreme example:

```
angles harmonic
*      *      *      418.4      109.4
end
```

will match any angle. Be careful - when Aten is creating expressions and searching for specific interactions between atom types, as soon as an intramolecular definition is found that matches it is used, and no further searching is done. So, loose definitions involving wildcards should be put near to the end of the block in which they occur.

12.4. Rule-Based Forcefields

Forcefields exist where individual intramolecular parameter definitions (i.e. those provided by the **bonds**, **angles**, and **torsions** blocks) are not necessary. Instead, such parameters are constructed as and when necessary using a set of parameters that depend only on the atomtypes involved. These forcefields are so-called ‘rule-based’, and are often able to describe enormously varied systems from a small set of defining parameters.

Rule-based forcefields are defined in exactly the same way as normal forcefields, save for the lack of blocks that define intramolecular terms. Instead, the per-atomtype parameters must be provided instead, and for all atomtypes defined in the **types** section(s). This generator data is then used by the equations defined within the code to construct the necessary intramolecular terms when required. One or more **data** blocks should be used to define this data for each atomtype.

12.4.1. Functions

In a rule-based forcefield all the useful function declarations which calculate the correct parameters (usually from values supplied in a **data** block) must be made within a single **function** block in the forcefield file. When calling the functions, Aten provides the necessary structure in which the generated parameters should be stored. In the case of the VDW-generating function, the actual atomtype structure which is missing the data is passed (see the **FFAtom** variable type). In the case of intramolecular interactions, Aten creates and passes a new, empty **FFBound** container in which the functional form of the interaction and the relevant data values should be set. A number of **FFAtom** references are also provided, corresponding to the atom types involved in the bound interaction, and from which the necessary data values may be retrieved using the relevant data accessors. For bound interactions it is not necessary to set the equivalent names of the involved atom types since this is done automatically.

The recognised function names and their arguments are as follows:

anglegenerator

Syntax:

```
int anglegenerator ( FFBound newdata, Atom i, Atom j, Atom k )
```

Called whenever function data for an unrecognised angle (between the atom types currently assigned to atoms *i*, *j*, and *k*) is needed. Generated parameters should be stored in the passed *newdata* structure

bondgenerator

Syntax:

```
int bondgenerator ( FFBound newdata, Atom i, Atom j )
```

Called whenever function data for an unrecognised bond (between the atom types currently assigned to atoms *i* and *j*) is needed. Generated parameters should be store in the passed *newdata* structure

torsiongenerator

Syntax:

```
int torsiongenerator ( FFBound newdata, Atom i, Atom j, Atom k, atom l )
```

Called whenever function data for an unrecognised torsion (between the atom types currently assigned to atoms *i*, *j*, *k*, and *l*) is needed. Generated parameters should be store in the passed *newdata* structure

vdwgenerator

Syntax:

```
int vdwgenerator ( FFAtom data )
```

Called whenever descriptive VDW data is missing from an atom type (which is passed into the function and should have the correct data placed in it)

12.5. Typing

We are all familiar with talking about atoms being chemically different depending on the functional group in which they exist - e.g. ether, carbonyl, and alcoholic oxygens - and this categorisation of atoms forms basis of forcefield writing. That is, a large number of different molecules and types of molecule should be described by a small set of different atoms, i.e. atom *types*. At the simplest level, the connectivity of an atom is enough to uniquely identify its specific type.

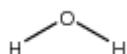
Some methods to use this information to uniquely assign types to atomic centres involve deriving a unique integer from the local connectivity of the atom (e.g. the SATIS method REF XXX), but including information beyond second neighbours is rather impractical. Others use a typing 'language' to describe individual elements of the topology of atoms in molecules, and are flexible enough to be able to describe complex situations in a more satisfactory way (e.g. that employed in Vega ref XXX). Aten uses the latter style and provides a clear, powerful, and chemically-intuitive way of describing atom types in, most importantly, a readable and easily comprehended style.

Type descriptions are used primarily for assigning forcefield types, but also make for an extremely useful way to select specific atoms as well.

12.5.1. Language Examples

Type descriptions in Aten use connectivity to other atoms as a basis, extending easily to rings (and the constituent atoms), lists of allowable elements in certain connections, atom hybridicities, and local atom geometries. Descriptions can be nested to arbitrary depth since the algorithm is recursive, and may be re-used in other atom's type descriptions to simplify their identification. Time to jump straight in with some examples. Note that these examples only serve to illustrate the concepts of describing chemical environment at different levels. They may not provide the most elegant descriptions to the problem at hand, don't take advantage of reusing types (see Section 12.5.4), and certainly aren't the only ways of writing the descriptions.

Example 1 - Water



Consider a water molecule. If you were describing it in terms of its structure to someone who understands the concept of atoms and bonds, but has no idea what the water molecule looks like, you might say:

A water molecule contains an oxygen that is connected two hydrogen atoms by single bonds

...or even...

It's an oxygen atom with two hydrogens on it

Given this degree-level knowledge, to describe the individual oxygen and hydrogen atoms in the grand scheme of the water molecule exactly, you might say:

A 'water oxygen' is an oxygen atom that is connected to two hydrogen atoms through single bonds

...and...

A 'water hydrogen' is a hydrogen that is connected by a single bond to an oxygen atom that itself is connected by a single bond to another (different) hydrogen atom

The extra information regarding the second hydrogen is necessary because otherwise we could apply the description of the 'water hydrogen' to the hydrogen in any alcohol group as well. Similarly, we might mistake the oxygen in the hydroxonium ion $[H_3O]^+$ as being a 'water oxygen', when in fact it is quite different. In this case, we could extend the description to:

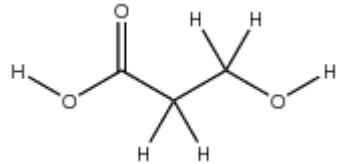
A 'water oxygen' is an oxygen atom that is connected to two hydrogen atoms through single bonds, and nothing else

An atom description in Aten is a string of comma-separated commands that describe this kind of information. So, to tell the program how to recognise a water oxygen and a water hydrogen, we could use the following type descriptions (written in the proper forcefield input style for the **types** block:

```
1      OW      O      "nbonds=2,-H,-H"          # Water oxygen
2      HW      H      "-O(nbonds=2,-H,-H)"      # Water hydrogen
```

Aten now recognises that a water oxygen (`OW`) is '*an oxygen atom that has exactly two bonds & is bound to a hydrogen & is bound to another hydrogen*'. Similarly, a water hydrogen (`HW`) is '*a hydrogen bound to an oxygen atom that; has two bonds to it, and is bound to a hydrogen, and is bound to another hydrogen*'. In the type descriptions above the dash '`-`' is short-hand for saying 'is bound to', while the bracketed part after '`-O`' in the water hydrogen description describes the required local environment of the attached oxygen. Using brackets to describe more fully the attached atoms is a crucial part of atom typing, and may be used to arbitrary depth (so, for example, we could add a bracketed description to the hydrogen atoms as well, if there was anything left to describe). If necessary, descriptions can be written that uniquely describe every single atom in a complex molecule by specifying completely all other connections within the molecule. This should not be needed for normal use, however, and short descriptions of atom environment up to first or second neighbours will usually suffice.

Example 2 - 3-hydroxypropanoic acid



Assuming that the OH group in the carboxylic acid functionalisation will have different forcefield parameters to the primary alcohol at the other end of the molecule, here we must describe the first and second neighbours of the oxygen atoms to differentiate them.

To begin, we can describe the carbon atoms as either two or three different types – either methylene/carboxylic acid, or carboxylic acid/adjacent to a carboxylic acid/adjacent to alcohol. For both, we only need describe the first neighbours of the atoms. For the first:

```

3      C(H2)  C      "nbonds=4,-H,-H,-C"          # Methylene Carbon
4      C_cbx  C      "nbonds=3,-O(bond=double),-O,-C"  # Carboxylic Acid C

```

Note the ordering of the oxygen connections for the carboxylic acid carbon, where the most qualified carbon is listed first. This is to stop the doubly-bound oxygen being used to match – O, subsequently preventing a successful match. This is a general lesson – bound atoms with the most descriptive terms should appear at the beginning of the type description (as it is read left-to-right) and those with the least left until the end.

Where all three carbons need to be identified separately, we may write:

```

5      C(OH)  C      "nbonds=4,-H,-H,-C,-O"        # CH2 adjacent to OH
6      C(COOH) C      "nbonds=4,-H,-H,-C,-C"       # CH2 adjacent to COOH
7      C_cbx   C      "nbonds=3,-O(bond=double),-O,-C"  # Carboxylic Acid C

```

Let us now assume that the hydrogens within the alcohol and carboxylic acid groups must also be seen as different types. In this case, the second neighbours of the atoms must be considered:

```

8      HO     H      "-O(-C(-H,-H))"           # Alcoholic H
9      H_cbx  H      "-O(-C(-O(bond=double)))"  # Carboxylic acid H

```

The assignment is thus based entirely on the nature of the carbon atom to which the OH group is bound since this is the next available source of connectivity information. The determination of the three different oxygen atoms is similar:

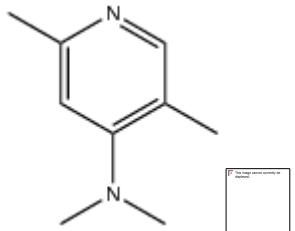
```

10     OH     O      "-H,-C(-H,-H)"          # Alcoholic O
11     O_cbx  O      "-C(-O(-H))"           # Carboxylic acid =O
12     OH_cbx O      "-H,-C(-O(bond=double))"  # Carboxylic acid O(H)

```

Of course, we could just have specified nbonds=1 for the doubly-bound oxygen of the carboxylic acid group, but this ‘hides’ information as to the true connectivity of the atom.

Example 3 - *N,N,2,5-tetramethylpyridin-4-amine*



At last, a proper problem - an asymmetric substituted pyridine. Lets assume that we need to distinguish between every non-hydrogen atom – we'll skip describing the hydrogen atoms for now, but note that this is most easily achieved by specifying directly the atomtype that the H is bound to (see later on). Let's start with the pyridine nitrogen. We basically need to say that its in a 6-membered aromatic ring:

```
13      N_py    N      "ring(size=6,aromatic)"          # Pyridine N
```

TODO

12.5.2. Description Depth

Many subtleties related to the form of type descriptions are perhaps evident from the examples given above. It is useful to think of type descriptions as having many different ‘depths’, loosely corresponding to the number of bonds followed away from a central atom (the target atom, or the one currently being tested against the type description). The target atom of the type description is the root of the description since all connections are defined relative to this atom. A type description requiring specific connections to this target atom is using the target atom’s bonds in order to identify it - atoms to a depth of 1 bond are being used to describe the atom. If these bound atoms are in turn described by their bound neighbours then atoms to a depth of two bonds are being used.

For example:

Table 12-1 NETA Descript Depth Examples

Example Description	Effective Depth
<code>nbonds=4</code>	Zero - contains specifications relevant to the root atom only
<code>nbonds=4, -H, -C</code>	1 - root commands and first bound neighbours
<code>nbonds=4, -H, -C (-H (n=3))</code>	2 - root commands, first and second bound neighbours

Any depth of description can be handled by Aten, becoming as complex as is necessary to uniquely identify the target atom.

12.5.3. Type Scores

In a forcefield with many defined types, more than one type may match an atom in a molecule. In order to best assign types to atoms, Aten scores each type description according to the number of terms defined within it, one point for each term satisfied. Once a matching type for an atom is located in the forcefield it is assigned to that atom, but the search for a better-scoring type continues. If a match with a higher score is found it replaces the previously-assigned type. If a match with the same score is found, the previous type is *not* replaced.

Non-Matching Types (Score = -1)

When a type description is tested for a given atom, it accumulates points for each term in the description that is satisfied by the environment of the atom. As soon as a term is found that is not satisfied, however, the score is reset to -1 and the match will fail. All terms in a type description must be satisfied in order for the type to be assigned to an atom.

Empty Types (Score = 1)

A type description containing no terms has a maximum score of 1 (coming from a match of the element type). Hence:

```
99      Cgen      C      ""                      # Generic carbon
```

matches any carbon in any system, but will be replaced fairly easily by other types since it has such a low score.

Normal Types (Score > 1)

For a type in which all terms are matched successfully, one point is scored for each individual term. All of the following types have a potential maximum score of 3 (don't forget, one point comes from matching the element):

```
100    C1      C      "nbonds=2, linear"          # Carbon A
101    C2      C      "-C,-C"                      # Carbon B
102    C3      C      "-C(n=2)"                   # Carbon C
102    C4      C      "=C"                         # Carbon D
```

Moreover, they all potentially match the same atom (for example the central carbon in 1,2-propadiene). Since they have the same score, the first type `C1` will match and persist over the other three, since only types with higher (not equivalent) scores can replace it.

12.5.4. Reusing Types

Once a complex NETA definition has been made for a given atom type, it is often useful to be able to reference this type in order to save repeating the same description for a closely-bound atom. The ampersand symbol allows you to do this, and specifies an integer type id to match rather than an element. As an example, consider the following definitions for trifluoroethanol from the OPLS-AA forcefield where the environment for each atom is described in full for each type:

160	CT	C	"-H,-H,-O(-H),-C(-F(n=3))"	"CH2 in trifluoroethanol"
161	CT	C	"-F(n=3),-C(nh=2,-O(-H))"	"CF3 in trifluoroethanol"
162	OH	O	"-H,-C(nh=2,-C(-F(n=3)))"	"OH in trifluoroethanol"
163	HO	H	"-O(-C(nh=2,-C(-F(n=3))))"	"HO in trifluoroethanol"
164	F	F	"-C(-F(n=3),-C(nh=2,-O(-H)))"	"F in trifluoroethanol"
165	HC	H	"-C(nh=2,-O(-H),-C(-F(n=3)))"	"H in trifluoroethanol"

For each atom type, the whole of the trifluoroethanol molecule is described, but each type tends to share a common chunk of NETA definitions with the other types. As an alternative, then, we can define one or two of the involved types explicitly as above, and then specify the rest of the types relative to this one:

160	CT	C	"-H,-H,-O(-H),-C(-F(n=3))"	"CH2 in trifluoroethanol"
161	CT	C	"-&160"	"CF3 in trifluoroethanol"
162	OH	O	"-&160"	"OH in trifluoroethanol"
163	HO	H	"-O(-&160)"	"HO in trifluoroethanol"
164	F	F	"-C(-&160)"	"F in trifluoroethanol"
165	HC	H	"-&160"	"H in trifluoroethanol"

Much neater! Or should that be ‘not much NETA’? Hmm. Anyway, reusing types in this way is a powerful way to reduce the complexity of type descriptions required for a given molecule or fragment. Typically it is advisable to pick an atom that is fairly central to the molecule and bears a lot of connections, and provide an unambiguous type description.

12.6. NETA Reference

This section lists all the available commands that may make up a type description. Many keywords only make sense within the bracketed parts of keywords that expand the depth of the description. For instance, it is meaningless to specify the connection type with the **bond** keyword in the root of a typing command since no connections are relevant at this point.

~X (any bond to X)

~x specifies that a connection to x must exist, but makes no demand of the type (bond order) of the connection. x may be an element symbol, an id for another type specifier, or a list in square brackets containing one or both of these to allow more flexible specifications. Specifying an unknown connection with **~x** is often useful in, for example, aromatic rings or conjugated systems where the connection might be either a double or single bond.

Table 12-2 NETA ‘~X’ Keyword Examples

Command	Meaning
~C	Any bond to a carbon atom
~&101	Any bond to an atom which matches type ID 101 (see Reusing Types in section 12.5.4)
~[N, S, P]	Any connection to either nitrogen, sulfur, or phosphorous
! ~O	Explicitly states that there should not be a bond to an oxygen atom

-X (single bond to X)

-x specifies that a single bond to x must exist. x may be either an element symbol, an id for another type specifier, or a list containing one or both of these to allow more flexible specifications. If used inside the bracketed part of a **ring** description this only indicates that the atom/type should be present within the cycle – the connection to the target atom is unimportant. If used in a **chain** keyword the connection type is honoured.

Table 12-3 NETA ‘-X’ Keyword Examples

Command	Meaning
-H	A single bond to a hydrogen atom
-&120	A single bond to an atom which matches type ID 120 (see Reusing Types in section 12.5.4)
-[C, N]	A single bond to either a carbon or a nitrogen
-[F, Cl, Br, I, At]	A single bond to any halogen atom
-[&10, &11, &18, -Kr]	A single bond to an atom with type ID 10, 11, or 18, or a krypton atom

=X (double bond to X)

=x specifies that a double bond to x must exist. Equivalent to writing `~x (bond=double) . x` may be either an element symbol, an id for another type specifier, or a list containing one or both of these to allow more flexible specifications. If used inside the bracketed part of a **ring** description this only indicates that the atom/type should be present within the cycle – the connection to the target atom is unimportant. If used in a **chain** keyword the connection type is honoured.

Table 12-4 NETA '=X' Keyword Examples

Command	Meaning
<code>=O</code>	A double bond to an oxygen atom
<code>=&4</code>	A double bond to an atom which matches type ID 4 (see Reusing Types in section 12.5.4)

bond

The **bond** keyword defines the specific *type* of the connection (see Bond Types in Section 16.2) required for a bound atom. The keyword should be used inside bracketed parts of bound atom descriptions. It is important to note that the bond keyword should only be used in conjunction with the `~x` (any bond to) specifier, since the specific connection demanded by the `-x` and `=x` specifiers will override any **bond** declarations.

Table 12-5 NETA 'bond' Keyword Examples

Command	Meaning
<code>~O (bond=double)</code>	A double bond to an oxygen atom (equivalent to <code>=O</code>)
<code>~&55 (bond=triple)</code>	A triple bond to an atom which matches type ID 55 (see Reusing Types in section 12.5.4)
<code>~C (bond=single)</code>	A single bond to a carbon atom (a very explicit way of writing simply <code>-C</code>)

chain

The **chain** command provides an easy was of specifying a linear sequence of atoms from the current atom forward. Within the bracketed part, a sequence of connections are listed in the order in which they are to appear. Atoms in the chain are specified in the same way as other connections (e.g. `-C (nbonds=2)`) but should *not* be separated by commas. The repeat keyword **n** may also be specified at some point in the bracketed part to define that more than one of the defined chains is required. Note that, if all atoms specified for the chain are matched, but the actual chain in the model is longer, a positive match will be returned. Thus, it is usually a good idea to define the last atom in the chain more explicitly to prevent false matches.

Table 12-6 NETA ‘chain’ Keyword Examples

Command	Meaning
<code>chain (-C-C-C-C)</code>	Specifies a (minimum) four-carbon chain of any

<code>chain (-C (nh=2) -C (nh=2) -C (nh=2) -C (nh=3))</code>	degree of saturation Explicitly specifies an all-atom butyl chain
--	--

n

The **n** keyword, when placed in the bracketed parts of bound atom, ring, and chain descriptions requires that they are matched a number of times rather than just once.

Table 12-7 NETA ‘n’ Keyword Examples

Command	Meaning
<code>-C (n=4, -H (n=3))</code>	Describes the central carbon in neopentane (2,2-dimethylpropane) which is bound to four methyl groups
<code>ring (size=4, n=3)</code>	Specifies that the target atom should be present in three unique four-membered rings
<code>chain (-C-C-C~N (bond=triple), n=4)</code>	Requests that the atom has four cyanoethyl groups hanging off it

nbonds

nbonds specifies the exact number of connections that an atom must possess.

Table 12-8 NETA ‘nbonds’ Keyword Examples

Command	Meaning
<code>nbonds=2</code>	Demand that the atom has exactly two connections
<code>~N (nbonds=1)</code>	Describes a nitrogen with only one bond, perhaps an sp ¹ nitrogen with a triple bond

nh

The **nh** keyword is shorthand for explicitly specifying the number of attached hydrogens to the target atom or a bound atom. It is equivalent to stating `-H (n=m)`.

Table 12-9 NETA ‘nh’ Keyword Examples

Command	Meaning
<code>-C (nh=2)</code>	Atom is bound to a methylene carbon with exactly two hydrogens on it

aromatic

aromatic indicates either that an atom must be present in an aromatic environment (e.g. in an aromatic ring), or that a ring should itself be aromatic.

Table 12-10 NETA ‘aromatic’ Keyword Examples

Command	Meaning
<code>~N(aromatic)</code>	Specifies a nitrogen connected by any bond which is present in an aromatic environment
<code>-C(ring(aromatic))</code>	Single bond to a carbon atom which is in an aromatic ring

noring

noring indicates that the atom must not be present in any rings.

Table 12-11 NETA ‘noring’ Keyword Examples

Command	Meaning
<code>-O(noring)</code>	Single bond to an oxygen which is not present in a ring

planar

The **planar** keyword specifies that the atom target should be planar, which is to say no bond from the atom may be more than 15° out of the plane formed by the first two bonds to the atom)

Table 12-12 NETA ‘planar’ Keyword Examples

Command	Meaning
<code>-C(planar)</code>	Single bond to a carbon atom which is roughly planar

ring

ring denotes that the target atom (if specified in the root of the description) or a bound atom (if used inside the associated bracketed part) should be present in a ring structure of some kind. The **ring** keyword alone simply demands that the atom is present inside a ring of some size, but may take an optional bracketed part describing more fully the number of atoms in the ring, and the individual nature of each of these atoms. Within the bracketed part, bound atoms may be specified as usual in the contained *neta*, but the connection type is irrelevant as it is only the presence of those particular atoms within the ring that is considered important.

Bound atom descriptions given inside the bracketed part should again be listed in order of decreasing complexity. Multiple rings may be specified with separate **ring** keywords, allowing the location of fused ring atoms.

Table 12-13 NETA ‘ring’ Keyword Examples

Command	Meaning
<code>ring(size=6,~C(n=6),aromatic)</code>	Benzene-style carbon
<code>-C(ring(size=6,-C(n=6,nh=2)))</code>	Carbon atom in cyclohexane

size

Only relevant in the bracketed part of a `ring` keyword, `size` requests the exact number of atoms comprising the cycle. Note that this may be used independently of the implicit size suggested by the number of atom descriptions supplied, or in conjunction with a partial list of atoms.

Table 12-14 NETA ‘size’ Keyword Examples

Command	Meaning
<code>ring(size=7)</code>	Specifies a 7-membered ring

12.6.1. Geometries

These keywords requests that the target atom (if specified in the root of the type description) or a bound atom (if used inside the associated bracketed part) should possess a certain physical geometry in terms of its connections. The number of bonds to the target atom and the angles between them are used to determine the geometry.

Note that, for some geometry types, there are several ways for the atom to have this geometry.

unbound

An `unbound` atom, i.e. one with zero bonds to other atoms. Only makes sense when used in the root of a type description.

onebond

`onebond` requests that the atom has exactly one bond (of any type).

linear

Two bonds to the atom in a `linear` arrangement (angle $i-j-k > 170^\circ$).

tshape

Three bonds to the atom in a `tshape` geometry (with two bonds making an angle $> 170^\circ$).

trigonal

Three bonds in a **trigonal** planar arrangement, with the largest of the three angles between 115 and 125°.

tetrahedral

tetrahedral geometries are possible for atoms with; exactly two bonds making an angle between 100 and 115°; exactly three bonds with the largest of the angles between 100 and 115°, and; exactly four bonds to the atom, with the average of the angles laying between 100 and 115°.

sqplanar

Four bonds to the atom in a square planar (**sqplanar**) arrangement, with the average of the angles laying between 115 and 125°.

tbp

Five bonds to the atom are assumed to be trigonal bipyramidal (**tbp**) geometry.

octahedral

Six bonds to the atom are assumed to be in an **octahedral** arrangement.

13. Functional Forms

All forcefield term functional forms are listed in the following sections, along with their parameters (default values for which follow in brackets, if they exist).

13.1. VDW Functional Forms

Table 13-1 VDW Functional Forms

Name	Keyword	Form $U_{ij} =$	Param	Rule ¹
Lennard-Jones 12-6	<code>lj</code>	$4\epsilon \left[\left(\sigma/r_{ij} \right)^{12} - \left(\sigma/r_{ij} \right)^6 \right]$	ϵ σ	Geom Arith
Lennard-Jones 12-6 (Geometric rules)	<code>ljgeom</code>	$4\epsilon \left[\left(\sigma/r_{ij} \right)^{12} - \left(\sigma/r_{ij} \right)^6 \right]$	ϵ σ	Geom Geom
Inverse Power	<code>inversepower</code>	$\epsilon \left(r/r_{ij} \right)^n$	ϵ r n (1.0)	Geom Arith Arith
Lennard-Jones AB	<code>ljab</code>	$A/r_{ij}^{12} - B/r_{ij}^6$	A B	Geom Geom
UFF Lennard-Jones 12-6	<code>ufflj</code>	$D_{ij} \left[\left(\sigma/r_{ij} \right)^{12} - n \left(\sigma/r_{ij} \right)^6 \right]$	D_{ij} σ n (2.0)	Geom Geom Arith
Buckingham exp6	<code>buck</code>	$Ae^{-Br_{ij}} - C/r_{ij}^6$	A B C	Geom Geom Geom
Morse	<code>morse</code>	$E_0 \left[\left(1 - e^{-k(r_{ij}-r_0)} \right)^2 - 1 \right]$	E_0 k r_0	Geom Arith Geom

¹ Combination rule used to generate cross-terms – either Geometric or Arithmetic.

13.2. Bond Functional Forms

Table 13-2 Bond Functional Forms

Name	Keyword	Form	Param
Ignore	ignore		
Constraint	constraint	$\frac{k}{2}(r - r_0)$	k r_0
Harmonic	harmonic	$\frac{k}{2}(r - r_0)^2$	k r_0
Morse	morse	$E_0 \left(1 - e^{-k(r_{ij} - r_0)}\right)^2$	E_0 k r_0

13.3. Angle Functional Forms

Table 13-3 Angle Functional Forms

Name	Keyword	Form	Param
Ignore	ignore		
Harmonic	harmonic	$\frac{k}{2}(\theta - \theta_0)^2$	k θ_0
Cosine	cos	$k(1 + s \cos(n\theta - \theta_0))$	k n θ_0 $s (1.0)$
Double Cosine	cos2	$k(C_0 + C_1 \cos(\theta) + C_2 \cos(2\theta))$	k C_0 C_1 C_2
Harmonic Cosine	harmcos	$\frac{k}{2}(\cos(\theta) + \cos(\theta_0))^2$	k θ_0
Constraint (1-3 Bond)	bondconstr aint	$\frac{k}{2}(r - r_0)^2$	k r_0

13.4. Torsion Functional Forms

Table 13-4 Torsion Functional Forms

Name	Keyword	Form	Param
Cosine	cos	$k(1 + s \cos(n\varphi - \varphi_{eq}))$	k n φ_{eq} $s (1.0)$
Triple Cosine	cos3	$\frac{1}{2} \sum_{i=1}^3 k_i (1 + (-1)^{i+1} \cos(i\varphi))$	k_1 k_2 k_3
Quadruple Cosine	cos4	$\frac{1}{2} \sum_{i=1}^4 k_i (1 + (-1)^{i+1} \cos(i\varphi))$	k_1 k_2 k_3 k_4
Triple Cosine + Constant	cos3c	$k_0 + \frac{1}{2} \sum_{i=1}^3 k_i (1 + (-1)^{i+1} \cos(i\varphi))$	k_0 k_1 k_2 k_3
Cosine Product	coscos	$\frac{1}{2} k (1 - \cos(n\varphi_{eq}) \cos(n\varphi))$	k n
Dreiding Cosine	dreiding	$\frac{1}{2} k (1 - \cos(n(\varphi - \varphi_{eq})))$	φ_{eq} k n

14. External Programs

14.1. Movie Generation

14.2. MOPAC

Aten can make use of MOPAC (any version) if it is installed correctly on your machine. All that is required is for the path to the binary to be set in the program preferences, along with a suitable temporary directory in which the calculations will run. Either go to the **External Programs** tab in the **Preferences** window (**Settings→Preferences**)

15. Methods

Some things in Aten are implemented from existing routines and algorithms. Some have been written from scratch, even when existing algorithms were available, either as an attempt to improve those existing algorithms or simply to learn more by working out how best to go about solving a given problem. A selection of algorithms are detailed in the following pages, grouped into those that were re-used from the literature, and those that were written specifically for Aten.

It should be pointed out that, in the eventuality that somebody notices that one of Aten's 'custom' algorithms is actually a reproduction of an existing method, then fair enough - send me the reference and I'll be happy to move it to the proper section.

15.1. Custom Algorithms

15.1.1. NETA

NETA stands for the Nested English Typing Algorithm – a fairly dull acronym, all said and done, but with the advantage that it is 'Aten' backwards. NETA is an attempt to provide a descriptive atom typing language that is:

- Easily readable
- Easily written from a small subset of keywords
- Recursive and able to describe complex molecules

It's closest relative that I'm aware of in the literature is the ATDL as implemented in Vega-ZZ,[[1]] but was (genuinely) conceived without prior knowledge of that system. NETA tries to keep the language simple enough that it can almost be read aloud and make sense, given one or two special syntactic tokens, rather than needlessly use numerical codes and spurious symbols to signify certain quantities or create an ultra-compact language. The former destroys readability and the latter promotes convolution, neither of which help when trying to interpret old rules or write new ones. So, for the most part NETA is keyword-based, with a limited number of fairly 'natural' symbols employed to denote common terms.

Typing begins from a provided set of atoms and bonds (i.e. the chemical graph). The connectivity between atoms must be 'set' prior to typing, either by automatic calculation of bonds based on distance criteria, manually adding them by hand, or reading them from the input model file. The typing algorithm itself makes no additions or changes to the connectivity of the input structure.

NETA requires a knowledge of species/molecule types in the model is required. For single molecule systems there is 1 distinct molecule (species) and 1 occurrence of it. For condensed phases, e.g. liquids, there are 1 or more species each with many copies of the molecule. In the interests of efficiency for the following routines, Aten attempts to generate a valid pattern description of the system if one is not present already. This essentially picks out the individual species and the number of molecules of each, and permits the typing routines to consider only one molecule of each species when determining atom types etc. The assumption here is that,

since all molecules in a given species will have the same chemical graph, atom types can be worked out for a single molecule and then duplicated on all of the others.

Following detection of a suitable pattern description, several tasks are then performed:

Cycle Detection

Firstly, any cyclic structures within a single molecule are detected up to a maximum (adjustable) ring size. This is achieved from a series of simple recursive searches beginning from each atom with more than one bond. A sequence of 'walks' along bonds are made in order to form a path of some specified length (i.e. ring size). If the final atom in this path shares a bond with the starting atom, a cycle has been found. If not, the final atom is removed and replaced with another. If there are no more atoms to try in this final position, the preceding atom in the path is removed and replaced with another, and so on. Each unique ring (its size and pointers to the sequence of constituent atoms) is stored in the pattern.

Assignment of Atom Environment

From the list of bound neighbours, each atom is assigned a simple hybridicity based on the character of the bonds it is involved in, mainly used for the determination of aromatic cycles in the next step.

Ring Types

Once atom hybridicities have been assigned, ring types can be determined. Rings are classed as either aliphatic, aromatic, or non-aromatic (i.e. a mix of resonant and aliphatic bonds that is not itself aromatic).

Now, working only with the representative molecule of each pattern, associated (or current) forcefield(s) are searched for types that match the contained atoms. Each NETA description whose character element is the same as a given atom is tested, and a score is obtained. If this score is non-zero and positive then the atomtype is a match and is assigned to the atom if it has no previous type, or if the score is higher than the previous one. See atom type scoring in Section 12.5.3 for more information.

[1] Pedretti, A.; Villa, L.; Vistoli, G. "Theoretical Chemistry Accounts", 109, 229-232 (2003).

15.1.2. Augment

Augmentation of bonds, as far as Aten is concerned, means to take a collection of atoms with basic connectivity (i.e. all single bonds, as per the result of rebonding) and assign multiple bonds where necessary. The method is based loosely on previously described algorithms.[1]

The basis of the method involves modifying the bond order of a particular connection to best satisfy the bonding requirements of the two involved atoms, for example making sure all carbon atoms possess an optimal total bond order of 4. However, many atoms (in particular S and P) happily exist with more than one total bond order (e.g. P) - the methodology borrowed from [1] solves this problem by scoring the total bond order for each particular element ranging from zero (meaning 'natural' or 'no penalty') to some positive number. The higher the positive number, the more 'unhappy' the element is with this number of bonds. For example, hydrogen atoms score 0 for a total bond order of 1, a small positive number (2) for

no bonds (hydrogen ion) and a very large positive value (here, 32) for any other bond order. In this way we penalise the total bond orders that an atom does not naturally take on, and always tend towards the lowest score (i.e. the natural total bond order) wherever possible. When modifying the bond order of a particular connection, the total bond order scores of both atoms are calculated once for the current connection and again for the potential new bond order of the connection. If the new score is lower, the change of bond order is accepted.

Pattern Detection

As with many other routines in Aten, a suitable pattern description is first detected for the system in order to isolate individual molecular species and make the algorithm as efficient as possible.

Augmentation of Terminal Bonds

Bonds that involve a heavy (i.e. non-hydrogen) atom connected to no other atoms (e.g. C=O in a ketone) are treated before all others. The bond order is modified such that the total bond order score for both atoms is as low as possible.

Augmentation of Other Bonds

Following optimisation of terminal bonds, all other bonds are modified using exactly the same procedure.

Second Stage Augmentation

The above two steps are enough to correctly determine multiple bonds in a chemically-correct molecule, provided no cyclic moieties are present in the system. The second stage is designed to correct improper augmentations within cycles, or shift existing augmentations around cycles such that other (missing) multiple bonds may be created.

For each existing multiple bond in each cyclic structure in each pattern's molecule, a simple re-augmentation of the constituent bonds is first attempted in order to try and lower the total bond order score for the whole ring (i.e. the sum of the individual bond order scores of every atom present in the cycle). Then, each bond in the ring is considered in sequence. If the bond is a double bond, then we attempt to convert this into a single bond and make the two adjacent bonds in the ring double bonds in an attempt to ‘aromaticise’ the ring. The total bond order score is checked and, if lower than the previous score, the change is accepted. If not, the change is reversed and the next bond is considered. By performing these secondary adjustments the double-bond pattern of many complex (poly)aromatics can be correctly (and fully automatically) detected.

[1] “Automatic atom type and bond type perception in molecular mechanical calculations”, J. Wang, W. Wang, P. A. Kollman, and D. A. Case, *Journal of Molecular Graphics and Modelling*, 25 (2), 247-260 (2006).

15.1.3. Autoellipsoids

TODO

15.1.4. Autopolyhedra

TODO

15.1.5. Rebond

The most common means of determining connectivity between a collection of atoms is based on simple check of the actual distance between two atoms and the sum of their assigned radii:

{img align="center" src=show_image.php?id=116}

The two σ represent the radii of atoms i and j which have coordinates x_i, y_i, z_i and x_j, y_j, z_j . The parameter α is an adjustable tolerance value to enable fine-tuning, and using Aten's set of built-in radii[1] usually lays between 1.0 and 2.0. For molecules or periodic systems of modest size the method can be used as is, but for large systems of many atoms the use of a double loop over atoms results in a very slow algorithm.

Aten overcomes this slowdown for larger systems by partitioning the system up into a series of overlapping 'cuboids'. For a system of N particles in a periodic box (or an isolated system with an orthorhombic pseudo-box determined by the extreme positions of atoms), the volume is partitioned into a number of subvolumes of some minimum size in each direction. The minimum size of any one of the subvolume's dimensions is chosen relative to the maximum bond length possible given the largest elemental radius and the current bond tolerance α . A single loop over atoms is then performed to associate them to these subvolumes. Each atom belongs to at least one cuboid, determined by its absolute position in the system, and commonly belongs to one other cuboid, determined by adding half of the cuboids dimensions on to the atoms position. While a little counterintuitive, potentially adding atoms to a neighbouring cuboid along this diagonal vector allows the final calculation of distances between pairs of atoms to consider only eight neighbouring (more correctly, overlapping) subvolumes rather than the 26 needed if each atom belongs exclusively to only one cuboid. For atoms that exist in subvolumes along the edges of the whole volume, these are also added to the subvolume(s) on the opposite side(s) to account for minimum image effects in periodic systems.

Once the effort has been made to assign atoms to cuboids, the final loops to calculate distances runs over a much reduced subset of atom pairs owing to the partitioning. A loop over cuboids is performed, first considering all atom pairs within the same cuboid, and then extending this to consider distances between a particular atom of this central cuboid and its eight 'overlapping' neighbours.

There is some redundancy of atom pairs since the same pair may be considered twice when taking into account the overlapping cuboids. However, in the interests of facile book-keeping this is not checked for explicitly during the running of the algorithm.

[1] "Covalent radii revisited", B. Cordero, V. Gómez, A. E. Platero-Prats, M. Revés, J. Echeverría, E. Cremades, F. Barragán and S. Alvarez, *Dalton Trans.* (2008) (DOI: <http://dx.doi.org/10.1039/b801115j>)

15.1.6. Disorder Builder

TODO

15.2. Literature Methods

The following table lists common literature methods implemented in Aten, noting any implementation differences etc.

Table 15-1 Literature Methods

Name	Description / Notes
Ewald Sum	Electrostatic energy and forces for periodic systems. Implemented as described in: XXX
Conjugate Gradient	
Marching Cubes	Isosurface generation from regular gridded data. No pathological cases are accounted for. See implementation details in: XXX
Steepest Descent	

16. Enumerations

The following tables list sets of keywords relevant to various data (e.g. bond types), to be used when setting or checking such values.

16.1. Basis Shell Types

A list of recognised types of basis shell recognised by the `BasisShell` type.

Table 16-1 Basis shell type keywords

Value	Description	nCartesians
S	Standard S-shell	1
L	Combined S/P shell	4
P	Standard P-shell	3
D	Standard D-shell	5
F	Standard F-shell	7

16.2. Bond Types

A bond between atoms is of one of the following types:

Table 16-2 Bond type keywords

Value	Description
any	Special case, never assigned to an actual bond but used in bond-matching functions
single	A single bond
double	A double bond
triple	A triple bond
aromatic	An aromatic bond, assigned automatically by the atom typing routines

16.3. Bound Types

Possible intramolecular bound interaction types are:

Table 16-3 Bound type keywords

Value	Description
angle	A normal angle interaction between three bound atoms
bond	A normal bond interaction between two bound atoms
improper	An improper torsion between four (not necessarily bound) atoms
torsion	A proper torsion interaction between four bound atoms

16.4. Cell Types

A model's unit cell is, at any given time, one of the following:

Table 16-4 Cell type keywords

Value	Description
none	The model has no unit cell (i.e. it is non-periodic)
cubic	The model's unit cell is cubic, with A=B=C and alpha=beta=gamma=90
orthorhombic	The model's unit cell is an orthorhombus, with any cell lengths and alpha=beta=gamma=90
parallelepiped	The model's unit cell is monoclinic or triclinic, with any cell lengths and angles

16.5. Colour Schemes

Available atomic colouring schemes are:

Table 16-5 Colour scheme keywords

Value	Description
charge	Atoms and connected bonds are coloured according to their charge
element	Atoms and connected bonds are coloured according to their element
force	Atoms and connected bonds are coloured according to the force acting on the atom
velocity	Atoms and connected bonds are coloured according to the velocities of the atoms
custom	Atoms and connected bonds are coloured according to the custom colours set on individual atoms

16.6. Combination Rules

Combination rule equations as used when combining similar Lennard Jones parameters.

Table 16-6 Combination rules

Keyword	Description
arithmetic	The arithmetic mean of two values : $a = 0.5 * (b * c)$
geometric	The geometric mean of two values : $a = \sqrt{b * c}$
custom1	Custom combination rule
custom2	Custom combination rule

16.7. Drawing Styles

Available drawing styles for individual atoms are:

Table 16-7 Draw style keywords

Value	Description
stick	Atoms and connected bonds are drawn using simple lines
tube	Atoms and connected bonds are drawn using 'capped' cylinders
sphere	Atoms and connected bonds are drawn using spheres and cylinders, with all atoms the same size
scaled	Atoms and connected bonds are drawn using spheres and cylinders, with all atoms sized according to their atomic radius set in the preferences

16.8. Energy Units

A list of energy units supported by Aten

Table 16-8 Energy unit keywords

Value	Description	Joule Equivalent
j	Joules per mole (J/mol)	1.0
kJ	KiloJoules per mole (kJ/mol)	1000.0
cal	Calories per mole (cal/mol)	4.184
kcal	KiloCalories per mole (kcal/mol)	4184.0
K	Kelvin (K)	1.0 / (503.2166 / 4184.0)
ev	Electronvolts per mole (eV/mol)	96485.14925
ha	Hartrees per mole (Ha/mol)	2625494.616

16.9. Glyph Types

Available glyph types (drawing objects) are:

Table 16-9 Glyph type keywords

Value	Description
arrow	Simple arrow
cube	A cuboid
ellipsoid	An ellipsoid, oriented by face and edge vectors
ellipsoidxyz	An ellipsoid, oriented by an explicit set of provided axes

line	A line between two points
quad	A quad (made up of two triangles)
sphere	A sphere or spheroid
svector	A sense vector
tetrahedron	Regular tetrahedron
text	Text rendered at 2D screen coordinates
text3d	Text rendered at 3D model coordinates
triangle	Regular triangle
tubearrow	A fully 3D arrow
vector	An arrow centred on a point and pointing along a specified local vector

16.10. Grid Styles

Available drawing styles for surface/grid data:

Table 16-10 Grid style keywords

Value	Description
grid	All data points are drawn as simple dots at their coordinate positions (the cutoff value is ignored)
points	Data points above the cutoff are drawn as simple dots at their coordinate positions
triangles	A triangulated wireframe isosurface is drawn between points above the cutoff value
solid	A triangulated solid isosurface is drawn between points above the cutoff value

16.11. Grid Types

Available grid types:

Table 16-11 Grid type keywords

Value	Description
regularxy	Two-dimensional surface data on a regularly-spaced XY grid. The grid size must be specified on creation (with the <code>initGrid</code> command).
regularxyz	Three-dimensional volumetric data on a regularly spaced XYZ grid. The grid size must be specified on creation (with the <code>initGrid</code> command).
freexyz	Three-dimensional data not restricted to a regular grid. No grid size specification is necessary.

16.12. Label Types

Valid label types are as follows:

Table 16-12 Label type keywords

Value	Description
charge	Atomic charge currently assigned to the atom
element	Element symbol of the atom
ffequiv	Forcefield equivalent name from the assigned forcefield atom type (or the forcefield atom type name if no equivalent applies)
id	Integer atom ID
type	Assigned forcefield atom type name (if any)

16.13. Monte Carlo Move Types

Valid move types for the Monte Carlo minimiser are as follows:

Table 16-13 Monte Carlo move type keywords

Value	Description
translate	Molecule translations
rotate	Molecule rotations
zmatrix	Molecule z-matrix moves (not yet implemented)
insert	Insertion moves
delete	Deletion moves (not yet implemented)

16.14. Output Types

Valid output types (or debug modes) are as follows:

Table 16-14 Output type keywords

Value	Description
all	Enable output of all types listed in this table
calls	Print out entrances and exits to most subroutines to enable quick tracing of crash locations
commands	Trace execution of commands in command lists (e.g. filters) and print information on variable access paths
gl	Debug OpenGL calls and graphics capabilities as best as is possible
parse	Debug file-reading and argument parsing routines
typing	Print (lots of) information regarding setting and matching of atom type descriptions
verbose	Enable a little extra output (but not much)

16.15. Parse Options

These options determine how general parsing of plain text files proceeds, as well as controlling delimited argument parsing.

Table 16-15 Parse option keywords

Value	Description
noescapes	Treat backslash as a normal character
normalcommas	Create commas as normal characters rather than delimiters
skipblanks	Blank lines (or those containing comments) are automatically skipped
stripbrackets	Normal parentheses are automatically stripped from the input file
stripcomments	Remove comments from file (text starting with ‘//’ or '#')
usecurlyies	Data within curly brackets will be parsed as a single argument
usequotes	Phrases enclosed in quotes will be parsed as a single argument

16.16. Read Success Integers

Integer return values for many read/write operations.

Table 16-16 Read/write return values

Value	Description
0	Success - no errors encountered
1	The read/write operation failed before it was completed
-1	End-of-file was encountered during read operation

16.17. ZMapping Types

Table 16-17 Map type keywords

Value	Description
alpha	Convert based on the alpha part of the name only. Leading or trailing numbers and symbols are discarded. The alpha part is assumed to be an element symbol
auto	Attempts several of the other conversions of increasing complexity until a match is found
ff	Search through the names of atomtypes in currently-loaded forcefields, assigning elements based on matches found
firstalpha	Convert based on the first alpha part of the name only (i.e. until a non-alpha character is found). The alpha part is assumed to be an element symbol
name	The name is assumed to be an actual element name, and is converted to the relevant element number
numeric	Use the numeric part of the name as the element number
singlealpha	Convert based on the first alphabetic character encountered - useful only

when single-character element symbols are likely to be found (e.g. for pure organic CHON systems)

batch processing, 47, 48, 52, **53** --help, 49
Command Line Switches -i, 50
 --atendata, 46 --int, 50
 -b, 46 --interactive, 50
 --batch, 47 -k, 50
 --bohr, 46 --keepnames, 50
 --bond, 47 --keeptypes, 50
 -c, 47 --keepview, 50
 --cachelimit, 47 -m, 50
 --centre, 48 --map, 50
 --command, 47 -n, 51
 -d, 48 --nobond, 51
 --debug, 48 --nocentre, 51
 --dialogs, 48 --nofold, 51
 --double, 48 --nofragmenticons, 51
 --export, 48 --nofragments, 51
 --exportmap, 48 --noincludes, 51
 --expression, 49 nolists, 51
 -f, 49 --nolists, 51
 --ff, 49 --nopack, 51
 --filter, 49 --nopartitions, 51
 --fold, 49 --noqtsettings, 51
 --format, 49 --pack, 52
 -g, 49 --pipe, 52
 --grid, 49 process, 52
 -h, 49 quiet, 52

-s, 52	atoi, 264
--script, 52	atomStyle, 148
--string, 52	augment, 154
-t, 52	autoConversionUnit, 187
--trajectory, 52	autoEllipsoids, 199
-u, 52	autoPolyhedra, 199
--undolevels, 52	axisRotate, 275
-v, 53	axisRotateView, 282
vbo, 53	beforeStr, 265
--vbo, 53	bohr, 158
--verbose, 53	bondDef, 187
-z, 53	bondTolerance, 154
--zmap, 53	cell, 165
Command Line Switches:, 52	cellAxes, 165
commands	centre, 275
abs, 216	cgMinimise, 225
addFrame, 272	chain, 159
addGridPoint, 205	charge, 170
addHydrogen, 158	chargeFF, 170
addNextGridPoint, 205	chargeFromModel, 170
addPoint, 172	chargePAtom, 171
addReadOption, 242	chargeType, 171
adjustCell, 165	clearBonds, 154
afterStr, 264	clearCharges, 171
angleDef, 187	clearExportMap, 188
atof, 264	clearExpression, 188

clearLabels, 214
clearMap, 188
clearMeasurements, 219
clearPatterns, 237
clearPoints, 172
clearSelectedBonds, 155
clearTrajectory, 272
clearTypes, 188
contains, 265
converge, 225
copy, 177
cos, 216
createAtoms, 230
createDialog, 222
createExpression, 189
createPatterns, 237
createScheme, 240
currentAtom, 148
currentFF, 189
currentModel, 230
currentPattern, 237
cut, 177
debug, 269
defaultDialog, 222
delete, 177
deleteFF, 189
deleteModel, 231
deSelect, 253
deSelectF, 254
deSelectFor, 254
deSelectType, 254
disorder, 175
do, 182
dotProduct, 216
drillPores, 240
elec, 179
endChain, 159
eof, 242
equivalents, 189
error, 222
exp, 216
expand, 255
exportMap, 189
ffModel, 190
ffPattern, 190
filterFilename, 242
finaliseFF, 191
finaliseGrid, 205
finaliseModel, 231
find, 243
firstFrame, 272
firstModel, 231

fix, 148
fixType, 191
flipX, 276
flipY, 276
flipZ, 276
fold, 166
foldMolecules, 166
for, 182
frameEnergy, 179
frameForces, 198
free, 149
freeType, 191
ftoa, 266
generateAngle, 191
generateBond, 192
generateTorsion, 192
geometry, 219
getAtom, 149
getCombinationRule, 192
getEnv, 269
getEnvF, 269
getEnvI, 270
getFF, 193
getLine, 243
getModel, 231
getPattern, 238
getSite, 262
getView, 282
glyphAtomF, 200
glyphAtomR, 200
glyphAtomsF, 201
glyphAtomsR, 201
glyphAtomsV, 201
glyphAtomV, 201
glyphColour, 202
glyphData, 202
glyphSolid, 202
glyphText, 203
gridAlpha, 206
gridAxes, 206
gridColour, 206
gridColourScale, 207
gridColourSecondary, 207
gridCubic, 207
gridCutoff, 208
gridCutoffSecondary, 208
gridLoopOrder, 208
gridOrigin, 209
gridOrtho, 209
gridStyle, 209
gridUseZ, 210
gridVisible, 210

gui, 270
help, 270
hide, 149
if, 184
info, 232
initGrid, 210
interDef, 193
invert, 255
itoa, 266
label, 214
lastFrame, 272
lastModel, 232
lineTol, 225
listComponents, 175
listMeasurements, 219
listModels, 232
listPatterns, 238
listScales, 172
listScripts, 252
listSites, 262
ln, 216
loadFF, 193
loadGrid, 211
loadModel, 233
loadScript, 252
loadTrajectory, 273
locate, 160
log, 216
logInfo, 233
lowerCase, 266
map, 194
matrixConvert, 276
matrixTransform, 277
mcMinimise, 226
measure, 220
measureSelected, 220
millerCut, 166
mirror, 281
modelEnergy, 179
modelForces, 198
modelTemplate, 233
mopacMinimise, 226
move, 160
moveToEnd, 160
moveToStart, 160
newAtom, 161
newAtomFrac, 161
newBasisShell, 228
newBond, 155
newBondId, 156
newEigenvector, 228
newFF, 194

newGlyph, 203
newGrid, 211
newModel, 234
newPattern, 238
newSite, 262
newVibration, 228
nextArg, 243
nextFrame, 273
nextModel, 234
nint, 217
normalise, 217
null, 270
orthographic, 282
pack, 167
parentModel, 234
paste, 177
peekChar, 244
peekCharI, 244
perspective, 282
prevFrame, 273
prevModel, 235
printCell, 167
printElec, 180
printEnergy, 181
printEwald, 180
printf, 223
printForces, 198
printInter, 180
printIntra, 180
printSetup, 194
printSummary, 181
printType, 194
printVdw, 181
printZMatrix, 228
quit, 271
random, 217
randomI, 217
readChars, 244
readDouble, 245
readDoubleArray, 245
readInt, 246
readIntArray, 246
readLine, 246
readLineF, 247
readNext, 247
readVar, 248
readVarF, 248
rebond, 156
rebondpatterns, 156
rebondSelection, 157
recreateExpression, 195
redo, 178

removeCell, 168
removeLabel, 214
removeLabels, 215
removePoint, 173
removeReadOption, 249
removeStr, 267
reorder, 162
reorient, 278
replaceChars, 266
replaceStr, 267
replicate, 167
resetView, 283
return, 185
rewind, 249
rotateView, 283
rotX, 162
rotY, 162
rotZ, 163
runScript, 252
saveBitmap, 212
saveExpression, 195
saveModel, 235
saveMovie, 212
saveSelection, 235
scale, 168
scaleMolecules, 168
scaleName, 173
scaleVisible, 173
sdMinimise, 226
searchCommands, 271
seed, 271
seekFrame, 274
select, 255
selectAll, 256
selectF, 256
selectFFTType, 256
selectFor, 256
selectInsideCell, 256
selectionAddHydrogen, 163
selectionCog, 257
selectionCom, 257
selectLine, 257
selectMiller, 258
selectMolecule, 258
selectNone, 258
selectOutsideCell, 259
selectOverlaps, 259
selectPattern, 259
selectPores, 241
selectRadial, 260
selectTree, 260
selectType, 260

setAngle, 278
setCell, 169
setCharge, 150
setCombinationRule, 195
setCoords, 150
setDistance, 279
setElement, 150
setForces, 150
setFx, 151
setFy, 151
setFz, 151
setId, 151
setName, 235
setPoint, 174
setPointColour, 174
setPointValue, 174
setRx, 151
setRy, 152
setRz, 152
setTorsion, 279
setupComponent, 175
setVelocities, 152
setView, 283
setVx, 152
setVy, 152
setVz, 152
shiftDown, 163
shiftUp, 163
show, 153
showAll, 236
showDefaultDialog, 223
sin, 217
siteAxes, 263
skipChars, 249
skipLine, 249
spacegroup, 169
speedTest, 284
sprintf, 267
sqrt, 218
stripChars, 268
tan, 218
terminate, 241
toa, 268
torsionDef, 196
translate, 279
translateAtom, 280
translateCell, 280
translateView, 284
translateWorld, 280
transmute, 164
TypeDef, 196
typeModel, 196

typeTest, 196
undo, 178
units, 197
upperCase, 268
verbose, 223
viewAlong, 284
viewAlongCell, 284
while, 186
writeLine, 250
writeLineF, 250
writeVar, 251
writeVarF, 251
zoomView, 285
forcefield keywords
 angles, 313
 bonds, 314
 convert, 313
 data, 314
 defines, 315
 equivalents, 316
 function, 316
 generator, 317
 inter, 317
 name, 312
 torsions, 318
 types, 318
uatypes, 319
units, 312
NETA keywords
 any bond to, 329
 aromatic, 332
 bond, 330
 chain, 330
 double bond to, 330
 linear, 333
 n, 331
 nbonds, 331
 nh, 331
 noring, 332
 octahedral, 334
 onebond, 333
 planar, 332
 ring, 332
 single bond to, 329
 size, 333
 sqplanar, 334
 tbp, 334
 tetrahedral, 334
 trigonal, 334
 tshape, 333
 unbound, 333
Patterns, 15, **288**

Default, 288	FFAtom, 113
types	FFBound, 114
Aten, 105	Forcefield, 115
Atom, 106	Glyph, 117
BasisPrimitive, 107	GlyphData, 119
BasisShell, 108	Grid, 119
Bond, 108	Measurement, 120
Bound, 109	Model, 120
ColourScale, 109	MonteCarlo, 131
ColourScalePoint, 110	Pattern, 133
Dialog, 110	Prefs, 134
Eigenvector, 112	UnitCell, 138
Element, 112	Vector, 139
EnergyStore, 113	Widget, 140