Aten version 0.92
User's Guide

T. Youngs

November 25, 2007

Contents

0.1 Introduction

Aten is first and foremost an application to ease the creation of coordinates suitable for use as the starting configurations for molecular simulation codes, ranging from single molecules *in vacuo* to condensed-phase systems such as gases or liquids. Forcefield methods can be applied, for example, to minimise the energy of systems with respect to their coordinates and create 'good guesses' for initial configurations as input to *ab initio* codes. More so, multi-molecule models can be created with Monte Carlo techniques to provide periodic systems for use in molecular dynamics.

For the computional chemist, it is often the case that coordinates (new or existing) need to be edited in a specific way to create the configuration necessary to perform the job at hand, and in the opinion of the author, this is a job not handled well by current the current stock of free codes available. Aten was initially created as a method of learning various programming, visual, and algorithmic techniques, but has grown into an application suitable for use in many situations by the simulation community.

Comments and suggestions on the facilities available (or not) in the code are welcome and will be warmly accepted. If a feature is missing or inadequate and of general applicability, I will most likely include it in the program. Please don't hesitate to write to me at the address provided below. In particular, if you take the time to write a file filter for use with your favourite code I would be happy to include as part of the distribution.

Aten is released under the GPL, meaning you may copy, change, and distribute it as you see fit, provided you acknowledge the original copyright. If, however, you have a burning desire to improve or extend the code in some way, and wish to do some under your own steam, I would be happy to consider including your modifications into the official stream. Contact me to discuss your brilliance at interpreting my code!

Tristan Youngs
9$^{\text{th}}$ November 2007
t.youngs@qub.ac.uk

Chapter 1

Installation

1.1 *Disclaimer*

Aten has been compiled successfully on Linux machines running SuSE 9.3 (with the GTK gui), Open-SuSE 10.2, and RedHat Enterprise 4, an Intel iMac running OS X 10.4.10, and XXX. No doubt with (or without) tweaking it will run on other systems.

Aten is in development, and as such might contain a bug or two. Should you come across a crash or an issue that needs to be fixed, *please* report them so they can be fixed.

In addition, I can't accept responsibility for any loss of data / work / sleep / expensive computers / cats / coordinates etc., use at your own risk etc., and so on...

1.2 Ingredients for Linux

- GNU Autotools 2.6
- A C++ compiler (e.g. g++, icc)
- OpenGL and glut (e.g. from freeglut)
- pkg-config
- GTK+2.0 (version 2.4 or above) or Qt4 (version 4.2 or above) if you want a GUI
- GtkGLExt if you want the GTK+ GUI

1.3 Ingredients for Mac OS X

- GNU Autotools 2.6
- A C++ compiler (e.g. g++, icc)
- OpenGL and glut frameworks
- pkg-config (if using Fink or MacPorts versions of GTK+ or Qt)
- GTK+ (version 2.4 or above) or Qt (version 4.1 or above) if you want a GUI

1.3.1 Which Qt4?

To build against the qt4-x11 package available *via* Fink run:

    configure --with-qt=fink ...

This will set most of the necessary environment variables, namely:

- QTGUI_CFLAGS='-I/sw/lib/qt4-x11/include -I/usr/X11R6/include'

- QTGUI_LIBS='-L/sw/lib/qt4-x11/lib -lQtGui -lQtCore -lQtOpenGL'

- QTOPENGL_CFLAGS='-I/sw/lib/qt4-x11/include'

- QTOPENGL_LIBS='-I/sw/lib/qt4-x11/lib'

You may also need to direct the build process to the correct Qt4 development binaries either by setting your $PATH to '/sw/lib/qt4-x11/bin:${PATH}' or by using the --with-qtdir option to configure.

The native Qt4 package is available as a disk image installer at no cost directly from http://trolltech.com. Specify that the build use the frameworks provided instead:

    configure --with-qt=framework ...

1.3.2 Autotools

You may find that the version of GNU Autotools you have is not recent enough. Check with autoconf --version - 2.59 is not sufficient! An up-to-date version can be installed via Fink. Known good versions are listed here, but don't assume that these are the only versions that will work:

    libtool : 1.5.22-1000
    autoconf : 2.60-4
    aclocal : XX
    automake : 1.9.6-3
    m4 : XXX

1.4 Configuration Options

Possible configure options are listed here. Most are present just to work around compilation issues, but some provide functionality in the program (in particular, the choice of GUI to build in).

--with-gl-includedir Sets the location of the OpenGL headers.

> **e.g. --with-gl-includedir=/usr/include** Sets the location of the OpenGL headers (GL/GL.h etc.) to /usr/include.

--with-gui Specifies GUI to compile with the program. Valid options are:

> **--with-gui=none**
> No GUI should be compiled (i.e. create a command-line only version)

> **--with-gui=gtk**
> Compile the legacy GTK+ GUI.

> **--with-gui=qt**
> Compile the Qt4 GUI (default).

--with-qt Specifies which installation of Qt4 to use on Apple Macs. Valid options are:

> **--with-qt=framework** Use the Qt4 framework provided by TrollTech's own installation.
>
> **--with-qt=fink** Use the Fink installation of Qt4 (qt4-x11).

--with-qtdir Sets the location of the Qt4 development tools (moc, uic, and rcc). This should only need to be set if you have Qt3 installed and your $PATH favours the older version's binaries.

> **e.g. --with-qtdir=/usr/local/bin** Sets the location of the Qt4 development tools to /usr/local/bin.

1.5 Four½Step Installation

Step 1:
> Get the source.
>
> - To download the tarball: go to http://aten.googlecode.com and select the relevant file from the Downloads page.
> - To get via svn: run 'svn co http://aten.googlecode.com/svn/trunk aten' to download the current tree to a new directory 'aten' in the current directory.

Step 1a:
> Set up the build (if you downloaded the source with svn)
>
> - Run ./autogen.sh to generate the configure script..
>
> Common issues:
> *(MAC) Running ./autogen.sh, autoconf 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 around line 89 with the 'PKG_CHECK_MODULES(GTK28, ..., [AC_DEFINE...' command).
>
> *(MAC) Running ./autogen.sh, aclocal '/sw/share/aclocal/pkg.m4:5: warning: underquoted definition of PKG_CHECK_MODULES'.*
>
> - If there is no pkgconfig installed, this is likely to give rise to said spurious error. Install pkgconfig to proceed.
>
> *(MAC) autogen.shm seems to work, but aclocal complains about 'underquoted definition of AM_PATH_*'.*
>
> - These warnings should nothave affected the generation of a working ./configure script. So you may as well move on to the next step.

Step 2:
> Configure the build (Run ./configure)
>
> - Most all dependencies should be detected by the scripts provided, ensuring a pain-free build (or quick detection of problems).
> - Select a GUI. Running ./configure with no options detects the operating system and builds in the Qt GUI. This can be overriden with the --with-gui switch:
>   - ./configure --with-gui=qt builds in the Qt GUI (default).
>   - ./configure --with-gui=gtk builds in the legacy GTK+ GUI.
>   - ./configure --with-gui=none builds a command-line only executable.

Common issues:

configure complains "Could not find GLUT header. Is freeglut/Mesa installed?"

- You need some form of OpenGL implementation installed. Try the freeglut packages, and don't forget to install the freeglut-devel package as well.

Step 3:

Compile the source (Run make)

- Once configured successfully, run make to compile the source and build the program.

- Go make some tea or brew some coffee.

Common issues:

When linking I get "Undefined references to 'pango_x_font_subfont_xlfd'" (or similar).

- The version of gtkglext you have installed is a bit retro (version 1.0.6 uses tokens from the pango subsystem that have since been made obsolete) so download and install version 1.2.0 from gtkglext.sourceforge.net.

When the Qt GUI is building, I get a version error for moc along the lines of 'uic: File generated with too recent version of Qt Designer (4.0 vs. 3.*.*)'.

- If you have both Qt4 and Qt3 installed $PATH is often set so that the Qt3 binaries are found first. Reconfigure the build with configure --with-qdtir=*path* where *path* is the location of the Qt4 binaries moc, uic, and rcc.

Step 4:

Install (Run make install or cp src/aten <destination>)

- After compilation is complete, you're left with the Aten executable in the src/ directory.

- Either run make install to place the program in /usr/local/bin if you have root priveledges, or copy the program to a local location of your choice and run it from there.

- The default stock of filters and spacegroup definitions is installed in the default locations if you run make install (on Linux systems this is typically /usr/local/share/aten).

- Aten is directed to this location via the $ATENDATA environment variable which should be set before running the program.

- If you haven't (or can't) run make install then set $ATENDATA to the /data directory in the source tree.

Chapter 2

Quickstart

2.1 Startup

Run from the command line without arguments, Aten starts up with the full GUI. If model files are given as arguments these are all loaded in to Aten, provided they are of a compatible format. Whether a model file is readable and/or writable by Aten is determined by the current stock of filters available to the program (see **??**). These are read from the location pointed to by $ATENDATA and, if found, from $HOME/.aten/filters. In addition, a user preferences file named 'prefs.dat' may exist in $HOME/.aten and control many aspects of the program (in fact, the preferences file is treated as a script file – see **??**).

Many command-line options exist to allow for immediate loading of models, scripts, forcefields etc., and to run the program in batch mode etc. See Section 10 for a full list.

The following section refers to the Qt interface.

2.2 Main Window

Most of the main window is taken up with a canvas which is used to display and edit models and take input from the user. Each of the mouse buttons has a different action on the canvas, each of which can be set to the users taste in the preferences (menu item Settings→Preferences on Linux). In addition the Shift, Ctrl, and Alt keys modify or augment these default actions. See the following sections for brief descriptions of the different modes. At the foot of the window is a message box (where output and errors are thrown) and a status bar reflecting the content of the current model displayed, 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 of the model (if it is periodic).

A stack of icons on the right-hand side of the window provides quick access to different tool panels for building, analysing etc. See the relevant sections of the manual for descriptions of each of these panels.

2.3 Changing the View

At its most basic the canvas 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 molecule in the plane of the screen (right-click and hold on an empty area of the canvas and move the mouse to rotate the model) and the mouse wheel zooms in and out. Note that right-clicking on an atom brings up the atom menu (see Section **??**). The middle mouse button translates the model in the plane of the screen – again, click-hold and drag. Rotation and translation operate on the position and orientation of the camera and no modifications to the actual coordinates of the model are made. 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)

and the View toolbar 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 centre of rotation is the geometric centre of the selected atoms.

2.4   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. Furthermore, single-clicking on a selected atom while holding Shift will deselect the atom.

To summarise mouse control, out of the box the standard settings are:

Table 2.1:

Button	Modifier	Action
Left	None	Click on individual atoms to select exclusively
		Click-hold-drag to exclusively select all atoms within rectangular region
		Double-click to show atom list
	Shift	Click on individual atoms to toggle selection state
		Click-hold-drag to inclusively select all atoms within rectangular region
Right	None	Click-hold-drag to rotate camera around model
		Click on atom to show atom menu
	Ctrl	Click-hold-drag to rotate selection in local (model) space
Middle	None	Click-hold-drag to translate camera
	Ctrl	Click-hold-drag to translate selection in local (model) space

2.5   Load it!

If the names of model files are supplied on the command line, all are loaded into the workspace, their file formats determined by file extensions and/or content and parsed accordingly. If none of the supplied files can be loaded, the program will exit without starting the GUI. The GUI can be suppressed for when batch command-line operation is required.

Aten uses 'filters' to achieve import and export of model file types, potentially providing the ability to load and save any format of model data available. A filter consists of a series of commands describing the layout of the data in a given file type, and may be thought of as a small, simplified C program typically of a few tens of lines in size. Thus, filters may be written by the user to enable the import/export of data in formats tailored to the need of the individual, or to read proprietary formats not in common usage within the community. See section 8 for full details on using and writing filters.

All input files (the preferences file, forcefields, scripts etc.) for Aten are entirely free-format. Comments (lines beginning with a hash '#') are always ignored, as are blank lines (or lines containing only delimiters) in the majority of cases. Valid delimiters between datum are spaces, commas, and tabs, although all this behaviour may be overridden by the use of double or single quotes to specify, for example, filenames or titles that contain any of the delimiters. For model loading, any or all of these rules may

not apply since specific formatting may bypass any or all of these rules.

Chapter 3

Features

Aten was designed with the needs of the molecular dynamics user in mind, but is more than likely to be useful in more general situations as well. At its simplest, it is (another) visualiser – at its most complex, it is a forcefield-based tool for the creation and editing of gas- and liquid-phase configurations for use in computational software, offering an advanced atom typing system, energy minimisation, Monte Carlo methods for the creation of disordered systems, and energy / force calculation.

In a nutshell:

Load, display, save (images or coordinates) model files.

Draw, edit, add, delete, move, position, and rotate atoms.

Display, add, and edit unit cells (includes crystal packing).

Load and apply atom typing and forcefield terms to models.

Calculate energies / forces.

Geometry minimise structures.

Detailed feature information:

Visualising

Visualise molecules / systems / trajectories in glorious line, tube, sphere and scaled sphere styles (or an arbitrary mix).

Play movies of molecular dynamics trajectories.

Save image snapshots.

Chapter 4

GUI

Chapter 5

Forcefield Expressions

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.

On the way to generating an expression, several preliminary steps are performed:

1. Detect molecule patterns for efficient usage of forcefield terms

2. Augment bonds within (otherwise chemically 'correct') molecules

3. Find rings and assign aromaticity
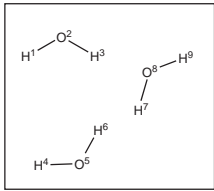
4. Assign atom hybridisation

The first serves to enable the creation of optimal forcefield expressions for systems containing many copies of the same molecules (e.g. liquids), while the last three provide additional data beyond basic topology parameters for use in atom typing. All are performed automatically unless you specifically request them not to be (see Section XXX). The most important of these is the pattern description of the system, without which the expression cannot be created.
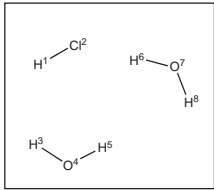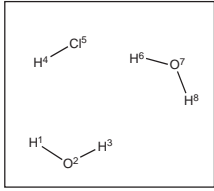
5.1 Patterns

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 subexpressions 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 the molecule.
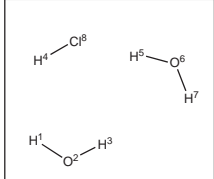
From the atoms and the basic connectivity between them Aten will automatically determine the pattern of systems of arbitrary complexity. The ordering of atoms is important, however, and the atoms belonging to a single molecule must not be interspersed by those from others. In other words, if a bond 'crosses an atom' that doesn't belong to the molecule, then the benefits of fine-graining the expression in this way is lost somewhat. 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:

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 H_2O. Obvious, huh? The two-component system

1.

Atom Order: H^1 O^2 H^3 H^4 O^5 H^6 H^7 O^8 H^9
Pattern: $H_2O(3)$

2.

Atom Order: H^1 Cl^2 H^3 O^4 H^5 H^6 O^7 H^8
Pattern: $HCl(1)$ $H_2O(2)$

3.

Atom Order: H^1 O^2 H^3 H^4 Cl^5 H^6 O^7 H^8
Pattern: $H_2O(1)$ $HCl(1)$ $H_2O(1)$

4.

Atom Order: H^1 O^2 H^3 H^4 H^5 O^6 H^7 Cl^8
Pattern: $H_5O_2Cl(1)$

illustrated in 2) and 3) is described by two different patterns since, in 3), the two water molecules are 'separated' by the hydrogen chloride. Thus, we end up with a three-pattern description to describe the contents of the system as opposed to the optimal two-pattern description, but remember that both are acceptable and will give the same total energies and atomic forces – only the partitioning of the molecules has changed, and their order is not important (the actual positions of the atoms of course remains constant). This illustrates the point that molecules of the same type need not exist next to each other in terms of the overall atom list, but that the optimal pattern description will not be possible. 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) – there are four ways to describe the three separate molecules:

1. $H_2O(3)$

2. $H_2O(1)$ $H_2O(1)$ $H_2O(1)$

3. $H_2O(2)$ $H_2O(1)$

4. $H_2O(1)$ $H_2O(2)$

Again, 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.

Recall that bonds 'crossing atoms' will prevent the optimal description of atoms, but now consider that patterns can 'cross molecules'. For instance, consider again the example 1) above. Three further (again, reiterating the point, equivalent) ways of writing the pattern description are:

1. $H_6O_3(1)$

2. $H_4O_2(1)$ $H_2O(1)$

3. $H_2O(1)$ $H_4O_2(1)$

Here, we have encompassed individual molecular entities into supermolecular groups, and as long as there are no bonds 'poking out' of the pattern element, 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 two valid patterns for this arrangement of atoms:

1. $H_5O_2Cl(1)$

2. $H_2O(1)$ $H_3OCl(1)$

Note that, when automatically creating patterns, if Aten stumbles across a situation like this, it will assume the default pattern of one supermolecule (the whole system of atoms), i.e. X(1) where 'X' is all atoms. This will work, but will result in rather inefficient calculations.

5.2   Creating the Expression

For each element in the pattern the required intramolecular terms and atom types are determined for the representative molecule of the element. Atom typing is a complex process and is discussed in detail in Section 6. For now, let us assume that each atom in the molecules of each pattern element has been successfully assigned a forcefield type (if this cannot be achieved, then the expression cannot be created). Once Aten has convinced itself that every atom has type, a skeleton list of intramolecular terms (currently only bonds, angles, and torsions are supported) necessary to describe the element molecules is constructed and subsequently filled ('fleshed out') with parameters. These parameters come either from the forcefield associated to the model as a whole, or from individual forcefields associated to individual elements of the pattern (see Section XX), based on the assigned types of the atoms. If one or more suitable terms

cannot be found in the forcefield(s), energies and forces cannot be calculated. Admittedly, this is a little Draconian, and will be addressed in a future release.

Once constructed, a complete expression persists until a change is made to the model which is potentially invalidates it. For example, moving atoms around is acceptable, but changes in bonding are not. Nor is adding or deleting atoms, since this obviously requires regeneration of the pattern description. In such cases, the expression is marked as out-of-date and will be automatically recreated when it is next required.

Chapter 6

Atom 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 one of the key tenets of forcefield writing. That is, as large a number of different molecules and types of molecule should be described by a simple set of different atoms, i.e. atom *types*. At the simplest and most common level, the connectivity of atoms completely describes the different types it may (or may not) be.

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 rether impractical. Others use a typing 'language' to describe individual elements of the topology of atom, and are flexible and 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 extremely useful tools for selecting specific atoms as well.

6.1 Language Basics

Type descriptions in Aten use connectivity to other elements 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, and may be re-used in other atom's type descriptions to simplify the identification of these atoms. Time to jump straight in with some examples.[1]

6.1.1 Example 1 – Water
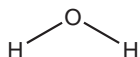
Consider a water molecule. If you were describing it in terms of its structure to someone who understands the concept of atoms and bonds, but has no idea what the water molecule looks like, you might say:

> A water molecule contains an oxygen that is connected by single bonds to two hydrogen atoms.

To describe the atoms in the grand scheme of the water molecule, you might say:

> A 'water oxygen' is an oxygen atom that is connected to two hydrogen atoms *via* single bonds.

---

[1]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 **??**), and certainly aren't the only ways of writing the descriptions. They're just plain 'ol examples of the language!

...and...

> A 'water hydrogen' is a hydrogen that is connected *via* a single bond to an oxygen atom, which is also connected by a single bond to another 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 as being a 'water oxygen', when in fact it is quite different. We should extend the description in this case to:

> A 'water oxygen' is an oxygen atom that is connected to two hydrogen atoms *via* single bonds, and nothing else.

An atom description in Aten is a string of comma-separated commands that explains this kind of criteria. So, to tell the program how to recognise a water oxygen and a water hydrogen, we could use the following type descriptions:[2]

```
1 OW O "nbonds=2,-H,-H" # Water oxygen
2 HW H "-O(nbonds=2,-H,-H)" # Water hydrogen
```

Aten reads this literally as "A water oxygen has exactly two bonds AND is bound to a hydrogen AND another hydrogen" and "A water hydrogen is bound to an oxygen that; has two bonds to it, is bound to a hydrogen, and is bound to another hydrogen". The '–X' is short-hand for saying 'is bound to X', 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.

6.1.2   Example 2 – 3-hydroxypropanoic acid
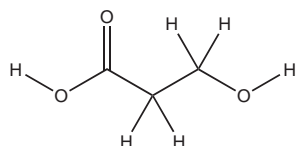
Assuming that the OH group in the carboxylic acid functionalisation will have different forcefield parameters to the primary alcohol at the other end of the molecule, here we must describe the first and second neighbours of the oxygen atoms to differentiate them.

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.

Where all three carbons need to be identified separately, we may write:

---

[2]This and all following type descriptions are written in the proper forcefield input style (see section XXX). Briefly, the line consts of a unique type id number, a type name, the element symbol, and then the type description in double-quotes. Comments have also been added to describe the types.

```
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:
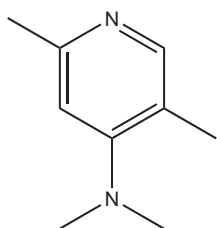
```
10 OH O "-H,-C(-H,-H)" # Alcoholic O
11 O_cbx O "-C(-O(-H))" # Carboxylic acid =O
12 OH_cbx O "-H,-C(-O(bond=double))" # Carboxylic acid O(H)
```

Of course, we could just have specified "nbonds=1" for the doubly-bound oxygen of the carboxylic acid group, but that wouldn't be very instructive, would it?

6.1.3 Example 3 – N,N,2,5-tetramethylpyridin-4-amine

Ah, now here we have 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
```

XXX

6.2 Command Reference

All type descriptions in files should be enclosed in double-quotes to prevent delimited parsing of strings. Commands may be separated by spaces or commas, but type descriptions may not span more than one line.

'-' (bound to) Specifies a connection to another element, but not the exact type of the connection itself (i.e. the bond type, which may be specified with the 'bond' keyword in brackets after the element specifier). Immediately following the '–' should be either an element symbol or a type id (the latter must be preceeded by a distinguishing '$'). Lists of the elements/type ids/type names may be given in square brackets, separated by commas or spaces, allowing multiple possibilities for matching the connection.

Examples:

bond, sp2, sp3, sp, unbound, geometry, ring (size)

XXX Repetition of descriptions - water example above. The dollar sign.

XXX Type scoring

XXX Art of type writing

XXX Order of atom descriptions

Chapter 7

Forcefields

7.1   File Format

7.2   Functional Forms

Chapter 8

Filters Handbook

Filters are essentially just a series of text commands describing the format and structure of data in an arbitrary file. Plug-ins, if you like. While this might seem needlessly complex and a lot of extra work, for a little effort your (personal or enforced) favourite formats can be added to the program. The reward is a set of coordinates that can be thrown straight in to your code of choice. No support for writing out commands relevant to the running of calculations is provided, for three principal reasons:

1. Input to codes is complicated, and I don't want to spend my evenings putting down endless combo boxes and spin buttons to cater for every basis set / method / obscure sub-option available.

2. Putting commands in is best left for the user to do by hand, since there's no better way to learn the ins and outs of a code than to write commands, make mistakes, and be forced into the manual.

3. The focus of Aten is on generating coordinates, pure and simple.

But enough ranting! A filter contains (in order to be useful) sets of commands which describe how a file should be imported or exported from the program, and encompasses model files, forcefields, and trajectory files. However, different filters for the same file type may be provided if necessary, each performing a slightly different set of import or export commands (if it is convenient not do to 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.

Filters are stored in one of two locations – in a stock common to all users (as determined by the –prefix variable, or in the default location), and in $HOME/.aten/filters. In both locations the filters read in by the program must be listed in the 'index' file – if you create your own, you must add them to the index file for them to be loaded in on startup.

Read on and discover how to enable input / output of 'stuff' as you see fit. The following sections discuss the different aspects of filters and the types of file content that filters can be applied to, including examples. Although many of the commands are common to all of them, examples are good!

8.0.1 Sections

Commands to read / write different things are split up into sections within a filter file. You can specify as many filters as you like in any one file, each acting on a different set of models / forcefields / trajectories, etc., but it probably makes sense to have one file per model format where possible. All filter commands must be givin within one of these sections.

importmodel
> Describes how to read model data from files, including atoms, cell and spacegroup data, and bonds. No variables are prepared by the command. Variables used after the command has completed:

title
> Used to set the new title of the model.

sg.group **and** *sg.setting*
> Set the spacegroup and optionally the spacegroup setting so crystal packing can be performed.

exportmodel
> Write model data to files, including atoms, cell and spacegroup data, and bonds.
> Variables prepared by the command:

title
> Title of the model.

natoms
> Number of atoms in the model.

cell.type
> Geometry of unit cell, one of 'none', 'cubic', 'orthorhombic', or 'parallelepiped'.

cell.x, *cell.y*, *cell.z*
> Lengths of the principal cell axes (if one is defined).

cell.alpha, *cell.beta*, *cell.gamma*
> Angles between the principal cell axes (if one is defined).

cell.xx, *cell.xy*, *cell.xz*

cell.yx, *cell.yy*, *cell.yz*

cell.zx, *cell.zy*, *cell.zz*
> Vector components of the unit cell matrix (if one is defined).

importgrid
> Describes how to read gridded volumetric data from files. No variables are prepared or used by the command.

exportgrid
> Describes how to write gridded volumetric data to files.

trajectory Read trajectory files. Note that, since trajectory files often consist of a short descriptive header followed by frames listed end to end afterwards, Variables prepared by the command:

natoms
> Number of atoms in the model.

cell.type
> Geometry of unit cell, one of 'none', 'cubic', 'orthorhombic', or 'parallelepiped'.

frame Read trajectory frame.
> Variables prepared by the command:

natoms
> Number of atoms in the model.

celltype
> Geometry of unit cell, one of 'none', 'cubic', 'orthorhombic', or 'parallelepiped'.

field Write forcefield dsecription of current model.
> Variables set by the command:

energyunit
> The current internal energy unit of the program, and hence the units used in the forcefield parameters.

title
> Current title of the model.

npatterns
> Number of different molecular species (pattern elements) in the model.

8.1 Recognising Files

Filters need a name. They are lonely creatures, and as such seek an identity. Although it can appear at more or less any point in the filter description, it usually makes most sense to name it early on. Consequently, the first line in most filters will be a short name describing what type of files the filter acts on:

```
name "ChikaanMol Coordinates File"
```

This name will appear in the filter list of file dialogs in the GUI, and also in the program output when reading / writing files of the type. The next obvious question might be "How does the filter actually *recognise* files of this type?" First off, there's the traditional way by filename extension:

```
extension chik
```

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 the filters. So, the example here would match all files ending with '.ucf'. Sometimes one extension is not enough and there might be others that are used for the same type of file:

```
extension "chik,egg,duk"
```

Commas separate extensions, and the double-quotes are important to prevent Aten from interpreting the list as a list of arguments instead. Note that the comparison of filename extensions is case-insensitive, so '.chk', '.EGG', and '.dUK' will all be matched by this filter.

Occasionally (and annoyingly) files have no extension at all, instead having short, fixed names. These exact names can be matched by the filter:

```
exact config
```

Exact filename matches are also case-insensitive, and lists of potential matches can be given, as before:

```
exact "config,configuration,coordinates"
```

Thirdly, the type of file can be matched by the contents of the file, 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:

```
within 10 "ChikaanMol version 2.2"
```

This states that if the phrase "ChikaanMol version 2.2" can be found somewhere within the first 10 lines of the file, then the filter is a match. One or more of the above criteria can be given, but it only takes a match with one of them to determine the type of file – that is, they are not logically ANDed together.

All of the above is relevant to filters that import data. For exporting, 'exact' property is unused, but all others are relevant. When saving data without the aid of the GUI, the desired filter can only be specified through the use of its nickname to avoid ambiguity.

```
nickname chikaan
```

Used in an 'exportmodel' section, this allows one to specify the saving of data in the ubiquitous ChikaanMol coordinates format. In summary, use 'name', 'extension', and 'filter' in all sections, 'exact' (optionally) for filters that import data, and 'nickname' for filters that export data.

8.1.1 Filter Partners

When filters have been provided that allow both the import and export of model data it is prudent to associate the pair together so that Aten knows how to immediately save data that it has loaded in. Each filter has a numerical ID associated to 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 loading with import filter ID 7 and then saving with export filter ID 7 should lead to exactly (or nearly) the same file. If the ID for a filter is not set it defaults to -1, and it is assumed that no partner exists.

```
id 7
```

IDs are local to specific filter sections, so they may be reused for different types of import / export. So, for example, a grid import / export filter pair may also have use ID of 7.

8.2 Getting Data – Format Strings

So, now the file has been recognised, what happens next? We either read data in, or write data out. Enter 'format strings'. A lot like a hybrid between FORMATs in Fortran and C/C++ $printf$ formats, these provide the interface to pass data between files and the program, and the sequence of commands is almost exactly like you would do in either of these programming languages. But for the uninitiated, don't panic! It's dead easy. Of course, in either language you use variables to store the data in the interim, and within filters this is no different. The basic (sub)processes in a filter are:

1. Read data from a file.

2. Split (parse) data into a number of variables.

3. Tell Aten to act on the current values of variables.

Variables thus form the bulk of the interface layer between numbers in a file, a filter to read them, and a program to use them (read–parse–act model). There are a handful of what you might call 'recognised' variables that mean something to the program in some way or other, but you are free to define your own at will to aid in the process of deconvoluting atomic data in awkward formats. See Section ?? for a long waffle about variables in Aten. For now it is enough to know that variables are either generic character strings (converted to real or integer numbers as and when necessary) or special types such as, for example, references to atoms or patterns within the model. They are not declared at the beginning of the filter, instead being dynamically created and typed when they are required. All must begin with a '$' symbol to distinguish them from commands or plain text arguments to commands.

To demonstrate how these variables are used, lets assume that you have a file which is, for argument's sake, a coordinates file that has in it the following data:

```
Na      0.0     1.0     0.0
Cl      1.0     0.0     0.0
Na      0.0    -1.0     0.0
```

Simply, elements and coordinates. Assuming that there's a filter command called 'readline' (there is) that reads in a line from a file and splits up the data on the line into a list of variables that you provide, if you were writing a command to describe one line of this data you would just say:

```
readline "$e $r.x $r.y $r.z"
```

The quoted part of the command is considered as the $formatstring$, and each element within it denotes a variable. After the command has completed, within the filter you can access the first item on the line that was read in by referencing the variable 'e', the second with 'r.x', and so on. These variables are all properties of an atom, but how do they actually become an atom? Well, these four are examples of so-called 'recognised' variables – ones that the program looks for when storing or retrieving data (which

depends on whether the filter itself is importing or exporting data). The command to create a new atom in the model we're currently reading in is 'addatom', which, when run, looks for a number of specific variables from which to take data to set the properties of the new atom. In our example, it will find the variables 'e', 'r.x', 'r.y', and 'r.z' and use them to set the element number and coordinates (it will also look for other variables which we have **not** set to any values, but won't complain when it can't find them). The situation is similar for other aspects of the model (*e.g.* the unit cell), as well as for when we are writing data to a file instead of reading it – the reverse process occurs, where we select an atom from the model (which sets the relevant variables) and then write out data using a format string containing these variables.

Lists of variables used or set by the various filter commands are given in the command reference in Section 8.3. Also, see the chapter on variables (Section **??**) for a full description and for details on how to specify the character length to read in, etc.

8.2.1 Example – XMol XYZ

As always, things are best illustrated with an example. Take the XYZ format as popularised by XMol – multiple models may be contained within a single file, each having a fixed structure with three basic data records:

1. Number of atoms (1 line) – Integer value specifying the number of atoms (N).

2. Title (1 line) – Long name or description of the system.

3. Atom data (N lines) – Atom data, one per line, usually containing an aotomic element and its coordinates, but occasionally the atomic charge as well.

For example:

```
3
Water
O    0.271   0.912   0.961    -0.8
H    0.978   1.244   1.295     0.4
H    0.126   0.149   1.307     0.4
```

The corresponding filter can utilise the number of atoms provided in the file to read in the atom data, but this is not a general requirement since arbitrary loops can be defined to read in data until a certain point in the file (e.g. until a certain string is found, or simply until the end of the file is reached). The fully-specified filter (providing both import and export capabilities) for this type of file can be written as:

```
import
  name "XMol XYZ"
  extension xyz
  nickname xyz
  glob *.xyz
  readline "$natoms"
  readline "$title@100"
  repeat $n $natoms
    readline "$e $r.x $r.y $r.z $q"
    addatom
  end
  rebond
end
export
  name "XMol XYZ"
  glob *.xyz
  nickname xyz
```

```
  writeline "$natoms"
  writeline "$title"
  foratoms $i
    get $i
    writeline "$i.e@3 *@4 $i.r.x@12 $i.r.y@12 $i.r.z@12"
  end
end
```

To begin, a descriptive name of the filter (*name "XMol XYZ"*), the common filename extension of the format (*extension xyz*), and a shell-style glob for the GUI file dialogs (*glob *.xyz*) are provided, giving the filetype description of the format. One post-load request is made, *rebond*, which indicates that bonds should be calculated for the model once all atoms have been loaded. Following this the list of commands to run when importing the file is given, beginning with the keyword *import*, and terminated with a matching *end* command. Recalling the structure of the xyz format, we expect to find the number of atoms on the first line, followed by the title of the model on the next, and then the atom data one per line after that. So, step-by-step the actions of the commands in this section are as follows:

readline "$natoms"

> The *readline* command reads a line from the file and parses it according to some *format string*. Many commands rely on the use of format strings to interpret and store data. For now, we just appreciate that in this case the first item found on the line read from the file is placed in a variable named 'natoms', which can then be used elsewhere.

readline "$title@100"

> Similar to the first command – here we take the first 100 characters from the line and store it in a variable named 'title' – here, 'title' is a variable name recognised internally by the program, and will be used to set the name of the model after import has finished.

repeat $n $natoms

> The repeat loop runs a section of commands for a specified number of iterations (storing the value of the current iteration in a variable called 'n'). In this case the number of iterations is determined by the value of the variable *natoms* to allow us to read in the expected number of atomic data.

readline "$e $r.x $r.y $r.z $q"

> Here, the format string given to the readline command contains five parameters in line with the expected ordering of data in the XMol XYZ format. That is, the element symbol followed by the atomic coordinates, and then (possibly) an atomic charge.

addatom

> At this point the current values of the temporary atom variables in the filter (in this example only the element, atomic coordinates, and atomic charge have been set) are saved in a new atom created in the model space.

end

> Signals the end of the repeat loop.

rebond

> Automatically calculate bonds between the atoms of the model.

XXX

8.3   Filter Commands

8.3.1   File Description

Definitions of the target file type and how to recognise it. Should be supplied in each filter section defined in a file.

extension *list*

Sets the filename extension that identifies files to be read / written by this filter.

- *e.g.* 'extension xyz' means that files with extension 'xyz' will be recognised by this filter.
- *e.g.* 'extension ''xyz,abc,foo''' means that files with extensions 'xyz', 'abc', and 'foo' will be recognised by this filter.

exact *list*

Defines one or more exact filenames that identify files to be read / written by this filter.

- *e.g.* 'exact coords' associates any file called 'coords' to this filter.
- *e.g.* 'exact ''results,output''' associates any files called 'results' or 'output' to this filter.

filter *glob*

Sets the file dialog filter extension to use in the GUI to the shell-style glob *glob*.

- *e.g.* 'glob *.doc' filters any file matching '*.doc' in the relevant GUI file selector dialogs.

name *name*

Sets the long name of the filter to *name*, to be used as the filetype description of files identified by the filter.

- *e.g.* 'name ''SuperHartree Coordinates File'''.

nickname *nick*

Sets the nickname of the filter to *nick*, which allows the filter to be identified in scripts.

- *e.g.* 'nickname shart' sets the nickname of the filter to 'shart'.

8.3.2 Reading / Writing Data

Commands to parse data from files and variables. Note that the style of the formatting strings differs between read and write commands.

readline *format*

Reads in the next line of the file (i.e., up to the carriage-return / newline marker) and splits up the data it finds into the variables specified in the *format*. The format string contains variables with optional length specifiers, separated by delimiters (e.g. spaces, commas) which serve only to separate variables – they do not represent actual whitespace between data items in the file.

- *e.g.* 'readline ''$n $m $title''' puts the first three whitespace-delimited items found on the next line in the file into the variables n, m, and *title*.

writeline *format*

Constructs a string according to the *format* given and writes it out to the file. A newline is automatically appended to the end.

- *e.g.* 'writeline ''NATOMS = $natoms''' outputs a new line to the file, looking like 'NATOMS = 20'.

print *format*

Print a message to the screen, optionally containing variables. Here, whitespace has its usual meaning since it is a simple text string. Variables to print out are written in the standard way, beginning with '$' and optionally having a length specifier. Individual variables are assumed to be terminated by either whitespace or the end of the line. If the output of characters directly after the variable is required (or variables sitting side-by-side are necessary), the variable name may be enclosed in curly brackets (e.g. ${*name*[@*format*]}). A newline is automatically appended to the end.

- *e.g.* 'print ``Number of atoms = $natoms''' writes a line to the screen looking like "Number of atoms = 11", assuming that the variable *natoms* has a value 11.
- *e.g.* 'print ``Atoms (${natoms}), bonds (${nbonds})''' writes a line to the screen looking like "Atoms (11), bonds (8)", assuming that the variables *natoms* and *bonds* have values of 11 and 8 respectively.

8.3.3 Storing Atom Data

Create and set data for atoms in the model, and create bonds between them.

addatom
The standard way of creating a new atom in the model. The command requires no arguments, and looks for variables set in the filter to store in the atom.

- *e.g.* 'addatom'

Variables recognised by the command are:

e
Element symbol/Z/fftype of the atom

r.x, *r.y*, *r.z*
Coordinates of the atom.

f.x, *f.y*, *f.z*
Forces of the atom.

v.x, *v.y*, *v.z*
Velocities of the atom.

q
Charge on the atom

id
Numeric ID of the atom, used for bonding (see later).

If any of the variables cannot be found then the relevant properties are not set, otherwise the variables are reset afterwards ready for the next atom. Should the element variable *e* be blank then the atom is set to element 0 ('XX'). The numeric ID of the atom given by *id* is a special case. If set it is stored as the atom's internal ID and can be referenced by the 'addbond' command, but as soon as loading has been finalised the atomic IDs are renumbered internally – atom IDs set from the *id* variable will not persist into the code.

addbond *id*1 *id*2 [*type*
Bond two atoms with the specified IDs together with a singlne bond (if no bond type is specified) or of the type specified in *type*. The bond type may be given as a bond order (e.g. single = 1, double = 2, triple = 3) or as a string ('single', 'double', or 'triple'). If the supplied *type* is not recognised a single bond is used instead.

- *e.g.* 'addbond 2 3' creates a single bond between atoms 2 and 3.
- *e.g.* 'addbond 10 5 *btype*' creates a bond between atoms 10 and 5 whose type is held in the variable *btype*.
- *e.g.* 'addbond 5 6 2' creates a double bond between atoms 5 and 6.

createatoms *n*
Creates a number *n* of atoms at once in the model, whose properties may then be set individually (in any order) with the 'setatom' command. Atoms are all created initially with coordinates, forces, and velocities set to (0.0,0.0,0.0), a charge of 0.0, and with element type 0.

- *e.g.* 'createatoms 52' creates 52 new atoms in the model.

- *e.g.* 'createatoms *natoms*' creates new atoms in the model, the number depending on the value of the variable *natoms*.

modeltemplate

> For use when reading trajectory frames, this command creates n new atoms in the frame (where n is the number of atoms currently in the parent model of the trajectory) and copies the element, colour, and rendering style data from the atoms in the parent model. Other properties such as the coordinates, velocities, and forces can then be defined with one of the 'set*' commands.
>
> - *e.g.* 'modeltemplate'.

setatom *id*

> Set the properties of the atom with the specified ID from variables set in the filter. The variables searched for are the same as for 'addatom'.
>
> - *e.g.* 'setatom 9' sets the data for atom ID 9 based on the values of variables already set in the filter.
> - *e.g.* 'setatom *i*' sets the data for atom *i* based on the values of variables already set in the filter.

setrx *id value*

setry *id value*

setrz *id value*

setfx *id value*

setfy *id value*

setfz *id value*

setvx *id value*

setvy *id value*

setvz *id value*

> Set individual properties of the specified atom referenced by its ID.
>
> - *e.g.* 'setrx 8 *posx*' sets the x-coordinate of atom 8 to the value of *posx*.
> - *e.g.* 'setfz 1 0.0' sets the z-component of the atomic forces of atom 1 to zero.

8.3.4 Loops

Commands to perform plain repeat loops and loops over patterns, molecules, atoms, and forcefield terms. For loops that run over aspects of the model (e.g. atoms) the counter variable, elements of the control variable (e.g. the atom's name, coordinates etc.) may be accessed by referencing in the form *variable.property*. For example, the name and element symbol of an atom variable named 'i' may be retrieved with 'i.name' and 'i.symbol' respectively. Full lists of the variables set by the loops are given in the following list. All loop blocks must be terminated with an 'end' command.

foratoms *var* [*pattern* [*molecule*]

> Loop that runs over atoms in a model, pattern, or pattern molecule. The required variable *var* is used to reference the current atom, while the second (optional) variable determines a specific pattern over which to loop (which may further be given a molecule number over which to restrict to). If only thne control variable is supplied the loop runs over all atoms in the current model.

> - *e.g.* to loop over all atoms i in the model
>
>   ```
>   foratoms $i
>   ...
>   end
>   ```
>
> - *e.g.* to loop over each atom i in the representative molecule of a pattern referenced by *patternvar*
>
>   ```
>   foratoms $atomnum $patternvar
>   ...
>   end
>   ```
>
> - *e.g.* to loop over all atoms i in a specific molecule m of pattern p
>
>   ```
>   foratoms $i $p $m
>   ...
>   end
>   ```
>
> - Properties available for the atom control variable are:
>
>   *symbol*
>   > Element symbol of the atom.
>
>   *z*
>   > Atomic number Z of the atom.
>
>   *mass*
>   > Atomic mass of the atom.
>
>   *name*
>   > Element name of the atom.
>
>   *r.x*, *r.y*, *r.z*
>   > Coordinates of the atom.
>
>   *f.x*, *f.y*, *f.z*
>   > Forces of the atom.
>
>   *v.x*, *v.y*, *v.z*
>   > Velocities of the atom.
>
>   *q*
>   > Charge on the atom.
>
>   *fftype*
>   > Forcefield type name associated to the atom (or '???' if none).
>
>   *ffequiv*
>   > Forcefield equivalent type name associated to the atom (or '???' if none).

forffbonds *var pattern*

forffangles *var pattern*

forfftorsions *var pattern*

> Loops that run over the different forcefield elements of a pattern.

> - *e.g.* to loop over all patterns p in the model
>
>   ```
>   forpatterns $p
>   ...
>   end
>   ```

Properties available for the control variable are:

$funcform$
>   Functional form of the term, corresponding to the keyword names in the original forcefield file (see Section XXX).

id_i
id_j
...
>   Molecule-relative ids of the atoms i, j, etc. involved in the term.

$param\_a$
$param\_b$
...
>   Parameters of the interaction, corresponding to the input order in the forcefield file (see Section XXX).

$type\_i$
$type\_j$
...
>   Forcefield types of the atoms i, j, etc. involved in the term.

formolecules *var pattern*
>   Specialised loop that runs over molecules in the patterns of a model, and so is of use when exporting data. The variable *var* is used to store the current molecule number.

>   - *e.g.* to loop over all molecules m in a pattern *thispat*
>
>   ```
>   formolecules $m $thispat
>   ...
>   end
>   ```

forpatterns *var*
>   Specialised loop that runs over the patterns of a model, and so is of use when exporting data.

>   - *e.g.* to loop over all patterns p in the model
>
>   ```
>   forpatterns $p
>   ...
>   end
>   ```

>   Properties available for the pattern control variable are:

>   *name*
>   >   Name of the pattern.

>   *nmols*
>   >   Number of molecules encompassed by the pattern.

>   *nmolatoms*
>   >   Number of atoms per molecule.

>   *name*
>   >   Element name of the atom.

>   *nbonds*
>   >   Number of forcefield bond terms used in the expression for the pattern.

>   *nangles*
>   >   Number of forcefield angle terms used in the expression for the pattern.

>   *ntorsions*
>   >   Number of forcefield torsion terms used in the expression for the pattern.

repeat *var [niter*
>   Repeat commands up to the next 'end' for a number of iterations, storing loop iteration number in the variable *var*. *niter* may either be a previously-initialised variable containing the number of iterations to perform, or an integer value. If *niter* is not supplied the loop runs indefinitely and can only be terminated by the 'XXX' command or by reaching the end of the file.

- *e.g.* 'repeat $n' loops indefinitely, storing the current loop iteration in n.
- *e.g.* 'repeat $natoms 164' loops 164 times storing the current loop iteration in a variable *natoms*.
- *e.g.* 'repeat $atomno $size' loops for the integer value of the *size* variable, storing the current loop iteration in a variable *atomno*.

while *var test value*
 Repeats the commands in the following block while the condition provided evaluates to true (tested at the start of the block).

- *e.g.* 'while element = Cl' loops while the variable *element* contains a value of 'Cl'.

8.3.5 Storing a Unit Cell

In particular, methods to set a handful of related variables for use in the filter, and to create / describe atoms from sets of known variables.

setcell
 Set the unit cell for the model / trajectory frame from cell length and angle variables already set in the filter. Lengths should be given in Angstroms and angles in degrees. The command expects to find values in:

 cell.a

 cell.b

 cell.c
  Lengths of the three cell axes.

 cell.alpha

 cell.beta

 cell.gamma
  Angles between cell axes.

setcellaxes
 Set the unit cell for the model / trajectory frame from an axis matrix definition already set in the filter. Unit of length for axis definitions is Ångstroms. The command expects to find values in:

 cell.a.x

 cell.a.y

 cell.a.z
  Vector components of the cell's A vector.

 cell.b.x

 cell.b.y

 cell.b.z
  Vector components of the cell's B vector.

 cell.c.x

 cell.c.y

 cell.c.z
  Vector components of the cell's C vector.

setspacegroup *spgrp*
 Set the spacegroup for a periodic model, allowing symmetry copies of atoms to be generated with the 'pack' command.

8.3.6　Testing Conditions

Ways to perform tests on variables within the filter.

if *var test value*

　　Tests the contents of the variable *var* against either another variable named *value*, or a literal string *value* (which it is assumed to be if a variable named *value* does not currently exist) and executes the commands in the following block if the test is true. See Section **??** for a list of tests and the way they are performed.

- *e.g.* '`if $natoms = 30`' checks if the variable *natoms* equals '30'.
- *e.g.* '`if $style <> formatted`' checks if the variable *style* equals the string 'formatted'.

else

　　Closes the command block of the most recent (as yet unterminated) 'if' or 'elseif' and begins another block, the contents of which will be executed if none of the preceeding tests evaluate to true.

- *e.g.* '`else`'

elseif *var test value*

　　Closes the command block of the most recent (as yet unterminated) 'if' or 'elseif' and begins another 'if' block, the contents of which will be executed if its condition evaluates to true.

- *e.g.* '`elseif $natoms = 50`' checks if the variable *natoms* equals '50', executing the commands within the following block if this is so.

```
 e.g. if $name = alan
        ...do something...
      elseif $name = bill
        ...do something else...
      elseif $job = done
        ...test on different variable...
      else
        ...contingency plan...
      end
```

By default, the atomic numbers of imported atoms are decoded from element symbols since this is the most common way in which they are represented in molecule file formats. Case is unimportant, so 'na', 'Na', and 'NA' will all be interpreted as atomic number 11 (sodium). When this proves to be inadequate, several extra options are available (set through the command line or file import dialog) allowing atomic numbers to be determined in different ways. The 'ff' option is particularly useful when loading in coordinate files which have atom names which do not necessarily correspond trivially to element information (e.g. DL_POLY configurations) and where the atomic numbers of atoms cannot uniquely be determined from a simple comparison with element symbols. When decoding forcefield types, all forcefields which have been imported prior to model loading will be searched for the type names in the model file, and the atomic number set accordingly to the first exact match found.

Chapter 9

Scriptors Handbook

The scripting and interactive modes of Aten allow most program procedures and actions to be controlled *via* a series of text commands contained in a file or typed in directly. Files containing script commands are passed with the '-s *file*' or '--script *file*' command-line options and are executed immediately following parsing of the commands in the file. Alternatively, a compound sequence of commands can be provided on the command line with the '-c' or '--command' switches. As with filters, variables must be preceded with a '$', so compound commands given on the command line must, for example, be enclosed with single-quotes to prevent the shell from expanding these variables itself.

9.1 Inside the Script

When performing a whole sequence of commands to perform actions without a visible workspace (i.e. the GUI), managing and using objects becomes more difficult. When dealing with models and forcefields in these cases, Aten does not use explicit references to different objects in order to accomplish tasks. For example, typically one might wish to:

- Load a model X

- Edit X

- Edit X a bit more

- Load a forcefield Y

- Associate a forcefield Y with X

- Minimise a model X with a forcefield Y

- Save a model X

- Go and have tea

Here, explicit references to the model X and forcefield Y are required every time they are used, which is verbose and rather incoherent with the same procedure when 'at the desk'. Instead, Aten uses the notion of *current* objects (*i.e.* those on the desk in front of you, as opposed to those waiting on the shelf behind you) which are the focus of all the commands, and means that no references to objects (models, forcefields etc.) are given to the commands, in the same spirit as the GUI where commands are implicitly executed on the model currently displayed. Obvious exceptions to this are those commands that provide methods to select other objects, thereby making them *current*. Thus, whenever an object is created or loaded it becomes the current object of that type, receiving the effects of all commands relevant to it, and remains current (on the desk) until replaced (moved to the shelf) by another object of the same type being loaded, created, or selected. In the example above, all references to X and Y could then be removed since when the model and forcefield are loaded they become the current objects of their respective type, and the remaining commands work implicitly on these current objects.

Objects in scripts are referred to by names associated with them on their creation or loading, and may then be selected by reference to this name at a later stage.

9.2 Quick Command Examples

9.3 Command Reference

9.3.1 Analysis

Adds quantities / properties to be calculated to a pending joblist. The 'modelanalyse' and 'frameanalyse' subcommands calculate the pending properties in the joblist for the current model / frame and accumulate the data. The 'finalise' subcommand is used to normalise the calculated data and generate suitable averages from the accumulated data.

finalise
> Finalises the quantities in the pending jobs list, performing the necessary averages.
>
> - *e.g.* '`finalise`'.

frameanalyse
> Calculates all properties in the pending joblist for the current frame.
>
> - *e.g.* '`frameanalyse`'.

modelanalyse
> Calculates all properties in the pending joblist for the current model.
>
> - *e.g.* '`modelanalyse`'.

pdens *name site1 site2 spacing npoints*
> Adds a 3D probability distribution function *name* to the pending jobs list for the model. Two sites must be specified, the first being the central species about which to calculate the distribution of the second. Axes must be specified for the first site with the 'setaxes' command. *spacing* gives the spacing between grid points (assuming a cubic grid) while *npoints* determines the number of grid points to use in each positive / negative direction.
>
> - *e.g.* '`pdens mydens OHO centre 0.1 40`' requests the calculation of a probability density called 'mydens' between the central site 'OHO' and the site 'centre', with 40 gridpoints in each positive / negative direction with a spacing of 0.1 Å (giving a grid extent of 4 Å in each positive / negative direction, i.e. a grid of 81x81x81 points).

printjobs
> Prints a summary of the pending jobs list.
>
> - *e.g.* '`printjobs`'.

rdf *name site1 site2 rmin binwidth nbins*
> Adds a radial distribution function *name* to the pending jobs list for the model. Two sites must be specified. The RDF is calculated over the distance range bounded by $rmin$ and $rmin + binwidth * nbins$.
>
> - *e.g.* '`rdf cog N 0.0 0.1 200`' calculates an RDF between the sites 'cog' and 'N' over the distance range 0.0 - 20.0 Å.

9.3.2 Bonding

Create bonds and perform automatic bonding operations.

augment
>   Augments bonds in the current model.
>
>   - *e.g.* '`augment`'.

rebond
>   Calculate bonding in the current model.
>
>   - *e.g.* '`rebond`'.

clearbonds
>   Delete all bonds in the current model.
>
>   - *e.g.* '`clearbonds`'.

bondpatterns
>   Calculate bonds within pattern molecules.
>
>   - *e.g.* '`bondpatterns`'.

bondselection
>   Calculate bonds restricted to current atom selection.
>
>   - *e.g.* '`bondselection`'.

bondtol *tol*
>   Adjust the bond calculation tolerance.
>
>   - *e.g.* '`bondtol 1.20`'.

9.3.3 Building

Tools to build molecules from scratch, or finalise unfinished models. The drawing frame is represented as a set of three orthogonal vectors defining the reference coordinate system (set initially to the Cartesian axes) centred at an arbitrary origin (the pen position). Subsequent rotations operate on these coordinate axes.

addhydrogen
>   Satisfy the valencies of all atoms in the current model by adding hydrogens to heavy atoms.
>
>   - *e.g.* '`addhydrogen`'.

addatom *el*
>   Create a new atom of element *el* at the current pen position.
>
>   - *e.g.* '`addatom N`' places a nitrogen atom at the current pen coordinates.

addchain *el* [*bondtype*
>   Create a new atom of element *el* at the current pen position, bound to the last drawn atom with a single bond (or of type *bondtype* if one was specified).
>
>   - *e.g.* '`addchain C`' places a carbon atom at the current pen coordinates, and creates a single bond with the last drawn atom.
>   - *e.g.* '`addchain O double`' places an oxygen atom at the current pen coordinates, and creates a double bond with the last drawn atom.

endchain

'Ends the current chain (so that the next drawn 'chain' atom will not be bound to the last drawn atom).

- *e.g.* 'endchain'.

locate *dx dy dz*

Sets the pen position to the coordinates specified (in Angstroms).

- *e.g.* 'pen set 0.0 0.0 0.0' moves the pen back to the coordinate origin.

move *dx dy dz*

Moves the pen position by the amounts specified (in Angstroms).

- *e.g.* 'pen move 1.0 1.0 0.0' moves the pen +1 Angstrom in both the x and y directions.

rotx *angle*

Rotates the reference coordinate system about the x axis by *angle* degrees.

- *e.g.* 'pen rotx 90.0' rotates around the x axis by $90°$.

roty *angle*

Rotates the reference coordinate system about the y axis by *angle* degrees.

- *e.g.* 'pen roty 45.0' rotates around the y axis by $45°$.

rotz *angle*

Rotates the reference coordinate system about the z axis by *angle* degrees.

- *e.g.* 'pen rotz 109.5' rotates around the z axis by $109.5°$.

9.3.4 Cell Editing

Unit cell actions.

printcell

Prints the cell parameters of the current model.

- *e.g.* 'printcell'

removecell

Clears any cell description (removes periodic boundary conditions) from the current model.

- *e.g.* 'removecell'

scalecell *x y z*

Scale unit cell (and centres-of-geometry of molecules within it) by the scale factors x, y, and z.

- *e.g.* 'scalecell 1.0 2.0 1.0' doubles the length of the y-axis of the system, stretching the positions of the molecules to reflect the new size. x and z axes remain unchanged.

setcell *a b c α β γ*

Set cell lengths and angles of current model. This command will add a cell to a model currently without a unit cell specification.

- *e.g.* 'setcell 20.0 10.0 10.0 90.0 90.0 90.0' adds an orthorhombic cell with side lengths 20x10x10 Åto the current model.

9.3.5 Charges

Assign partial charges to models, atoms, and patterns. Charges are specified in units of e.

chargeatom id q
    Assign a charge of q to atom id in the current model.

- *e.g.* 'chargeatom 12 -0.2' assigns a charge of -0.2 to the twelfth atom.

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.

- *e.g.* 'chargeff'.

chargefrommodel
    Copies charges of all atoms in the current model to the atoms of the current trajectory frame.

- *e.g.* 'chargefrommodel'.

chargepatom id q
    Assigns a charge of q to atom id in each molecule of the current pattern.

- *e.g.* 'chargepatom 3 0.1' assigns a charge of 0.1 to the third atom in each molecule of the current pattern.

chargeselection q
    Assigns a charge of q to each selected atom in the current model.

- *e.g.* 'chargeselection 1.0' gives each atom in the current model's selection a charge of 1.0.

chargetype $fftype$ q
    Assigns a charge of q to each atom that is of type $fftype$ in the current model.

- *e.g.* 'chargetype OW -0.8' gives a charge of -0.8 to every atom that is of type 'OW'.

clearcharges
    Clears all charges in the current model, setting them to zero.

- *e.g.* 'clearcharges'.

9.3.6 Disordered Building

Build periodic disordered systems from individual components using Monte Carlo methods.

addcomponent *model nmols name*
    Specify that *nmols* copies of the *model* are to be added (or attempted to be added) during the build process. The component is referenced by the other commands from the provided *name*.

- *e.g.* 'addcomponent butane 300 bulk' requests that the 'butane' model should be entered into the list of components, referenced by the name 'bulk', and that the disordered builder should attempt to create 300 copies of the model in the new system.

setcellcentre *name*
    Sets the coordinates of the centre of the region defined for component *name* to the center of the cell.

- *e.g.* 'setcellcentre propanol' sets the centre of the shape for 'propanol' to be the centre of the unit cell.

setcenter *name x y z*

> Sets the coordinates of the centre of the region defined for component *name*.
>
> - *e.g.* 'setcenter propanol 5.0 7.0 6.0' sets the centre of the 'propanol' region to [5.0 7.0 6.0].

setgeometry *name x y z [l*

> Sets the geometry of the region for component *name*. The x, y, and z values determine the total extent of the region along each principal axis.
>
> - *e.g.* 'setgeometry propanol 10.0 10.0 3.0' sets the geometry of the region for the 'propanol' component. For example, if the region was of type 'sphere' this would create an elongated ellipsoid.

setoverlap *name true|false*

> Determines whether additions into the region are allowed to overlap with regions defined for other components. Default is true.
>
> - *e.g.* 'setoverlap lysine false' restricts the 'lysine' component to the subspace of its defined region that does not overlap with any other region.

printcomponents

> Prints the current component list to be used in the disordered builder.
>
> - *e.g.* 'printcomponents'.

setshape *name shape*

> Sets the type of the allowed insertion region for the specified model (which should have already been 'add'ed). Valid *shape*s are 'cell', 'cuboid', 'spheroid', and 'cylinder'.
>
> - *e.g.* 'setshape propanol sphere' restricts the component 'propanol' to a spherical region of the cell.

disorder *ncycle*

> Start the disordered builder, requesting *ncycle* cycles of Monte Carlo moves.
>
> - *e.g.* 'disorder 50' runs 50 cycles of the disordered builder.

vdwscale *scale*

> Sets the scaling factor for VDW radii to use in the disordered builder.
>
> - *e.g.* 'vdwscale 0.75' scales all VDW radii by 0.75 in the calculation.

9.3.7 Energy Calculation

Calculate energies for models and trajectory frames. All printing commands refer to the last energy calculated for either the model or a trajectory frame.

frameenergy

> Calculate energy of the current frame of the trajectory associated with the current model.
>
> - *e.g.* 'frameenergy'.

modelenergy

> Calculate the energy of the current model, which can then be printed out (in whole or by parts) by the other subcommands.
>
> - *e.g.* 'modelenergy'.

printelec

> Prints out the electrostatic energy decomposition matrix.

- *e.g.* 'printelec'.

printewald

Prints the components of the Ewald sum energy.

- *e.g.* 'printewald'.

printinter

Prints out the total inter-pattern energy decomposition matrix.

- *e.g.* 'printinter'.

printintra

Prints out the total intramolecular energy decomposition matrix.

- *e.g.* 'printintra'.

printenergy

Prints the elements of the calculated energy in a list.

- *e.g.* 'printenergy'.

printsummary

Print out a one-line summary of the calculated energy.

- *e.g.* 'printsummary'.

printvdw

Prints out the VDW energy decomposition matrix.

- *e.g.* 'printvdw'.

9.3.8 Expression

Manage the energy setup and forcefield expression.

createexpression

Creates a suitable energy description for the current model.

- *e.g.* 'createexpression'.

ecut *distance*

Sets the electrostatic cutoff (for Coulomb sum and real-space part of Ewald sum) to *distance*.

- *e.g.* 'ecut 14.5' sets the electrostatic cutoff distance to 14.5 Å.

elec *style...*

Selects the method of calculation for electrostatic energy and forces (default is 'none').

- 'elec none' turns off electrostatics.
- 'elec coulomb' uses the coulomb sum.
- 'elec ewald *alpha kx ky kz*' selects the Ewald sum with convergence parameter *alpha* and *k*-vectors specified.
- *e.g.* 'elec ewald 0.2 9 9 9'.
- 'elec ewaldauto *precision*' selects Ewald sum with automatic parameter generation governed by precision specified.
- *e.g.* 'elec ewaldauto 5.0e-6' auto-calculates *alpha* and *kmax* with XXX precision of 5.0e-6.

intra *on|off*

Controls calculation of intramolecular terms in energy / force calculations (on by default).

- *e.g.* 'intra off' turns intramolecular energy / force calculation off.

printexpression

Prints the current expression setup.

- *e.g.* 'printexpression'

vcut *distance*

Sets the van der Waals cutoff to *distance*.

- *e.g.* 'vcut 20.0' sets van der Waals cutoff distance to 20.0 Å.

vdw *on|off*

Controls calculation of van der Waals terms in energy / force calculations (on by default).

- *e.g.* 'vdw off' turns van der Waals energy / force calculation off.

9.3.9 Forcefields

Basic forcefield management.

ffmodel

Associates current forcefield to the current model.

- *e.g.* 'ffmodel'.

ffpattern

Associates current forcefield to the current pattern.

- *e.g.* 'ffpattern'.

ffpatternid *id*

     e.g. 'ffpatternid 3'.

loadff *file name*

Load a forcefield from *file* and reference it by *name*. Becomes the current forcefield.

- *e.g.* 'loadff /home/foo/complex.ff waterff' loads a forcefield called 'complex.ff' and names it 'waterff'.

selectff *name*

Selects the forcefield *name* and makes it the current forcefield. If a forcefield of that name is not loaded an error is returned.

- *e.g.* 'selectff organicff' makes the forcefield 'organicff' current.

9.3.10 Forces

Calculate forces for models and trajectory frames.

frameforces

Calculate the atomic forces of the current frame of the trajectory associated with the current model.

- *e.g.* 'frameforces'.

modelforces

Calculate the atomic forces of the current model.

- *e.g.* 'modelforces'.

printforces

Print out the forces of the current model.

- *e.g.* 'printforces'.

9.3.11 Monte Carlo

Change parameters for Monte Carlo-based calculations. Energy values are given in the current working unit of energy in the program.

mcaccept *move emax*

Sets the energy difference *emax* for the movetype *move* above which moves will be rejected.

- *e.g.* 'mcaccept translate 0.0' requests that only translation moves that lower the overall energy will be accepted.
- *e.g.* 'mcaccept insert 200.0' requests that insertion moves will be accepted provided the total energy does not rise more than 200.0 units.

mcmaxstep *move size*

Sets the maximal stepsize for the move type *move*.

- *e.g.* 'mcmaxstep translate 5.0' sets the maximum translation displacement to 5 Å.
- *e.g.* 'mcmaxstep rotate 30.0' sets the maximum rotation to 30°.

mcntrials *move n*

Sets the number of times *n* that the move type *move* should be attempted in each cycle.

- *e.g.* 'mcntrials insert 50' requests that there will be 50 insertion attempts per cycle per molecule type.

printmc

Prints the current Monte Carlo parameters.

- *e.g.* 'printmc'

9.3.12 Minimisation

Perform geometry minimisation on models.

converge *econv 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.

- *e.g.* 'converge 1e-6 1e-4' sets the energy and RMS force convergence criteria to 1×10^{-6} and 1×10^{-4} respectively.

mcminimise *maxsteps*

Optimises the current model using a Monte Carlo minimisation method.

- *e.g.* 'mcminimise 20' runs a geometry optimisation for a maximum of 20 cycles.

sdminimise *maxsteps maxtrials stepsize*

Optimises the current model using the Steepest Descent method.

- *e.g.* 'sdminimise 100 50 0.5' minimises the current model for a maximum of 100 Steepest Descent steps, with the maximum line trials per step set to 50, and the XXX

9.3.13 Models

Basic model management.

listmodels
>   Lists all models currently available in the workspace.
>
>   - *e.g.* 'listmodels'.

loadmodel *file name*
>   Load a model from *file*, referenced by *name*, which becomes the current model.
>
>   - *e.g.* 'loadmodel /home/foo/coords/test.xyz mymodel' loads a model called test.xyz and gives it the name 'mymodel'.

newmodel *name*
>   Create a new model, referenced by *name*, which becomes the current model.
>
>   - *e.g.* 'newmodel emptymodel' creats a new model called 'emptymodel' and makes it current.

printmodel
>   Print out information on the current model and its atoms.
>
>   - *e.g.* 'printmodel' outputs something like:
>
>   XXX Some crap here.

savemodel *format file*
>   Save the current model to *file* in *format*.
>
>   - *e.g.* 'savemodel xyz /home/foo/newcoords/test.config' saves the current model in 'xyz' format to the filename given.

selectmodel *name*
>   Make *name* the current model. If *name* cannot be found an error is returned.
>
>   - *e.g.* 'selectmodel othermodel' makes the model called 'othermodel' current.

9.3.14 Patterns

Automatically or manually create pattern descriptions for models.

addpattern *nmols natoms name*
>   Add a new pattern node to the current model, spanning *nmols* molecules of *natoms* atoms each, and called *name*.
>
>   - *e.g.* 'addpattern 100 3 water' creates a new pattern description of 100 molecules of 3 atoms each (i.e. 100 water molecules) in the current model.

clearpatterns
>   Delete the pattern description of the current model.
>
>   - *e.g.* 'clearpatterns'.

createpatterns
>   Automatically detect and create the pattern description for the current model.
>
>   - *e.g.* 'createpatterns'.

printpatterns
>   Prints out the current pattern description of the current model.

- *e.g.* 'printpatterns'.

selectpattern *name*
> Selects the pattern *name* in the current model to be the current pattern. If the pattern cannot be found an error is returned.

- *e.g.* 'selectpattern formate' selects a pattern named 'formate'.

9.3.15   Selection

Select atoms or groups of atoms.

invert
> Inverts the selection of all atoms in the current model.

- *e.g.* 'invert'.

selectall
> Select all atoms in the current model.

- *e.g.* 'selectall'.

selectatom *id*
> Select atom *id* in the current model.

- *e.g.* 'selectatom 45' selects the 45th atom.

selectelement *el*
> Select all atoms that are element *el* in the current model.

- *e.g.* 'selectelement Br' selects all bromine atoms.

selectfftype *fftype*
> Select all atoms with forcefield type *fftype* in the current model.

- *e.g.* 'selecttype CT' selects all atoms that are of type 'CT'.

selectnone
> Deselect all atoms in the current model.

- *e.g.* 'selectnone'.

selecttype *element description*
> Select atoms that are *element* and match the type *description* given.

- *e.g.* 'selectatomtype C ''-H(n=2)''' selects all carbon atoms that are bound to two hydrogens.

9.3.16   Sites

Describe sites within molecules for use in analysis.

addsite *name pattern* "*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.

- *e.g.* 'addsite watercentre h2o' adds a site called 'watercentre' based on the pattern called 'h2o' and located at the centre of geometry of all atoms.

- *e.g.* 'addsite oxy methanol 5' adds a site called 'oxy' based on the pattern called 'methanol' and located at the fifth atom in each molecule.

printsites

    Prints the list of sites defined for the current model.

- *e.g.* 'printsites'.

selectsite *name*

    Selects (makes current) the site referenced by *name*. If the site cannot be found an error is returned.

- *e.g.* 'selectsite carb1' makes the site 'carb1' the current site.

setaxes '*atomlist*' '*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 (defined by the list of atoms given to 'addsite') 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 two orthogonal vectors.

- *e.g.* 'setaxes '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.

9.3.17 Options

Control miscellaneous aspects of the program and its behaviour. Boolean arguments are specified as on/off, but may also take the form yes/no or true/false.

atomdetail *i*

    Sets the number of slices / stacks to use in the rendering of polygonal spheres.

bonddetail *i*

    Sets the number of slices / stacks to use in the rendering of polygonal tubes.

densityunits *units*

    Set the density units to use in output. See Section **??** for a list of possibilities.

- *e.g.* 'densityunits gpercm3' gives densities in g cm^{-3}.

energyunits *units*

    Set the unit of energy to use in output. See Section **??** for a list of possibilities.

- *e.g.* 'energyunits kcal' shows energy values in units of kcal mol^{-1}.

gl *option on|off*

    Enables various OpenGL features (all are disabled by default).

    linealias Enable line aliasing

    polyalias Enable polygon aliasing (Note - has odd effects in the current version)

    backcull Enable backface culling of polygons

    fog Enable depth cueing.

key *button action*

    Sets the action of the modifier keys that change normal mouse actions into their alternatives. Valid modifier keys are 'shift', 'ctrl', and 'alt'. Valid actions are:

    zrotate Changes a normal x/y rotation action into rotation about the world z-axis

transform Performs rotations and translations on the model coordinates rather than the view (i.e. a it translate action will translate the selected model coordinates instead of the camera)

none The key does nothing. At all.

mouse *button action*

Sets the action of the specified button. Valid *button*s are 'left', 'middle', and 'right'. Valid actions are:

none Gives you a useless button

rotate Free rotation of the model around the x and y axes

zrotate Rotation of the model around the z axis

translate Shifts the model in the world xy plane of the current view (i.e. in the plane of the screen)

zoom Moves the model back and forth along the local z-axis (into the screen) on moving the mouse up and down respectively

interact Performs almost every other action including selection, drawing etc.

movestyle *viewstyle*

The drawing style to use for models when rotating, manipulating etc. For those with lots of atoms and mid-range graphics cards, *stick* is probably a sensible option.

radius *viewstyle f* Sets the standard atom radius to use in the different drawing styles, although the effect of the radius differs between styles. For 'tube' and 'sphere' f defines the actual radius (and consequently the selection hotspot radius) of on-screen atoms in Angstroms. For the 'stick' style it determines only the hotspot radius of the atoms. The 'scaled' style uses f as a scaling factor applied to the elemental radii defined in the elements definition file 'elements.dat'.

shininess *i*

Sets the shininess of materials. i should be in the range 0-255.

show *object on|off*

Sets the visibility of objects in the rendering canvas. Valid *object*s are:

atoms Atoms and bonds within the model

cell The unit cell

cellaxes Cell axis arrows at the origin of the unit cell

cellrepeat Repeat units of the unit cell

forcearrows Atomic force arrows

globe The rotation globe displaying the cartesian axes

labels Atomic labels

measurements Geometry measurements

regions Regions for components in the disordered builder

style *viewstyle*

The default drawing style to use for all new models. Valid *viewstyle*s are 'stick', 'tube', 'sphere', and 'scaled'.

9.3.18 Trajectories

Open and select frames from the trajectory file associated to the current model.

firstframe

Select the first frame from trajectory of current model.

- *e.g.* 'firstframe'.

lastframe

  Select last frame in trajectory of current model.

- *e.g.* 'lastframe

loadtrajectory *file*

  Associate trajectory in *file* with current model.

- *e.g.* 'loadtrajectory /home/foo/md/water.HISf' opens and associated the DL_POLY trajectory file 'water.HISf' with the current model.

nextframe

  Select next frame from trajectory of current model.

- *e.g.* 'nextframe'.

prevframe

  Select the previous frame from the trajectory of the current model.

- *e.g.* 'prevframe'.

Chapter 10

Command Line Options

Import

-f *file*, --ff *file* Loads the specified forcefield file before any models specified on the command line.

--fold, --nofold
> Force or prevent initial folding of atoms to within the boundaries of the unit cell (if one is present) in loaded models, overriding filter directives.

--pack, --nopack
> Force or prevent generation of symmetry-equivalent atoms from spacegroup information in loaded models, overriding filter directives.

--bond, --nobond
> Force or prevent (re)calculation of bonding in loaded models, overriding filter directives.

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

-b, --bohr
> Specifies that the unit of length used in loaded models is Bohr rather than Angstrom, and should be converted to the latter.

Output / Debugging

-d, --debug
> Enables debugging of subroutine calls.

--debugtyping
> Enables output from the atom typing routines.

--debugfilters
> Enables output from the filter routines.

--debugall
> Enables output from the most all routines.

--debugparse
> Enables output from the file parsing routines.

--verbose
> Switch on verbose reporting of program actions.

Modes

-c *"commands..."*, --command *"commands..."*

> The command or compound command given is executed directly after processing of all other command line options is complete, including the loading of models. Commands should be separated with semicolons.

-s $file$, --script $file$

> Specifies that the script $file$ is to be loaded and run immediately following loading of all models on the command line (if any are specified). Once the script is completed, unless it is terminated by the 'quit' command the GUI will start (if it has not been started already), with all models (and the results of their manipulation by commands in the script) available.

-i, --interactive

> Starts Aten in interactive mode, where script commands are typed and executed immediately. The GUI is not started by default, but may be invoked (see Section **??**).

XXX Other options
-c, –convert $format$

Index