CONQTOUR: User Manual

A. S. Torralba[1,2,3], and the rest of the CONQUEST team (see CONQUEST manual)

[1]London Centre for Nanotechnology, UCL, 17-19 Gordon St, London WC1H 0AH, UK
[2]Department of Physics & Astronomy, UCL, Gower St, London, WC1E 6BT, UK
[3]National Institute for Materials Science, 1-2-1 Sengen, Tsukuba, Ibaraki 305-0047, Japan

http://www.conquest.ucl.ac.uk/

September 16, 2012

Abstract

This manual describes how to build and use CONQTOUR, a CONQUEST utility for visualising CONQUEST density files. It also helps in creating input files for CONQUEST, as well as in converting between several standard file formats. CONQTOUR is written in OpenGL and provides a python interface to make scripting easy.

Contents

1 Introduction

1.1 Purpose

The linear scaling Density Functional Theory (DFT) code CONQUEST achieves efficient paralelisation by, among other things, carefully distributing functions on a grid among computer processes. As a result, memory dumps of the charge density are scrambled and need to be rearranged for visualisation. Furthermore, many visualisation programs, such as VMD, understand a limited number of standard 3D data formats, so conversions of CONQUEST output files are necessary. As a further complication, the information required to perform such conversions is stored in several CONQUEST input and output files, making the task more cumbersome for the user. A number of external tools exist that plot 3D data, but for quick visualisation of CONQUEST results a lightweight application is desirable.

CONQTOUR aims at providing a fast and as transparent as possible tool for visualisation of CONQUEST charge densities and other volumetric data. It tries to gather information from configuration and output files without user intervention, so that charge density files can be loaded immediately. Users can explore the data using a mouse and keyboard, or they can introduce commands in a python terminal to interact with the display. Scripts can be executed to automate common tasks, either at statup or at any later time. The display is written in OpenGL and GLUT, to improve portability.

1.2 Functionality

CONQTOUR provides the following functions:

- Slicing of the density, by clicking on three atoms.
- Navigation along a direction using the cursor keys.
- Intuitive navigation using a mouse.
- Basic image export (into a ppm file).
- Several colouring schemes.
- Through the python terminal (pyCQ), CONQTOUR:
  - Reads 3D Data files in CQ or xplor formats (the latter, also from the command line).
  - Reads coordinates in xyz (also from the command line).
  - Sets the view orientation in a precise fashion.

Planned features (a.k.a wish list):

- Import/export of PDB coordinate files
- More flexible colouring of data (including manual intervals)
- Load pdb files from the command line
- Forgetful mode, so that only one density slice is stored at a time (useful in systems with limited memory).
- Flexible resolution of image files
- Direct output of png images (using libpng)

2 Building and running

CONQTOUR is written in C and uses standard libraries. It has been tested in unix systems (Mac OS, Linux), but it should compile in other systems, provided that the dependencies are met. It is part of the CONQUEST distribution and it can be found in the `utilities` directory. System parameters are stored in a `system.make` file, similar to the one used for the main code.

2.1 Dependencies

CONQTOUR requires the following libraries:

- An OpenGL implementation (e.g. mesa), including GLU and GLUT
- Posix threads (pthreads)
- A python installation, with accessible Python.h (readline module recommended)

2.2 Preparing the `system.make` File and Compiling

Configuration parameters are stored in `system.make` files, that are selected according to the environment variable CQ_SYSTEM, i.e. the `Makefile` will import `system.make.$(CQ_SYSTEM)`, if defined, or just `system.make`, if not. The user may need to modify one of the existing files, if the required libraries are not in standard locations.

A basic `system.make` file may look like the following:

```
CQ_USRBIN=$(HOME)/bin

CC=gcc
CFLAGS=-pthread
INCDIRS=-I/usr/include/ -I/usr/include/python2.6
LIBDIRS=-L/usr/lib/
LIBS=-lGL -lGLU -lglut -lpython2.6
```

To build the tool, just go into the source code directory (`utilities/cq_ut_conqtour` in the CONQUEST source code distribution) and say

```
CQ_SYSTEM=<your_system> make
```

where `<your_system>` could be something like `linux-x86_64-py26-usr`. Check the source code installation to find other examples.

If the environment variable `CQ_USRBIN` is defined, the binary (`cq_ut_conqtour`) will be copied there. For convenience, this should be in your `PATH`.

2.3 Command Line

The CONQTOUR command line is as follows:

```
cq_ut_conqtour [-v] [-d xplor_density_file] [-c xyz_coordinates]
               [python scripts...]
```

After running the command, a python terminal and, if a density was loaded, a display window, will open. If CONQTOUR is run without options, the display will be innactive until a 3D data file is loaded from the python terminal. Loading a coordinates file prepares a molecule, but is not enough to activate the display. At present, molecules are drawn in a very crude way, only as a reference to navigate CONQUEST densities.

CONQTOUR will always try to collect as much information about a CONQUEST run as possible. For this reason, it is best to run the command from within a directory that contains configuration and output files. However, this is not necessary and much of the required information can be provided manually from the python terminal (pyCQ) at a later time. This is explained in section 4.2.

The optional parameters are:

- `-v`: Make the output verbose. Among other things, information about a CONQUEST run will be listed as it is collected.
- `-d`: Preload an xplor density file.
- `-c`: Preload an xyz coordinates file.
- Python scripts: Any other parameters will be interpreted as python scripts and will be run in the order given.

3 Display and Navigation

The display window will not open until a 3D data file is loaded, even when coordinates are available. Xplor density files and xyz coordinates files can be loaded directly at start-up, using command line options (section 2.3). Other files, including CONQUEST densities, can be loaded later from the python terminal. Only one molecule can be displayed. Atoms are represented as spheres. Every time a density is loaded, it replaces the previous one in the memory, and a slice at the center of the new bounding box is calculated and displayed. The slice can be moved along the normal of the original plane, or a new normal can be calculated be selecting (double-clicking) three atoms to define a new plane. This is often convenient to display the density on the plane of a molecule or chemical group. It is also possible to recalculate the slice with the functions `cq_view_pick_atoms_slice` or `cq_view_set_reference_plane`.

3.1 Objects

The display window may contain several objects whose visibility can be toggled, either using a key or a command. The display becomes active once a 3D data file is loaded.

Object	Command	Key
Axes	`cq_toggle_axes`	x
Bounding box	`cq_toggle_box`	b
Legend	`cq_toggle_legend`	l
Molecule	`cq_toggle_molecule`	m

The **density slice** (not listed above, as for the moment it is always visible) is probably the most important object in CONQTOUR. It displays 3D data on a polygon (triangle to hexagon) within the bounding box and can be moved with the arrow keys or recalculated by selecting three atoms. Slices are calculated in steps (which depend on the available grid points and are set automatically) along a given direction. The initial direction is the z-axis. Because slices can intercept the bounding box near the vertices, sometimes they are very small. If the slice seems to be missing, try moving it with the arrow

keys. Keep in mind that each new slice is stored in an index, until the direction is reset, so memory could be exhausted for very large data sets. To save memory, it is possible to choose directly one of the slices calling `cq_view_set_slice` from the python terminal (see section 4.1).

The **bounding box** encloses all the 3D data, but not necessarily all the atoms in the molecule.

The **axes** are represented in a very simple way, with three lines for X (red), Y (green) and Z (blue) at the centre of the box. This is just for orientation and it is not intended to be very sofisticated. The axes are disabled by default.

The **legend** is active by default and gives information about the colours used to represent the data. Several methods exist to customise it from the python terminal (see section 4.3). In general, positive and negative values are treated separately and values out of range (both underflow and overflow) are represented in the bar with a striped pattern. The goal is to give a sense of *actual* values in the data set. The legend covers all data in the file, and cannot (at least at present) be restricted to a subinterval. Hence, if a logarithmic method is used, all orders of magnitude present in the file are shown.

The **molecule**, if loaded and if the display is active, is represented with spheres of several sizes that can be controlled with the left and right arrow keys. Atoms are clickable, and selecting three of them will redraw the density slice. If the central atom is within the bounding box, the slice will go through it; otherwise, the center of the box will be used.

3.2 Mouse Controls and Window Areas

The view can be controlled with a three-button mouse. Alternatively, if you have only two buttons (or even just one!), holding a modifier key while pressing the left button will emulate the others.

Button	Modifier key	Action
Left	None	Rotate
Middle	Shift	Translate center
Right	Control	Zoom in and out

Translation and scaling are fairly intuitive. Rotation is similar to other visualisation tools and also quite straightforward. Perhaps it requires some explanation. The display window is divided in two areas:

- The central area is a circle that spans 95% of the height of the window. You may think of this area as the surface of a spheric world map. If you press your finger against the surface and move it, the map will rotate accordingly. This movement is convenient to orient the view in any direction, but tends to slant the cell.
- The area outside of the central circle behaves in a simpler way: movements of the mouse rotate the whole system *around the normal to the screen*. This is useful to correct the "slant" or to orient some relevant part of the system in line with your eyes.

If you get lost, you can always reset the view to something sensible by pressing Shift+r. If a molecule is present and displayed, double-clicking any atom will "select it". The atom will then be shown in yellow. Selecting three atoms will trigger a recalculation of the density slice plane . Or you can run the commands `cq_view_pick_atoms_slice`, for the same effect, and `cq_view_set_reference_plane`, to set the plane manually with a point and vector.

3.3 Keyboard Controls

Several keys provide shortcuts for some functionality. The up and down cursor keys move the density slice within the bounding box, whereas the right and left keys increase or decrease the radius of the

atoms when a molecule is displayed. This is useful to see local details of the density while keeping a reference to the atom centres. Keys to toggle objects were introduced in section 3.1

A full list of active keys is listed below.

Key	Action
Up arrow	Move slice "up" in the direction of slice normal
Down arrow	Move slice "down" in the direction of slice normal
Left arrow	Decrease the size of the atomic spheres
Right arrow	Increase the size of the atomic spheres
s	Equivalent to "Up arrow"
a	Equivalent to "Down arrow"
R	Reset the view, to make the slice parallel to the screen and to show the whole cell
c	Recenter the view
C	Recenter the view and make the slice parallel to the screen
i	Write an image (screenshot), in ppm format
p	Write a python script[1] that can be run in PyMol to display the slice
b	Show/hide the bounding box
l	Show/hide the legend
m	Show/hide the molecule
x	Show/hide the XYZ axes (red, green and blue lines at the center of the box)
r	Show/hide a rectangle containing the slice (for debugging, mainly)
ESC	Exit the tool

[1] The name of the file is `slice.py`

4 Python Command Line and Scripting

CONQTOUR provides a flexible way for interacting with the display via a python terminal. An embeded module, `conqtour`, defines an interface to the display and other functionality that can be used as part of a general python script. This is convenient for automating frequent tasks or as a means to log a work session.

HINT *Saving your session*

If module `readline` is available (CONQTOUR will try to load it at start-up), you can say:

```
pyCQ> readline.write_history_file("mylog.py")
```

This will store your command history (including the `readline...` command, which you could edit out). Then, you can re-execute the log from the CONQTOUR command line (2.3) or, within a session, you can do

```
pyCQ> execfile("mylog.py")
```

The terminal will have autocompletion if your python distribution includes the `readline` and `rlcomplete` modules. In that case, just type `cq_` and then Tab. CONQTOUR will display a list of all the `conqtour` functions (section A.1). Similarly, a number of helper lists start with a `CQ_` prefix (section A.2).

Commands are grouped by areas, e.g. CONQUEST information can be obtained and manipulated using functions starting with `cq_info_`. The following sections describe the most important interface functions. The rest are listed in appendix A.

4.1 Getting and setting CONQTOUR Parameters

A number of parameters, mostly related to the current view of the display, can be accessed or modified from the command line. For example, if a molecule is loaded, a call to the function `cq_get_number_atoms()` returns the number of atoms of the molecule.

Some of these functions are very convenient for scripting. See, for instance, the example for `cq_get_number_slices` (section A.1.2.1).

Among the most important functions are `cq_view_get` and `cq_view_set`, which control the current view in an accurate way. `cq_view_get` returns a structure (a list of lists) that provides complete information of the current orientation of the displayed objects. This structure can be stored and reused later, or it can be modified and used to show a different view. For example, one could get closer to the box by doing:

```
pyCQ> view=cq_view_get()
pyCQ> view[3][0] -= 5      # Move 5 angstroms closer
pyCQ> cq_view_set(view)    # Do it
```

Also important are the functions to choose a slice. You can do this using three atom numbers as a reference, e.g. `cq_view_pick_atoms_slice(1,10,3)`, or you can set a completely arbitrary plane, defined by a normal vector and a point, with `cq_view_set_reference_plane`. To get the number of an atom, select it (double-click it) and the number will be shown in the python terminal. To deselect, double-click some empty space.

4.2 Getting and setting CONQUEST Information

CONQTOUR is about handling CONQUEST data in as transparent a way as possible. Because of this, if you run it in a directory containing the configuration and output files of a CONQUEST run, most of the neccessary information to load a density and an atomic structure will be gathered automatically. In most cases, you should be able to run `cq_load_cq_density()` and `cq_load_cq_coordinates()` without any further intervention.

If, for some reason, some files or parameters where missing when you started CONQTOUR, you can always try to find the parameters again using `cq_info_find`. But keep in mind that all previous parameters will be kept, so it you need a fresh start, do run `cq_info_reset`. If you need to actually see the current information, turn on verbosity (`cq_toggle_verbosity` or start with the `-v` command line option).

However, in some situations you will want to change some parameters by hand. Imagine, for instance, that you have several alternative input files with non-standard names and you want to select one of them. Then you can use `cq_info_set_input_file` to give the name splicitly, instead of using the default.

Another possibility is that you run CONQTOUR in an arbitrary directory, where you may have some `.xplor` and `.xyz` files. In this case, CONQUEST information is not available, but if you want to "cut" a piece of your system (see Appendix B), you might have to set the cell dimensions explicitly, with `cq_info_set_cell_dimensions`. For more details, see `cq_write_cq_coordinates` in section A.1.4.1.

A complete reference of these functions is in A.1.3.

4.3 Customising the Legend

CONQTOUR provides several colouring schemes to hopefully make the data easier to understand. In the future, more flexible methods may be provided, but for the moment the main idea behind the legend is that it should cover *all* the data, and it should also give an idea of the orders of magnitude spanned.

It is not an easy task to emphasise the most relevant aspects of the data. Our experience is that it is easy to get almost monochromatic plots that say little about the details of the system. Our approach is to divide the data in colour intervals (which in principle could be arbitrary, but for the moment are only conventionally equispaced). Each color interval extreme is assigned a colour and points within an interval are interpolated. These three aspects (*interval, interpolation, and colour*) can be handled separately in CONQTOUR. In addition, one can choose different ways to tick the legend. The interpolation, ticking and interval-generation methods can also be treated collectively, in a more conventional way, whereas the colouring method has to be chosen separately.

In the simplest approach, just set a global and color method:

```
pyCQ> CQ_LEGEND_METHOD
['linear','log']
pyCQ> cq_legend_set_method('linear')
pyCQ> CQ_LEGEND_COLORS
['redblue', 'rb', 'redgreenblue', 'rgb', 'rainbow']
pyCQ> cq_legend_set_method('rainbow')
```

You can choose among the methods listed in a relevant python list (e.g. `CQ_LEGEND_METHOD` for the collective method). For a complete reference, see sections A.1.2.2 and A.2.

As an example, if you choose the `'log'` method, intervals are calculated using the logarithm of the absolute value of the data, times the sign of the original datum. This allows for negative numbers to be represented, but obviously excludes zero.

It is important to understand exactly how this works:

- When the data are loaded, four extremes are found: the maximum and minimum positive and negative values present in the file.
- Intervals are created for positive and negative values separately.
- Zero (i.e. underflow values) are excluded, and will be represented in a striped pattern
- If the method is logarithmic, the minimum and maximum orders of magnitude are found, separately for negative and positive numbers.
- *All* orders of magnitude are represented in the legend. This means that if you see `1.0e-8`, it means you have values of that order of magnitude in your data, and if you don't see `1.0e2`, it means that you don't have e.g. 103.

In the future, a mechanism might be provided to restrict the range of data shown in the legend, but for the moment this is a fixed feature.

There is more flexibility than just using a collective method. A typical case is when you want e.g. linear intervals and ticks, but you want colours to be constant within each interval, so that flat colour areas are shown, instead of smooth gradients. This is useful to distinguish boundaries between colour intervals. To do just that, set the collective method as above, and then do:

```
pyCQ> cq_legend_set_interpolation_method('constant')
```

You could also choose e.g. linear intervals with log interpolation. Sometimes these weird combinations help the eye, so it is up to you to play the options.

TO DO: In the future, a way to set precise colour intervals (and extreme colours) manually will be provided. That is likely to be helpful to focus on particular intervals of interest (e.g. creating more subintervals within a particular range).

4.4 Input and Output

CONQUEST is usually run in parallel and it generates separate files for each MPI process. CONQTOUR attempts to facilitate exploring and managing these files, by gathering as much information about a run as possible (section 4.2). In many instances, CONQTOUR will be able to load density files in one go, if the necessary files are in the current directory, just by calling the parameter-less function `cq_load_cq_density()`. Similarly, to load an atomic structure from a CONQUEST coordinates file, it is usually enough to call `cq_load_cq_coordinates()`.

Since the systems that CONQUEST can handle can be quite large, it is sometimes convenient to filter the input somehow, so as to be able to make data fit in the available local memory, or to make visualisation faster. In order to achieve this goal, CONQTOUR provides a set of limits, that determine the exact data that will be loaded (or written). The next section describes these limits, that will condition how files (section 4.4.2) are handled and converted to other formats.

4.4.1 Limits

CONQTOUR limits are a set of parameters that reduce the amount of data to be read or written. There are two types of limits: grid limits and real limits. Grid limits refer to grid points, whereas real limits refer to space dimensions. They are used differently: grid limits usually affect I/O operations on densities; real limits change the way coordinates are treated. Both kinds are, of course, not entirely independent.

Grid limits are defined by a grid window, a grid shift and a stride.

The **stride** is a vector of integers that defines the periodicity with which grid points will be read (or written). Thus, a `[1,1,1]` stride means all points will be read, whereas `[1,2,3]` means all points will be read in the X direction, but only one every 2 or 3, in the Y and Z directions, respectively. You can change the value of this vector with `cq_limits_set_grid_stride`. Read section A.1.4.2 for more details.

The **grid window** is the number of points to be read or written in an I/O operation. This number is *after applying the stride*, so it does not refer to the number of points in a file. To get this, you must check the file itself, or, for CONQUEST files, use the `cq_info_find` function. To change this window, use `cq_limits_set_grid_window`.

The **grid shift** is the origin of points to be read/written. Together with the other grid limits, it determines the range of data that will be considered. The grid shift is useful to choose a piece of data for closer inspection. It can be set with `cq_limits_set_grid_shift`.

Real limits are only a real window and a real shift, always in Ångstrom. A stride does not make sense for real limits.

The **real window** determines the atoms that will be loaded from or written to a coordinates file. It will generally agree with the grid window "in the CONQUEST sense", i.e. only if enough information (cell dimensions, grid points...) is known, as reported by `cq_info_find`. To change it, use `cq_limits_set_real_window`.

The **real shift** is just a real vector that sets the origin of the window. In this case, it is completely independent of the grid shift, in all cases. This is so that the real shift can be used to store a reference of the original shift that was used to load a set of data.

If CONQUEST information is available the `cq_limits...` functions can carry out some consistency checks, such as for agreement between real and grid windows. But note that these checks will not be performed if the only available information comes from a loaded density or coordinates file. This is because the limits do not describe in-memory data, and for example a grid window could be larger than the current number of loaded density grid points, but it cannot be larger than the potential number of grid points to be loaded from CONQUEST density files. If you are trying to load data from other formats,

you should probably reset the CONQUEST information with `cq_info_reset`. And **always check the current limits with `cq_limits_get`** before loading or writing files: many functions modify the limits without (much) notice. Read section A.1.4.2 for the specific behaviour of each function.

4.4.2 Files

CONQTOUR can handle several formats (and hopefully more in the future) for both densities (or 3D data in general) and coordinates. For specific functions to handle them, see section A.1.4.1. Of course, the most important formats in this tool are those that CONQUEST uses, but there are many situations when you might prefer or need other formats.

For instance, usually you will want to load the density files from a CONQUEST run, perhaps applying some limits (4.4.1), visualise the data and then write all or part of the density into an `.xplor` file, to use it in e.g. VMD or PyMol.

Conversely, you may have a model in XYZ format and you may need to generate the CONQUEST coordinates file for a run. You could just read in the `.xyz` file with `cq_read_xyz_coordinates` and then write out the structure with `cq_write_cq_coordinates`. As you probably know, CONQUEST coordinates files do not contain all the necessary information to create the atomic structure. In particular, coordinates can be fractional or absolute (depending on a flag in the CONQUEST input file), and the atoms are listed with a reference number that corresponds to the labels in the `ChemicalSpeciesLabel` block in the input file. `cq_write_cq_coordinates` will generate these few pieces of information from your XYZ coordinates, so that the flags are easy to cut and paste into the `Conquest_input` file.

Step-by-step examples of common situations can be found in Appendix B.

A The `conqtour` python module

This section is the complete reference to the `conqtour` module, which is embedded in CONQTOUR and provides and interface to the display. It also handles input/output operations. A number of helper lists help call some of the functions. Help for any function can be obtained from the terminal with the usual python `help` function. Note that some of the functions are only active when certain kinds of data (3D, coordinates...) are loaded.

A.1 Functions

A.1.1 Display

`cq_view_get` : Get the parameters of the current view
Input : None
Output : A list of the form [[center],[up],[normal],[distance,field,near,far]]
Requires : 3D Data

Comments : A view is defined by a center and two directions, relative to the screen and provided as separate sublists, plus the distance to the center, a visual field angle, and distances to two clipping planes, near and far (at present, hard-coded, but this may change in the future).

Example : Store the view and restore it later

```
pyCQ> view=cq_view_get()    # Now change the view using the mouse
pyCQ> cq_view_set(view)     # This restores the previous view
```

`cq_view_pick_atoms_slice` : Recalculate the slice using three atoms as a reference
Input : Three atom numbers (as three parameters, not one tuple)
Output : None
Requires : Coordinates and 3D Data

`cq_view_recenter` : Restore the center of the view
Input : None
Output : The new center as a list
Requires : 3D Data

`cq_view_rotate` : Rotate the view around the up vector
Input : Angle in degrees
Output : None
Requires : 3D Data

Comments : The slice is rotated with the rest of the objects. To keep the slice parallel to the screen, use `cq_view_rotate_reslice`

Example : Turn around the view once

```
# While this loop is running,
# the display is still responsive to the mouse
```

```
pyCQ> for i in range(360):
.....   cq_rotate(1)
.....
```

cq_view_rotate_reslice : Rotate the view and keep the slice facing you
Input : Angle in degrees
Output : None
Requires : 3D Data

Comments : The slice is recalculated to keep it parallel to the screen (non-tracking mode).

Example : Turn around the view once

```
# While this loop is running,
# the display is still responsive to the mouse
pyCQ> for i in range(360):
.....   cq_view_rotate_reslice(1)
.....
```

cq_view_set : Set the current view
Input : A list of the form [[center],[up],[normal],[distance,field,near,far]]
Output : None
Requires : 3D Data

Comments : See `cq_view_get` for an explanation of the input list and an example.

cq_view_set_center : Set the center of the view
Input : A list or tuple with the center coordinates (not three coordinates)
Output : None
Requires : 3D Data

Comments : The function `cq_view_recenter` returns a list of the expected form.

cq_view_set_reference_plane : Set the plane of a new slice by hand
Input : A list of the form [[normal],[in-plane point]]
Output : None
Requires : 3D Data

Comments : If the point is outside of the bounding box, the centre of the box will be used. To get the current normal and point, use `cq_get_slice_plane`

Example : Show a slice on the XZ plane

```
pyCQ> cq_view_set_reference_plane([[0,1,0],[7,7,7]])
```

cq_view_set_slice : Set the slice number
Input : The slice number, starting at 0

Output : None
Requires : 3D Data

Comments : The get the number of slices in the current direction, use `cq_get_number_slices`.

Example : See `cq_get_number_slices` in section A.1.2.1.

A.1.2 CONQTOUR State

A.1.2.1 General

`cq_get_current_slice` : Get the index of the slice on display
Input : None
Output : The slice index
Requires : 3D Data

`cq_get_number_atoms` : Get the number of atoms of the molecule
Input : None
Output : The number of atoms
Requires : Coordinates

`cq_get_number_slices` : Get the number of slices that can be created along the current direction
Input : None
Output : The maximum number of slices
Requires : 3D Data

Comments : The slice index starts at zero, so the maximum returned by this function is not allowed

Example : View all slices, write ppm files and make a movie

```
# NOTE: To create the movie, you need ImageMagick
#       and mencoder in your system!!
import os

i=0
while i < cq_get_number_slices():
  # Show (and calculate if necessary) slice i
  cq_view_set_slice(i)
  # Write an image file
  frame=str(i).zfill(5)
  cq_write_ppm_image("scripted"+frame+".ppm")
  # Convert the image ppm to png (more useful) with ImageMagick
  os.system("convert scripted"+frame+".ppm scripted"+frame+".png")
  os.system("rm scripted"+frame+".ppm")
  i += 1
  print "Frame "+frame
# Merge the images into a movie, and delete the image files
os.system("mencoder mf://scripted0*.png -mf fps=25:type=png \
```

```
    -ovc copy -oac copy -o video.avi")
os.system("rm scripted0*.png")
```

cq_get_slice_plane : Get a point in the slice and a normal to it
Input : None
Output : A list of the form [[normal to slice][point in slice]]
Requires : 3D Data

Comments : To set a new slice direction and plane, use `cq_view_set_reference_plane`

A.1.2.2 Legend

cq_legend_get_label_format : Get the current format of the legend labels
Input : None
Output : A string with a C-style `printf` format
Requires : 3D Data

Comments : To set the format, use `cq_legend_set_label_format`

cq_legend_set_color_method : Change the method to assign colours to colour intervals
Input : One of the strings listed in `CQ_LEGEND_COLORS`
Output : None
Requires : 3D Data

Example : Get the list of methods and choose one

```
pyCQ> CQ_LEGEND_COLORS # List the valid method identifiers
['redblue', 'rb', 'rainbow'] # This may be different
pyCQ> cq_legend_set_color_method('rb')                # One way
pyCQ> cq_legend_set_color_method(CQ_LEGEND_COLORS[0]) # Another
```

cq_legend_set_interpolation_method : Change the method to interpolate colours in colour intervals
Input : One of the strings listed in `CQ_LEGEND_INTERPOLATION`
Output : None
Requires : 3D Data

cq_legend_set_interval_method : Change the method to create colour intervals
Input : One of the strings listed in `CQ_LEGEND_INTERVALS`
Output : None
Requires : 3D Data

cq_legend_set_label_format : Set the current format of the legend labels
Input : A string with a C-style `printf` format
Output : None

Requires : 3D Data

Comments : To get the format, use `cq_legend_get_label_format`

`cq_legend_set_method` : Change the general method to create a legend
Input : One of the strings listed in `CQ_LEGEND_METHOD`
Output : None
Requires : 3D Data

Comments : The general method sets in one go the ticks, interval and interpolation methods. Ticks will be recalculated using default values.

`cq_legend_set_tick_method` : Change the method to tick the legend
Input : One of the strings listed in `CQ_LEGEND_TICKS`
Output : None
Requires : 3D Data

A.1.2.3 Toggles

`cq_toggle_axes` : Toggle the visibility of the axes
Input : None
Output : The state after the call (True/False)
Requires : 3D Data

`cq_toggle_box` : Toggle the visibility of the bounding box
Input : None
Output : The state after the call (True/False)
Requires : 3D Data

`cq_toggle_coordinates_shift` : Toggle the shifting method for coordinates
Input : None
Output : The state after the call (True/False)
Requires : 3D Data

Comments : Coordinates can be shifted or not according the the real shift of the limits (see 4.4.1). The real shift will always be used to shift the I/O window, but the coordinates may be left unchanged, depending on this toggle.

`cq_toggle_legend` : Hide/Show the legend
Input : None
Output : The state after the call (True/False)
Requires : 3D Data

`cq_toggle_molecule` : Toggle the visibility of the molecule
Input : None
Output : The state after the call (True/False)
Requires : Coordinates

cq_toggle_verbosity : Toggle the level of output
Input : None
Output : The verbosity level (for the moment 0 or 1, but this may change)
Requires : Nothing

A.1.3 CONQUEST Information

cq_info_find : Gather information about a CONQUEST run
Input : None
Output : None
Requires : Nothing

Comments : Unless otherwise specified, it will search for files in the current directory. If the output is verbose (change with `cq_toggle_verbosity`), it will print a summary of what it finds. If some information was available from a previous call or from the user, it will keep it. For this reason, if you want a fresh set of parameters from the files in your directory, call `cq_info_reset` before, to clear all values.

cq_info_reset : Forget all the known information about a CONQUEST run
Input : None
Output : None
Requires : Nothing

Comments : This function forces `cq_info_find` to get all the information afresh, so if you are getting confused about the behaviour of CONQTOUR, it is probably a good idea to call it.

cq_info_set_block_points : Set the number of grid points per block
Input : A tuple containing (non-negative) values for all three axis.
Output : None
Requires : Nothing

Comments : To unset the value for any componenet, so that `cq_info_find` will update it, use 0.

cq_info_set_blocks_file : Set the blocks file associated to a CONQUEST run
Input : The file name.
Output : None
Requires : Nothing

Comments : Typical names look like `make_blk.dat`.
Use the empty string to force `cq_info_find` to update it.

cq_info_set_cell_dimensions : Set the cell dimensions associated to a CONQUEST run
Input : A tuple containing the new dimensions (the modules of the lattice vectors).
Output : None
Requires : Nothing

Comments : This is *not* the actual value after the data is loaded, which depends on the limits

(4.4.1, A.1.4.2), but the dimensions found in the coordinates file. To unset the value for any component, so that `cq_info_find` will update it, use 0.0.

`cq_info_set_coordinates_file` : Set the file name of the coordinates for a CONQUEST run
Input : A file name.
Output : None
Requires : Nothing

Comments : Use the empty string to force `cq_info_find` to update it.

`cq_info_set_cores` : Set the number of cores used in a particular CONQUEST run
Input : A non-negative integer.
Output : None
Requires : Nothing

Comments : Use 0 to force `cq_info_find` to update it.

`cq_info_set_density_stub` : Set the stub for the density files of a CONQUEST run
Input : A string.
Output : None
Requires : Nothing

Comments : Use the empty string to force `cq_info_find` to update it.

`cq_info_set_fractional_coordinates` : Set whether CONQUEST coordinates are fractional
Input : A True or False python object.
Output : None
Requires : Nothing

Comments : True objects include True and 1; false objects include False and 0.

`cq_info_set_grid_points` : Set the number of grid points associated to a CONQUEST run
Input : A tuple containing three non-negative integers
Output : None
Requires : Nothing

Comments : This is *not* the number of grid points that will be loaded, which depends on the limits (4.4.1, A.1.4.2). Use 0 for any component to force `cq_info_find` to update it.

`cq_info_set_input_file` : Set the parameter (input) file for a CONQUEST run
Input : A string.
Output : None
Requires : Nothing

Comments : This is likely to be `Conquest_input`. Use the empty string to force `cq_info_find` to update it.

`cq_info_set_output_file` : Set the output file for a CONQUEST run
Input : A string.

Output : None
Requires : Nothing

Comments : This file is most useful to get cell dimensions and the numbers of cores and grid points of a run. It is a good idea to set this right and then try `cq_info_find`. A call with the empty string will force `cq_info_find` to search this name in the input file.

A.1.4 Input/Output

A.1.4.1 General

`cq_load_cq_coordinates` : Load the coordinates associated to a CONQUEST run
Input : None
Output : None, or an IOError exception if there was a problem
Requires : The name of the file and whether coordinates are fractional. This information is gathered from a CONQUEST run (see `cq_info_find`) or manually set by the user (A.1.3)

Comments : Only atoms within limits will be loaded. The real shift and window limits are used to decide about that. The real shift will also be used to move coordinates back to the origin, if the last call to `cq_toggle_coordinates_shift` returned `True`. If an atom is outside of the simulation cell (because of periodic boundary conditions), it will be wrapped into the cell.

`cq_load_cq_density` : Load a density from a CONQUEST run
Input : None
Output : None, or an IOError exception if there was a problem
Requires : Consistent information, gathered from a CONQUEST run (see `cq_info_find`) or manually set by the user (A.1.3)

Comments : This function rearranges the density files of a CONQUEST run using information from the input and output files, so that it is displayed correctly. Limits (4.4.1, A.1.4.2) are applied to reduce the amount of data or to restrict it to a given space region. If some information is missing, it will fail until the user provides the necessary piece manually. Depending on the level of verbosity, it will show a "progress bar" or it will proceed silently. For large systems, it could take a while. As a side effect, the limits are reset according to the new data.

`cq_load_xplor_density` : Load an xplor density file
Input : The file name
Output : None, or an exception if there was a problem
Requires : Nothing

Comments : The limits (4.4.1, A.1.4.2) may change.

`cq_load_xyz_coordinates` : Load a molecule from an xyz file
Input : The file name
Output : None, or an exception if there was a problem
Requires : Nothing

cq_write_cq_coordinates : Write a CONQUEST coordinates file
Input : The file name
Output : None
Requires : Coordinates and some information about a CONQUEST run, limits, etc...

Comments : The behaviour can be quite complicated, so verbosity is recommended, so you know exactly what you are doing.

The cell dimensions that are written to the file depend on what you have. It will use the first piece of information that you have defined, in the following order:

- CONQUEST info about cell dimensions (check with `cq_info_find` and set with `cq_info_set_cell_dimensions`).
- Real window of the limits (set or unset with `cq_limits_set_real_window`).
- Density dimensions (if a density is loaded).
- The maximum values for the current coordinates (if everything fails, you must have at least this).

Take the value of the `coordinates_shift` toggle into account. If this is True, or in any case if limits were used, the coordinates will be shifted.

You can set whether the coordinates are fractional with `cq_info_set_fractional_coordinates`.

After writting the file, it will give you a few lines that you should copy into your `Conquest_input` file. Something like this:

```
----- FYI: You could copy the next few lines into your Conquest_input
IO.FractionalAtomicCoords F
General.NumberOfSpecies 4
%block ChemicalSpeciesLabel
 1  12.01   C
 2  15.99   O
 3  14.01   N
 4   1.01   H
%endblock
----- FYI END ------------------------------------------------------
```

cq_write_ppm_image : Write a ppm file with the image on display
Input : The file name
Output : None, or an exception if there was a problem
Requires : 3D Data

Comments : For the moment, the resolution is that of the openGL window, so make it larger (if you can!) to get a better picture.

cq_write_pymol_slice : Write a PyMol script to show the current slice
Input : The file name
Output : None
Requires : 3D Data

Comments : The script must be run within PyMol using the run command (e.g. `run slice.py`). The same script is written from the keyboard (press p) with a fixed name (`slice.py`).

cq_write_xplor_density : Write an xplor file from the current 3D data
Input : The file name
Output : None, or an exception if there was a problem
Requires : 3D Data

Comments : Limits are used and may be changed to fit the data, e.g. to reduce grid windows if they are too large.

cq_write_xyz_coordinates : Write an XYZ coordinates file
Input : The file name
Output : None
Requires : Coordinates and some information about a CONQUEST run, limits, etc...

Comments : See comments for `cq_write_xyz_coordinates`, except those about fractional coordinates, which are not relevant in an XYZ file.

A.1.4.2 Limits This section deals with getting and setting information about data limits. For a detailed explanation of the various limits, see section 4.4.1. Most of these functions can be called at any time (even without any loaded data). However, checks of bounds are only possible when CONQUEST information is available (see sections 4.2 and A.1.3).

cq_limits_get : Get the current data limits
Input : None
Output : A list of the form [[grid window],[grid shift],[strides],[real window],[real shift]]
Requires : Nothing

Comments : To set all the limits at once, use `cq_limits_set`.

cq_limits_reset : Reset I/O limits
Input : None
Output : None
Requires : Nothing

Comments : All the limits will be zero, except the strides, which will be 1.

cq_limits_set : Set all data limits in one go
Input : A list of the form [[grid window],[grid shift],[strides],[real window],[real shift]]
Output : None
Requires : Nothing

Comments : The function `cq_limits_get` gives a list of the same form.

cq_limits_set_grid_shift : Change the shift of the 3D data grid
Input : A tuple containing the three shifts
Output : None
Requires : Nothing

Comments : Not to be confused with `cq_limits_set_real_shift`.

cq_limits_set_grid_shift_from_real : Adjust the grid shift using the real shift
Input : None
Output : None
Requires : The dimensions and grid points of the simulation cell for a CONQUEST run. See sections 4.2 and A.1.3

Comments : The real shift will be changed slightly to make it match the grid shift exactly. If no CONQUEST information is available, the grid shift will be set to zero. To choose arbitrary values, use `cq_limits_set_grid_shift`.

cq_limits_set_grid_strides : Change the strides of the 3D data grid
Input : A tuple containing the three strides
Output : None
Requires : Nothing.

Comments : Grid points are read from/written to files every *stride* points. The default is 1 (all points are read). This is useful to save memory. Use 0 to keep the current value for a component. Note that there are no real strides.

cq_limits_set_grid_window : Change the number of actual read or written grid points
Input : A tuple containing the grid points per coordinate
Output : None
Requires : Nothing, but adjustment of the real window will only happen if enough CONQUESTinformation is known.

Comments : The minimum is 2, but use 0 or 1 to reset the window to 0 points (and the real one to 0.0), and negative values to keep the current value. Resetting a component is useful, for example, to load unconditionally all atoms in a file. This function is different from but related to `cq_limits_set_real_window`: The grid and real window will agree if cell dimensions and number of grid points are known from CONQUEST information.

cq_limits_set_real_shift : Change the real shift
Input : A tuple containing the real shift per component (in Ångstrom)
Output : None
Requires : Nothing

Comments : At present, this is applied to the output only. Not to be confused with `cq_limits_set_grid_shift`

cq_limits_set_real_shift_from_grid : Change the real shift using the grid shift
Input : None
Output : None
Requires : The dimensions and grid points of the simulation cell for a CONQUEST run. See sections 4.2 and A.1.3

Comments : If no CONQUEST information is available, the real shift will be set to zero. To choose arbitrary values, use `cq_limits_set_real_shift`.

cq_limits_set_real_window : Change the real window
Input : None
Output : None

Requires : Nothing

Comments : Use negative values to keep the current value for a component, and zero (or very small) to reset a component to zero. The latter is useful, for example, to load unconditionally all atoms in a file. Changing the real window affects the grid window, as both agree (if enough CONQUEST information about the grid and lattice is known). Note that the real window is *not* the cell dimensions.

A.2 Lists

CQ_LEGEND_COLORS : *string*
List of available color methods for the legend, to be used with `cq_legend_set_color_method`.

Value	Meaning
`redblue`	Red colours for positive values and blue colours for negative values.
`rb`	Short for `redblue`.
`redgreenblue`	Similar to redblue, but with green colours around zero.
`rgb`	Short for `redgreenblue`.
`rainbow`	The colors of the rainbow.

CQ_LEGEND_INTERPOLATION : *string*
List of available interpolation methods for the legend, to be used with `cq_legend_set_interpolation_method`.

Value	Meaning
`linear`	Linear interpolation between the extremes of each colour interval.
`log`	Logarithmic interpolation, using the absolute value and keeping the sign of the interpolated point.
`constant`	Constant value for each interval (the maximum, i.e. more positive value is used).

CQ_LEGEND_INTERVALS : *string*
List of available methods for automatic generation of colour intervals, to be used with `cq_legend_set_intervals_method`. In all these methods, positive and negative values are treated separately, so interpolation happens between the minimium and maximum positive values (or orders of magnitude), etc.

Value	Meaning
`linear`	Linear interpolation between minimum and maximum data values.
`log`	Logarithmic interpolation between data value extremes (in absolute value, but keeping the sign).

CQ_LEGEND_METHOD : *string*
List of available (collective) methods to create a legend. These are equivalent to setting ticks, interpolation and intervals methods at once. To be used with `cq_legend_set_method`.

Value	Meaning
`linear`	Linear ticks, intervals and interpolation.
`log`	Logarithmic methods (in absolute value, but keeping the sign).

CQ_LEGEND_TICKS : *string*
List of available methods to add ticks to the legend. To be used with `cq_legend_set_tick_method`.

Value	Meaning
`linear`	Ticks at linear intervals.
`log`	Ticks at logarithmic intervals (using absolute values, but keeping the sign).

B Step by step common tasks

B.1 Creating CONQUEST coordinates from an xyz file

Case: You have a structure model in an xyz file and you need CONQUEST coordinates to run a linear-scaling DFT calculation.

Procedure:

1. Start CONQTOUR and load the `.xyz` file.

```
pyCQ> cq_load_xyz_coordinates("model.xyz")
```

   In verbose mode, you will get some information about the number of atoms read. The display will *not* open, since you have not loaded a density, which is required.

2. Choose whether the coordinates should be fractional or not.

```
pyCQ> cq_info_set_fractional_coordinates(False)
```

3. To make sure that all atoms are written, reset the limits. (This step is not really needed in this example, but you should keep in mind that some atoms may be missing in the ouput file if you have some limits set, even if they correspond to the desired cell window, because atoms on the boundary might be discarded due to rounding. This step prevents that.)

```
pyCQ> cq_limits_reset()
```

4. Write the CONQUEST coordinates file. You probably want to turn on verbosity with `cq_toggle_verbosity()`.

```
pyCQ> cq_write_cq_coordinates("model.in")
Info: Using largest coordinates to decide the cell dimensions
      If you don't want this, set the CQ cell dimensions, with cq_info_set_cell_dimensions
Info: Writing a file with the folowing dimensions: 53.124000 61.488000 61.177000 A
----- FYI: You could copy the next few lines into your Conquest_input file -----
IO.FractionalAtomicCoords F
General.NumberOfSpecies 7
%block ChemicalSpeciesLabel
 1  14.01   N
 2   1.01   H
 3  12.01   C
 4  15.99   O
 5  32.06   S
 6  30.97   P
 7  22.99  Na
%endblock
----- FYI END ------------------------------------------------------------------
```

   In this case, there was no CONQUEST information available, the limits were set to zero and there was no density either, so the cell dimensions were calculated using the maximum values for each coordinate component. Often, this is not what you want, you can then set coordinates explicitly:

```
pyCQ> cq_info_set_cell_dimensions([49.5630, 57.1220, 56.9529])
```

   Then you will get a different message.

```
Info: Using cell dimensions from the Conquest info structures
      If you don't want this, set the CQ cell dimensions to zero, with cq_info_set_cell_dimensions
Info: Writing a file with the folowing dimensions: 49.563000 57.122001 56.952998 A
```

5. Copy the labels to your `Conquest_input` file.

B.2 Creating an xplor file from CONQUEST densities

Case: You have run CONQUEST and now have a lot of files in your current directory, including some density files, with names like `chden.001`, and you want to visualise them in CONQTOUR and then make an isosurface of the central piece in PyMol.

Procedure:

1. Go to the directory where you have the files and run `cq_ut_conqtour -v`. You should get a message like the following:

```
-------------------------------------------------------------------
-------------------------------------------------------------------
----                 KNOWN CONQUEST PARAMETERS                 ----
----   Note: this is NOT what you choose in the command line   ----
----         and it is only used if you load densities and/or  ----
----         coordinates from the pyCQ prompt                  ----
-------------------------------------------------------------------
-------------------------------------------------------------------
Info: Setting input file to default: Conquest_input
Info: Setting output file to default: Conquest_out
Info: Using Conquest_input to set coordinates file name: testfrac.out
Info: Setting flag: IO.FractionalAtomicCoords = False
Info: Number of species = 7
Info:  Species   1 = H
Info:  Species   2 = C
Info:  Species   3 = N
Info:  Species   4 = O
Info:  Species   5 = P
Info:  Species   6 = S
Info:  Species   7 = Na
Info: Setting flag to default: General.DistanceUnits = a0
Info: Setting flag to default: ReadBlocksFromFile = True
Info: Setting flag: General.PartitionMethod = Hilbert
Info: Using Conquest_input to set blocks file name: make_blk.dat
Info: Setting density file prefix to default: chden
Info: Number of processes = 256
Info: All density files are available for reading
Info: Number of grid points = 432 500 500
Info: Number of grid points per block = 4 4 4
Info: Dimensions (according to Conquest_out): a = 93.660490 b = 107.944930 c = 107.625560 a0
Info: Grid limits: Shifts = 0 0 0
Info: Grid limits: Window = 432 500 500
Info: Real limits: Shifts =    0.000    0.000    0.000 a0
Info: Real limits: Window =   93.660  107.945  107.626 a0
Info: Strides = 1 1 1
pyCQ>
```

2. If this information looks complete, run:

```
pyCQ> cq_load_cq_density()
```

   You should see a text progress bar and, finally, the display window will open and show your density. Here, we could write the whole density to an `.xplor` file, but we are going to cut a piece.

3. Check your limits (always a good idea before writing a file):

```
pyCQ> l=cq_limits_get()
pyCQ> l
[[432, 500, 500], [0, 0, 0], [1, 1, 1], [49.563000438192006,
57.122001100897563, 56.952997762884429], [0.0, 0.0, 0.0]]
```

4. Choose a smaller window, say 10x10x10 $Å^3$. For the density, we need to change the grid limits, so we could use `cq_limits_set_grid_window`, but since we have CONQUEST information, we can use `cq_limits_set_real_window`, which is more convenient than counting grid points. But first, we set a real shift with `cq_limits_set_real_shift` and then `cq_limits_set_grid_shift_from_real` to explicitly adjust the grid shift from that.

```
pyCQ> cq_limits_set_real_shift([(i-10.0)/2.0 for i in l[3]])
pyCQ> cq_limits_get()
[[432, 500, 500], [0, 0, 0], [1, 1, 1], [49.563000438192006, 57.122001100897563,
56.952997762884429], [19.781500219096003, 23.561000550448782, 23.476498881442215]]
```

(Here, we have used the real window, `l[3]`, to calculate a real shift, before changing the real window to 10 Å.)

```
pyCQ> cq_limits_set_grid_shift_from_real()
WARNING: User-provided grid window is not valid for coord. x; setting to maximum
WARNING: User-provided grid window is not valid for coord. y; setting to maximum
WARNING: User-provided grid window is not valid for coord. z; setting to maximum
pyCQ> cq_limits_get()
[[260, 295, 295], [172, 205, 205], [1, 1, 1], [29.783798407173389, 33.655046740809382,
33.555473631839725], [19.77920203101862, 23.466954360088177, 23.397524131044705]]
```

Note how the grid window was reduced to the maximum value after changing the shift. Finally, we change the real window.

```
pyCQ> cq_limits_set_real_window([10,10,10])
pyCQ> cq_limits_get()
[[87, 88, 88], [172, 205, 205], [1, 1, 1], [9.8896010155093101, 9.9591464845252258,
9.9296809726872652], [19.77920203101862, 23.466954360088177, 23.397524131044705]]
```

The grid window has been set automatically, using known parameters from the CONQUEST run (and the real window is no longer 10.0 Å, but it is as close as possible).

5. Write the piece of density to a file.

```
pyCQ> cq_write_xplor_density("density.xplor")
```

6. Open this density in PyMol and draw an isosurface.

```
bash$> pymol density.xplor
PyMOL> isosurface map1, density, 0.5
```

B.3 Creating an xyz file that matches an xplor file

Case: You have written an xplor file from a piece of your density and need an `xyz` file that matches

Procedure:

1. Follow the steps in B.2 to write a piece of a density into an xplor file.
2. Load the corresponding coordinates.

```
pyCQ> cq_load_cq_coordinates()
```

Because limits are still set to the central part of your cell, only the atoms in that area will be loaded. Furthermore, because the toggle for coordinates shift is set to `False` by default, the atoms will appear at the center of the display, just where they should.

3. However, the `.xplor` we just wrote will be interpreted as being at the origin, so before writing the coordinates to a file, we set the coordinates shift toggle to `True`. This will bring the coordinates (in the file, not in the memory) back to the origin.

```
pyCQ> cq_toggle_coordinates_shift()
True
```

   We have CONQUEST information, so CONQTOUR would use to determine the size of the cell.

   However, for an `xyz` file this is irrelevant, so we can proceed. If you need to use the dimensions in the current limits (for example, if you want to write a CONQUEST coordinates file of this small piece), call `cq_info_reset` or just set the CONQUEST cell dimensions to zero with `cq_info_set_cell_dimensions`. Limits will then be used.

4. Write the coordinates to an `xyz` file.

```
pyCQ> cq_write_xyz_coordinates("coords.xyz")
```

5. You can check that the density piece and the coordinates indeed match by opening them with CONQTOUR.

```
bash$> cq_ut_conqtour -d density.xplor -c coords.xyz
```

Index