

sculptor.biomachina.org

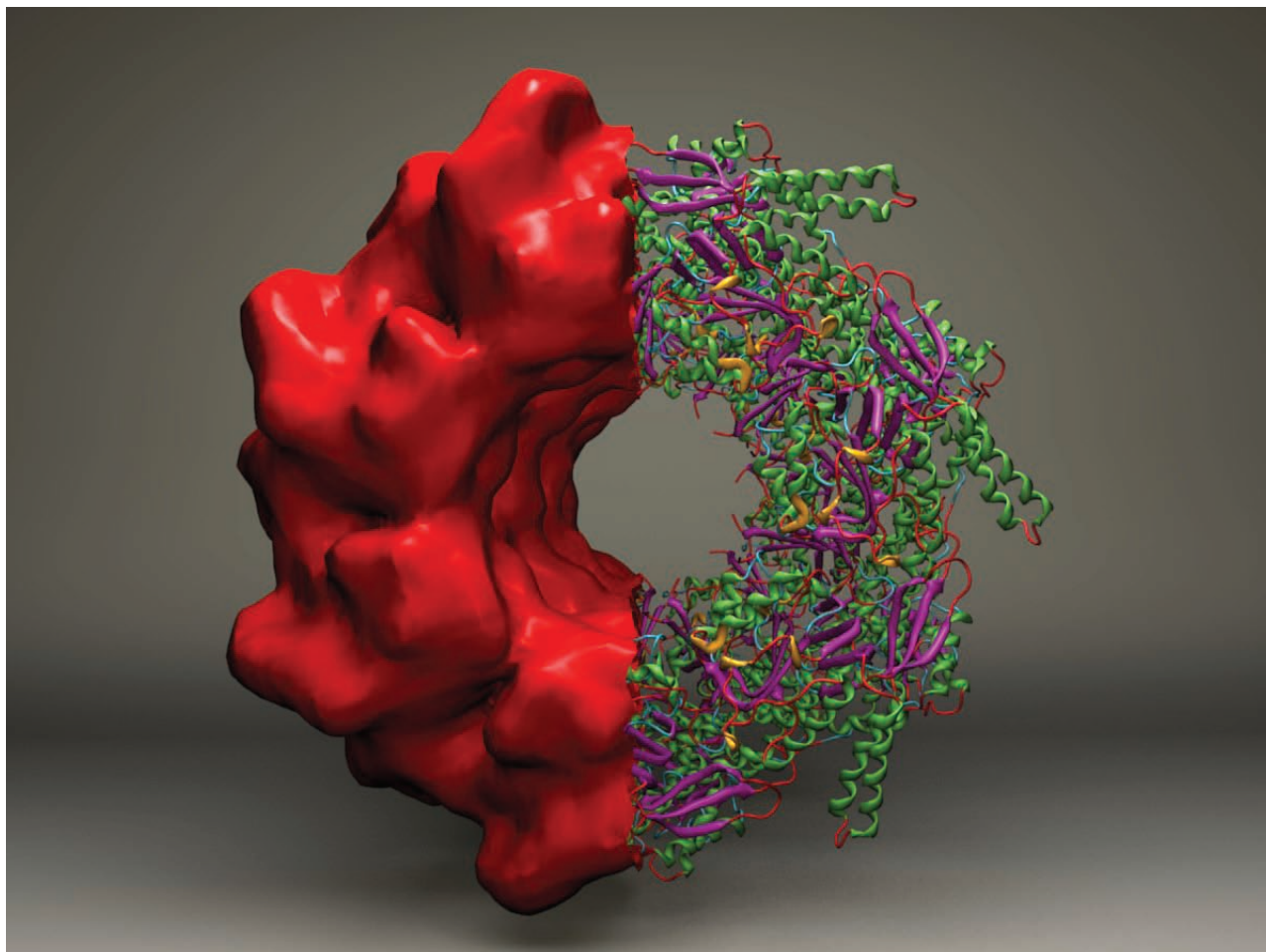


# SCULPTOR

VERSION 2

---

## USER MANUAL



March 26, 2010

**Sculptor** has been developed from 2001-2012 at [biomachina.org](http://biomachina.org) with contributions from the following authors (in alphabetic order):

**Stefan Birmanns, Maik Boltes, Paul Boyle, Jan Deiterding, Frank Delonge, Sayan Ghosh, Jochen Heyd, Oliver Passon, Mirabela Rusu, Francisco Serna, Zbigniew Starosolski, Manuel Wahle, Willy Wriggers, and Herwig Zilken.**

The main reference for Sculptor v. 2 is

- **Stefan Birmanns, Mirabela Rusu, and Willy Wriggers.** [Using Sculptor and Situs for Simultaneous Assembly of Atomic Components into Low-Resolution Shapes.](#) *J. Struct. Biol.*, Vol. 173, pp. 428–435, 2011.

In addition, we list here some relevant published articles that describe specific visualization features of Sculptor:

- Mirabela Rusu and Willy Wriggers. [Evolutionary Bidirectional Expansion for the Tracing of Alpha Helices in Cryo-Electron Microscopy Reconstructions.](#) *J. Struct. Biol.*, Vol. 177, pp. 410-419, 2012.
- Manuel Wahle and Stefan Birmanns. [GPU-Accelerated Visualization of Protein Dynamics in Ribbon Mode.](#) *SPIE Proceedings Vol. 7868, Visualization and Data Analysis (2011)*, Pak Chung Wong; Jinah Park; Ming C. Hao; Chaomei Chen; Katy Börner; David L. Kao; Jonathan C. Roberts, Editors.
- Oliver Passon, Maik Boltes, Stefan Birmanns, Herwig Zilken, and Willy Wriggers. [Laplace-Filter Enhanced Haptic Rendering of Biomolecules.](#) In: *Proceedings Vision Modeling and Visualization*, G. Greiner, J. Hornegger, H. Niemann, M. Stamminger, Editors, 2005, pp. 311-318 & 516, IOS Press, Netherlands, ISBN 1-58603-569-X.
- Stefan Birmanns, Maik Boltes, Herwig Zilken, and Willy Wriggers. [Adaptive Visuo-Haptic Rendering for Hybrid Modeling of Macromolecular Assemblies.](#) In: *Proceedings Mechatronics and Robotics*, P. Drews, Editor, 2004, Vol. 4, pp. 1351-1356, Eysoldt Verlag, Germany, ISBN 3-938153-30-X.
- Stefan Birmanns and Willy Wriggers. [Interactive Fitting Augmented by Force-Feedback and Virtual Reality.](#) *J. Struct. Biol.*, 2003, Vol. 144, pp. 123-131.

# *Contents*

*Sculptor*      5

*Visualization*      19

*The Sculptor Scripting Interface*      21

**NOTE:** For the latest information see the [online tutorials](#) and [online documentation](#).

# Sculptor

## Concept

### Graphical User Interface


#### Main Window


The Sculptor window is partitioned into two columns, with the left column showing the current list of loaded documents and their properties, whereas the right column is reserved for the 3D and text output. The columns are separated with thin dividers that can be moved to adjust the size of the different areas of the GUI. They can also be used to completely collapse sub-elements, which is useful to maximize the space for the 3D window. Of course the collapsed elements can be brought back by moving the divider again.

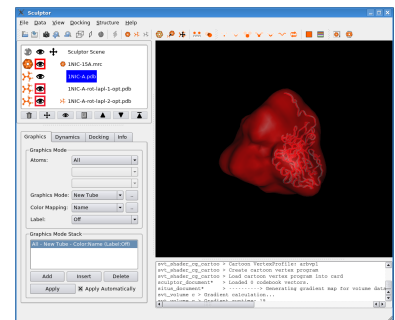
#### Document Window

The document window maintains a list of all loaded data sets. It is also the main GUI window, the place where one can adjust most of the global properties of the program, and also a number of settings relating to an individual document.

Here you will find only a basic description of the icons of the main document window - details about the usage of the features of the program are described in the other tutorials. The icons can also be found as functions in the menubar.


 - enables or disables the rendering of this document (the menu item "data/view" can also be used to toggle the state).

 - if this icon is displayed, the document will be the center of the transformation. If the document is rotated, for example by using the mouse, the rotation will be around the geometrical center as pivot point. Attention: The coordinates of the document will change if a



rotation is applied. If the "Sculptor Scene" document is the center, a global rotation around the center of the 3D scene will get carried out. In that case, the actual coordinates of the documents will not change, only the viewpoint gets adjusted.

One can change the center of the transformation by selecting a document and by clicking on the menu item "data->move". Alternatively one can also directly click into the document list, into the area where the crosshair would be. The 'm' key on the keyboard also allows to toggle the status of the current document - if is already the center, the scene will be the new center. In a visual multi-scale docking process one very often wants to adjust the position of one or multiple probe molecules relative to a 3D volumetric map. To check the position and orientation of the molecule it is often beneficial to alternate between a local and global transformation. This can also be accomplished very efficiently by selecting the probe molecule in the document list and by

toggeling the status via the  key.



- deletes the selected document(s).



- opens a dialog box with which the visualization of the selected document can be adjusted. See other tutorials for details.



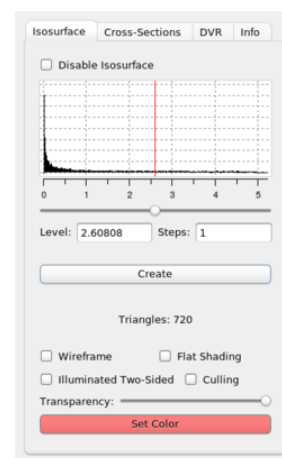
- moves the selected document up in the document list. The geometric center of the first item will at the same time also be the origin of the 3D scene. That way, the first document will always be clearly visible after being loaded. If more documents are loaded the origin can be changed by moving another file up and thereby the first in the list.



- moves the selected document down in the document list.



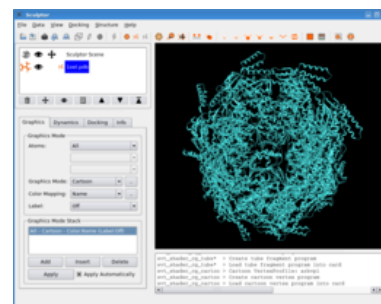
- makes the selected document the top in the document list.



### Save and Restore States

Sculptor can store the current state of the program in an xml file with the extension '.scl'. This state file will keep all the information about the current documents and the visualization modes and settings. One can save and load a state file with the menu items 'File->Load State' and 'File->Save State'.

Attention: The state file will only store the location of the document files like '.pdb' or '.mrc' files. If the location of the files changes, the state cannot be restored properly. The state files are ascii xml files, so by editing in a text editor one might be able to restore the file.




## Global rendering settings

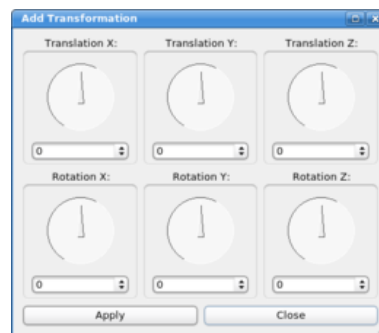
In Sculptor you can find under the view submenu all the global settings and rendering functions. One can take screenshots of the renderings, change the background color, zoom in and out or change to orthographic mode. The settings should be self-explanatory.



## Transformations

In multi-scale modeling applications a very precise control over the transformations applied to the probe molecule is necessary to ensure an accurate docking result. Therefore Sculptor provides two basic modes of interactions: With the mouse molecular structures can be moved on the X-Y plane (right mouse button), on the X-Z plane (middle mouse button) and rotate around the origin (left mouse button).

 - if precise control is needed, a transformation dialog can be activated (menu: 'Docking->Transformation'). Six dials control the six degrees of freedom of a transformation in 3D space. The top dials determine the three translation components, a translation along the X, the Y, and the Z axis. Each step on one of the dials corresponds to 0.1 Angstrom. The lower three dials determine the three components of the rotation, along the X, the Y, and the Z axis. Each step rotates the object by 1 degree.



## File Menu

### Open

With the file-open menu item one can load data sets into Sculptor. In most cases Sculptor automatically recognizes the file-type based on the extension. Sculptor supports the following file-types:

PDB - standard protein data base files. Sculptor supports multiple models and interprets the END statement as frame-end delimiter (for trajectories).

PDB/PSF - Sculptor can load in PDB+PSF file combos. Please select "Molecule (\*.pdb \*.psf)" from the file-type pull-down. Now double-click on the PDB file first and then double-click on the PSF file - Sculptor will immediately reopen the file dialog after the PDB file was selected.

PSF/DCD - Trajectory files based on a protein structure file and a binary DCD file. Please select "Trajectory (\*.psf \*.dcd)" from the file-type pull-down. Now double-click on the PSF file first and then double-click on the DCD file - Sculptor will immediately reopen the file dialog after the PSF file was selected.

SITUS - Sculptor loads volumetric maps in the Situs format.

CCP4/MRC/MAP - Standard MRC files are supported. We have tried to be consistent with most data files and other programs that are available. Unfortunately the file format is not really standardized and therefore incompatibilities can exist. Sculptor only supports orthogonal maps.

### *Save As*

The selected document can be saved using the "Save As" menu item in the file menu. The program will always open a file dialog to give you a chance to specify a new file name to avoid overwriting your original data.

### *Load and Save State*

Sculptor is able to save and restore it's current state. This will restore all loaded documents, all visualization settings and all docking results. Attention: If one generates a completely new data set in Sculptor (for example by blurring a high-resolution structure into a volumetric map), this new document needs to be saved before the state is written out into a file.

### *User Manual - Data Menu*

All menu-items in the Data sub-menu will always change the currently selected documents in the document list. The document list is a list of loaded files in the top left corner of the Sculptor window.

### *Close*



- Closes the selected documents - they will get removed from the list of loaded files.

### *Properties*

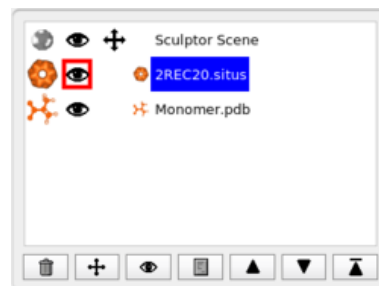


- Opens the properties dialog of the selected document.

### *View*




- Toggles the visibility of the selected document. If the "Sculptor Scene" document is currently selected all documents will be set to invisible.






The visibility can also be toggled by clicking directly into document list onto the "eye" icon.

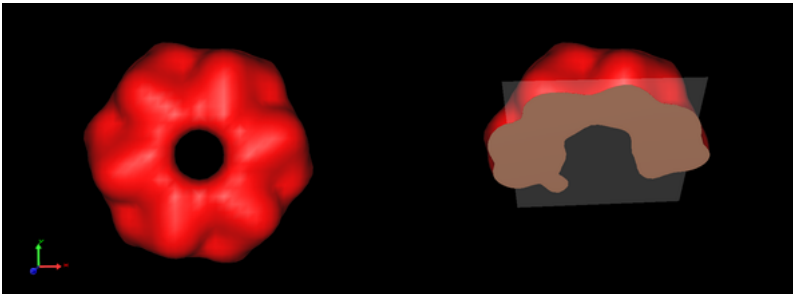
### *Top*

 - The top-most document in the document list determines the origin of the entire scene - its origin will be the origin of the Sculptor 3D scene and all other documents will be positioned relative to this one. If the origins of your files are very different, this can also mean that only some files are visible. Other are positioned so far away that they cannot be seen. By moving the document to the top position one can inspect those files.


### *View Menu*

#### *Add Clipping Plane*

 - The clipping plane cuts the 3D visualization into two areas, everything left of the plane will not get rendered anymore. The clipping plane is stored as a standard document that can be manipulated (moved around, rotated, visibility on/off, etc.) like any other document. To facilitate the positioning of the clipplane a semi-transparent rectangle is drawn on top of the plane. The rendering of the rectangle can be turned off in the property dialog of the clipplane document.



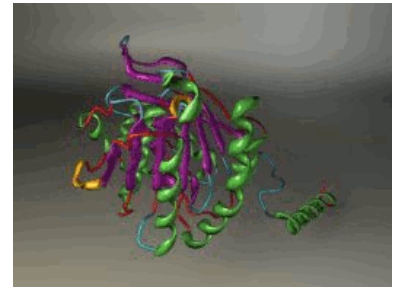
#### *Save Screenshot*

 - Saves the content of the 3D window as a jpg or png file to disk. Please make sure that the 3D window is not occluded by any other window or dialog box.

### *Export as Wavefront OBJ file*

Saves the 3D visualization as alias wavefront obj file. The file format is supported by most 3D rendering packages, which can be utilized to create high-quality images. The format does not support all features of the opengl rendering shown in the interactive 3D window, the direct volume rendering is an example of a Sculptor rendering which cannot be exported using the alias wavefront format.

The image on the right was exported from Sculptor and rendered with the free 3D modeling package Blender.



### *Change Background Color*

Opens a color chooser dialog with which one can set a new background color.

### *Zoom In / Zoom Out*



- Scales the entire Sculptor scene.

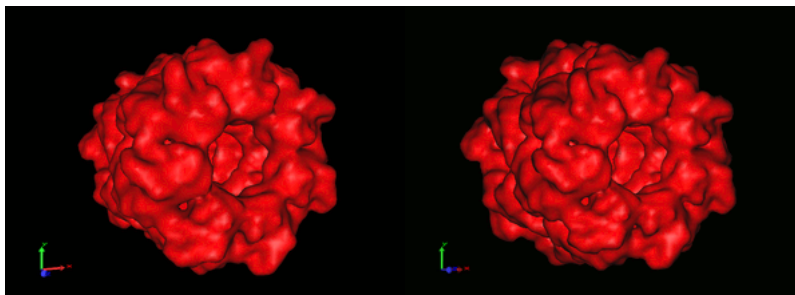
### *Center Mode*

If center mode is enabled, the rotations are applied around the center of the document (this is the default). If the mode is disabled the rotations are relative to the mouse point (or the coordinate of the input device, tracking device, etc).

### *Orthographic Projection*

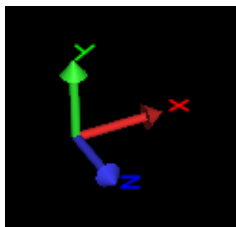


- Switches between perspective projection (default) and orthographic projection. The following example shows the same system first in perspective projection and then in orthographic projection:



### *Show 3D Coordinate System*

Turns the coordinate system on or off.



### *Show 3D Cursor*

Turns the 3D cursor on or off.

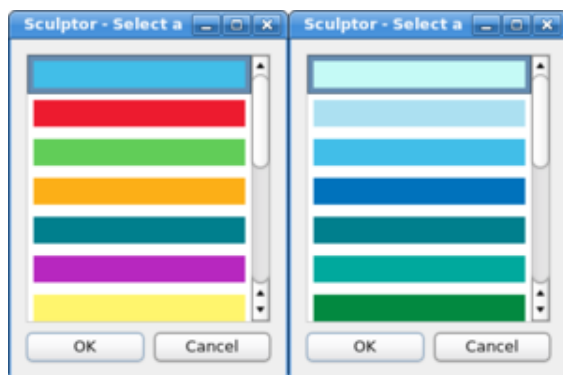


### *High Contrast Palette*

The coloring of the molecular structures is based on a standard palette - properties of the structure, for example the b-factor, chain ID, etc., are mapped to a color using a palette.

With the menu item one can switch between a high and low contrast palette. If the high contrast palette is selected, adjacent values are mapped onto very different colors, whereas the low color palette changes smoothly over the range of the values.

The following two dialog-boxes show the same region of the high-contrast and low-contrast palette:



### *Closed Surfaces at Clip-Planes*

This menu item can be used to close the borders of surfaces at clip-planes (default: off):



### *Docking Menu*

The docking menu includes various tools with which one can accomplish a multi-scale fitting. In the process a high-resolution structure (from now on called "probe-molecule") is docked into a low-resolution ("target map") volumetric map.

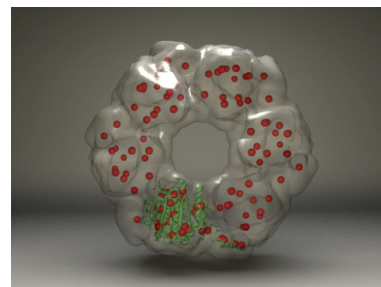
The program implements a variety of established techniques, most of which are reviewed in the following paper about hybrid modeling methods:

**Willy Wriggers and Pablo Chacón. Modeling Tricks and Fitting Techniques for Multi-Resolution Structures. Structure, 2001, Vol. 9, pp. 779-788.**

The efficient feature-based M-to-N docking algorithm is described in the following article:

**Stefan Birmanns and Willy Wriggers. Multi-Resolution Anchor-Point Registration of Biomolecular Assemblies and Their Components. J. Struct. Biol. 2007, Vol. 157, pp. 271-280.**

The user manual describes the individual menu-items one by one - the tutorial section reports in a more application oriented fashion how one can carry out a multi-resolution fitting.



### *Set Target Map*



- As one can load multiple data sets into Sculptor, the user has to inform the program which of the files is the target map for the multi-resolution fitting. Once the document is selected, a small icon next to its file-name will also show its special state as target map:

The volumetric map which was loaded last will automatically be selected as target map.



 1oel.situs

### *Set Probe Molecule*



- As one can load multiple data sets into Sculptor, the user has to inform the program which of the files is the probe-molecule for the multi-resolution fitting.

The structure which was loaded last will automatically be selected as probe molecule.

### *Cross Correlation*



- In signal processing cross-correlation is an established criterion to determine the agreement or similarity of two signals. It can be used in the context of multi-resolution fitting as scoring function to compare multiple docking solutions.

Select two volumetric maps from the list of documents and click on the cross-correlation menu item. The program will calculate and report the cross-correlation coefficient between the two maps. If a structure was docked into a map and the quality of the fit should be measured using the CCC, the structure has to be converted into a volumetric description first. Please click on the document and selected "Structure->Blur" to generate a volumetric map. The resulting volume can then be utilized in a second step for the correlation coefficient calculation.

### *Feature Extraction*

The items in the sub-menu extract feature-points from multi-resolution data sets. These feature-points can then in a second step be utilized to find solutions to the multi-resolution docking problem (see feature-based docking below).

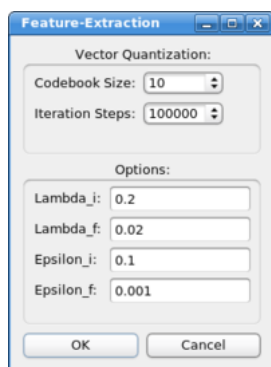
Feature-points can be determined using the neural gas algorithm or laplacian quantization. The algorithmic details are described in the following article:

**Stefan Birmanns and Willy Wriggers. Multi-Resolution Anchor-Point Registration of Biomolecular Assemblies and Their Components. J. Struct. Biol. 2007, Vol. 157, pp. 271-280.**

### *Neural Gas*

The neural gas is a classic artificial neural network algorithm, from the category of the self-organizing maps. The neural net is trained

to represent the original data in a best possible way (least information loss). After clicking on "Feature Extraction->Neural Gas" the following dialog box appears:



The main parameter is the size of the codebook, which determines the number of the features the algorithm will extract from the data. The other parameters concern the inner workings of the algorithm, they should not be changed in common docking applications. The following article describes the algorithm and the parameters:

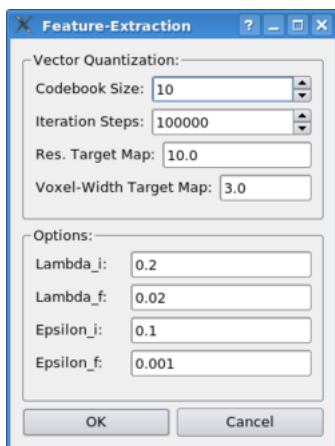
**A "neural-gas" network learns topologies. In T. Kohonen, K. Mäkisara, O. Simula, and J. Kangas, editors, *Artificial Neural Networks*, pages 397-402. North-Holland, Amsterdam, 1991.**

### *Laplacian Quantization*

The laplacian quantization combines the laplacian filter with the vector quantization algorithm, yielding in superior performance for volumetric maps with a low resolution. In order to be able to apply the laplace filter the algorithm will convert the high-resolution structures into volumetric maps internally. In order to be consistent, the algorithm needs to know the resolution and voxel width of the target map when quantizing a structure:

### *Atomic Coordinates*

This option is typically not useful for a normal multi-resolution docking application - it provides a simple feature extraction based on the atomic coordinates of a molecular structure. This is only useful for the comparison of two structures, do not use this option and compare the feature vectors with neural gas generated points!

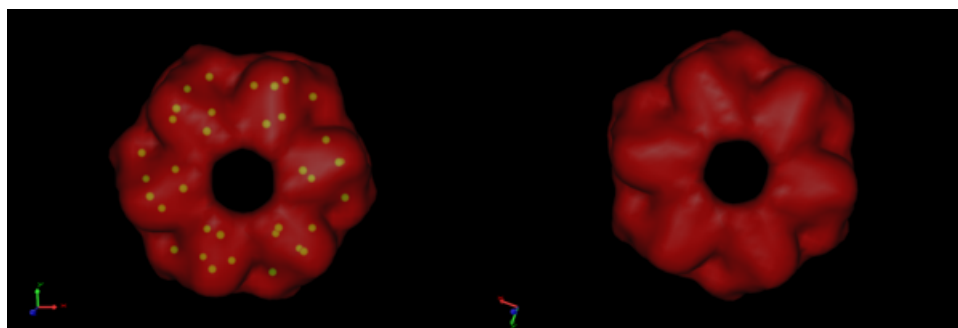


### *Import / Export Feature Vectors*

The feature vectors are stored by default in the Sculptor state file ("File->Save State" and "File->Load State", see the documentation of the File menu). If one would like to use already previously calculated feature-points from another program (for example from the Situs qpdb and qvol tools), or if one would like to export the coordinates to other programs, one can use Sculptor to import or export the points in PDB format.

### *Render Feature Vectors*

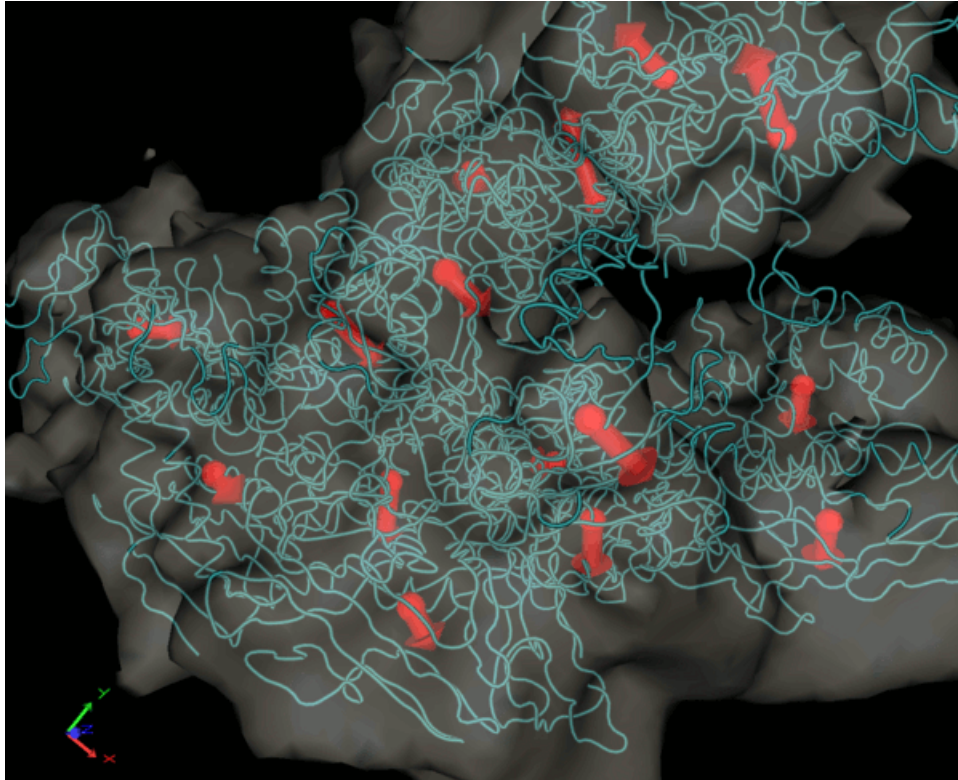
The visualization of the feature vectors can be turned on or off with this menu item:



### *Render Displacements*

In addition to the normal feature-vector rendering one can also render the deviation of two feature-point sets using arrows (for example in order to visualize the conformational change between two data sets). Please select two documents in the document list (hold down the ctrl

key and click onto the two documents). The two documents have to have feature-point sets associated with them (for example from a feature extraction with the neural gas algorithm). Once you click on the menu item, the visualization should change and arrows between the two point-sets will appear:





# *Visualization*

## *Volume Rendering*

### *Iso Surfaces*

#### *Direct Volume Rendering*

The before mentioned iso surface renderings convert the volumetric data into a triangular mesh. This is done by essentially thresholding the map - wherever the voxel densities reach a certain threshold, triangles are added and the surface is drawn. Unfortunately thresholding essentially converts the floating-point data set into a simple binary map, all the density variations are lost / not shown in the triangular iso surfaces.

An alternative representation for volumetric data is

### *Map Explorer*



# *The Sculptor Scripting Interface*

## *Introduction*

SCULPTOR VERSION 1.3 and above features an embedded Lua interpreter which is used as scripting language. Sculptor internally is fully implemented in C++, which is a standard, object-oriented, compiled language. We are happy with C++ and do not plan to switch to Lua or any other interpreted language for our main algorithm development. Lua, as a true scripting language on the other hand, allows us to quickly test new ideas that we later might port to C++. It is also very useful for end-users as they can implement their own molecular modeling techniques, without having to set-up a C++ programming environment. In fact, Lua routines can be programmed directly in Sculptor, in the internal script-editor, and can be executed immediately without any compilation step. The scripting interface is also very convenient to automatize certain routine procedures, like for example loading a series of files and setting up their visualization. Why Lua and not XXX (fill in your favorite interpreted programming language here)? There are literally hundreds of scripting languages available, whereby most of them nowadays are not only praised and advertised for light programming tasks, but also for real application development. If one intends to write a significant portion of the code of a larger application in a scripting language, this changes the requirements and makes certain languages more attractive than others. On the other hand, as mentioned above, Sculptor is implemented in C++ and the scripting language is used by users and collaborators to write small external routines and short scripts. For Sculptor it is therefore important that the language can be embedded seamlessly into the main application, that it does not create any problems when it is ported to other platforms, and that it does not create any links to external libraries. Lua is ideal in this respect as it is very compact and stable. Lua is fully embedded into Sculptor and does not rely on any other operating system components or external libraries.

Sculptor version 1.3 and above features an embedded Lua interpreter which is used as scripting language. See <http://www.lua.org> for more information. Sculptor itself is fully implemented in C++.

## Why Lua?

The following text from <http://www.lua.org/about.html> summarizes important properties of the Lua scripting language and thereby indirectly provides some additional reasons why we chose Lua over other alternatives:

**What is Lua?** Lua is a powerful, fast, lightweight, embeddable scripting language. Lua combines simple procedural syntax with powerful data description constructs based on associative arrays and extensible semantics. Lua is dynamically typed, runs by interpreting byte-code for a register-based virtual machine, and has automatic memory management with incremental garbage collection, making it ideal for configuration, scripting, and rapid prototyping.

**Why choose Lua?** Lua is a proven, robust language. Lua has been used in many industrial applications (e.g., Adobe's Photoshop Lightroom), with an emphasis on embedded systems (e.g., the Ginga middleware for digital TV in Brazil) and games (e.g., World of Warcraft). Lua is currently the leading scripting language in games. Lua has a solid reference manual and there are several books about it.

**Lua is fast!** Lua has a deserved reputation for performance. To claim to be "as fast as Lua" is an aspiration of other scripting languages. Several benchmarks show Lua as the fastest language in the realm of interpreted scripting languages. Lua is fast not only in fine-tuned benchmark programs, but in real life too. A substantial fraction of large applications have been written in Lua.

**Lua is portable.** Lua is distributed in a small package and builds out-of-the-box in all platforms that have an ANSI/ISO C compiler. Lua runs on all flavors of Unix and Windows, and also on mobile devices (such as handheld computers and cell phones that use BREW, Symbian, Pocket PC, etc.) and embedded microprocessors (such as ARM and Rabbit) for applications like Lego MindStorms.

**Lua is embeddable.** Lua is a fast language engine with small footprint that you can embed easily into your application. Lua has a simple and well documented API that allows strong integration with code written in other languages. It is easy to extend Lua with libraries written in other languages. It is also easy to extend programs written in other languages with Lua. Lua has been used to extend programs written not only in C and C++, but also in Java, C#, Smalltalk, Fortran, Ada, and even in other scripting languages, such as Perl and Ruby.

**Lua is powerful (but simple).** A fundamental concept in the design of Lua is to provide meta-mechanisms for implementing features, instead of providing a host of features directly in the language. For example, although Lua is not a pure object-oriented language, it does provide meta-mechanisms for implementing classes and inheritance.

"Lua is a powerful, fast, lightweight, embeddable scripting language." "Lua is a proven, robust language [and] has been used in many industrial application (e.g., Adobe's Photoshop Lightroom)." <http://www.lua.org/about.html>



Lua's meta-mechanisms bring an economy of concepts and keep the language small, while allowing the semantics to be extended in unconventional ways.

**Lua is small.** Adding Lua to an application does not bloat it. The tarball for Lua 5.1.4, which contains source code, documentation, and examples, takes 212K compressed and 860K uncompressed. The source contains around 17000 lines of C. Under Linux, the Lua interpreter built with all standard Lua libraries takes 153K and the Lua library takes 203K.

**Lua is free.** Lua is free software, distributed under a very liberal license (the well-known MIT license). It can be used for any purpose, including commercial purposes, at absolutely no cost. Just download it and use it.

## *Syntax*

Users familiar with other programming languages should not have any difficulties to learn Lua. It is very similar to other high-level languages. A full documentation of Lua is available for free online at <http://www.lua.org>, in addition various books are available for purchase at bookstores and Amazon.

In the following the main syntax elements are quickly highlighted:

### *Comments*

Simply start with two hyphens –:

```
-- this is a comment
```

### *Variables*

Lua knows strings, numbers and booleans as basic types. As Lua is dynamically typed, i.e. one can just start using variables without having to declare them:

```
x = 5
text = "Hello"
question = false
```

There are no integer variables (only doubles) and variables are by default always global. Using the local keyword one can make them local to a function.

```
function localTest()
  local counter = 0
  print( counter )
end
```

More documentation can be found on the Lua website:

<http://www.lua.org/docs.html>.

There are also several books about Lua available, especially "Programming in Lua by Roberto Ierusalimsky, March 2006, ISBN 85-903798-2-5" can be recommended as standard textbook.



Attention: Variables are by default global. After a function returns, the variables that were created inside the function stay visible (unless the local keyword is used).

## *Arrays*

Arrays can be accessed using the square-bracket operator and **start with index 1** and not 0, like in C!

Attention: Arrays start with the index 1!

```
text = "Hello"
print( text[1] )
```

## *Strings*

Strings can be concatenated using two dots:

```
text = "Hello" .. "World"
print( text )
```

## *Control the program flow*

The syntax of if statements is very similar to other programming languages:

```
if (x==5) then
  print("OK")
end
```

## *Loops*

For loops use an index variable and three parameters. The first parameter is the starting value of the variable, the second is the final value and the third the amount by which the index should increase at every iteration. The third parameter is optional and will be 1 by default:

```
for i=1,10,2 do
  print(i)
end
```

## *Functions*

Functions can be defined by using the function keyword:

```
function f(a, b)
  print( a, b )
end
```

Using the return keyword the function can also return one or several values:

```
-- function definition
function f(a, b)
```

```
print( a, b )
return a*b,a+b
end
-- now lets test the function
c,d = f(1,2)
print(c,d)
```

### *Print / Printf*

A special lua printf function was added that prints directly to std-out instead of into the sculptor log-window. The behaviour is identical to the normal print function. Such a printf function is useful for regression-test-scripts, which can output values (e.g. rmsds, correlation) that can be piped into files and then be compared with pre-computed values. The test-script can then be started from the command-line "sculptor test.lua" and as last statement the script kills the sculptor application with sculptor:quit(). One can run series of those regression-tests before a new release to make sure that the introduction of new features did not break any older algorithms.

The information above should enable you to write simple scripts, for a more in-depth description of Lua please go to <http://www.lua.org> or buy one of the books about the language.

### *Reference Manual*

In the following the special Sculptor-Lua classes are documented and their member-functions are listed with a simple example that illustrates their Usage. The molecule, volume and matrix classes are all normal, dynamic classes, which means that one first has create objects/instances before one can use them. The Sculptor class is different in this respect as there is always a sculptor object available (and one should not attempt to create any new object of the type sculptor). The idea is that the sculptor object represents the main application program, whereas one of course can create and delete molecular models or volume data dynamically.

New objects of the types volume, molecule and matrix are typically created using get functions of other objects: For example

```
vol = sculptor:getDoc(2)
```

returns the second document. One can also generate new objects using the constructor:

```
vol = volume:new( 10, 10, 10 )
```

## SCULPTOR

The Sculptor class encapsulates the main application program and allows the user to load and retrieve documents and to make global adjustments to the program.

---

### *Load a file*

**Name:** load

**Desc.:** Loads a file into the main application. This function call is equivalent to clicking on "File->Load" in the graphical user interface. The extension of the file will be used to determine the file type (e.g. in case of "file.pdb" Sculptor will attempt to load the file as an atomic model). The loaded file will appear in the document list, just like any other file loaded interactively through the main user interface.

**Param.:** String with the filename

**Usage:**

```
sculptor:load("test.pdb")
```

---

### *Save the sculptor state*

**Name:** save

**Desc.:** Saves the state of the Sculptor program in a scl state file.

**Param.:** String with the filename

**Usage:**

```
sculptor:save("current.scl")
```

---

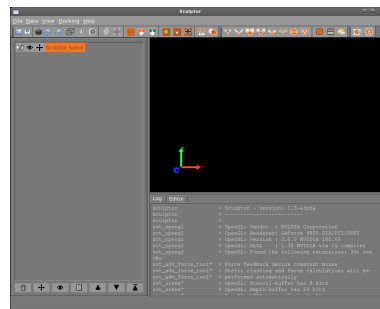
### *Get the number of currently loaded documents.*

**Name:** numDoc

**Desc.:** Returns the number of currently loaded files / existing documents in Sculptor.

**Param:** None

**Usage:**





```
num = sculptor:numDoc()
```

---

#### *Get a document*

**Name:** getDoc

**Desc.:** Retrieves a document from the main applications and returns it either as a volume or molecule object to the Lua program.

**Param.:** Index of the document. First document is the Sculptor scene, so the first document loaded by the user is index 2.

**Usage:**

```
mol = sculptor:getDoc( 2 )
```

---

#### *Delete a document*

**Name:** delDoc

**Desc.:** Deletes a document from the main program. This is equivalent to clicking on "Data->Close" or clicking on the little trash-can icon.

**Param.:** Index of the document. First document is the Sculptor scene, so the first document loaded by the user is index 2.

**Usage:**

```
sculptor:delDoc( 2 )
```

---

#### *Delete all documents*

**Name:** delAllDocs

**Desc.:** Delete all documents in the main program.

**Param.:** None

**Usage:**

```
sculptor:delAllDocs( )
```

---

*Get the sculptor version number*

**Name:** version

**Desc.:** Returns a string with the version number

**Param.:** None

**Usage:**

```
print( sculptor:version() )
```

---

*Print the current svt tree*

**Name:** printTree

**Desc.:** Prints the internal svt tree to the log window. This function is only useful for internal debugging.

**Param.:** None

**Usage:**

```
sculptor:printTree()
```

---

*Make a document visible*

**Name:** showDoc

**Desc.:** This function makes a document visible and is equivalent to clicking on the eye symbol.

**Param.:** Number of document

**Usage:**

```
sculptor:showDoc(2)
```

---

*Hide a document*

**Name:** hideDoc

**Desc.:** This function makes a document invisible and is equivalent to clicking on the eye symbol.

**Param.:** Number of document

**Usage:**

```
sculptor:hideDoc(2)
```

---

*Redraw of the 3D window***Name:** redraw**Desc.:** This function triggers a redraw of the 3D window.**Param.:** None**Usage:**

```
sculptor:redraw()
```

---

*Sleep***Name:** sleep**Desc.:** Lets sculptor sleep for a certain number of milliseconds.**Param.:** Number of milliseconds.**Usage:**

```
sculptor:sleep(10)
```

---

*Clear log window***Name:** clearLog**Desc.:** Deletes all the output from the log window.**Param.:** None**Usage:**

```
sculptor:clearLog()
```

---

*Get the current directory*

**Name:**     getCurrentDir

**Desc.:**     Returns the current directory where Sculptor carries out its functions

**Param.:**    None

**Usage:**

```
sculptor:getCurrentDir()
```

---

*Set the current directory*

**Name:**     setCurrentDir

**Desc.:**     Sets the current directory where Sculptor operates

**Param.:**    String with the new current directory.

**Usage:**

```
sculptor:setCurrentDir("tmp")
```

---

*Save a screenshot to disk*

**Name:**     saveScreenshot

**Desc.:**     Saves a screen shot to disk. The routine can save jpg and png files and will attempt to determine the file type using the extension of the filename.

**Param.:**    String with the filename of the image file.

**Usage:**

```
sculptor:saveScreenshot("scr.png")
```

---

*Get global scene transformation matrix*

**Name:**     getTrans

**Desc.:**     Returns the current global scene transformation matrix (similar to a camera matrix).

**Param.:** None

**Returns:** matrix4 object

**Usage:**

```
mat = sculptor:getTrans()  
mat:print()
```

---

*Sets global scene transformation matrix*

**Name:** setTrans

**Desc.:** Sets the current global scene transformation matrix (similar to a camera matrix).

**Param.:** matrix4 object.

**Usage:**

```
mat = sculptor:getTrans()  
mat:rotate(2, 10)  
sculptor:setTrans( mat )
```

---

*Close sculptor*

**Name:** quit

**Desc.:** This function will still check for new documents that were created, but not saved yet and will prompt the user. In a test-script that might not be desirable - just delAllDocs first to make sure that the quit() function will succeed without any user intervention.

**Param.:** None

**Usage:**

```
sculptor:quit( )
```

---

### *Open a file-open-dialog*

**Name:** guiFileOpenDlg

**Desc.:** Opens a file-dialog and returns a string with a file name. If user cancelled the dialog, the string will be empty.

**Param.:** Message

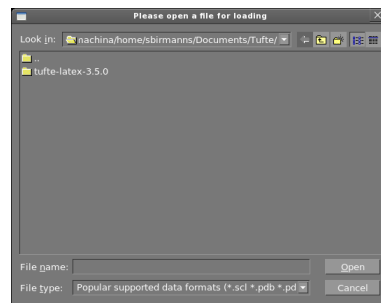
**Returns:** String with selected file name

**Usage:**

```
file = sculpтор:guiFileOpenDlg("Please select a file")
sculpтор:load( file )
```

---

sculpтор:guiFileOpenDlg opens a file-open dialog



### *Open a file-save-dialog*

**Name:** guiFileSaveDlg

**Desc.:** Opens a file-save dialog and returns a string with a file name. If user cancelled the dialog, the string will be empty.

**Param.:** Message

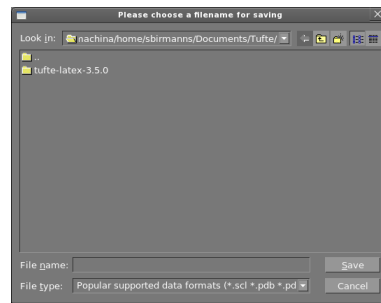
**Returns:** String with selected file name

**Usage:**

```
file = sculpтор:guiFileSaveDlg("Please choose a file")
```

---

sculpтор:guiFileSave opens a file-save dialog



### *Open a warning-dialog*

**Name:** guiWarning

**Desc.:** Opens a warning dialog with a message for the user.

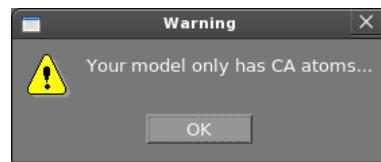
**Param.:** Message

**Usage:**

```
sculpтор:guiWarning("Your model only has CA atoms...")
```

---

sculpтор:guiWarning opens a warning dialog



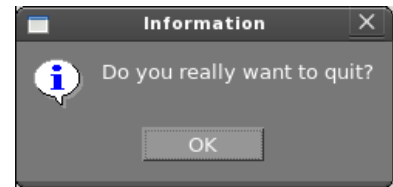
### *Open an information-dialog*

**Name:** guiInfo  
**Desc.:** Opens an information dialog with a message for the user.  
**Param.:** Message  
**Usage:**  

```
sculptor:guiInfo("Your model only has CA atoms...")
```

---

sculptor:guiInfo opens an information dialog



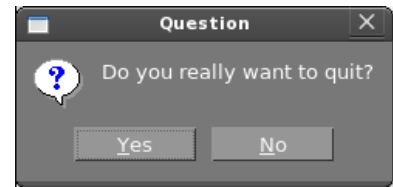
### *Open a yes/no question-dialog*

**Name:** guiYesNo  
**Desc.:** Opens a yes/no dialog with a question for the user.  
**Param.:** Message  
**Returns:** Boolean with the value true if the user clicks on Yes.  
**Usage:**  

```
answer = sculptor:guiYesNo("Do you really want to quit?")
```

---

sculptor:guiYesNo opens a dialog box with a question for the user



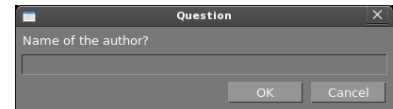
### *Open a free-text question-dialog*

**Name:** guiQuestion  
**Desc.:** Opens a question dialog, where the user can reply with a text  
**Param.:** Message  
**Returns:** String with answer  
**Usage:**  

```
answer = sculptor:guiQuestion("Name of the author?")
```

---

sculptor:guiQuestion opens a question dialog

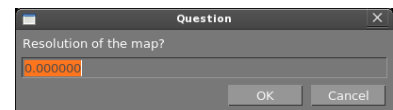


### *Open a value question-dialog*

**Name:** guiGetValue  
**Desc.:** Opens a dialog where the user can type in a value.  
**Param.:** Message  
**Usage:**  

```
answer = sculptor:guiGetValue("Resolution of the map?")
```

sculptor:guiGetValue opens a dialog that requests a value



## ATOM

The atom class encapsulates a single atom. An atom object is typically retrieved from a molecule object.

---

### *Get displaymode*

**Name:** getDisplayMode

**Desc.:** Retrieves the current display mode of an atom object.

**Param.:** None

**Returns:** String with mode name, e.g. CARTOON

**Usage:**

```
mode = atm:getDisplayMode()
```

---

### *Set displaymode*

**Name:** setDisplayMode

**Desc.:** Sets the current display mode of an atom.

**Param.:** String with the display mode, e.g. VDW or CARTOON

**Usage:**

```
atm:setDisplayMode('CARTOON')
```

---

### *Set/get the x coordinate of the position of the atom*

**Name:** x

**Desc.:** If no parameter is given, the function just returns a number, otherwise it will set the x coordinate using the parameter provided by the user.

**Param.:** Optional, x coordinate

**Returns:** If no parameter given, it returns a number, the x coordinate.

**Usage:**

```
atm_a:x( 15.0 )  
print( atm_a:x() )
```

---



*Set/get the y coordinate of the position of the atom*

**Name:** y

**Desc.:** If no parameter is given, the function just returns a number, otherwise it will set the y coordinate using the parameter provided by the user.

**Param.:** Optional, y coordinate

**Returns:** If no parameter given, it returns a number, the y coordinate.

**Usage:**

```
atm_a:y( 15.0 )  
print( atm_a:y() )
```

---

*Set/get the z coordinate of the position of the atom*

**Name:** z

**Desc.:** If no parameter is given, the function just returns a number, otherwise it will set the z coordinate using the parameter provided by the user.

**Param.:** Optional, z coordinate

**Returns:** If no parameter given, it returns a number, the z coordinate.

**Usage:**

```
atm_a:z( 15.0 )  
print( atm_a:z() )
```

---

*Print content to stdout*

**Name:** print

**Desc.:** Prints information about the atom to the sculptor log window.

**Param.:** None

**Usage:**

```
atm:print()
```

---

*Measure distance to another atom.*

**Name:** distance

**Desc.:** Measures the distance from one atom to another one.

**Param.:** Atom object.

**Returns:** Number.

**Usage:**

```
dist = mol:getAtom(123):distance( mol:getAtom(124) )
```

*Powell refinement*

**Name:** powell

**Desc.:** Compute the powell off-lattice refinement of an array of molecules against a volume object. The user has to specify an array of molecules, which of course can just consist of a single molecule. In case of multiple models in the array, a simultaneous refinement is carried out.

**Param.:** array of molecule objects  
volume object,  
number with resolution of volumetric map,  
boolean - should the model be low-pass filtered or not (attention: takes significantly more time!)

**Usage:**

```
sculptor:load("sculptor_powell.scl")
vol = sculptor:getDoc(2)
a = {}
a[1] = sculptor:getDoc(3)
a[2] = sculptor:getDoc(4)
a[3] = sculptor:getDoc(5)
a[4] = sculptor:getDoc(6)
--
-- union sphere correlation, blur = false
--
sculptor:powell(a,vol,10, false)
```

---

## MOLECULE

The molecule class encapsulates a single molecule, i.e. a Sculptor atomic model document.

---

### *Add the molecule to the list of documents*

**Name:** addDoc

**Desc.:** Add a Lua-internal molecule to the Sculptor list of documents.

**Param.:** String with the name

**Usage:**

```
addDoc( "test.pdb" )
```

---

### *Get an atom from the molecule*

**Name:** getAtom

**Desc.:** Retrieve a specific atom out of all the atoms of the molecule

**Param.:** Index

**Returns:** An atom object

**Usage:**

```
atm = mol:getAtom( 123 )
```

---

### *Set an atom in the molecule*

**Name:** setAtom

**Desc.:** The atom contains the index, so it knows where it was retrieved from and will go back to exactly the same position. If a new atom is supposed to be stored in the molecule, please use addAtom.

**Param.:** Atom object

**Usage:**

```
atm = mol:getAtom( 123 )  
atm.x( 10.5 )  
mol:setAtom( atm )
```

---

*Get a series of atoms from the molecule*

**Name:** getAtoms

**Desc.:** Retrieve a subset of atoms from the molecule

**Param.:** Two indices

**Returns:** An array of atoms

**Usage:**

```
atm = mol:getAtoms( 123, 135 )
```

---

*Set an array of atoms in the molecule*

**Name:** setAtoms

**Desc.:** The atoms contain the index, so they know where they were retrieved from and will go back to exactly the same position. If a new atom is supposed to be stored in the molecule, please use addAtom.

**Param.:** Array of atoms

**Usage:**

```
atm = mol:getAtoms( 123, 135 )  
atm[5]:x( 10.5 )  
mol:setAtoms( atm )
```

---

*Add an atom to the molecule*

**Name:** addAtom

**Desc.:** The atom is stored as a new atom in the molecule, will therefore get a new svt index.

**Param.:** Atom

**Usage:**

```
atm = mol:getAtom( 123 )  
atm:x( 10.5 )  
mol:addAtom( atm )
```

---

*Add an array of atoms to the molecule*

**Name:** addAtoms

**Desc.:** The atoms are stored as new atoms in the molecule, and will therefore get a new svt index.

**Param.:** Array of atoms

**Usage:**

```
atm = mol:getAtoms( 123, 135 )
atm[1]:x( 10.5 )
mol:addAtoms( atm )
```

---

*Get the number of atoms in the molecule*

**Name:**

**Returns:** Number

---

*Compute RMSD*

**Name:** rmsd

**Desc.:** Compute the rmsd with another molecule object.

**Param.:** molecule object,  
boolean: align the two structures before rmsd (true/false)

**Usage:**

```
rmsd = mol:rmsd(mol2, false)
```

---

*Create volumetric map*

**Name:** blur

**Desc.:** Blur an atomi model by convoluting it with a Gaussian kernel

**Param.:** voxelwidth, resolution

**Returns:** a volume

**Usage:**

```
vol = mol:blur( 3.0, 10.0 )
```

---

*Get transformation matrix*

**Name:** getTrans

**Desc.:** Get the transformation matrix

**Returns:** matrix4 object

**Usage:**

```
mat = mol:getTrans()  
mat:print()
```

---

*Sets the transformation matrix*

**Name:** getTrans

**Desc.:** Set the transformation matrix

**Param.:** matrix4 object

**Usage:**

```
mat = mol:getTrans()  
mat:rotate(2, 10)  
mol:setTrans( mat )
```

---

*Update the rendering of the molecule*

**Name:** updateRendering

**Desc.:** Updates the visual rendering of the molecule. Should be called if the data was manipulated internally.

**Param.:** None

**Usage:**

```
mol:updateRendering()
```

---

### *Save the molecule to disk*

**Name:** save

**Desc.:** This function saves the atomic model to disk as a pdb file.

**Param.:** Filename

**Usage:**

```
mol:save("test.pdb")
```

---

### *Load an atomic model from disk*

**Name:** load

**Desc.:** The molecule will not get added to the document list of Sculptor (can be done later using the addDoc function). This function is therefore not identical to the function load of the Sculptor class, which one typically would call. The function here is useful if one needs to have access to a certain atomic structure temporarily and would like to avoid loading it into Sculptor as a real document (for which e.g. a visualization is created, etc).

**Param.:** Filename

**Usage:**

```
mol = molecule:new()  
mol:load("test.pdb")
```

---

### *Vectorquantization*

**Name:** vectorquant

**Desc.:** Create codebookfeature vectors using the neural gas / TRN algorithm.

**Param.:** None

**Usage:**

```
mol:vectorquant()
```

---

### *Set displaymode*

**Name:** setDisplayMode

**Desc.:** Set the global display-mode of the molecular structure.

**Param.:** String with the name of the display mode, e.g. "CAR-  
TOON"

**Usage:**

```
mol:setDisplayMode("CARTOON")
```

---

### *Set colmapmode*

**Name:** setColmapMode

**Desc.:** Set the color mapping mode.

**Param.:** Colormapping mode, e.g. "SOLID"  
Number of color (from the Sculptor palette)

**Usage:**

```
mol:setColmapMode("SOLID", 5)
```

---

### *Project-Mass Correlation*

**Name:** projectMassCorr

**Desc.:** This routine only projects the atoms onto the volume object and calculates the correlation - it will not convolute the molecule with a Gaussian to bring it to the same resolution as the volumetric data.

**Param.:** Volume object

**Usage:**

```
cc = mol:projectMassCorr( volume )
```

---



*Get secondary structure information*

**Name:** getAtomSecStruct

**Desc.:** Retrieve the secondary structure information of atom i.

**Param.:** Index

**Returns:** String

**Usage:**

```
ss = mol:getAtomSecStruct( 123 )
```

---

*Get atom type information*

**Name:** getAtomType

**Desc.:** Get the atom type information for a specific atom in the molecule

**Param.:** Index

**Returns:** String

**Usage:**

```
type = mol:getAtomType( 123 )
```

---

*Set atom type information*

**Name:** setAtomType

**Desc.:** Set the atom type information for a specific atom in the molecule

**Param.:** Index,  
String with type information

**Usage:**

```
mol:setAtomType( 123, "H" )
```

---

*Get atom model information*

**Name:** getAtomModel

**Desc.:** Get atom model information for atom i

**Param.:** Index

**Returns:** Integer

**Usage:**

```
model = mol:getAtomModel( 123 )
```

---

*Set atom model information*

**Name:** setAtomModel

**Desc.:** Set atom model information for atom i

**Param.:** Index,  
Integer with the new model identifier

**Usage:**

```
mol:setAtomModel( 123, 3 )
```

---

*Get remoteness information*

**Name:** getAtomRemoteness

**Desc.:** Get remoteness information for atom i (e.g. alpha for a carbon alpha atom).

**Param.:** Index

**Returns:** String

**Usage:**

```
rem = mol:getAtomRemoteness( 123 )
```

---

*Set remoteness information*

**Name:** setAtomRemoteness

**Desc.:** Set remoteness information for atom i (e.g. alpha for a carbon alpha atom).

**Param.:** Index

**Usage:**

```
mol:setAtomRemoteness( 123 )
```

---

*Get branch information*

**Name:** getAtomBranch

**Desc.:** Get branch information of atom i.

**Param.:** Index

**Returns:** String

**Usage:**

```
branch = mol:getAtomBranch( 123 )
```

---

*Set branch information*

**Name:** setAtomBranch

**Desc.:** Set branch information of atom i.

**Param.:** Index

**Usage:**

```
mol:setAtomBranch( 123 )
```

---

*Get alternate location indicator*

**Name:** getAtomAltLoc

**Desc.:** Get alternate location indicator information of atom i.

**Param.:** Index

**Returns:** String

**Usage:**

```
alt = mol:getAtomAltLoc( 123 )
```

---

*Set alternate location indicator*

**Name:** setAtomAltLoc

**Desc.:** Set alternate location information of atom i.

**Param.:** Index

**Usage:**

```
mol:setAtomAltLoc( 123 )
```

---

*Get residue name*

**Name:** getAtomResName

**Desc.:** Get residue name information of atom i (e.g. ALA for alanin).

**Param.:** Index

**Returns:** String

**Usage:**

```
res = mol:getAtomResName( 123 )
```

---

*Set residue name*

**Name:** setAtomResName

**Desc.:** Set residue name information of atom i (e.g. ALA for alanin).

**Param.:** Index

**Usage:**

```
mol:setAtomResName( 123 )
```

---

*Get residue number*

**Name:** getAtomResNum

**Desc.:** Get residue number information of atom i.

**Param.:** Index

**Returns:** Integer

**Usage:**

```
res = mol:getAtomResNum( 123 )
```

---

*Set residue number*

**Name:** setAtomResNum

**Desc.:** Set residue number information of atom i.

**Param.:** Index

**Usage:**

```
mol:setAtomResNum( 123, 123 )
```

---

*Get chain id*

**Name:** getAtomChain

**Desc.:** Get chain id information of atom i (e.g. A, B, C, ...).

**Param.:** Index of the atom

**Returns:** String

**Usage:**

```
chain = mol:getAtomChain( 123 )
```

---

### *Set chain id*

**Name:** setAtomChain

**Desc.:** Set chain id information of atom i (e.g. A, B, C, ...).

**Param.:** Index of the atom  
String with the chain identifier

**Usage:**

```
mol:setAtomChain( 123, "A" )
```

---

### *Get icode*

**Name:** getAtomICode

**Desc.:** Get icode (insertion of residues) information of atom i.

**Param.:** Index

**Returns:** String

**Usage:**

```
chain = mol:getAtomICode( 123 )
```

---

### *Set icode*

**Name:** setAtomICode

**Desc.:** Set icode (insertion of residues) information of atom i.

**Param.:** Index

**Usage:**

```
mol:setAtomICode( 123 )
```

---

### *Get occupancy*

**Name:** getAtomOccupancy

**Desc.:** Get occupancy information of atom i.

**Param.:** Index

**Returns:** Number

**Usage:**

```
occ = mol:getAtomOccupancy( 123 )
```

---

*Set occupancy*

**Name:** setAtomOccupancy

**Desc.:** Set occupancy information of atom i.

**Param.:** Index, Number

**Usage:**

```
mol:setAtomOccupancy( 123, 1.0 )
```

---

*Get temperature factor*

**Name:** getAtomTempFact

**Desc.:** Get temperature factor information of atom i.

**Param.:** Index

**Returns:** Number

**Usage:**

```
temp = mol:getAtomTempFact( 123 )
```

---

*Set temperature factor*

**Name:** setAtomTempFact

**Desc.:** Set temperature factor information of atom i.

**Param.:** Index  
Number with the temperature factor

**Usage:**

```
mol:setAtomTempFact( 123, 1.0 )
```

---

### *Get note*

**Name:** getAtomNote

**Desc.:** Get note of atom i

**Param.:** Index

**Returns:** String

#### **Usage:**

```
note = mol:getAtomNote( 123 )
```

---

### *Set note*

**Name:** setAtomNote

**Desc.:** Set note of atom i (at least three characters long!)

**Param.:** Index of the atom  
String with the note

#### **Usage:**

```
mol:setAtomNote( 123, "ABC" )
```

---

### *Get segment*

**Name:** getAtomSegID

**Desc.:** Get segment id of atom i.

**Param.:** Index

**Returns:** String

#### **Usage:**

```
seg = mol:getAtomSegID( 123 )
```

---



### *Set segment id*

**Name:** setAtomSegID

**Desc.:** Set segment id of atom i (at least four characters long!)

**Param.:** Index of the atom  
String with the segment id

**Usage:**

```
mol:setAtomSegID( 123, "ABCD" )
```

---

### *Get element*

**Name:** getAtomElement

**Desc.:** Get element information of atom i.

**Param.:** Index of atom

**Returns:** String

**Usage:**

```
element = mol:getAtomElement( 123 )
```

---

### *Set element*

**Name:** setAtomElement

**Desc.:** Set element information of atom i (at least two characters, add a space in front if you need just one!)

**Param.:** Index of atom  
String with the element information

**Usage:**

```
mol:setAtomElement( 123, " H" )
```

---

### *Get charge*

**Name:** getAtomCharge

**Desc.:** Get charge of atom i.

**Param.:** Index of atom

**Returns:** String

**Usage:**

```
element = mol:getAtomCharge( 123 )
```

---

### *Set charge*

**Name:** setAtomCharge

**Desc.:** Set charge of atom i (two characters at least!).

**Param.:** Index of atom  
String with the charge information

**Usage:**

```
mol:setAtomCharge( 123, "12" )
```

---

### *Is hydrogen?*

**Name:** isAtomHydrogen

**Desc.:** Is atom i a hydrogen?

**Param.:** Index of atom

**Returns:** Boolean

**Usage:**

```
hydro = mol:isAtomHydrogen( 123 )
```

---

*Is atom codebook vector?*

**Name:** isAtomQPDB

**Desc.:** Is atom i a codebook vector?

**Param.:** Index of atom

**Returns:** Boolean

**Usage:**

```
hydro = mol:isAtomQPDB( 123 )
```

---

*Is water molecule?*

**Name:** isAtomWater

**Desc.:** Is atom i part of a water molecule?

**Param.:** Index

**Returns:** Boolean

**Usage:**

```
hydro = mol:isAtomWater( 123 )
```

---

*Is carbon alpha?*

**Name:** isAtomCA

**Desc.:** Is atom i a carbon alpha?

**Param.:** Index

**Returns:** Boolean

**Usage:**

```
hydro = mol:isAtomCA( 123 )
```

---

*Is atom i on the backbone?*

**Name:** isAtomBackbone

**Desc.:** Is atom i part of the backbone?

**Param.:** Index of atom

**Returns:** Boolean.

**Usage:**

```
hydro = mol.isAtomBackbone( 123 )
```

---

*Is atom i a nucleotide?*

**Name:** isAtomNucleotide

**Desc.:** Is atom i a nucleotide?

**Param.:** Index of atom

**Returns:** Boolean.

**Usage:**

```
hydro = mol.isAtomNucleotide( 123 )
```

---

*Get mass*

**Name:** getAtomMass

**Desc.:** get the atomic mass

**Param.:** Index of atom

**Returns:** Number

**Usage:**

```
mass = mol.getAtomMass( 123 )
```

---

*Set mass*

**Name:** setAtomMass

**Desc.:** Set the atomic mass of an atom.

**Param.:** Index of atom  
Number

**Usage:**

```
mol:setAtomMass( 123, 1.2 )
```

*Adjust the atomic mass based on a (simple) periodic table*

**Name:** adjustAtomMass

**Desc.:** Automatically adjust the atomic mass of an atom using a simple periodic table.

**Param.:** Index of atom

**Usage:**

```
mol:adjustAtomMass( 123 )
```

---

*Get vdw radius of atom i*

**Name:** getAtomVDWRadius

**Desc.:** Get the van der Waals radius of an atom

**Param.:** Index of atom

**Returns:** Number

**Usage:**

```
rad = mol:getAtomVDWRadius( 123 )
```

---

*Get a model from the molecule*

**Name:** getModel

**Desc.:** Returns a new molecule object, with an extracted model from the current molecule.

**Param.:** Modelnumber

**Returns:** Molecule object.

**Usage:**

```
mol_b = mol:getModel( 1 )
```

---

*Get a chain from the molecule*

**Name:** getChain

**Desc.:** Extracts a chain from a molecule and returns it as a new molecule object

**Param.:** Chain ID

**Returns:** Molecule object.

**Usage:**

```
mol_b = mol:getChain( "A" )
```

---

*Compute Internal Model Information*

**Name:** calcAtomModels

**Desc.:** Builds an internal array with the different atom model numbers - typically this array is automatically build during loadPDB. If a pdb is build by hand, call this function after assembly of the structure!

**Param.:** None

**Usage:**

```
mol:calcAtomModels()
```

---

*Add another molecule*

**Name:** add

**Desc.:** Add a another molecule to this molecule object.

**Param.:** Molecule object

**Usage:**

```
mol_a:add( mol_b )
```

---

*Add a bond between two atoms*

Param.: Index a and index b of the two atoms.

Usage:

```
mol:addBond( 5, 10 )
```

---

*Is there a bond between two atoms?*

Param.: Index a and index b of the two atoms.

Usage:

```
bond = mol:isBond( 5, 10 )
```

---

*Remove a bond between two atoms*

Param.: Index a and index b of the two atoms.

Usage:

```
mol:delBond( 5, 10 )
```

---

*Remove all atoms*

Usage:

```
mol:delAllAtoms( )
```

---

*Remove all bonds*

Usage:

```
mol:delAllBonds( )
```

---

*Match two point-clouds*

**Name:** match

**Desc.:** This function should not be used directly on atomic models, but only on feature-point-clouds. First use vector-quantization to extract a small number of feature points from both data sets / models and then match those feature-points. The resulting transformation matrix can then be applied to the original atomic models. Returns an array of matrices, with the first matrix representing the most likely match.

**Param.:** Other molecule object,  
tolerance for anchor point determination (15.0),  
nearest neighbor matching zone (12.0),  
zonesize (3),  
wildcards (0)

**Usage:**

```
mats = trnmol_a:match( trnmol_b, 15, 12, 3, 0 )  
realmol_a:setTrans( realmol_a.getTrans():mult( mats[1] ) )
```

---

*Flex a molecule*

**Name:** flexing

**Desc.:** Flex a molecule according to the coordinates of two feature-point sets. The first feature-point set describes the molecule in its original conformation, the other one the target conformation (for example from a cryo-em map)

**Param.:** Two molecule objects

**Returns:** New molecule object

**Usage:**

```
flex = mol:flexing( trn_a, trn_b )
```



## VOLUME

The volume class encapsulates a single volumetric data set.

---

### *Allocate memory*

**Name:** allocate

**Desc.:** This function throws the current map of the object away and allocates memory for a new volumetric map. Attention, the old content of the volume will get erased! Typically one would rather use the constructor new function instead of allocate.

**Param.:** x,y,z with the size of the volumetric map

**Usage:**

```
vol = volume:new()  
vol:allocate(10,8,12)
```

better:

```
vol = volume:new(10,8,12)
```

---

### *Add the volume data set to the list of documents*

**Name:** addDoc

**Desc.:** Add the volume object to the list of documents in Sculptor.

**Param.:** String with the name

**Usage:**

```
vol:addDoc( "test.situs" )
```

---

### *Set the isosurface threshold level*

**Name:** setIsoThreshold

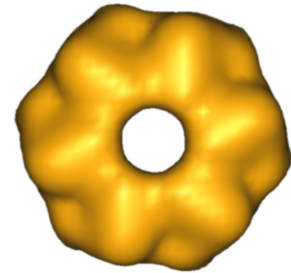
**Desc.:** Set the isosurface threshold level of the volume.

**Param.:** Value of the new isosurface threshold

**Usage:**

```
vol:setIsoThreshold( 1.0 )
```

---



### *Set wireframe*

**Name:** setWireframe

**Desc.:** Enables or disables the wireframe mode of the isosurface volume display.

**Param.:** boolean

**Usage:**

```
vol:setWireframe( true )
```

---

### *Set color*

**Name:** setColor

**Desc.:** Set the color of the isosurface rendering.

**Param.:** red, green, blue (0.0 - 1.0)

**Usage:**

```
vol:setColor( 1.0, 1.0, 1.0 )
```

---

### *Size of the Map*

**Name:** size

**Desc.:** Get the number of voxels in the volume

**Param.:** None

**Returns:** Number

**Usage:**

```
size = vol:size()
```

---

### *Get a voxel value*

**Name:** getValue

**Desc.:** The function retrieves a voxel value from the 3D volume. No interpolation takes place, x,y and z are indices and not coordinates.

**Param.:** x,y,z index

**Usage:**

```
voxel = vol:getValue(10,8,12)
```

---

*Get a voxel value*

**Name:** getIntValue

**Desc.:** This function takes a normal angstroem world coordinate and interpolates the voxel values tri-linearly.

**Param.:** x,y,z angstroem coordinate

**Usage:**

```
voxel = vol:getIntValue(11.5, 8.24, 12.21)
```

---

*Set a voxel value*

**Name:** setValue

**Desc.:** The function allows to directly manipulate a single voxel value. No interpolation takes place, x,y and z are indices and not coordinates.

**Param.:** x,y,z index and value

**Usage:**

```
vol:setValue(10,8,12, 0.567)
```

---

*Copy*

**Name:** copy

**Desc.:** Copies the current volume data set. It will not automatically get added to the list of loaded documents, but stays initially just on the Lua side (and will get deleted again, once the Lua interpreter finishes). AddDoc can be used to push the new object into the Sculptor document list and make it permanent.

**Param.:** None

**Usage:**

```
volCopy = vol:copy()
```

---

*Crop the volume***Name:** crop**Desc.:** Cuts the volume to a smaller size, by removing voxels from each dimension.**Param.:** min/max x, min/max y, min/max z**Usage:**

```
vol:crop(2,12,5,24,3,34)
```

---

*Update rendering***Name:** updateRendering**Desc.:** Update the rendering of the volume data. Should be called if the data was manipulated internally, for example with setValue, so that the rendering will reflect the new / changed data.**Param.:** None**Usage:**

```
vol:updateRendering()
```

---

*Save the volume***Name:** save**Desc.:** This function writes the volume data to a file on the disk.**Param.:** Filename**Usage:**

```
Usage: vol:save("test.situs")
```

---

### *Load a volume from disk*

**Name:** load

**Desc.:** Loads a volume from disk. The volume will not get added to the document list of Sculptor (can be done later using the addDoc function). Typically one would prefer to use the sculptor:load() function.

**Param.:** Filename

**Usage:**

```
vol = volumw:new()  
vol:load("test.situs")
```

---

### *Compute Feature-Vectors*

**Name:** vectorquant

**Desc.:** Create codebookfeature vectors using the neural gas TRN algorithm. Returns a new "molecule" with the feature vectors. Can start from an already existing configuration and will in that case also return a matched feature point set (can be used to flex the molecule). The start configuration is in that case the second parameter of the function call.

**Param.:** Number of feature vectors

**Usage:**

```
vectors = vol:vectorquant(10)
```

---

### *Cross-Correlation*

**Name:** correlation

**Desc.:** Calculate the correlation with another volume object.

**Param.:** Other volume object. The second parameter specifies if the correlation coefficient should only be computed under the current molecule (aka local correlation), or over the entire volume. Boolean, if true only under the molecule, false standard correlation. Default: false = standard cc.

**Usage:**

```
cc = vol_a:correlation( vol_b )
```

---

### *Delete a spherical subregion*

**Name:** cutSphere

**Desc.:** Remove / set to 0 a spherical subregion within the volume. Useful for example for virus maps, where one might only be interested in the capsid for docking.

**Param.:** center voxel coordinate for the spherical region x,y,z and radius of sphere

**Usage:**

```
vol:cutSphere( 10,12,8, 3.5 )
```

---

### *Threshold*

**Name:** threshold

**Desc.:** Threshold the volumetric map. All voxel below and above certain values are cut off and set to those values.

**Param.:** New minimum and maximum values

**Usage:**

```
vol:threshold( 0, 3.5 )
```

---

### *Get Maximal Density*

**Name:** getMaxDensity

**Desc.:** Get the maximal voxel value in the map.

**Param.:** None

**Usage:**

```
max = vol:getMaxDensity( )
```

---

### *Get Minimal Density*

**Name:** getMinDensity

**Desc.:** Get the minimal voxel value in the map.

**Param.:** None

**Usage:**

```
max = vol:getMinDensity( )
```

---

### *Get the voxelwidth of the volume*

**Name:** getVoxelwidth()

**Desc.:** Returns the size / width of the voxels in the map. Sculptor assumes that the maps are cubic and orthogonal.

**Param.:** None

**Usage:**

```
vw = vol:getVoxelwidth( )
```

---

### *Set the voxelwidth of the volume*

**Name:** setVoxelwidth()

**Desc.:** This will not re-interpolate the map, but only set the internal voxelwidth variable! See interpolate function.

**Param.:** Nmber, new voxelwidth

**Usage:**

```
vol:setVoxelwidth( 3.0 )
```

---

### *Get the size of the volume in x dimension*

**Name:** getSizeX()

**Desc.:** Returns the size / number of voxels in x direction.

**Param.:** None

**Usage:**

```
sizeX = vol:getSizeX( )
```

---

*Get the size of the volume in y dimension*

**Name:** getSizeY()

**Desc.:** Returns the size / number of voxels in y direction.

**Param.:** None

**Usage:**

```
sizey = vol:getSizeY( )
```

---

*Get the size of the volume in z dimension*

**Name:** getSizeZ

**Desc.:** Returns the size / number of voxels in z direction

**Param.:** None

**Usage:**

```
sizez = vol:getSizeZ( )
```

---

*Normalize the map*

**Name:** normalize

**Desc.:** Normalize the voxel values in the volumetric map to [0..1].

**Param:** None

**Usage:**

```
vol:normalize( )
```

---

*Mask with another volume object*

**Name:** mask

**Desc.:** Applies a mask to the volume. All the voxels in this vol are multiplied with the mask volume voxels (multiplied by 0 for not overlapping voxels).

**Param:** Mask volume object

**Usage:**

```
vol:mask( maskvol )
```

---



### *Create a binary mask*

**Name:** makeMask

**Desc.:** Create a binary mask using a threshold value. Every voxel below the threshold will get set to 0, the rest to 1.

**Param.:** Number, threshold value

**Usage:**

```
vol:makeMask( 1.0 )
```

---

### *Interpolate map to different voxelsize*

**Name:** interpolate

**Desc.:** Interpolate the map to a different voxelsize.

**Param:** Number, new voxelsize

**Usage:**

```
vol:interpolate( 2.0 )
```

---

### *Convolve map*

**Name:** convolve

**Desc.:** Convolve map with another volume (kernel).

**Param.:** Volume, kernel

**Usage:**

```
vol:convolve( kernelvol )
```

---

### *Create Gaussian Kernel*

**Name:** createGaussian

**Desc.:** Create a Gaussian kernel volume within SigmaFactorfSigma. Attention: This will overwrite the current content of the volume object with the filter kernel. It will allocate the memory internally.

**Param.:** sigma of map and sigma factor

**Usage:**

```
kernel:createGaussian( sigma, sigmafactor )
```

---

### *Create Laplacian Kernel*

**Name:** createLaplacian

**Desc.:** Create a laplacian kernel volume (3x3x3).

**Param.:** None

**Usage:**

```
kernel:createLaplacian()
```

---

### *Set / Get Position*

**Name:** x, y, z

**Desc.:** Set/get the x/y/z coordinate of the position of the map  
If no parameter is given, the function just returns a number, otherwise it will set the x/y/z coordinate using the parameter provided by the user.

**Param.:** Set functions: Coordinate in Angstroem

**Usage:**

```
vol:x( 15.0 )  
print( vol:x() )  
vol:y( 5.0 )  
print( vol:y() )  
vol:z( 1.0 )  
print( vol:z() )
```

## *Examples*

### *Hello World*

```
print( "This is a very simple Sculptor script.")
print( "Sculptor version:", sculptor:version() )
```

---

### *Animation*

```
mat = sculptor:getTrans()
for j=1, 10 do mat:rotate( 0, 1 )
  sculptor:setTrans( mat )
  sculptor:redraw()
  sculptor:sleep(20)
end
```

---

### *Number of atoms*

```
mol_A = sculptor:getDoc(2)
mol_B = molecule:new(mol)
print("Mol_A: ", mol_A:size())
print("Mol_B: ", mol_B:size())
```

---

### *Save a screenshot*

```
sculptor:saveScreenshot("/tmp/test.png")
os.execute("display /tmp/test.png")
```

This can be easily combined with the animation script above to create a movie. In that case one would use a movie-encoder like ffmpeg in the os.execute command to create the final movie file.

---

### *Creation of a synthetic cryo-EM map and cut out a spherical region*

```
sculptor:clearLog()
sculptor:delAllDocs()
sculptor:load("monomer.pdb")
mol_a = sculptor:getDoc( 2 )
vol_a = mol_a:blur( 3.0, 10.0 )
vol_a:cutSphere( 10,10,10,10.0 )
vol_a:addDoc("cutSphere.situs")
```

---

*Create a 10x10x10 volume with a cube*

```
vol = volume:new(10,10,10)
for x=1,10 do
  for y=1,10 do
    for z=1,10 do
      if x>2 and x<8 and y>2 and y<8 and z>2 and z<8 then
        vol:setValue(x,y,z,10)
      end
    end
  end
end
end
vol:addDoc("demo.situs") vol:setIsoThreshold( 0.5 )
```

---

*Extract only the helices from an atomic model*

```
sculptor:clearLog()
sculptor:delAllDocs()
sculptor:load("monomer.pdb")
mol_a = sculptor:getDoc( 2 )
mol_b = molecule:new()
for i=1,mol_a:size() do
  if (mol_a:getAtomSecStruct(i) == "H") then
    mol_b:addAtom( mol_a:getAtom( i ) )
  end
end
end
mol_b:addDoc("helices.pdb")
```

---

*Do a powell refinement only on a single chain*

<to be written>

---

*Matching*

This example code will load two molecules, vectorquantize them and match them.

```
sculptor:clearLog()
sculptor:delAllDocs()
```

```

--
-- Load molecule A
--
sculptor:load("mol_A.pdb")
mol_a = sculptor:getDoc( 2 )
mol_a:setDisplayMode("CARTOON")
mol_a:setColmapMode( "SOLID", 1 )
trn_a = mol_a:trn( 6 )
trn_a:addDoc("TRN_A.pdb")
trn_a:setDisplayMode("VDW")
trn_a:setColmapMode( "SOLID", 1 )
--
-- Load molecule B
--
sculptor:load("mol_B.pdb")
mol_b = sculptor:getDoc( 4 )
mol_b:setDisplayMode("CARTOON")
mol_b:setColmapMode( "SOLID", 2 )
trn_b = mol_b:trn( 6 )
trn_b:addDoc("TRN_B.pdb")
trn_b:setDisplayMode("VDW")
trn_b:setColmapMode( "SOLID", 2 )
sculptor:redraw()
--
-- Calculate the rmsd before the matching
--
print( "RMSD Before:", mol_a:rmsd( mol_b, false ) )
--
-- And now lets see if we can match it
--
mats = trn_b:match( trn_a )
ewmat = mol_b:getTrans():mult( mats[0] )
mol_b:setTrans( newmat )
sculptor:redraw()
--
-- Calculate the rmsd after the matching
--
print( "RMSD After:", mol_a:rmsd( mol_b, false ) )

```

---

*Flex an atomic model into a volumetric map*

```

sculptor:clearLog()
sculptor:delAllDocs()

```

```

--
-- Load molecule
--
sculptor:load("molecule.pdb")
mol = sculptor:getDoc( 2 )
mol:setDisplayMode("CARTOON")
mol:setColmapMode( "SOLID", 1 )
--
-- Vectorquantize
--
trnMol = mol:trn( 6 )
--
-- Create a synthetic map for tests
--
vol = mol:blur( 3.0, 10.0 )
--
-- Flex the molecule
--
for i=1,100 do
  -- vectorquantize with molecule features as starting point
  trnVol = vol:trn( 6, trnMol )

  -- OK, for fun introduce some random deviations
  for j=1,6 do
    atm = trnVol:getAtom( j )
    atm:x( atm:x() + ((math.random()-0.5) * 10) )
    atm:y( atm:y() + ((math.random()-0.5) * 10) )
    atm:z( atm:z() + ((math.random()-0.5) * 10) )
    trnVol:setAtom( atm )
  end

  -- now flex
  flex = mol:flexing( trnMol, trnVol )
  flex:addDoc("flexingTest.pdb")
  flex:setDisplayMode("CARTOON")
  flex:setColmapMode( "SOLID", 2 )
  sculptor:redraw()
  sculptor:sleep(500)
  sculptor:delDoc( 3 )
end

```