PyVOL Documentation

Release 1.6.8

Ryan H.B. Smith

Mar 23, 2020

CONTENTS

CHAPTER

ONE

INTRODUCTION

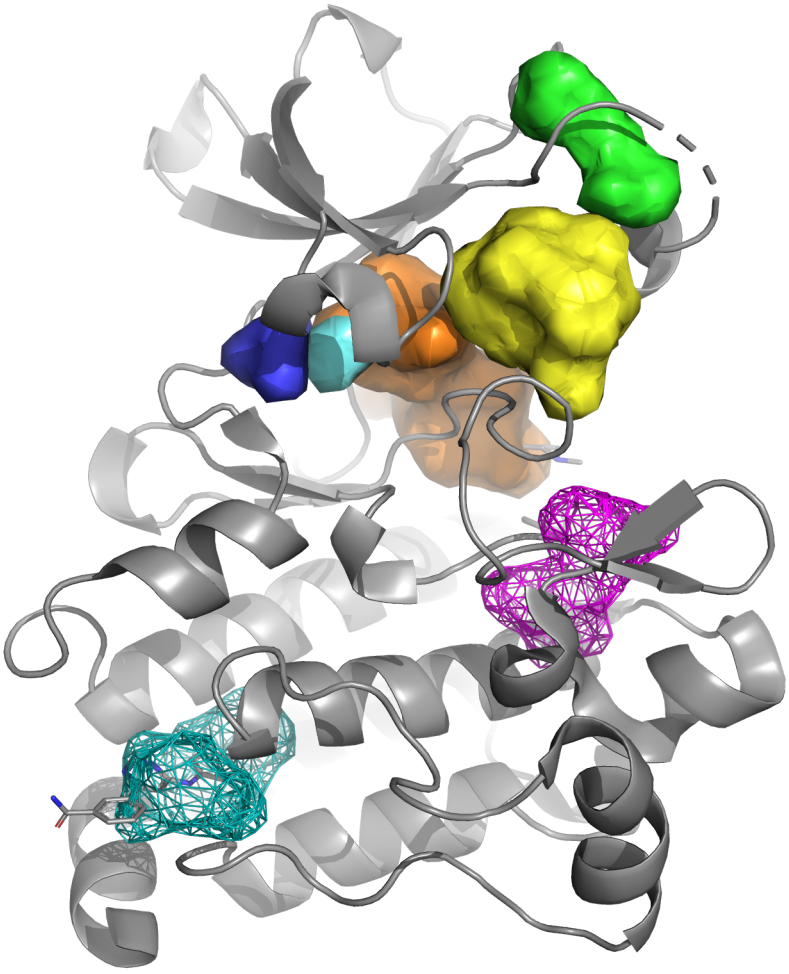

Fig. 1: PyVOL Pocket Identification

1.1 Overview

PyVOL is a python library packaged into a PyMOL GUI for identifying protein binding pockets, partitioning them into sub-pockets, and calculating their volumes. PyVOL can be run as a PyMOL plugin through its GUI or the PyMOL prompt, as an imported python library, or as a command-line program. Visualization of results is exclusively supported through PyMOL though exported surfaces are compatible with standard 3D geometry visualization programs. The project is hosted on github by the Schlessinger Lab. Please access the repository to view code or submit bugs. The package has been most extensively tested with PyMOL 2.3+ running Python 3.7. Support for all python versions 2.7+ is intended but not as thoroughly tested. Support for PyMOL 1.7.4+ without the GUI is as yet incomplete. Unfortunately, PyVOL can not currently run on MacOS Catalina due to its restrictions on running 32-bit executables. The Mac-compatible MSMS executable is not yet available in a 64-bit form.

1.2 Quick Installation into PyMOL 2.0+

PyVOL can be installed into any python environment, but installing directly into PyMOL 2.0+ is easiest. Download the `basic GUI installer` and then use the PyMOL plugin manager to install that file: *Plugins* → *Plugin Manager* → *Install New Plugin* → *Install from local file* → *Choose file...*

This installs the PyVOL GUI. Select *Plugins* → *PyVOL* → *Settings* → *Install PyVOL from PyPI* to fetch PyVOL and any missing dependencies. Once PyVOL has been installed, the location of MSMS must be added to the path. In the *MSMS Settings* panel, common locations for the executable can be searched. Once an executable has been identified and is displayed, *Change MSMS Path* can be clicked to make that executable visible to the back-end. The GUI should then display that it can find MSMS. For academic users and non-academic users with the Schrodinger incentive PyMOL distribution, installation is now complete. For all others *MSMS Installation*.

1.3 Example Basic Run

A simple calculation using the PyMOL prompt is to load a protein of interest and then run the *pocket* command. This is an example for the Sorafenib-bound structure of BRAF:

```
fetch '1uwh'
pocket protein="1uwh and chain B"
```

CHAPTER

TWO

INSTALLATION

PyVOL consists of a back-end and a GUI. The back-end has been packaged into installers that contain all dependencies, but normal distribution is through PyPI and accessed through *pip*. PyVOL can consequently be installed into any python environment. For convenience, the PyVOL GUI contains an installer for easy installation into PyMOL 2.0+.

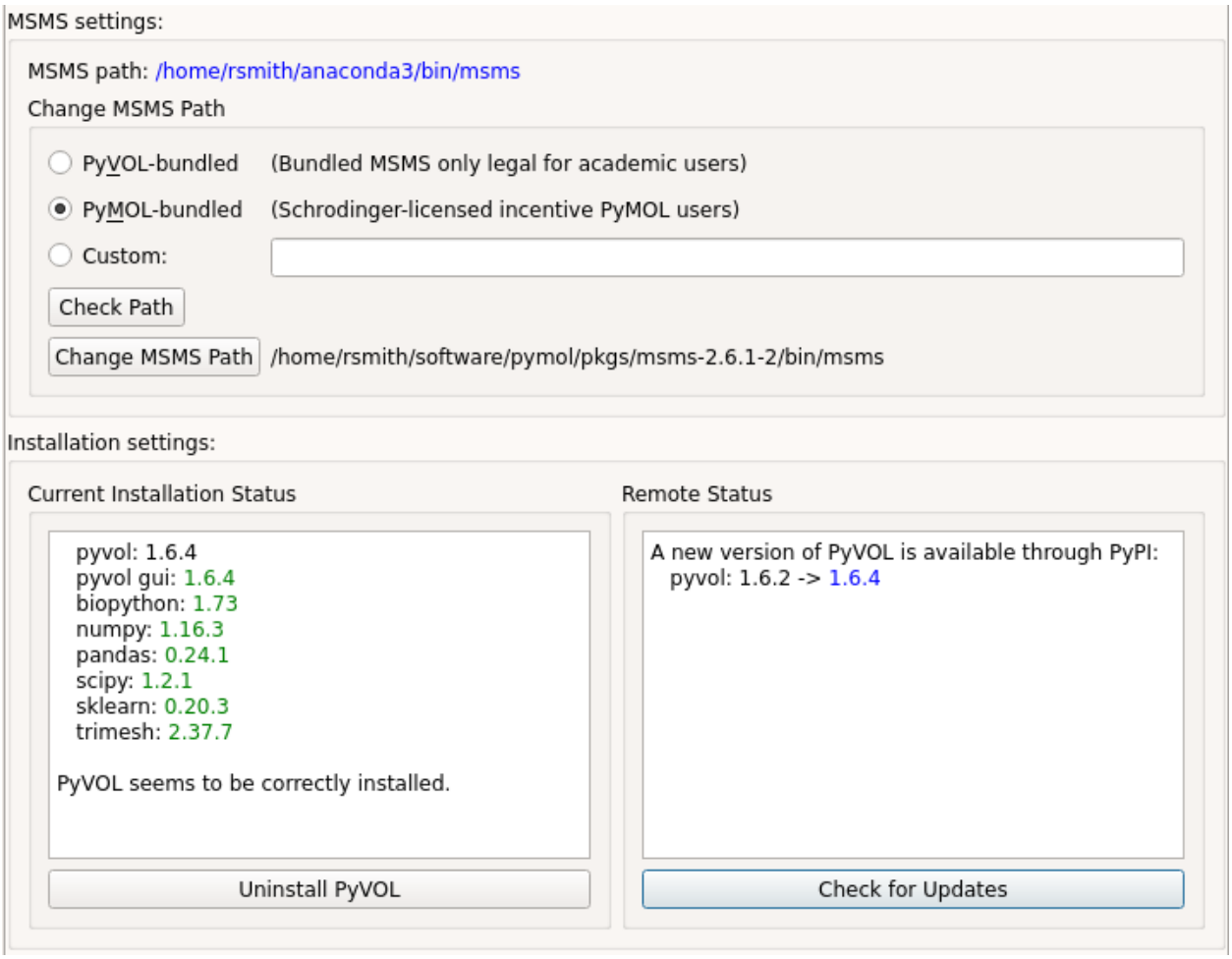

Fig. 1: GUI section that installs, updates, and uninstalls the PyVOL backend as well as confirming availability of the MSMS binary.

2.1 GUI Installation into PyMOL from PyPI

Download the `basic GUI installer` and then use the PyMOL plugin manager to install that file: *Plugins* → *Plugin Manager* → *Install New Plugin* → *Install from local file* → *Choose file...*

This installs the PyVOL GUI. Select *Plugins* → *PyVOL* → *Settings* → *Install PyVOL from PyPI* to fetch PyVOL and any missing dependencies. Once PyVOL has been installed, the location of MSMS must be added to the path. In the *MSMS Settings* panel, common locations for the executable can be searched. Once an executable has been identified and is displayed, *Change MSMS Path* can be clicked to make that executable visible to the back-end. The GUI should then display that it can find MSMS. For academic users and non-academic users with the Schrodinger incentive PyMOL distribution, installation is now complete. For all others refer to *MSMS Installation*.

2.2 GUI Installation into PyMOL from a Packaged Installer

A larger installer with cached copies of PyVOL and its dependencies is also available. This option is useful if deploying PyVOL onto computers without internet access or if accessing a stable snapshot of a working build is necessary for some reason. Download the `full GUI installer` and then use the PyMOL plugin manager to install that file: *Plugins* → *Plugin Manager* → *Install New Plugin* → *Install from local file* → *Choose file...*

This installs the PyVOL GUI. Select *Plugins* → *PyVOL* → *Settings* → *Install PyVOL from Cache* to install PyVOL and any missing dependencies from the installer. Once PyVOL has been installed, the location of MSMS must be added to the path. In the *MSMS Settings* panel, common locations for the executable can be searched. Once an executable has been identified and is displayed, *Change MSMS Path* can be clicked to make that executable visible to the back-end. The GUI should then display that it can find MSMS. For academic users and non-academic users with the Schrodinger incentive PyMOL distribution, installation is now complete. For all others refer to *MSMS Installation*.

2.3 PyMOL Prompt Installation into PyMOL

Installation of the PyMOL back-end using the PyMOL prompt is also supported. This should work even in earlier versions of PyMOL (1.7.4+) where the GUI is non-functional. Simply run the following command on the prompt:

```
install_pyvol
```

Installation from the packaged installer is also available using the PyMOL prompt:

```
install_pyvol_local
```

2.4 Manual Installation

PyVOL minimally requires *biopython*, *MSMS*, *numpy*, *pandas*, *scipy*, *scikit-learn*, *trimesh*, and *msms* in order to run. PyVOL is available for manual installation from github or through PyPI. Most conveniently:

```
pip install bio-pyvol
```

Again, for academic users and non-academic users with the Schrodinger incentive PyMOL distribution, installation is now complete. For all others, refer to manual *MSMS Installation*.

Note: When using command-line installation commands, make sure to use the right python environment. By default, pip will use the system python, but PyMOL often includes its own python environment. To check which python

environment to use, run *import sys; print(sys.executable)* on the PyMOL prompt. If that is anything besides the system default python, use *<PyMOL python executable> -m pip install bio-pyvol* to install PyVOL into the PyMOL-accessible environment.

2.5 MSMS Installation

MSMS is provided with PyVOL for ease of use for academic users. If MSMS is available on the system path, it is automatically detected. Common locations (including the bundled version for academic users) can be searched using the GUI *Settings* tab. Select the appropriate location to search and then click *Check Path*. If a viable MSMS executable is found at that location, it is displayed. In this case the *Change MSMS Path* button allows the default location for MSMS to be set. This stores the MSMS path under the PyMOL variable *pyvol_msms_exe* which can be manually accessed and edited via PyMOL's settings manager.

MSMS can also be manually installed and then added to the path or provided as the *custom* location (i.e. *pyvol_msms_exe* variable). MSMS can be downloaded from MGLTools on all systems or installed on MacOS and Linux using the bioconda channel:

```
conda install -c bioconda msms
```

2.6 Updating

PyVOL can be updated through the PyMOL GUI simply by navigating *PyVOL* → *Settings* → *Check for Updates*. This queries the PyPI server to detect if an update is available. If an update is available for download, the same button becomes *Update PyVOL* and will update the back-end. The new version of the PyVOL back-end will notify you if it expects an updated GUI. If the GUI also needs to be updated, uninstall the *pyvol_gui* using *Plugins* → *Plugin Manager* → *Installed Plugins* → *pyvol_gui x.x.x* → *Uninstall*. Restart PyMOL, download the updated GUI from `github`, and install the updated GUI as described above.

Alternatively, PyVOL can be manually updated via the command line:

```
pip update bio-pyvol
```

or the PyMOL prompt:

```
update_pyvol
```

2.7 Uninstalling

PyVOL can be uninstalled through its GUI by navigating *PyVOL* → *Settings* → *Uninstall PyVOL*. This uninstalls the back-end. Then use the plugin manager to uninstall the *pyvol_plugin*.

Again, PyVOL can also be uninstalled via the command line:

```
pip uninstall bio-pyvol
```

CHAPTER

THREE

BASIC USAGE

PyVOL accepts inputs from the PyMOL prompt, the PyMOL GUI, and configuration files via the system shell. The PyMOL prompt and configuration file inputs are fully featured while the GUI contains a slightly simplified interface with a few enforced defaults. Programmatic invocation is also supported and covered through module documentation.

The next few sections describe the parameters controlling Basic Usage, *Pocket Specification*, *Partitioning Options*, and *Output and Display Options*. With the exception of PyMOL-specific arguments, parameter names and argument handling are identical across all inputs.

Fig. 1: The Basic Parameters section of the PyVOL GUI with parameter mapping: Protein PyMOL Selection -> *protein*, Minimum Radius -> *min_rad*, and Maximum Radius -> *max_rad*

3.1 Protein Selection

The *Protein* selection for processing can be provided using one of two arguments. The *prot_file* argument directs PyVOL to read a structure from file. All included atoms are considered to be part of the pocket boundary (all atoms that occlude space). Alternatively, input through either PyMOL interface allows specification of the *protein* argument. This is a PyMOL selection string that defines the pocket boundary.

When providing input through PyMOL, the Boolean *protein_only* argument (checkbox in the GUI) can be set to restrict the *protein* selection to peptide atoms. This is enabled in the GUI by default and excludes all waters, solvent molecules, and other small-molecules from the pocket boundary. This is normally useful, but if a co-factor should be considered part of the binding site and excluded from the available volume, it can make sense to group it with the peptide for the purpose of calculating the solvent excluded surface.

```
# arguments: protein, prot_file
pocket prot_file=<protein_pdb_filename>
pocket protein=<"PyMOL selection string">, protein_only=True
```

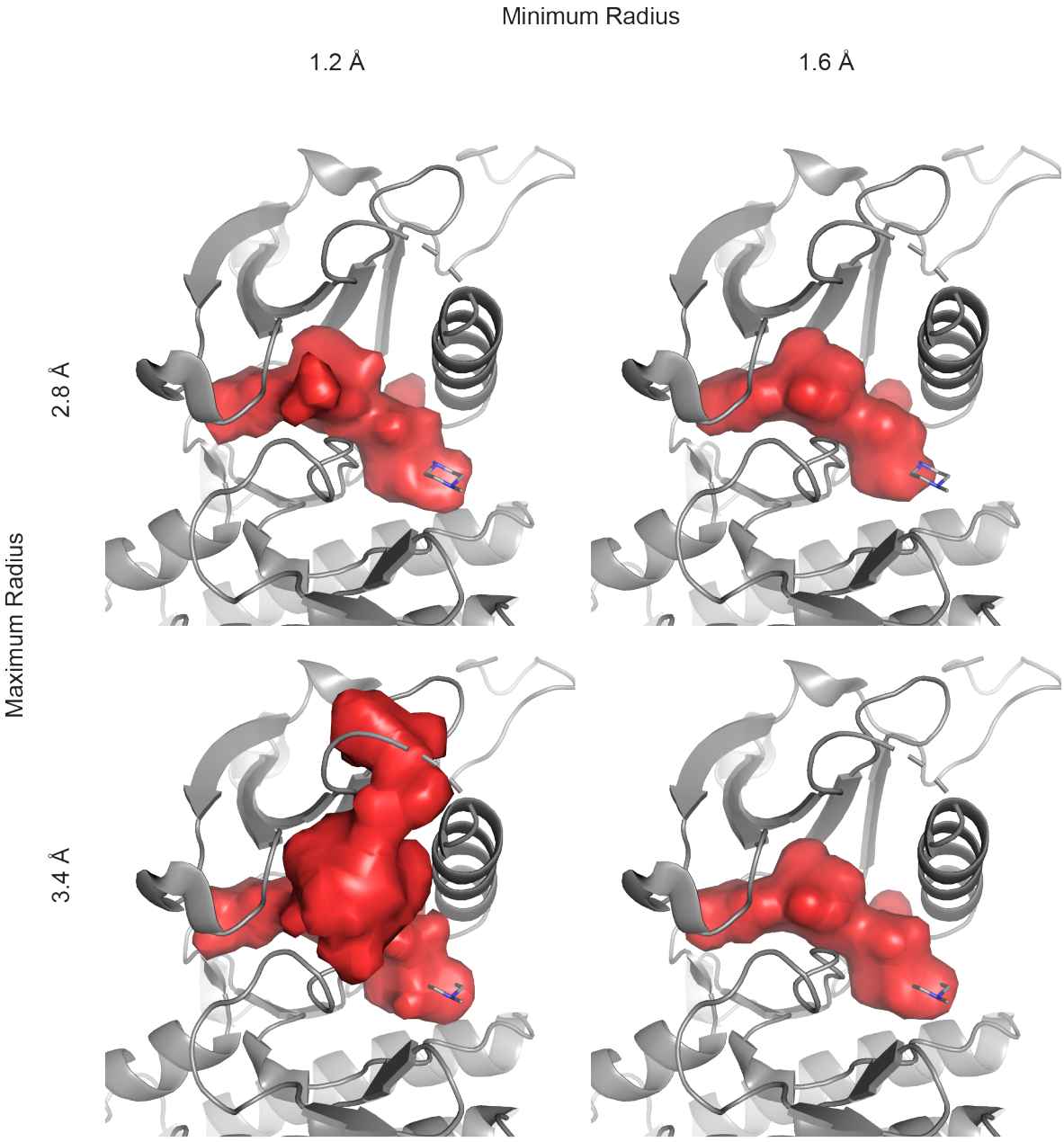

Fig. 2: Demonstration of the effects of varying the minimum and maximum probe radii. The smaller minimum radius of 1.2 Å (left column) shows surface topology slightly better than the larger minimum radius of 1.6 Å (right column). However, the smaller radius can connect regions for which the connections are smaller than that tolerated by small molecules. The smaller maximum radius of 2.8 Å (top row) includes excludes an extra region from the bulk solvent relative to the larger maximum radius of 3.4 Å (bottom row). In this case, all parameter combinations with the exception of the 1.2 minimum radius and 3.4 maximum radius identify a pocket closely conforming to the volume occupied by the bound small molecule. The top left panel was produced with the command: *pocket protein='3k5v and chain A', protein_only=True, min_rad=1.2, max_rad=2.8.*

3.2 Minimum and Maximum Probe Radii

The most important parameters controlling PyVOL pocket identification and boundary location are the minimum and maximum radii (*min_rad* and *max_rad* arguments) used for surface identification. The maximum radius determines the size of the probe used to identify regions accessible to bulk solvent. This parameter should be chosen to exclude any binding pockets of interest while not overly distorting the surface of the protein. Generally, values around 3.4 Å are reasonable, but this parameter can be set lower to increase pocket detection stringency or set higher to reduce stringency. The drawback to setting *max_rad* too high is that this can lead to the identification of unreasonable, shallow pockets that snake around the surface of the protein. The minimum radius controls two factors: the level of detail of the calculated binding pocket surfaces and the algorithmic lower limit to minimum internal radii of identified binding pockets. Lower minimum radii calculate the accessibility to smaller solvent molecules. This necessarily increases the number of nooks or crannies in the binding pocket surface that are calculated and can link adjacent pockets that can not accommodate even small organic molecules. If the purpose of volume calculations is to identify protein features relevant to compound binding, such behavior is undesirable. From such a perspective, setting the minimum radius to approximately that of the smallest pharmacophore radius of potential ligands makes sense for calculations. However, the radius of water is the default setting because it meets the typical expectation of users looking at solvent excluded surfaces.

```
# arguments min_rad, max_rad
pocket prot_file=<protein_pdb_filename>, min_rad=<1.4>, max_rad=<3.4>
pocket protein=<"PyMOL selection string">, min_rad=<1.4>, max_rad=<3.4>
```

3.3 Input Constraint

By default, basic input radii parameters are compared and constrained to tested ranges using the *constrain_radii* argument. This can be turned off when running outside of the GUI, but in practice it is never useful to disable this feature. While edge cases are possible in which violating constraints is useful, in practice these constraints represent effective ranges. In particular, if the minimum radius is set to absurdly low values, the software will start fitting pockets even within intramolecular spaces and provide meaningless output if not fully crashing the program.

```
# arguments constrain_radii
pocket prot_file=<protein_pdb_filename>, min_rad=<1.4>, max_rad=<3.4>, constrain_
↪radii=True
pocket protein=<"PyMOL selection string">, min_rad=<1.4>, max_rad=<3.4>, constrain_
↪radii=True
```

Note: Be careful about saving *.pse* PyMOL sessions with PyVOL-produced surfaces. PyMOL does not currently use plugins to load unfamiliar CGO objects, so calculated surfaces will not load correctly from a saved PyMOL session. On the other hand, it should be possible to recreate results using saved PyMOL *.pml* logs. Surfaces can be loaded back into a session using the *Load Pocket* (command-line *load_pocket*) commands.

CHAPTER

FOUR

POCKET SPECIFICATION

PyVOL runs in one of three modes (*largest*, *specific* or *all*). By default it runs in *largest* mode and returns only the single volume and geometry corresponding to the largest pocket identified when calculating *all* pockets. However, manual identification of the pocket of interest is often preferable. This can be done through specification of a ligand, a residue, or a coordinate. If any specification is given, the mode must be changed to *specific* in order to process that parameter. The *specific* mode is the fastest by a small margin because it calculates the fewest surfaces.

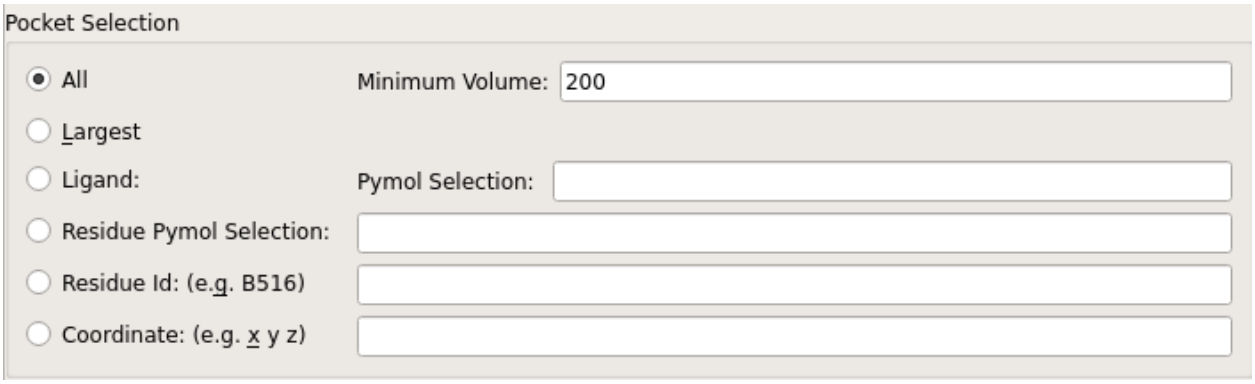

Fig. 1: GUI section controlling user specification of binding pockets with parameter mapping: primary radio buttons → *mode*, Minimum Volume → *min_volume*, Ligand: PyMOL Selection → *ligand*, Residue PyMOL Selection → *residue*, Residue Id → *resid*, and Coordinate → *coordinates*

4.1 All Mode

When running in *all* mode, PyVOL determines all surfaces with inward-facing normals with volumes over a minimum threshold defined by the *Minimum Volume* (*min_volume* argument). This functions similarly to the *largest* mode except that 1) all surfaces are returned rather than just the largest, 2) if the largest surface has a volume less than *Minimum Volume*, no surface will be returned at all, and 3) subpocket partitioning cannot occur on the output from this mode. By default the minimum volume is set to 200 Å:superscript:*3*. This is a heuristically determined threshold that is generally useful at distinguishing between artifacts and interesting pockets.

```
# arguments: mode, min_volume
pocket prot_file=<protein_pdb_filename>, mode=all, min_volume=<200>
pocket protein=<"PyMOL selection string">, mode=all, min_volume=<200>
```

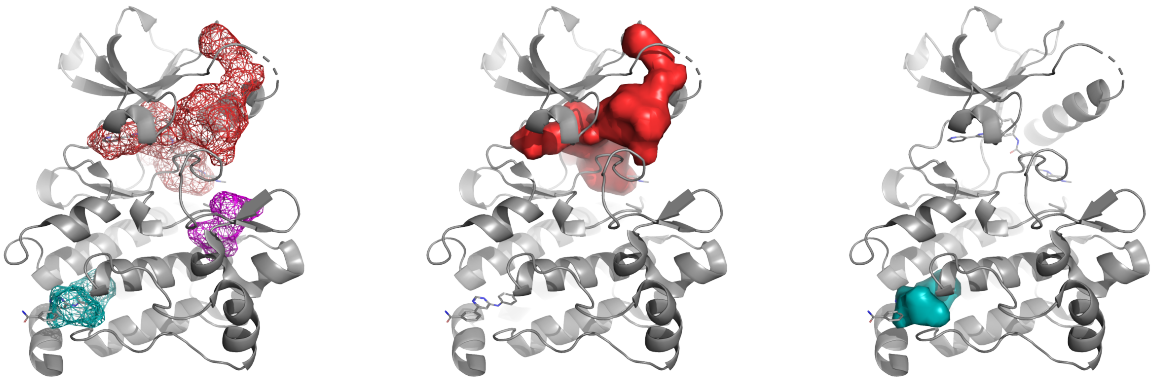

Fig. 2: Example of output using specification parameters. PyMOL commands to generate these panels consist of a specification argument in addition to this basic commmand: *pocket protein='3kvg and chain A', protein_only=True, palette='tv_red,teal,magenta'*. The first panel is produced by adding *mode=all*, the second with *mode=largest*, and the final with any of the following: 1) *mode=specific, resid=A487*, 2) *mode=specific, residue='3k5v and chain A and resi 487'*, 3) *mode=specific, ligand=<single atom from the bound small molecule>*, or 4) *mode=specific, coordinate='12.3,42.2,48.6'*.

4.2 Largest Mode

In the default *largest* mode, PyVOL determines all surfaces with inward-facing normals, calculates the volume of each, and selects the largest. The pocket selected with this mode is usually the pocket of pharmacological interest in a protein. However, sometimes changes in the minimum or maximum radius can lead to the unexpected selection of an alternative, normally superficial pocket.

```
# arguments: mode
pocket prot_file=<protein_pdb_filename>, mode=largest
pocket protein=<"PyMOL selection string">, mode=largest
```

4.3 Specific Mode

The final mode, the *specific* mode, is invoked through specification of a ligand, residue, or coordinate. PyVOL automatically switches to this mode if any specification is provided, but this behavior can be overriden. Manual specification of the *specific* mode is safer. There is an internal priority to which specification is used, but only a single option should be specified to avoid ambiguity.

4.3.1 Ligand Specification

A ligand occupying the binding pocket of interest can be specified similarly to protein definition. All inputs can accept a *lig_file* argument specifying a pdb file containing ligand geometry, and PyMOL inputs can accept a *ligand* argument containing a PyMOL selection string. If the *ligand* argument is provided, atoms belonging to the *ligand* are removed from the *protein* selection before pocket identification. In all cases, bulk solvent boundary determination includes the provided ligand, so ligands that extend somewhat beyond the convex hull of the protein can include some of that volume within the calculated binding pocket. In these cases the calculated volumes depend on the exact identity and

pose of the ligand provided. This option is improper for *apo* volumes with the trade-off that calculated volumes can be meaningfully compared to small molecule volumes.

```
# arguments: lig_file, ligand
pocket prot_file=<protein_pdb_filename>, mode=specific, ligand=<ligand_pdb_filename>
pocket protein=<"PyMOL selection string">, mode=specific, lig_file=<"ligand selection␣
↪string">

# Trivial case in which a single organic small molecule is present in the protein␣
↪selection
pocket protein=<"PyMOL selection string">, ligand=<"'PyMOL selection string' and org">
```

Supplying a ligand opens up two additional options. *Inclusion Radius* (*lig_incl_rad* argument) prevents the exterior surface of the protein (bulk solvent surface definition) from being defined within that distance from the ligand. In cases where a ligand extends somewhat into solvent and calculated volumes would otherwise be smaller than the volume of the known ligand, this can be used to produce a more useful surface. *Exclusion Radius* (*lig_excl_rad* argument) limits the maximum scope of the identified surface as the locus of points that distance from the supplied ligand. Both of these options introduce a heuristic that alters reported results. They are most useful when standardizing volumes across a series of similar structures as they provide a mechanism to limit volume variability due to variation in bulk solvent boundary determination.

```
# arguments: lig_incl_rad, lig_excl_rad
pocket prot_file=<protein_pdb_filename>, mode=specific, ligand=<ligand_pdb_filename>,␣
↪lig_incl_rad=<3.5>, lig_excl_rad=<5.2>
pocket protein=<"PyMOL selection string">, mode=specific, lig_file=<"ligand selection␣
↪string">, lig_incl_rad=<3.5>, lig_excl_rad=<5.2>
```

Note: SDF format ligand files are not currently supported for input using *lig_file* because that would increase the number of software dependencies. Reading the sdf file into PyMOL and then passing the ligand into PyVOL using the *ligand* argument is the current solution.

4.3.2 Residue Specification

A bordering residue can be supplied to localize a pocket. Once again, this can be done either by specifying a residue ID or with a PyMOL selection string when running through PyMOL. The *resid* argument accepts a string specifying a residue by chain and index (i.e., residue 25 of chain A would be "A35"). The chain is inferred if not explicitly included. Only sidechain atoms are considered. The PyMOL *residue* argument allows specification of a PyMOL selection bordering the pocket of interest. This selection can be of arbitrary size but has been primarily tested holding single residues. Only the sidechains of the provided selection are used for pocket specification. In both of these cases, PyVOL tries to identify the residue atom closest to an interior surface and uses that atom to specify the adjacent pocket of interest. Some residues are adjacent to multiple pockets and make specification computationally arbitrary and unpredictable. If having trouble, specify a single atom as a PyMOL selection string.

```
# arguments: resid, residue
pocket prot_file=<protein_pdb_filename>, mode=specific, resid=<chain/residue_index>
pocket protein=<"PyMOL selection string">, mode=specific, residue=<"residue selection␣
↪string">
```

4.3.3 Coordinate Specification

The final method for specifying a pocket of interest is through providing a coordinate that is within the pocket using the *coordinates* argument. PyVOL identifies the closest atom in the protein selection to the supplied coordinate and

uses it to define the surface of the calculated pocket. The coordinate value is accepted as a string of three floats with spaces in between values (e.g., “23.1 47.2 -12.7”).

```
# arguments: coordinates
pocket prot_file=<protein_pdb_filename>, mode=specific, coordinates="x,y,z"
pocket protein=<"PyMOL selection string">, mode=specific, coordinates="x,y,z"
```

CHAPTER

FIVE

PARTITIONING OPTIONS

PyVOL can deterministically divide a binding pocket into subpockets. This can be run on the output of any surface determination that results in a single returned surface. PyVOL currently calculates *de novo* complete binding pocket surfaces prior to partitioning because determination of the overall pocket is computationally trivial relative to subdivision. Processing time scales with the volume and complexity of the studied pocket. Most jobs take just seconds, but partitioning a pocket with total volume exceeding 1000-1500 Å:superscript:*3* can extend computation time past one minute.

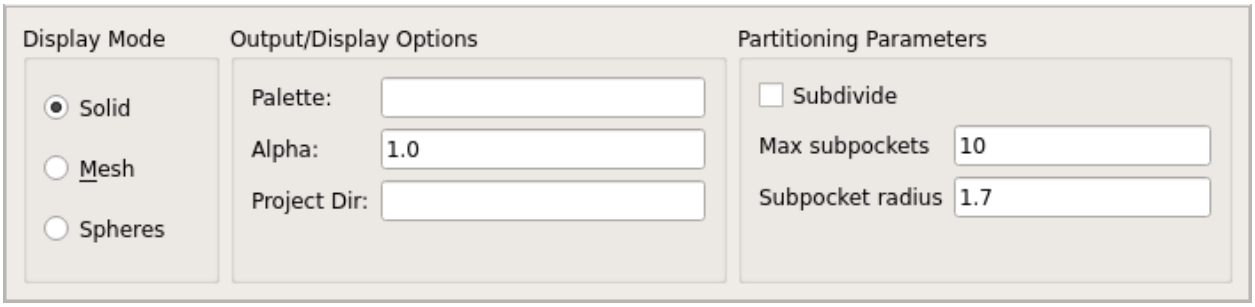

Fig. 1: GUI section controlling user binding pocket partition into subpockets with argument mapping: Subdivide → *subdivide*, Max subpockets → *max_clusters*, and Subpocket radius → *min_subpocket_rad*

5.1 Enabling Subpocket Partitioning

Subpocket partitioning is enabled by setting the *subdivide* argument to *True*. In the GUI, this is done by selecting the *Subdivide* checkbox.

```
# arguments: subdivide
pocket prot_file=<protein_pdb_filename>, subdivide=True
pocket protein=<"PyMOL selection string">, subdivide=True
```

5.2 Controlling the Number of Subpockets

Parameters controlling the number of sub-pockets identified generally perform well using defaults; however, they can be easily adjusted as needed. The two most important parameters are controlled with the *max_clusters* and *min_subpocket_rad* arguments. PyVOL clusters volume into the maximum number of regions that make physical sense according to its hierarchical clustering algorithm. This means that there is a maximum number of clusters that is determined by the *min_subpocket_rad* (the smallest sphere used to identify new regions). Larger values of the *min_subpocket_rad* can prohibit unique identification of smaller regions and can cause partitioning to fail altogether.

Setting the maximum number of clusters simply sets an upper bound to the number of subpockets identified. If the number of clusters originally determined is greater than the supplied maximum, clusters are iteratively merged using a metric that is related to an edge-biased surface area between adjacent clusters.

```
# arguments: min_subpocket_rad, max_clusters
pocket prot_file=<protein_pdb_filename>, subdivide=True, min_subpocket_rad=<1.7>, max_
↪clusters=<10>
pocket protein=<"PyMOL selection string">, subdivide=True, min_subpocket_rad=<1.7>,␣
↪max_clusters=<10>
```

5.3 Other Partitioning Parameters

The size of the probe used to calculate surface accesibility of subpockets can be set with the *min_subpocket_surf_rad*. Calculation stability is less sensitive to the value of this parameter than the overall minimum probe radius. In practice, it should be set to a value slightly smaller than the overall minimum radius but not less than 1.0 Å. Unless changing the minimum used for overall calculations, the default value should be left unchanged.

PyVOL currently defaults to performing radial sampling frequency at 10 Å:superscript:*-1* but this can be adjusted using the *radial_sampling* argument. Larger *radial_sampling* values should significantly improve calculation speed but at the cost of pocket resolution.

PyVOL isolates the pocket to be subdivided prior to running partitioning. The local environment of the pocket is isolated by identifying all atoms within a set distance of the surface calculated for the pocket of interest. This distance is set to the maximum radius used for bulk solvent surface identification plus a buffer. The magnitude of this buffer is by default 1 A and can be set using the *inclusion_radius_buffer* argument.

The maximum sampled internal radius of subpockets can be set with the *max_subpocket_rad* argument. Varying this parameter above ~2.7 Å is unlikely to alter results. The only practical scenario for setting this variable is when an unusually low maximum radius is used in determining bulk solvent surfaces. If internal pocket cross sections are larger than the external probe used, setting the *max_subpocket_rad* to a higher value can permit proper clustering. For the majority of users, this parameter should never be adjusted.

The minimum number of tangent surface spheres belonging to a subpocket can be set with the *min_cluster_size*. The purpose of this filter is to remove small, aphysical sphere groupings before clustering. In practice, this never needs to be adjusted.

```
# arguments: min_subpocket_surf_rad, radial_sampling, max_subpocket_rad, min_cluster_
↪size
pocket prot_file=<protein_pdb_filename>, subdivide=True, min_subpocket_rad=<1.7>, max_
↪clusters=<10>, min_subpocket_surf_rad=<10>, radial_sampling=<0.1>, max_subpocket_
↪rad=<3.4>, min_cluster_size=<50>
pocket protein=<"PyMOL selection string">, subdivide=True, min_subpocket_rad=<1.7>,␣
↪max_clusters=<10> min_subpocket_surf_rad=<10>, radial_sampling=<0.1>, max_subpocket_
↪rad=<3.4>, min_cluster_size=<50>
```

CHAPTER

SIX

OUTPUT AND DISPLAY OPTIONS

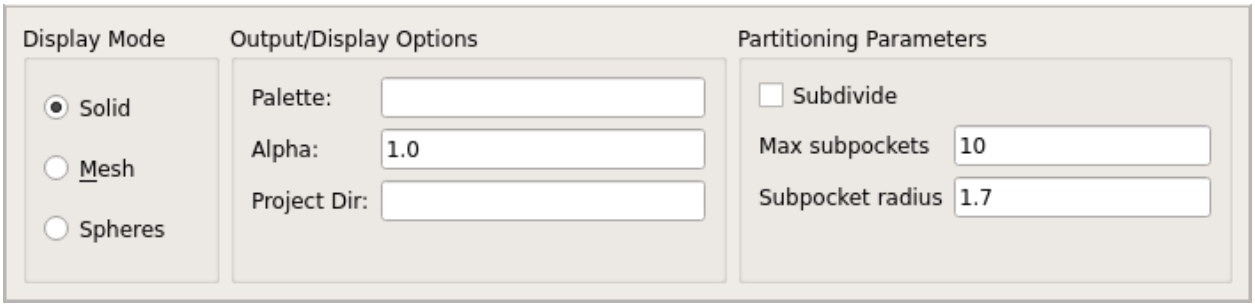

Fig. 1: GUI section controlling output and display options with argument mapping: Display Mode radio button → *display_mode*, Palette → *palette*, Alpha → *alpha*, and Project Dir → *project_dir*

6.1 File Output Options

PyVOL always creates an output directory. It looks for a project directory (*project_dir* argument) and then creates an output directory (*output_dir*) within it. If a project directory is not provided, the current working directory is used. The output directory and all included files are named by default using a prefix consisting of a timestamp followed by a system-compatible chunk of the protein identifier (filename or selection string). The prefix can be set using the *prefix* argument.

A minimum of six files are produced in a completed calculation: 1) input protein geometry (*prefix'_prot.pdb), 2) a configuration file recording all options to recapitulate a calculation ('prefix*.cfg), 3) a detailed log output (*prefix*.log), 4) a report of all calculated volumes (*prefix*.rept), and pocket geometry information contained in two files: 5) surface geometry (*prefix'_p'n*.obj) and 6) tangent sphere definitions (*prefix'_p'n*.xyzrg). If multiple pockets are calculated (e.g., *all* and *subdivide* runs), pocket geometry files are written for each.

```
# arguments: project_dir, output_dir, prefix
pocket prot_file=<protein_pdb_filename>, project_dir=<directory>, output_dir=<sub-
↪directory>, prefix=<identifier>
pocket protein=<"PyMOL selection string">, project_dir=<directory>, output_dir=<sub-
↪directory>, prefix=<identifier>
```

6.2 Logger Options

PyVOL output is done through loggers. Logger handler levels can be configured via the *logger_stream_level* and *logger_file_level* arguments. The *logger_stream_level* controls the level of information printed to the screen during

runs. The *logger_file_level* sets the amount of information written to the log file. Default levels of the stream and file handlers are respectively "INFO" and "DEBUG". This provides a more limited summary at run time with further details written to file.

```
# arguments: project_dir, output_dir, prefix
pocket prot_file=<protein_pdb_filename>, logger_stream_level="INFO", logger_file_
↪level="DEBUG"
pocket protein=<"PyMOL selection string">, logger_stream_level="INFO", logger_file_
↪level="DEBUG"
```

6.3 Display Options

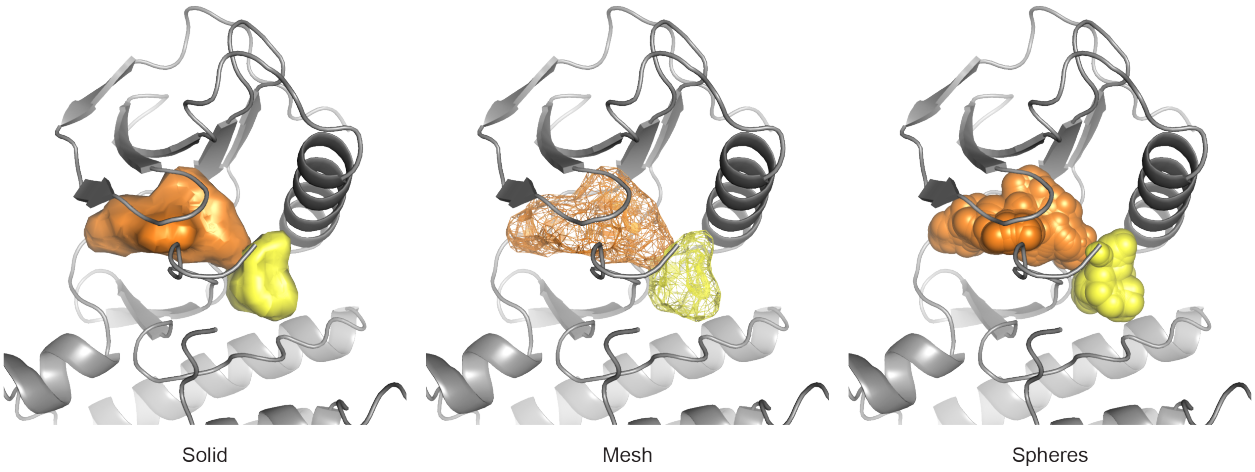

Fig. 2: Examples of the three primary display modes.

When running through PyMOL or loading previous results into a PyMOL session, calculated surfaces can be visualized in any of three different ways by setting the *display_mode* parameter. Surfaces can be represented using a solid mesh (*solid*), a wireframe mesh (*mesh*), or a group of spheres (*spheres*). Transparency (when applicable) is set with the *alpha* argument: a float [0,0, 1.0] that is equal to (1 - transparency). The displayed color of surfaces is controlled using the *palette* parameter. For non-programmatic invocation, the palette is interpreted as a comma-separated list of PyMOL color strings or space-separated RGB floats. A default palette is automatically selected, and additional colors are automatically interpolated when more surfaces need to be displayed. If 5 colors need to be chosen from an input palette of 4 colors, the first and fifth palette colors are the first and fourth of the input colors. The second color contains 2/3 of the first and 1/3 of the second input colors through linear interpolation. The second through fourth colors are similarly interpolated.

```
# arguments: display_mode, alpha, palette
pocket protein=<"PyMOL selection string">, display_mode=mesh
pocket protein=<"PyMOL selection string">, display_mode=solid, alpha=0.85
pocket protein=<"PyMOL selection string">, display_mode=spheres
pocket protein=<"PyMOL selection string">, mode=all, display_mode=solid, palette="tv_
↪red,tv_orange,0.34 0.26 0.74"
pocket protein=<"PyMOL selection string">, mode=largest, subdivide=True, display_
↪mode=mesh, palette="marine,forest_green,magenta,cyan"
```

Note: Specifying non-standard colors for display purposes can be syntactically difficult. The easiest way to match RGB values, for instance to other figure elements, is to edit the produced configuration file and reload the results into

PyMOL using *Loading Previous Results*.

CHAPTER

SEVEN

SHELL INTERFACE

PyVOL can also be run from the system command line using bash or any standard shell. If installed using `pip`, a `pyvol` entry point should be automatically installed and made available on the path. Otherwise, manual invocation of `pyvol/__main__.py` should work.

7.1 Running from the Shell

From the command-line, PyVOL is run exclusively using a configuration file.

```
python -m pyvol <input_parameters.cfg>
```

7.2 Template Configuration File Generation

A template configuration file with default values supplied can be generated using:

```
python -m pyvol -t <output_template.cfg>
```

7.3 Rerunning Previous Calculations

Each PyVOL job writes the configuration file to recapitulate the exact run. After modifying a configuration file, unset the *prefix* and *output_dir* parameters in order to direct the output of the new run into a new folder.

Note: When unsetting parameters in the configuration file, delete the entire line including the parameter name rather than just leaving the definition blank. For some parameters, leaving the definition blank confuses the configuration file reader.

CHAPTER

EIGHT

LOADING PREVIOUS RESULTS

PyMOL cannot load custom CGO objects back into sessions correctly, so any PyMOL session containing PyVOL surfaces will have issues. PyMOL log files can be used but can take a while to run on slower computers.

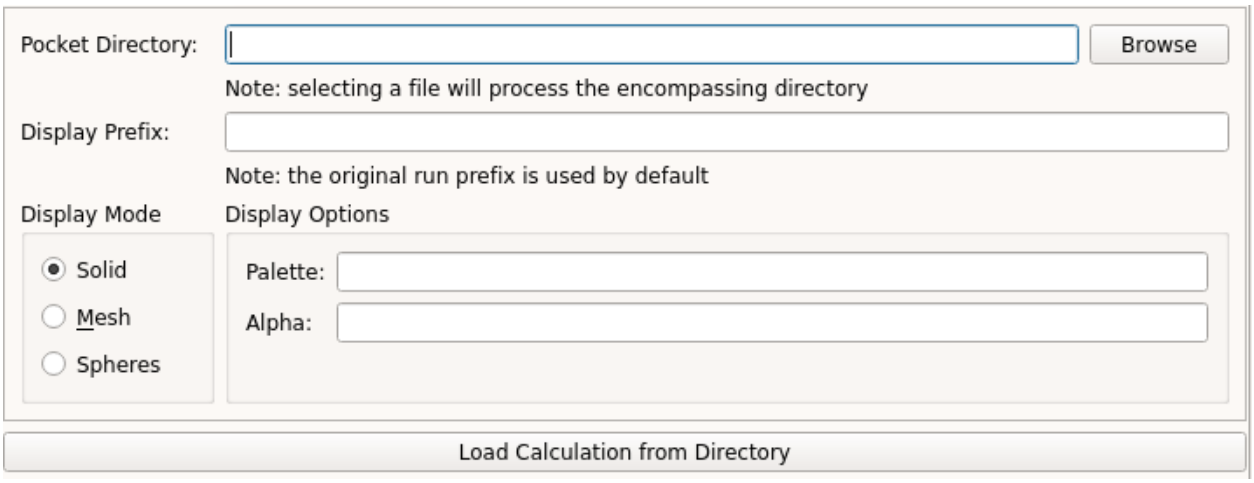

Fig. 1: GUI section that loads previous calculations into PyMOL for visualization with parameter mapping: Pocket Directory → *data_dir* and Display Prefix → *prefix*

The PyMOL *load_pocket* command allows previous results to be read back in from file and displayed. A display prefix and display options (previously described *Display Options*) can be provided to overwrite configuration file values. *load_pocket* requires the directory holding the data from a previous calculation as its first parameter. This corresponds to the *output_dir* for new calculations and by default ends in '.pyvol'. If a data file is instead provided, PyVOL instead processes the encompassing directory.

```
load_pocket <directory>
```

CHAPTER

NINE

DEVELOPMENT

9.1 Package Design

The main PyVOL algorithm is run from *identify.pocket*. Input option sanitization and logger configuration have been split into *identify.pocket_wrapper*. The pocket identification logic occurs within *identify.pocket* with almost all direct data manipulation handled by the class methods of *Spheres*. If enabled, subpocket clustering occurs in *identify.subpockets* with data manipulation occurring in *cluster*. Frequently used functions have been split into *utilities*. Configuration file reading and writing as well as input parameter checking is done in *configuration*.

The PyMOL interface is contained in *pymol_interface* though integration into the PyMOL environment is actually handled in *pyvol_gui.__init__*. Display and other PyMOL-specific functions are defined in *pymol_utilities*.

The two primary interfaces are via configuration file (invoked through the command line using the entry point in *__main__* that is created on installation) and via PyMOL. PyMOL is extended with all commands, and the GUI provides a limited interface to these functions. Programmatic invocation is also supported. If standard output options are reasonable, using the *identify.pocket_wrapper* entry point is better. For more customization, directly call *identify.pocket* after calling *configuration.clean_opts* on a dictionary containing all required options.

9.2 Algorithm Design

The primary algorithmic logic is supplied in *identify.py* which acts as the only interface between the user-facing modules and the computational back-end.

The Spheres class holds all of the geometric information about proteins and pockets. It represents any object as a collection of spheres by holding their coordinates, radii, and cluster identifications in a 5D numpy array. Surface triangulation using MSMS and many other convenience functions are included in the class itself. The methods contained in the separate *cluster.py* would largely work as methods in the Spheres class but have been separated due to that class becoming too large and the specificity of those methods to subpocket partitioning.

9.3 GUI Design

The GUI was developed using Qt Designer and run using PyQT5. PyQT does not run in PyMOL 1.x distributions, so the GUI is only available in PyMOL 2.0+.

9.4 Version Incrementation

PyVOL uses a standard incrementation scheme. The version of the back-end must be updated in *setup.py*, *pyvol/__init__.py*, and *docs/source/conf.py*. The GUI version is set in *pyvolgui/__init__.py*, and the the version of

the GUI that the back-end expects is set again in *pyvol/__init__.py*. Experimental code is pushed with an alpha or beta designation (a or b before the final digit). GUI versions should only change when the GUI files are changed, but the version is intended to catch up to the backend version rather than the next available incrementation.

9.5 Distribution

The code is hosted on github by the Schlessinger Lab. The PyVOL backend is distributed through PyPI. This process of uploading to PyPI is automated in the *dev/build.sh* script. Installers are packaged using the *dev/package_plugins.sh* script. Documentation is generated and pushed to the github-hosted documentation website with the *dev/document.sh* script. All three are combined in the *dev/rebuild.sh* script. The plugin will be available both from the github page and (eventually) through the official PyMOL wiki.

9.6 Documentation

Documentation is in the Sphinx/RTD style. Module documentation is collated using *sphinx-apidoc*. The documentation website is built using the *sphinx-rtd-theme* and maintained on the gh-pages branch of PyVOL. The *pyvol_manual.pdf* is generated using sphinx's evocation of pdfTeX. PyPI can apparently not parse rst files, so the README.rst is converted to a md file using pandoc just prior to deployment.

9.7 Testing

Integration testing of the non-PyMOL components is performed using pytest out of *tests/test_pyvol.py*. These are invoked by running *python -m pytest* in the root pyvol directory. These tests have been run using pytest version 5.3.5. Installing *pytest-xdist* is recommended for efficiency's sake.

CHAPTER

TEN

PYVOL PACKAGE

10.1 Submodules

10.2 PyVOL Cluster Module

Contains functions to cluster spheres objects in memory; used in subpocket clustering.

pyvol.cluster.**cluster_within_r**(*spheres*, *radius*, *allow_new=True*)
Cluster spheres with the same radius using DBSCAN, modifying input data in situ

Args: spheres (Spheres): complete set of input spheres radius (float): radius at which clustering is to occur allow_new (bool): permit new clusters? (Default value = True)

pyvol.cluster.**cluster_between_r**(*spheres*, *ref_radius*, *target_radius*)
Cluster spheres from a target radius to a reference radius, modifying input data in situ

Args: spheres (Spheres): complete set of input spheres ref_radius (float): radius from which cluster identities will be drawn target_radius (float): radius to which cluster identities will be propagated

pyvol.cluster.**cluster_improperly_grouped**(*spheres*, *radius*, *min_cluster_size=1*, *max_clusters=None*)
Reassigns improperly clustered spheres to 'proper' clusters, modifying input data in situ

Args: spheres (Spheres): complete set of input spheres radius (float): radius at which closest groups are identified min_cluster_size (int): minimum number of spheres in a 'proper' cluster (Default value = 1) max_clusters (int): maximum number of 'proper' clusters (Default value = None)

pyvol.cluster.**extract_groups**(*spheres*, *surf_radius=None*, *prefix=None*, *group_names=None*)
Extracts spheres belonging to each cluster from the complete input set and optionally calculates bounded surfaces

Args: spheres (Spheres): complete set of input spheres surf_radius: radius used to calculate bounding spheres for individual groups (Default value = None) prefix: prefix to identify new surfaces (Default value = None)

Returns: group_list ([Spheres]): a list of Spheres objects each corresponding to a different cluster

pyvol.cluster.**hierarchically_cluster_spheres**(*spheres*, *ordered_radii*, *min_new_radius=None*, *min_cluster_size=10*, *max_clusters=None*)
Cluster spheres by grouping spheres at large radius and propagating those assignments down to smaller radii

Args: spheres (Spheres): complete set of input spheres ordered_radii ([float]): list of radii ordered from largest to smallest min_new_radius (float): smallest spheres to keep (Default value = None) min_cluster_size (int): minimum number of spheres in a cluster (Default value = 10) max_clusters (int): maximum number of clusters (Default value = None)

pyvol.cluster.**identify_closest_grouped**(*spheres*, *group*, *radius*)
Identifies the closest 'properly' grouped cluster to a specified group

Args: spheres (Spheres): complete set of input spheres group (float): group for which to identify the closest clusters radius (float): radius at which to perform the search

Returns: group (float): passthrough of input group closest (float): id of the closest cluster magnitude (int): number of pairwise closest connections between the queried group and the closest identified cluster

pyvol.cluster.**merge_sphere_list**(*s_list*, *r=None*, *g=None*)

Args: s_list ([Spheres]): list of input spheres r (float): radius value to assign to output Spheres (Default value = None) g (float): group value to assign to output Spheres (Default value = None)

Returns: merged_spheres (Spheres): a single Spheres object containing the merged input lists

pyvol.cluster.**reassign_group**(*spheres*, *source_group*, *target_group*)
Reassign a group in place

Args: spheres (Spheres): complete set of input spheres source_group (float): group to change target_group (float): new group id

pyvol.cluster.**reassign_groups_to_closest**(*spheres*, *group_list*, *radius*, *iterations=None*, *preserve_largest=False*)
Reassign a group to the closest group as identified by maximum linkage; operates in place

Args: spheres (Spheres): complete set of input spheres group_list ([float]): list of group ids which are to be iteratively reassigned radius (float): radius at which searches are to take place iterations (int): number of times to attempt to reassign groups (Default value = None) preserve_largest: keep the group id of the group with more members? (Default value = False)

pyvol.cluster.**remove_interior**(*spheres*)
Remove all spheres which are completely enclosed in larger spheres; operates in place

Args: spheres (Spheres): complete set of input spheres

pyvol.cluster.**remove_included_spheres**(*spheres*, *ref_spheres*, *radius*)
Removes all spheres with centers within radius of ref_spheres

pyvol.cluster.**remove_overlap**(*spheres*, *radii=None*, *spacing=0.1*, *iterations=20*, *tolerance=0.02*, *static_last_group=False*)
Remove overlap between groups; operates in place

Args: spheres (Spheres): complete set of input spheres radii ([float]): radii at which to perform searches for overlap (Default value = None) spacing (float): binning radius (Default value = 0.1) iterations (int): number of times to attempt overlap removal (Default value = 20) tolerance (float): overlap tolerance (Default value = 0.02) static_last_group (bool): don't move the 'other' group but rather the first group twice as much (effectively leaves the group with the highest index in place while moving everything else around it)

10.3 PyVOL Identify Module

pyvol.identify.**load_calculation**(*data_dir*, *input_opts=None*)
load the results of a calculation from file

Args: data_dir (str): directory where previous calculation results are stored input_opts (dict): dictionary of pyvol options that is used to update the options read in from file

Returns: pockets ([Spheres]): a list of Spheres objects each of which contains the geometric information describing a distinct pocket or subpocket opts (dict): updated PyVOL options dictionary

pyvol.identify.**pocket**(**opts*)
Calculates the SES for a binding pocket

Args: opts (dict): dictionary containing all PyVOL options (see pyvol.pymol_interface.pymol_pocket_cmdline for details)

Returns: pockets ([Spheres]): a list of Spheres objects each of which contains the geometric information describing a distinct pocket or subpocket

pyvol.identify.**pocket_wrapper**(**opts*)
wrapper for pocket that configures the logger, sanitizes inputs, and catches errors; useful when running from the command line or PyMOL but split from the core code for programmatic usage

Args: opts (dict): dictionary containing all PyVOL options (see pyvol.pymol_interface.pymol_pocket_cmdline for details)

Returns: pockets ([Spheres]): a list of Spheres objects each of which contains the geometric information describing a distinct pocket or subpocket output_opts (dict): dictionary containing the actual options used in the pocket calculation

pyvol.identify.**subpockets**(*bounding_spheres*, *ref_spheres*, **opts*)

Args: bounding_spheres (Spheres): a Spheres object containing both the peptide and solvent exposed face external spheres ref_spheres (Spheres): a Spheres object holding the interior spheres that define the pocket to be subdivided opts (dict): a dictionary containing all PyVOL options (see pyvol.configuration.clean_opts for details)

Returns: grouped_list ([Spheres]): a list of Spheres objects each of which contains the geometric information describing a distinct subpocket

pyvol.identify.**write_cfg**(**opts*)
write the processed configuration to file

Args: output_dir (str): output directory, relative or absolute prefix (str): identifying prefix for the output files

pyvol.identify.**write_report**(*all_pockets*, **opts*)
Write a brief report of calculated volumes to file

Args: all_pockets ([Spheres]): a list of Spheres objects each of which contains the complete information about a distinct pocket or subpocket output_dir (str): output directory, relative or absolute prefix (str): identifying prefix for output files

10.4 PyVOL Pymol Interface Module

Front facing PyMOL functions

pyvol.pymol_interface.**display_pockets**(*pockets*, **opts*)
Display a list of pockets

Args: pockets ([Spheres]): list of spheres object to display opts (dict): a dictionary containing all PyVOL options (see pyvol.pymol_interface.pymol_pocket_cmdline for details)

pyvol.pymol_interface.**load_calculation_cmdline**(*data_dir*, *prefix=None*, *display_mode=None*, *palette=None*, *alpha=None*)
Loads a pocket from memory and displays it in PyMOL

Args: data_dir (str): directory containing PyVOL output (by default ends in .pyvol) prefix (str): internal display name (Default value = None) display_mode (str): display mode (Default value = "solid") palette (str):

comma-separated list of PyMOL color strings (Default value = None) alpha (float): transparency value (Default value = 1.0)

pyvol.pymol_interface.**pymol_pocket_cmdline**(*protein=None*, *ligand=None*, *prot_file=None*, *lig_file=None*, *min_rad=1.4*, *max_rad=3.4*, *constrain_radii=True*, *mode='largest'*, *coordinates=None*, *residue=None*, *resid=None*, *lig_excl_rad=None*, *lig_incl_rad=None*, *min_volume=200*, *subdivide=False*, *max_clusters=None*, *min_subpocket_rad=1.7*, *max_subpocket_rad=3.4*, *min_subpocket_surf_rad=1.0*, *radial_sampling=0.1*, *inclusion_radius_buffer=1.0*, *min_cluster_size=50*, *project_dir=None*, *output_dir=None*, *prefix=None*, *logger_stream_level='INFO'*, *logger_file_level='DEBUG'*, *protein_only=False*, *display_mode='solid'*, *alpha=1.0*, *palette=None*)

PyMOL-compatible command line entry point

Args: protein (str): PyMOL-only PyMOL selection string for the protein (Default value = None) ligand (str): PyMOL-only PyMOL selection string for the ligand (Default value = None) prot_file (str): filename for the input pdb file containing the peptide–redundant with protein argument (Default value =-) lig_file (str): filename for the input pdb file containing a ligand–redundant with ligand argument (Default value = None) min_rad (float): radius for SES calculations (Default value = 1.4) max_rad (float): radius used to identify the outer, bulk solvent exposed surface (Default value = 3.4) constrain_radii (bool): restrict input radii to tested values? (Default value = False) mode (str): pocket identification mode (can be largest, all, or specific) (Default value = "largest") coordinates ([float]): 3D coordinate used for pocket specification (Default value = None) residue (str): Pymol-only PyMOL selection string for a residue to use for pocket specification (Default value=None) resid (str): residue identifier for pocket specification (Default value = None) lig_excl_rad (float): maximum distance from a provided ligand that can be included in calculated pockets (Default value = None) lig_incl_rad (float): minimum distance from a provided ligand that should be included in calculated pockets when solvent border is ambiguous (Default value = None) min_volume (float): minimum volume of pockets returned when running in 'all' mode (Default value = 200) subdivide (bool): calculate subpockets? (Default value = False) max_clusters (int): maximum number of clusters (Default value = None) min_subpocket_rad (float): minimum radius that identifies distinct subpockets (Default value = 1.7) max_subpocket_rad (float): maximum sampling radius used in subpocket identification (Default value = 3.4) min_subpocket_surf_rad (float): radius used to calculate subpocket surfaces (Default value = 1.0) inclusion_radius_buffer (float): buffer radius in excess of the nonextraneous radius from the identified pocket used to identify atoms pertinent to subpocket clustering (Default value = 1.0) radial_sampling (float): radial sampling used for subpocket clustering (Default value = 0.1) min_cluster_size (int): minimum number of spheres in a proper cluster; used to eliminate insignificant subpockets (Default value = 50) project_dir (str): parent directory in which to create the output directory if the output directory is unspecified (Default value = None) output_dir (str): filename of the directory in which to place all output; can be absolute or relative (Default value = None) prefix (str): identifying string for output (Default value = None) logger_stream_level (str): sets the logger level for stdio output (Default value = "INFO") logger_file_level (str): sets the logger level for file output (Default value = "DEBUG") protein_only (bool): PyMOL-only include only peptides in protein file display_mode (str): PyMOL-only display mode for calculated pockets (Default value = "solid") alpha (float): PyMOL-only display option specifying translucency of CGO objects (Default value = 1.0) palette (str): PyMOL-only display option representing a comma separated list of PyMOL color strings (Default value = None)

pyvol.pymol_interface.**pymol_pocket**(***opts*)

Perform PyMOL-dependent processing of inputs to generate input files for PyVOL pocket processing

Args: opts (dict): dictionary containing all PyVOL options (see pyvol.pymol_interface.pymol_pocket_cmdline for details)

Returns: pockets ([Spheres]): a list of Spheres objects each of which contains the geometric information describing a distinct pocket or subpocket output_opts (dict): dictionary containing the actual options used in the pocket calculation

10.5 PyVOL Pymol Utilities Module

PyMOL convenience functions used by the front-end contained in pymol_interface.

pyvol.pymol_utilities.**construct_palette**(*color_list=None*, *max_value=7*, *min_value=0*)
Construct a palette

Args: color_list ([str]): list of PyMOL color strings (Default value = None) max_value (int): max palette index (Default value = 7) min_value (int): min palette index (Default value = 1)

Returns: palette ([str]): list of color definitions

pyvol.pymol_utilities.**display_pseudoatom_group**(*spheres*, *name*, *color='gray60'*, *palette=None*)
Displays a collection of pseudoatoms

Args: spheres (Spheres): Spheres object holding pocket geometry name (str): display name color (str): PyMOL color string (Default value = 'gray60') palette ([str]): palette (Default value = None)

pyvol.pymol_utilities.**display_spheres_object**(*spheres*, *name*, *state=1*, *color='marine'*, *alpha=1.0*, *mode='solid'*)
Loads a mesh object into a cgo list for display in PyMOL

Args: spheres (Spheres): Spheres object containing all geometry name (str): display name state (int): model state (Default value = 1) color (str): PyMOL color string (Default value = 'marine') alpha (float): transparency value (Default value = 1.0) mode (str): display mode (Default value = "solid") palette ([str]): palette (Default value = None)

pyvol.pymol_utilities.**mesh_to_solid_CGO**(*mesh*, *color*, *alpha=1.0*)
Creates a solid CGO object for a mesh for display in PyMOL

Args: mesh (Trimesh): Trimesh mesh object color (str): PyMOL color string (Default value = 'gray60') alpha (float): transparency value (Default value = 1.0)

Returns: cgobuffer (str): CGO buffer that contains the instruction to load a solid object

pyvol.pymol_utilities.**mesh_to_wireframe_CGO**(*mesh*, *color_tuple*, *alpha=1.0*)
Creates a wireframe CGO object for a mesh for display in PyMOL

Args: mesh (Trimesh): Trimesh mesh object color (str): PyMOL color string (Default value = 'gray60') alpha (float): transparency value (Default value = 1.0)

Returns: cgobuffer (str): CGO buffer that contains the instruction to load a wireframe object

10.6 PyVOL Spheres Module

Defines the Spheres class which holds geometric information and performs basic operations on its data

class pyvol.spheres.**Spheres**(*xyz=None*, *r=None*, *xyzr=None*, *xyzrg=None*, *g=None*, *pdb=None*, *bv=None*, *mesh=None*, *name=None*, *spheres_file=None*)
Bases: object

copy()
Creates a copy in memory of itself

calculate_surface(*probe_radius=1.4*, *cavity_atom=None*, *coordinate=None*, *all_components=False*, *exclusionary_radius=2.5*, *largest_only=False*, *noh=True*, *min_volume=200*)
Calculate the SAS for a given probe radius

Args: probe_radius (float): radius for surface calculations (Default value = 1.4) cavity_atom (int): id of a single atom which lies on the surface of the interior cavity of interest (Default value = None) coordinate ([float]): 3D coordinate to identify a cavity atom (Default value = None) all_components (bool): return all pockets? (Default value = False) exclusionary_radius (float): maximum permissibile distance to the closest identified surface element from the supplied coordinate (Default value = 2.5) largest_only (bool): return only the largest pocket? (Default value = False) noh (bool): remove waters before surface calculation? (Default value = True) minimum_volume (int): minimum volume of pockets returned when using 'all_components' (Default value = 200)

identify_nonextraneous(*ref_spheres*, *radius*)
Returns all spheres less than radius away from any center in ref_spheres using cKDTree search built on the non-reference set

Args: ref_spheres (Spheres): object that defines the pocket of interest radius (float): maximum distance to sphere centers to be considered nonextraneous

Returns: nonextraneous (Spheres): a filtered Spheres object

nearest(*coordinate*, *max_radius=None*)
Returns the index of the sphere closest to a coordinate; if max_radius is specified, the sphere returned must have a radius <= max_radius

Args: coordinate (float nx3): 3D input coordinate max_radius (float): maximum permissibile distance to the nearest sphere (Default value = None)

Returns: nearest_index: index of the closest sphere

propagate_groups_to_external(*coordinates*, *tolerance=3*)
Propagates group identifications to an external set of coordinates

Args: coordinates (Nx3 ndarray): coordinates of the external spheres tolerance (float): maximum distance exclusive of the radii of the internal spheres

Returns: prop_groups ([int]): list of group identifications for the supplied external coordinates

nearest_coord_to_external(*coordinates*)
Returns the coordinate of the sphere closest to the supplied coordinates

Args: coordinates (float nx3): set of coordinates

Returns: coordinate (float 1x3): coordinate of internal sphere closest to the supplied coordinates

remove_duplicates(*eps=0.01*)
Remove duplicate spheres by identifying centers closer together than eps using DBSCAN

Args: eps (float): DBSCAN input parameter (Default value = 0.01)

remove_ungrouped()
Remove all spheres that did not adequately cluster with the remainder of the set

remove_groups(*groups*)
Remove all spheres with specified group affiliations

Args: groups ([float]): list of groups to remove

write (*filename*, *contents='xyzrg'*, *output_mesh=True*)
Writes the contents of _xyzrg to a space delimited file

Args: filename (str): filename to write the report and mesh if indicated contents (str): string describing which columns to write to file (Default value = "xyzrg") output_mesh (bool): write mesh to file? (Default value = True)

xyzrg
Retrieve the coordinates, radii, and group ids

xyzr
Retrieve coordinates and radii

xyz
Retrieve the coordinates

r
Retrieve the radii

g
Retrieve the group indices

10.7 PyVOL Utilities Module

pyvol.utilities.**calculate_rotation_matrix** (*ref_vector*, *new_vector*)
Calculates the 3D rotation matrix to convert from ref_vector to new_vector; not used in main PyVOL calculations

Args: ref_vector (3x1 ndarray): original vector new_vector (3x1 ndarray): target vector

Returns: rot_matrix (3x3 ndarray): rotation matrix to convert the original vector to the target vector

pyvol.utilities.**closest_vertex_normals** (*ref_mesh*, *query_mesh*, *ref_coordinates=None*, *ref_radius=2*, *interface_gap=2*)
Returns the location and normal for the closest point between two meshes

Args: ref_mesh (trimesh): origin mesh query_mesh (trimesh): target mesh ref_coordinates (3xN ndarray): coordinates used to specify the pertinent subregion on the ref_mesh ref_radius (float): radius used to identify points on the ref_mesh that are sufficiently close to the ref_coordinates interface_gap (float): maximum distance between the ref and query meshes at the identified point

Returns: mean_pos (3x1 ndarray): coordinate of the central point between the meshes mean_normal (3x1 ndarray): normalized vector pointing from the ref_mesh to the query_mesh

pyvol.utilities.**check_dir** (*location*)
Ensure that a specified directory exists

Args: location (str): target directory

pyvol.utilities.**configure_logger** (*filename=None*, *stream_level=None*, *file_level=None*)
Configures the base logger

Args: filename (str): target filename is the log is to be written to file (Default value = None) stream_level (str): log level for the stream handler (Default value = None) file_level (str): log level for the file handler (Default value = None)

pyvol.utilities.**clean_logger** ()
Removes current handlers from the main PyVOL logger so that new ones can be assigned

pyvol.utilities.**coordinates_for_resid**(*pdb_file*, *resid*, *chain=None*, *model=0*, *sidechain_only=True*)

Extract the 3D coordinates for all atoms in a specified residue from a pdb file

Args: pdb_file (str): filename of the specified pdb file resid (int): residue number chain (str): chain identifier (Default value = None) model (int): model identifier (Default value = 0) sidechain_only (bool): return only sidechain atom coordinates? (Default value = True)

Returns: coordinates ([[float]]): 3xN array containing all atomic positions

pyvol.utilities.**run_cmd**(*options*, *in_directory=None*)

Run a program using the command line

Args: options ([str]): list of command line options in_directory (str): directory in which to run the command (Default value = None)

pyvol.utilities.**surface_multiprocessing**(*args*)

A single surface calculation designed to be run in parallel

Args:

args: a tuple containing: spheres (Spheres): a Spheres object containing all surface producing objects probe_radius (float): radius to use for probe calculations kwargs (dict): all remaining arguments accepted by the surface calculation algorithm

Returns: surface (Spheres): the input Spheres object but with calculated surface parameters

pyvol.utilities.**sphere_multiprocessing**(*spheres*, *radii*, *workers=None*, ***kwargs*)

A wrapper function to calculate multiple surfaces using multiprocessing

Args: spheres (Spheres): input Spheres object radii ([float]): list of radii at which surfaces will be calculated workers (int): number of workers (Default value = None) kwargs (dict): all remaining arguments accepted by surface calculation that are constant across parallel calculations

Returns: surfaces ([Spheres]): a list of Spheres object each with its surface calculated

PYTHON MODULE INDEX

p

INDEX

S

W

X