PyVOL Documentation

Release 1.2.45

Ryan Smith

Mar 17, 2020

CONTENTS

CHAPTER

ONE

INTRODUCTION

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 under development. 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 "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+.

2.1 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 *MSMS Installation*.

As an alternative, 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
```

2.2 Installation into PyMOL Using 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. `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 *MSMS Installation*.

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

```
install_pyvol_local
```

2.3 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, see 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.

2.4 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
```

2.5 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
```

2.6 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

The GUI and PyMOL prompt interfaces are all but identical. The shell interface is run using input configuration files and is best used to automate tasks first tested within PyMOL. Explanations for running PyVOL with the PyMOL GUI or prompt are covered first. Additional details covering shell invocation follow. Programmatic invocation of internal functions is supported and covered through the module documentation.

The next few section describe the parameters controlling Basic Usage, *Pocket Specification*, *Partitioning Options*, and *Output and Display Options* along with the corresponding GUI sections.

Fig. 1: The Basic Parameters section of the PyVOL GUI

3.1 Protein Selection

The *Protein Selection* (command-line *protein* required to be in the first position) is a PyMOL selection string that identifies all atoms that should be considered to occlude space. This can be any interpretable PyMOL selection. The *only include PyMOL 'poly'* checkbox (command-line *exclude_org*) is a convenience option that simply appends *'and poly'* to the given selection string to exclude water, solvents, and any other small molecule, non-peptide atoms. If a cofactor should be considered part of the binding site, it might make sense to group it in with the peptide for calculating the solvent accessible surface effectively present for the small molecule of interest.

```
pocket <protein selection>
```

3.2 Min and Max Radii

The most important parameters (command-line *min_rad* and *max_rad*) controlling PyVOL pocket identification and boundary location are the minimum and maximum radii 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 A are reasonable, but lower values can be provided to increase the stringency of pocket detection or higher values to reduce the stringency. 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. Minimum radii as large as the smallest pharmacophore radius of potential ligands make sense for calculations, but the radius of water is default and normally best because it is what users expect when looking at solvent accessible surfaces.

```
pocket <protein selection>, max_rad=<3.4>, min_rad=<1.4>
```

3.3 Input Constraint

By default, input quantitative parameters are compared and constrained to test ranges. The *Constrain Inputs to Tested Ranges* (command-line *constrain_inputs*) toggles this feature. While edge cases are possible in which violating constraints is useful, in practice these constraints represent effective ranges. In particular, setting the minimum radius to absurdly low values will start fitting pockets in even intramolecular spaces and provide meaningless output if not a crash.

```
pocket <protein selection>, constrain_inputs=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, saved PyMOL *.pml* logs should recreate results. 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 a single volume and geometry. However, manual identification of the pocket of interest is generally preferable. This can be done through specification of a ligand, a residue, or a coordinate. If any specification is given, the mode is changed to *specific* by default. The *specific* mode is the fastest by a small margin because it calculates the fewest surfaces. All parameters controlling pocket specification are contained in the corresponding GUI section.

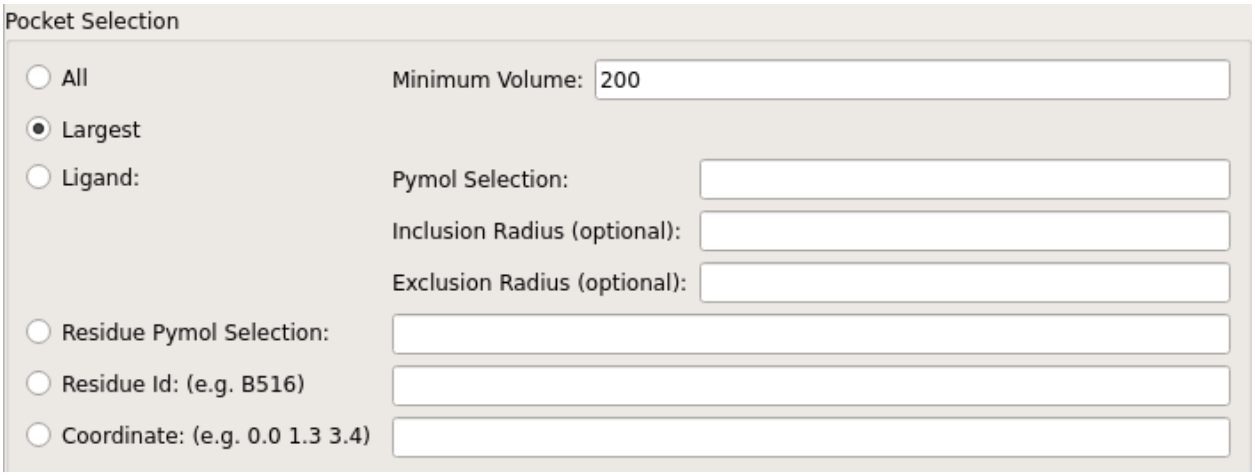

Fig. 1: GUI section controlling user specification of binding pockets

4.1 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 can normally choose the pharmacologically interesting pocket in a protein. However, sometimes changes in the minimum or maximum radius can lead to the unexpected selection of an alternative pocket.

```
# Equivalent expressions
pocket <protein_selection>
pocket <protein_selection>, mode="largest"
```

4.2 All Mode

When running in *all* mode, PyVOL determines all surfaces with inward-facing normals with volume over a minimum threshold defined by the *Minimum Volume* (command-line *minimum_volume*). 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, and 3) subpocket partitioning cannot occur on the output from this mode. By default the *Minimum Volume* is set to 200 A^3. This is a heuristically determined threshold that is generally useful at distinguishing between artifacts and interesting pockets.

```
pocket <protein_selection>, mode="all", minimum_volume=<200>
```

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. There is an internal priority to which specification is used, but only a single option should be used.

4.3.1 Ligand Specification

A ligand occupying the binding pocket of interest can be specified using the GUI's *Ligand:* → *PyMOL Selection* field (command-line *ligand*). If the ligand selection is included in the protein selection, it is removed from the protein selection before the algorithm runs.

```
# Equivalent expressions
pocket <protein_selection>, mode="specific", ligand=<ligand_selection>
pocket <protein_selection>, ligand=<ligand_selection>

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

Supplying a ligand opens up two additional options. *Inclusion Radius* (command-line *lig_incl_rad*) 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* (command-line *lig_excl_rad*) 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.

```
pocket <protein_selection>, ligand=<ligand_selection>, lig_incl_rad=<3.5>, lig_excl_
↪rad=<5.2>
```

4.3.2 Residue Specification

A residue can be supplied to localize a pocket. This can be done either with a PyMOL selection string or by specifying a residue ID. The *Residue PyMOL Selection* (command-line *residue*) takes an input PyMOL selection (which can be arbitrarily large or small but was designed to hold a single side chain). The *Residue Id* (command-line *resid*) accepts a string specifying an optional chain and a required residue index. For example, residue 35 of chain A would be specified by 'A35'. If only a single chain is present, the chain identifier can be omitted. PyVOL tries to identify the

residue atom closest to an interior surface and uses that atom to specify the adjacent pocket of interest. Sometimes a residue is adjacent to multiple pockets. That makes it a poor, unpredictable choice for specification. If having trouble, specify a single atom as a PyMOL selection string.

```
pocket <protein_selection>, resid=<A15>
pocket protein_selection, residue=<residue_selection>
```

4.3.3 Coordinate Specification

The final method for specifying a pocket interest is through providing a *Coordinate* (command-line *pocket_coordinate*) that is within the pocket. 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 ("x y z"). When running on the command-line, quotation marks are necessary given default argument processing.

```
pocket protein_selection, pocket_coordinate="5.0 10.0 15.0"
```

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.

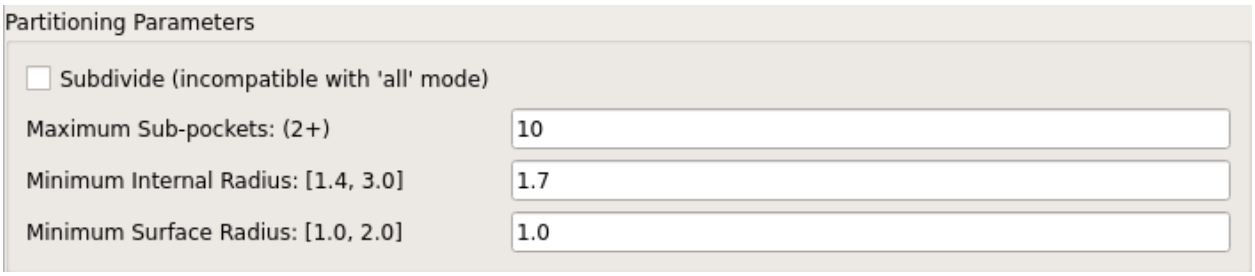

Fig. 1: GUI section controlling user binding pocket partition into subpockets

5.1 Enabling Subpocket Partitioning

Subpocket partitioning is enabled in the GUI by selecting the *Subdivide* checkbox (command-line *subdivide*).

```
pocket <protein_selection>, subdivide=True
```

5.2 Limiting 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 the *Maximum Sub-pockets* (command-line *max_clusters*) and the *Minimum Internal Radius* (command-line *min_subpocket_rad*). PyVOL clusters volume into the maximum number 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 *Minimum Internal Radius* (the smallest sphere used to identify new regions) and the radial sampling frequency. Through its PyMOL interface, PyVOL currently fixes the radial sampling frequency at 10 A^-1. Larger values of the *Minimum Internal Radius* prohibit unique identification of smaller regions. The other parameter, *Maximum Sub-pockets*, sets the maximum number of clusters calculated. If the number of clusters originally determined is greater than the supplied maximum, clusters are iteratively merged using a metric that is related to the edge-weighted surface area between adjacent clusters.

```
pocket <protein_selection>, subdivide=True, min_subpocket_rad=<1.7>, max_clusters=<10>
```

5.3 Smoothness of the Subpocket Surfaces

The size of the probe used to calculate surface accesibility of subpockets can be set with the *Minimum Surface Radius* (command-line *min_subpocket_surf_rad*) in the Partitioning Parameters section of the GUI. Calculation stability is less sensitive to the value of this parameter than the overall *Minimum Radius*. In practice, it should be set to a value slightly smaller than the overall *Minimum Radius* but not aphysically small. Unless changing the *Minimum Radius* used for overall calculations, the default value should be left unchanged.

```
pocket <protein_selection>, subdivide=True, min_subpocket_surf_rad=<1.2>
```

CHAPTER

SIX

OUTPUT AND DISPLAY OPTIONS

By default, PyVOL simply outputs a log containing volumes and, when invoked through PyMOL, displays pocket boundaries as semi-translucent surfaces. This behavior can be extensively customized.

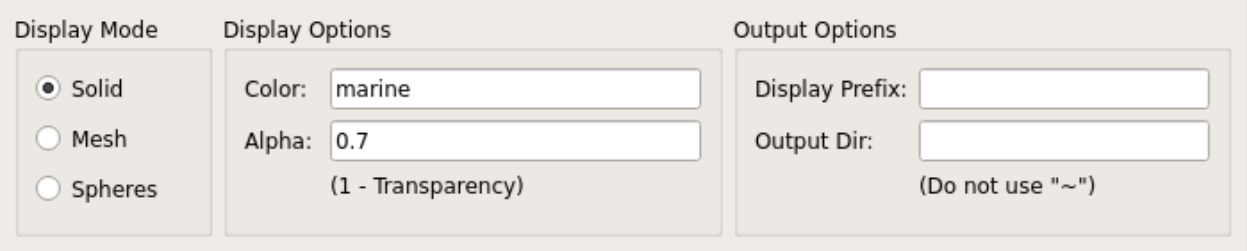

Fig. 1: GUI section controlling file output and PyMOL visualization of new results

6.1 File Output Options

The output name for all computed PyMOL objects and the base filename for any output files can be specified using the *Display Prefix* (command-line *prefix*) option. A prefix is otherwise derived from the input protein selection string.

```
pocket <protein_selection>, prefix=<output_prefix>
```

PyVOL can also write the input and output files to a directory specified with *Output Dir* (command-line *output_dir*). In this case it writes out the input protein and ligand structures, a csv report of all calcuated volumes, and paired csv/obj files containing tangent sphere collections and 3D triangulated mesh files respectively. Relative and absolute paths both work. However, in many cases referencing the home directory with ~ will not access the user's home directory. OBJ files can be read by almost all 3D file viewers.

```
pocket <protein_selection>, output_dir=<output_dir>
```

6.2 Display Options

Calculated surfaces can be visualized in three different ways by setting the *Display Mode* (command-line *display_mode*) parameter. The following three commands set the output as a solid surface with transparency, a wireframe mesh, and a collection of spheres. Color is set with the *Color* (command-line *color*) parameter that accepts any PyMOL color string. Transparency (when applicable) is set with the *Alpha* (command-line *alpha*) parameter that is a float [0.0, 1.0] that is equal to (1 - transparency).

```
pocket protein_selection, display_mode=solid, alpha=0.85, color=skyblue
pocket protein_selection, display_mode=mesh, color=red
pocket protein_selection, display_mode=spheres, color=firebrick
```

The presets should generally be sufficient, but custom colors can be chosen using the commands given on the PyMOL wiki. When creating multiple surfaces at the same time, PyVOL generates a palette and assigns each surface a different color. This can distinguish an arbitrary number of surfaces. A palette can be manually specified using the *Palette* (command-line *palette*) parameter. The palette should be provided as a comma-separated list of PyMOL color strings. If an insufficient number of colors are provided for the number of generated surfaces, additional values are interpolated automatically.

```
pocket protein_selection, palette="tv_red,tv_orange,marine,magenta"
```

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 Notes on Output

Currently, PyVOL only reports standard log output to stdout when run this way. So if an output directory is not provided, there is no easy way to retrieve the results.

CHAPTER

EIGHT

LOADING PREVIOUS RESULTS

PyMOL can not 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. The *Load Pocket* (command-line *load_pocket*) command allows previous results to be read back in from file. In order to generate these input files, an *Output Dir* (command-line *output_dir*) must be set.

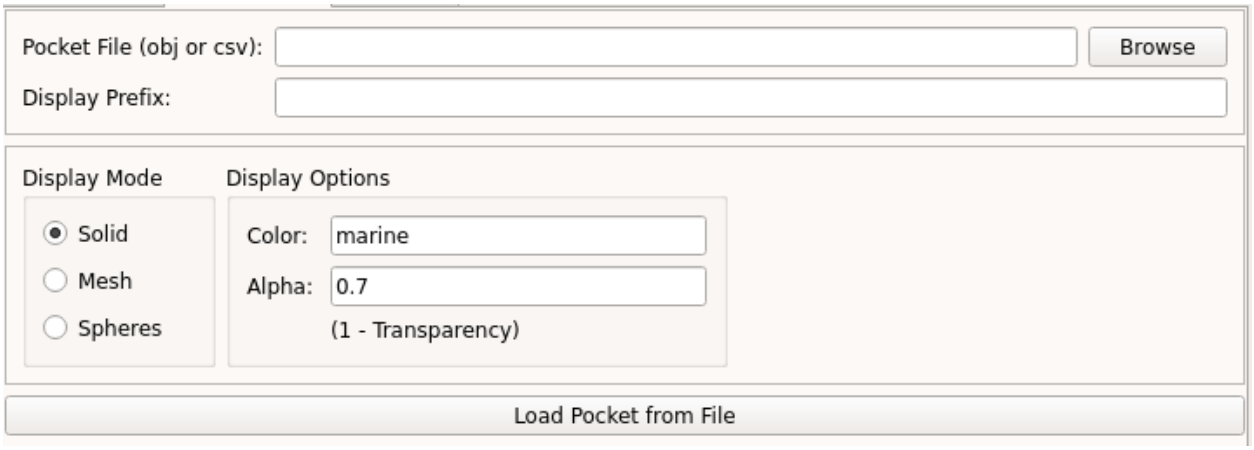

Fig. 1: GUI section that loads previous calculations into PyMOL for visualization

8.1 Selecting Saved Results

PyVOL saves its output as a collection of two files that hold all of the information to recreate the original *Spheres* object in memory. These files consist of a csv holding *Spheres* data and a 3D file holding surface triangulation information. If a surface has not been computed for a *Spheres* object, the 3D file will not be written out. However, all modes accessible through the PyMOL command-line and GUI interfaces will write a 3D file. The 3D files by default are written in the *OBJ* format. Either the 3D file or the csv can be provided to read in the collection of both.

```
load_pocket <3D or csv file>, name=<new PyMOL name>, display_mode=<"solid">, color=<
↪"marine">, alpha=<0.5>
```

8.2 Display Options

These options in the GUI are identical to the display options when performing a *de novo* calculation. On the command-line, *name* is accepted in place of *prefix*. Please see *Display Options* for more information

CHAPTER

NINE

DEVELOPMENT

9.1 Package Design

The main PyVOL algorithm is run from identify.py. There are two interfaces to this module which prepare user supplied inputs: *pyvol/pymol_interface.py* and the commandline entry point in *pyvol/__main__.py*. The PyMOL interface can be accessed directly through the PyMOL prompt or run using the included GUI. The command line entry point is evoked using a configuration file. The main entry point for accessing with the API is through *pyvol/identify.py*.

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 earlier PyMOL distributions.

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*.

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 through the official PyMOL wiki.

9.6 Documentation

Documentation is largely in the google style but being switched to sphinx 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.

CHAPTER

TEN

PYVOL PACKAGE

10.1 Submodules

10.2 PyVOL Cluster Module

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: prot_file (str): filename for the input pdb file containing the peptidee mode (str): pocket identification mode (can be largest, all, or specific) (Default value = "largest") lig_file (str): filename for the input pdb file containing a ligand (Default value = None) coordinate ([float]): 3D coordinate used for pocket specification (Default value = None) resid (str): residue identifier for pocket specification (Default value = None) coordinates ([float]): 3D coordinate of an atom in a surrounding residue used for pocket specification (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) 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) subdivide (bool): calculate subpockets? (Default value = False) min_volume (float): minimum volume of pockets returned when running in 'all' mode (Default value = 200) min_subpocket_rad (float): minimum radius that identifies distinct subpockets (Default value = 1.7) min_subpocket_surf_rad (float): radius used to calculate subpocket surfaces (Default value = 1.0) max_clusters (int): maximum number of clusters (Default value = None) min_cluster_size (int): minimum number of spheres in a proper cluster; used to eliminate insignificant subpockets (Default value = 50) 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) prefix (str): identifying string for output (Default value = None) output_dir (str): filename of the directory in which to place all output; can be absolute or relative (Default value = None) constrain_inputs (bool): restrict quantitative input parameters to tested values? (Default value = False)

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

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 min_rad (float): radius for original SES calculations (Default value = 1.4) max_rad (float): radius originally used to identify the outer, bulk solvent exposed surface (Default value = 3.4) min_subpocket_rad (float): minimum radius that identifies distinct subpockets (Default value = 1.7) min_subpocket_surf_rad (float): radius used to calculate subpocket surfaces (Default value = 1.0) max_subpocket_rad (float): maximum spheres radius used for subpocket clustering (Default value = None) sampling (float): radial sampling frequency for clustering (Default value = 0.1) inclusion_radius_buffer (float): defines the inclusion distance for nonextraneous spheres in combination with min_rad and max_rad (Default value = 1.0) min_cluster_size (int): minimum number of spheres that can constitute a proper clusterw (Default value = 50) max_clusters (int): maximum number of clusters (Default value = None) prefix (str): identifying string for output (Default value = None)

Returns: pockets ([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 Poses Module

pyvol.poses.**compound_occupancy**(*pose_file, pocket_file, output_dir, output_prefix=None, name_parameter='_Name', scoring_parameter='r_i_glide_gscore', pocket_tolerance=3, panelx=250, panely=200, molsPerRow=4, rowsPerPage=6, palette=[(1, 0.2, 0.2), (1, 0.55, 0.15), (1, 1, 0.2), (0.2, 1, 0.2), (0.3, 0.3, 1), (0.5, 1, 1), (1, 0.5, 1)]*)

Creates a report that highlights 2D compound representations by subpocket occupancy according to the poses in a provided sdf file

Args: pose_file (str): input SDF file containing docked compound poses pocket_file (str): input csv containing the spheres' 5 dimensional array describing subpocket geometry; output with a "_spa.csv" ending output_dir (str): output directory for all files output_prefix (str): output prefix name_parameter (str): SDF property key for the molecule name scoring_parameter (str): SDF property key for whichever property should be shown in the report pocket_tolerance (float): maximum distance (Angstrom) at which an atom outside of the defined subpocket volumes is still associated with a subpocket panelx (int): horizontal width of the drawing space for each molecule panely (int): vertical height of the drawing space for each molecule molsPerRow (int): number of molecules to fit on a row (total width is <= panelx * molsPerRow) rowsPerPage (int): number of rows of molecules to fit on each page (total height is <= panely * rowsPerPage) palette ([(float,float,float)]): list of tuples of fractional RGB values that controls the highlighting colors

pyvol.poses.**draw_molecules_with_subpockets**(*molecules, output_dir, output_prefix, palette=[(1, 0.2, 0.2), (1, 0.55, 0.15), (1, 1, 0.2), (0.2, 1, 0.2), (0.3, 0.3, 1), (0.5, 1, 1), (1, 0.5, 1)], panelx=250, panely=200, molsPerRow=4, rowsPerPage=6, name_parameter='_Name', scoring_parameter='r_i_glide_gscore'*)

Creates a report that highlights the 2D representation of docked molecules to indicate subpocket occupancy

Args: molecules ([rdkit.Chem.rdchem.Mol]): list of rdkit molecules to draw output_dir (str): output directory output_prefix (str): output prefix palette ([(float,float,float)]): list of tuples of fractional RGB values; TODO: allow input of pymol style color strings panelx (int): horizontal width of the drawing space for each molecule panely (int): vertical height of the drawing space for each molecule molsPerRow (int): number of molecules to fit on a row (total width is <= panelx * molsPerRow) rowsPerPage (int): number of rows of molecules to fit on each page (total height is <= panely * rowsPerPage) name_parameter (str): SDF property key for the molecule name scoring_parameter (str): SDF property key for whichever property should be shown in the report

pyvol.poses.**set_atom_subpockets**(*molecule*, *groups*)

Apply calculated subpocket groups to each atom in a molecule

Args: molecule (rdkit.Chem.rdchem.Mol): rdkit molecule object groups ([int]): list of subpocket identifiers

10.5 PyVOL Pymol Interface Module

10.6 PyVOL Pymol Utilities Module

pyvol.pymol_utilities.**construct_palette**(*color_list=None*, *max_value=7*, *min_value=1*)
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=0.7*, *mode='solid'*, *palette=None*)
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 = 0.7) mode (str): display mode (Default value = "solid") palette ([str]): palette (Default value = None)

pyvol.pymol_utilities.**mesh_to_solid_CGO**(*mesh*, *color='gray60'*, *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='gray60'*, *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

10.7 PyVOL Spheres Module

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*, *minimum_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

10.8 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

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**()

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

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

10.9 PyVOL Main Entry Point

PYTHON MODULE INDEX

p

INDEX

S

W

X