Implementation of a geometric algorithm to find proteins' binding sites

Courses: Structural Bioinformatics and Python

Jingqi Fu, Maria Tarrat Castells, Xareni Reyes Soto

22-April-2025

Content

1 Introduction

Identifying the binding pockets of a protein is crucial for determining which molecules can interact with it. As a result, considerable research has been devoted to localizing binding pockets using geometry, machine learning, and deep learning approaches.

For this project, we chose a geometric approach. We began by reviewing several geometric methods described in the literature. Our first focus was the algorithm proposed by Saberi Fathi and Tuszynski (2014), which we implemented in Python. However, we observed that it performed poorly. For instance, when applied to the test protein with PDB code 1a6u, the algorithm identified 73 potential binding sites—substantially more than the 16 final pockets reported in the original article. We arrived at this number after applying our own biochemical filtering conditions, as the article did not specify how such filters were applied. After ranking the predicted pockets based on volume, surface area, and residue diversity, we reduced the list to 39—still too many for a relatively small protein. Consequently, we decided to explore alternative methods.

Next, we examined the LIGSITEcsc algorithm (Huang & Schroeder, 2006), which builds upon and enhances the original LIGSITE method (Hendlich et al., 1997). LIGSITEcsc models the protein surface using a Connolly surface (Connolly, 1983), which intuitively represents the surface accessible to a probe sphere rolling over the protein. The intuitive idea behind the construction of this surface is easy to grasp. However, its rigorous implementation in a programming language is far from trivial, as it involves advanced geometric concepts that fall outside the scope and objectives of this project.

The final method we reviewed—and ultimately chose to implement—is based on the LIGSITE algorithm. LIGSITE, published in 1997 (Hendlich et al., 1997), is inspired by the earlier POCKET method (Levitt & Banaszak, 1992). Both algorithms identify binding pockets by scanning a protein structure on a cubic grid. However, the results from the POCKET method depended on the protein orientation in 3D space. LIGSITE addressed this limitation by performing a more robust and orientation-independent scanning process.

None of the three algorithms we studied had publicly available source code. Additionally, the first approach we evaluated lacked critical details, particularly in steps 7 (determining the physical properties of the pockets) and 9 (applying biochemical filtering conditions). Without clear parameter definitions or step-by-step specifications, it was not possible for us to accurately reproduce the results reported in the original paper. While we had a solid understanding of the general flow of the second algorithm, we encountered a major obstacle when attempting to implement the Connolly surface generation. We explored existing libraries to handle this computation, but unfortunately, no native Python solutions were available that supported the generation of Connolly surfaces.

The next section explains in detail the algorithm we implemented.

2 Algorithm

The algorithm models the protein surface using the van der Waals surface and simulates solvent accessibility by incorporating a probe sphere of radius 1.4 Å.

2.1 Parsing the PDB File

The program reads the input pdb file and extracts information about the residues (id, name, chain ID) and atoms (ID, name, coordinates). The program creates two dictionaries to store the atoms' information and the residues' information:

`atom_id.coords.dict`: maps atom IDs to their 3D coordinates.
`residue_info.dict`: stores detailed information for each residue, including the list of atoms, their coordinates, and atomic elements.

2.2 Creating the Bounding Box and Voxel Grid, and marking occupied voxels

In this step, the program creates a bounding box for the protein, extended by a padding value to account for atomic radii and the solvent probe. This 3D space is discretized into small cubes, called *voxels*, each with a side of length 0.5 Å for increased spatial resolution.

The padding value, `r_max`, is the sum of the maximum van der Waals radius among all atoms (`r_max_atom`) and the probe radius (`r_probe`). In our implementation:

`r_max_atom` = 1.85 Å,
`r_probe` = 1.4 Å.

Thus `r_max` = 3.25 Å. Let `x_coords`, `y_coords`, and `z_coords` represent lists of all the protein atoms' x, y, and z coordinates. The bounding box limits are:

```
xmin = min(x_coords) - r_max
xmax = max(x_coords) + r_max
ymin = min(y_coords) - r_max
ymax = max(y_coords) + r_max
zmin = min(z_coords) - r_max
zmax = max(z_coords) + r_max
```

The box boundaries are [`xmin`, `xmax`], [`ymin`, `ymax`], and [`zmin`, `zmax`]. The bounding box is then discretized into voxels, and a dictionary `voxel_grid` is initialized where each voxel is indexed by its position (i, j, k) along the x, y, and z axes. Initially, all voxels are marked with the value 0, indicating that they are *unoccupied*. The values for a subset of voxels will change in the following steps.

We used a voxel size of 0.5 Å to have a finer approximation of the protein surface. This value falls within the range suggested in (Hendlich et al., 1997).

2.2.1 Marking Occupied Voxels

We mark *occupied* voxels at this stage. A voxel is *occupied* if there are protein atoms sufficiently close to it, where *sufficiently close* means within a distance r defined as:

$$r = r_{\text{atom}} + r_{\text{probe}}.$$

Here, r_{atom} is the van der Waals radius of the atom's element and $r_{\text{probe}} = 1.4$ Å. To speed up the search, for each protein atom p, we compute a small bounding box around it and restrict the voxel search to it. Let x, y, z be the coordinates of atom p, and r defined as above. Then we restrict our search for each atom to the box with boundaries $[x - r, x + r]$, $[y - r, y + r]$, $[z - r, z + r]$. A 2D illustration of this concept is shown in Figure 1.

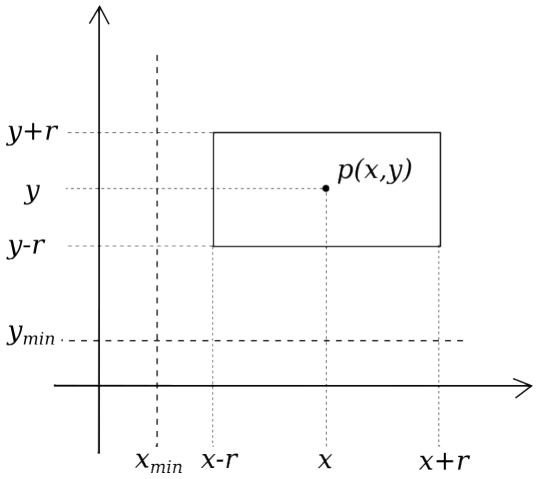

Figure 1: Small bounding box around the point $p(x, y)$.

The voxel indices corresponding to this smaller bounding box are computed as:

```
i0 = floor((x - r - xmin) / voxel_size)
i1 = floor((x + r - xmin) / voxel_size) + 1
j0 = floor((y - r - ymin) / voxel_size)
j1 = floor((y + r - ymin) / voxel_size) + 1
k0 = floor((z - r - zmin) / voxel_size)
k1 = floor((z + r - zmin) / voxel_size) + 1
```

Then, we iterate over all voxels within this subregion. At each step, we get the Cartesian coordinates of the *voxel center*. For this, we convert the voxel index to Cartesian coordinates and add `voxel_size/2` to each coordinate. If the Euclidean distance between the atom and the voxel center is less than or equal to r, we mark the voxel as *occupied* by setting `voxel_grid[(i,j,k)]` to -1. These voxels represent regions that are inaccessible to solvent.

2.3 Scanning Along Coordinate Axes

A sequence of voxels that starts with protein, is followed by a solvent, and ends with protein is called a *protein-solvent-protein (PSP) event*. In practice, we look for runs of zeros enclosed on both sides by -1.

During this phase, the program independently scans the voxel grid along the three principal axes (x, y, and z). For each scan direction, we iterate through all lines parallel to the selected axis. Along each line, we look for sequences of zeros (solvent voxels) enclosed on both sides by -1 (protein voxels). When such a sequence is found, the values of the voxels within the solvent region are incremented by 1, indicating that they are part of a PSP event.

Example

Consider the following sequence of voxel values along a line:

$$[0, 0, -1, 0, 0, 0, 0, -1, -1, -1, 0, 0, -1].$$

Between the two -1 voxels at indices 2 and 7, we find the solvent voxels at indices 3, 4, 5, and 6. These values are increased by 1. Similarly, the solvent voxels between indices 9 and 12 (indices 10 and 11) are also incremented.

2.4 Scanning Along Diagonal Directions

Additional scans are performed along four cubic diagonal directions to reduce the dependency on the protein's orientation in the 3D space. The diagonal vectors are $(1, 1, 1)$, $(1, 1, -1)$, $(1, -1, 1)$, and $(1, -1, -1)$. They are derived by considering a cube of side length 1 with one vertex at the origin. The cube has vertices at: $(0, 0, 0)$, $(1, 0, 0)$, $(0, 1, 0)$, $(1, 1, 0)$, $(0, 1, 1)$, $(1, 1, 1)$, $(1, 0, 1)$, $(0, 0, 1)$. The four cubic diagonals are obtained by connecting specific pairs of vertices, such as $(0, 0, 0)$ to $(1, 1, 1)$ for the $(1, 1, 1)$ direction, and so on. As with the axis-aligned scans, we search for solvent voxel sequences enclosed by occupied voxels and increment the voxel values accordingly when PSP events are detected.

After scanning in all seven directions (x, y, z, and the four cubic diagonals), each voxel will have a value between 0 (no PSP events detected) and 7 (points that are deeply buried inside a pocket). Note that occupied voxels (marked initially as -1) remain unchanged throughout the scanning process.

2.5 Pocket Identification and Filtering

We define a *pocket* as a contiguous region of grid points where each voxel has a minimum number of PSP events, `MIN_PSP`. We set this threshold to 3.

Pockets are clustered as follows:

1. We iterate over all voxels in the grid.

2. If a voxel has a PSP event count greater or equal to `MIN_PSP` and has not been visited yet, it becomes the start of a new pocket.

3. We check its six connected neighbors. These are voxels that share a face with the current voxel. If they satisfy the PSP threshold, we add them to the pocket.

4. We continue the process until all voxels with three or more PSP events are added to the list. Any grid points left with three or more PSP events will form one or more new pockets, and we repeat the process for them.

At the end of the clustering process, the pockets are returned as a list of lists, each containing the voxel coordinates belonging to a particular pocket. The list is sorted by pocket size, with the largest pockets appearing first.

Not all detected pockets are biologically relevant. Pockets with less than 40 voxels are discarded as they are too small. The remaining pockets are stored in a dictionary, where each key represents a pocket identifier, and the value is the list of voxel coordinates.

2.6 Identifying Surface Voxels

In this step, we determine which voxels lie on the surface of each detected pocket. A voxel is classified as a *surface voxel* if at least one of its six connected neighbors (i.e., voxels that share a face) is an occupied voxel, as defined in Section 2.2.1 (i.e., marked as -1 in the grid). We iterate through the voxels of each pocket and check their neighbors. If a neighboring voxel is occupied, we mark the original voxel as part of the surface. The program stores the surface voxels in a dictionary where the key is the pocket ID, and the value is a list of the corresponding surface voxel coordinates.

2.7 Identifying Surrounding Atoms and Residues

Once the surface voxels of each pocket have been determined, we proceed to identify the amino acids and atoms that are spatially close to the surface.

A protein atom is considered *near the pocket surface* if it lies within a distance r from any surface voxel center, where:

$$r = r_{\text{probe}} + a,$$

with $r_{\text{probe}} = 1.4$ Å, and a is the average of the van der Waals radii, computed from the atoms in the structure. Then:

1. For each pocket, we iterate over its surface voxels, and convert voxel indices to Cartesian coordinates (by computing the center of the voxel).

2. For each surface point, we iterate over all protein atoms and compute the Euclidean distance between the voxel center and the atom. If the distance is less than or equal to r, store the atom and its corresponding residue.

The result is a nested dictionary: the top-level key is the pocket ID. The value is another dictionary where: keys are (chain ID, residue ID) tuples.

Each value is a dictionary has:

1. name: the residue name (e.g., 'ARG')
2. atoms: a list of atoms from that residue involved in the pocket, sorted by atom ID.

The residues in each pocket are sorted by their residue ID for better readability.

2.8 Computing Pocket Properties

We compute several geometric properties for each detected and filtered pocket that help characterize its size, shape, and spatial position. These include:

1. Surface area: Approximated by multiplying the number of surface voxels by the area of a single voxel face.
2. Volume: Calculated by multiplying the total number of filled voxels in the pocket by the volume of a single voxel.
3. Depth: We compute the maximum distance from the filled pocket voxel to the nearest surface voxel. For each voxel in the filled pocket, compute the Euclidean distance to all voxels in the pocket surface and get the minimum. We store this data in a list. The depth is this list's maximum value, representing how deeply buried the pocket is.
4. Geometric center (centroid) of the pocket: Computed by averaging the Cartesian coordinates of all filled voxels: the x coordinate of the pocket center is the average of the x coordinates, and the same for y and z.

2.9 Final Pocket Selection

In this step, we generate the final list of predicted pockets. To ensure biological relevance, we apply an additional filter to retain only those pockets that are surrounded by at least 3 unique residues.

2.10 Output Generation and Visualization

After identifying the final set of pockets, the program prints a summary of the results to the console. It also generates:

- A text report (.txt file) containing pocket properties and residue-level information.
- A table summarizing the residues surrounding each pocket, including atom details.

- Chimera-compatible scripts for four different rendering modes to facilitate the visualization.

For details on the output files generated by the program and an execution example, refer to Section 4.

3 Evaluation of the results

3.1 Evaluation methodology

An evaluation is needed to asses the accuracy of our program in predicting ligand-binding sites. For this purpose, we used a dataset of 48 proteins, each with both bound and unbound structures.

This dataset was originally used to test and validate the LIGSITEcsc program (Huang & Schroeder, 2006)., which was then compared with other known pocket predictors that rely on geometric approaches. These include LIGSITEcs, LIGSITE, CAST, PASS, and SURFNET.

Initially, we attempted to recreate the validation method used by LIGSITEcsc, in which they take a distance-based approach. This method consists of finding the center point for each predicted pocket in the unbound proteins and calculating the distance between this point and the atoms of the ligand in the bound protein. For a prediction to be considered a "hit," the center atom of the predicted pocket must be within 4 Å of any atom in the ligand. The main problem we faced when implementing it was due to the fact that the unbound and bound structures tend to have slight conformational changes between them, making the center coordinates of the pockets predicted from the unbound protein unsuitable to use in the bound structure.

As a result, we modified the evaluation approach to something more practical and feasible to implement:

1. Use Chimera to identify which residues are within 4 Å of the ligand or ligands in the bound structures (Figure 2). A PDB file of the selected residues is obtained, from which we extract a list of the aminoacids nearby, excluding water molecules and other heteroatoms.

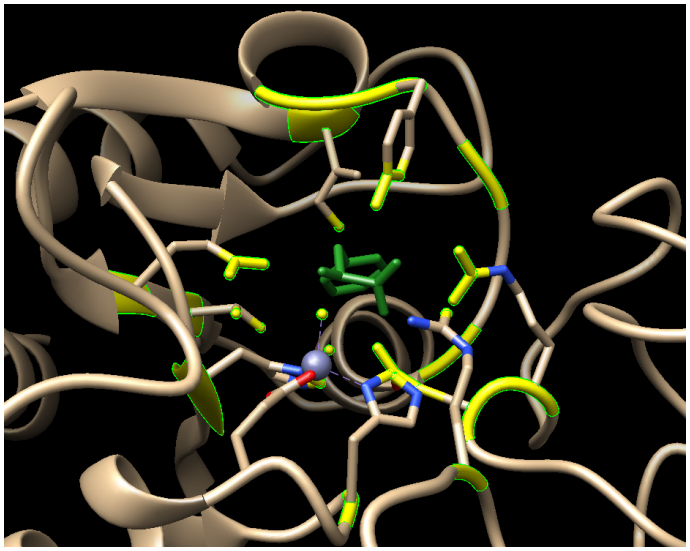

Figure 2: Chimera visualization of the 2ctc protein structure with the residues within 4 Å of the ligand (green) selected and colored yellow.

2. Run the program with the unbound protein and examine the residues obtained in the top 5 pockets. If at least 50% of the residues close to the ligand are found in a predicted pocket, we consider that prediction a hit (Figure 3).

Residues within 4 Å of the 2ctc ligand	Top 5 pockets predicted from the 2ctb unbound structure	% of matching residues
HIS69, ARG127, ASN144, ARG145, HIS196, ILE243, ILE247, TYR248, **ALA250, GLY253, THR268, GLU270**	Pocket1: **HIS69**, GLU72, **ARG127**, **ASN144**, **ARG145**, **HIS196**, SER199, **ALA250**, SER251, **GLY253**, SER254, ILE255, ASP256, **THR268**, **GLU270**	75
	Pocket2: VAL183, LYS184, GLY187, PHE189, LYS190, GLN261, GLY262, LYS264, THR304, LEU305, ASN307	0
	Pocket3: SER131, VAL132, THR133, SER134, TYR165, LYS168, TYR169, SER172, GLU173, VAL174	0
	Pocket4: ASP148, GLY150, PHE151, ALA170, ASN171, LYS184, TYR208, TRP257, ASN260, GLN261	0
	Pocket5: TYR204, PRO205, TYR206, THR210, GLN211, SER212, ILE213, LYS216, SER242, THR245, THR246	0

Figure 3: Table displaying the step 2 of the evaluation for the 2ctc/2ctb protein, where the residues within 4 Å of the ligand are compared to the top 5 predicted pockets and the percentage of the real pocket residues found in the predicted pocket is calculated. Figure created with Biorender.

We limited the evaluation to the top 5 pockets because with further testing we determined that these were the most significant pockets and were enough to assess whether the pocket prediction of that protein was successful. Multiple Python scripts were used to automatize

the process (these scripts can be found in our GitHub repository, see section Section 5).

3.2 How to run the evaluation

Step 1: Obtain all the bound PDB files

1. Create a new folder called `evaluate`
2. Download `batch_download.sh` from PDB official website
3. Create a text file with the liganded protein list, comma-separated: `list_of_pdbs.txt`

   ```
   1stp,2tmn,5p2p,2pk4,4dfr,1hyt,1fbp,3gch,1ivd,
   3mth,1cdo,4phv,1ida,1imb,6rsa,7cpa
   ```

4. Make the script executable: `chmod a+x batch_download.sh`
5. Run the command: `./batch_download.sh -f list_of_pdbs.txt -p`
6. Unzip the downloaded files: `gunzip *.pdb.gz`

Step 2: Process nearby residues with Chimera and extract residue list

1. Download `process_pdbs.py` from our GitHub repository
2. Place it in the folder containing all your PDBs (`evaluate`)
3. Make it executable: `chmod a+x process_pdbs.py`
4. Run the script with Chimera: `chimera --nogui --script "process_pdbs.py . ./nearby_residues 4.0"`
5. Check the output in `residue_summary.txt`

Step 3: Compare bound and unbound structures

1. Save the complex PDB files to a new folder, as they are no longer needed
2. Create a text file with the unliganded file list, comma-separated: `list_of_pdbs2.txt`
3. Run the command: `./batch_download.sh -f list_of_pdbs2.txt -p`
4. Unzip them: `gunzip *.pdb.gz`
5. Create a file `complex-unbound.txt` matching liganded proteins to their unliganded counterparts, like:

```
1stp        2rta
2tmn        1l3f
5p2p        3p2p
2pk4        1krn
...
```

6. Download `find_pockets.py` to the `evaluate` folder
7. Download `pocket_validator.py` from our GitHub repository
8. Run the validation:

```
python pocket_validator.py nearby_residues/residue_summary.txt
↪  complex-unbound.txt . ./hits_output
```

9. Check `pocket_detection_results.txt`

3.3 Evaluation results

Compared to the original approach, this modified evaluation continues to use the 4 Å distance parameter; however, instead of focusing on the pocket center, we examine the residues that constitute the pocket. To account for this difference, we required that at least 50% of the amino acids within 4 Å of the ligand have to be found in one of the predicted pockets. This modification may be more stringent than the original approach, so the percentage of hits obtained using our program might yield lower results than if it were evaluated with the original method.

From the evaluation of our program with the 48 proteins dataset we obtain that there is a success rate of 50%. Meaning 50% of the proteins have at least one pocket, of the top 5, with 50% of the residues within 4 Å from the ligand. In most cases, either pocket 1 or 2 ends up being the pocket with the higher matching percentage, and there are only 3 cases of pocket 4 or 5 being the best, although in those cases the percentages tends to be below 50%, meaning they are not a hit. For a more detailed view of the results, check Table 1.

Bound structure	**Unbound structure**	**Best Pocket (Top 5)**	**% Match**	**Hit**
2ctc	2ctb	1	75	1
1phd	1phc	1	65.52	1
1hew	1hel	1	45.45	0
1inc	1esa	None	0	0
1blh	1djb	None	0	0
1bid	3tms	1	83.3	1
1qpe	3lck	1	38.46	0
2sim	2sil	2	53.85	1
3ptb	3ptn	None	0	0

1byb	1bya	1	56.67	1
1rob	8rat	2	12.5	0
1mrg	1ahc	4	27.27	0
1mtw	2tga	1	12.5	0
1gca	1gcg	2	66.67	1
1stp	1swb	1	93.75	1
1pso	1psn	1	76.47	1
2h4n	2cba	2	40	0
1acj	1qif	2	63.64	1
1okm	4ca2	1	45.45	0
1a6w	1a6u	1	70	1
1apu	3app	1	55	1
2ifb	1ifb	1	50	1
1hfc	1cge	2	80	1
1dwd	1hxf	5	33.33	0
5cna	2ctv	None	0	0
1srf	1pts	1	75	1
1snc	1stn	1	36.36	0
1rbp	1brq	1	31.25	0
1pdz	1pdy	1	100	1
2ypi	1ypi	1	84.62	1
1rne	1bbs	2	60.87	1
1ulb	1ula	1	72.22	1
1stp	2rta	1	100	1
2tmn	1l3f	1	61.54	1
5p2p	3p2p	1	70.59	1
2pk4	1krn	None	0	0
4dfr	5dfr	1	56.25	1
1hyt	1npc	None	0	0
1fbp	2fbp	4	37.93	0
3gch	1chg	2	50	1
1ivd	1nna	1	4.76	0
3mth	6ins	2	16.67	0
1cdo	8adh	1	11.11	0
4phv	3phv	2	26.67	0
1ida	1hsi	1	42.86	0
1imb	1ime	1	80	1
6rsa	7rat	1	7.69	0
7cpa	5cpa	2	52.94	0
			Success rate	**50%**

Table 1: Results of Pocket Prediction using our algorithm and the obtained success rate. (Complete table with RMSD values, protein names, ligands, size and family can be found in our GitHub repository, see Section 5 for the link).

To assess whether the prediction success of the program is size-dependent —specifically, whether protein size predicts the probability of obtaining a hit— we performed a logistic regression analysis. The results indicated no significant relationship between protein length and the probability of a successful pocket prediction (Figure 4). The obtained coefficient

for protein length was of -0.001666, suggesting a slight decrease in the likelihood of success as protein size increases. However, this effect was not statistically significant, as indicated by a p-value of 0.215.

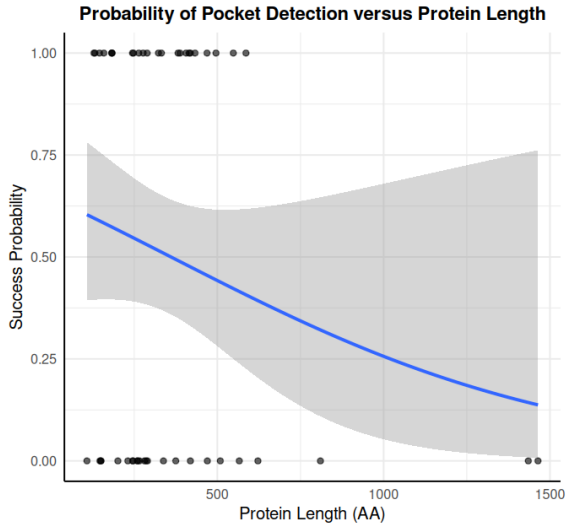

Figure 4: Logistic regression showing the relationship of protein length with the probability of obtaining a hit in our program. Figure created in R.

The prediction of pockets in the two biggest protein of the dataset, with a size of around 1500 AA, did not result in any hits. However, with the 48 protein dataset we don't have enough data to conclude whether there is a trend towards a worse pocket prediction in bigger proteins. A similar problem happened when checking the hit rate by family, the 48 protein datasets include a total of 32 different families so there aren't enough proteins of a single family to reach a specific conclusion. Our main hypothesis is that our geometry-based approach should be effective across a variety of protein families, particularly those with well-defined structural features, such as concavities or clefts, and minimal conformational changes which allows for an accurate prediction of the pocket in the absence of the ligand. Future studies should be made with a bigger dataset, with more well-defined families (Figure 5).

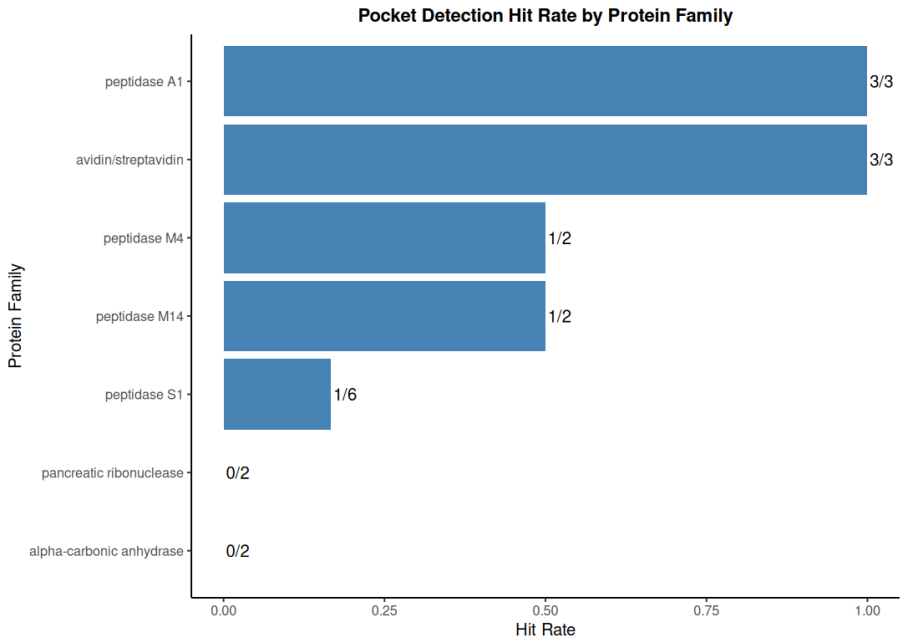

Figure 5: Hit rate by protein family of families that have at least 2 proteins in the 48-protein dataset. Figure created in R.

When comparing the success rate of pocket prediction of the different geometric-based programs, using the original evaluation approach, with the success rate of our program with the modified approach, we obtain a slightly lower percentage in our program. This happens both when checking the hits for only the top 1 and top 3 predictions. Generally LIGSITEcsc is the program with the better success rate followed by LIGSITEcs, PASS, LIGSITE, CAST, SURFNET and our program (see more details in Table 2). The lower success rate of our program may be attributed to reduced accuracy in pocket detection, but it could also stem from slight differences in the evaluation methodology, as we were unable to fully replicate the original approach.

Method	**Top 1 (%)**	**Top 3 (%)**
LIGSITEcsc	71	N/A
LIGSITEcs	60	77
LIGSITE	58	75
CAST	58	75
PASS	60	71
SURFNET	52	75
Our program	37.5	50

Table 2: Percentage of success rate for Top 1 and Top 3 predictions in the unbound State for each program. Figure obtained from the LIGSITEcsc paper (Huang & Schroeder, 2006).

4 Tutorial: Protein Pocket Detection Program

This section provides a comprehensive guide to using the protein pocket detection program, including input requirements, output descriptions, visualization methods, and advanced configuration options.

4.1 Program Usage Instructions

4.1.1 Input Requirements

```
python find_pockets.py <pdb_file> [output_directory]
```

PDB File Specifications The program accepts standard PDB (Protein Data Bank) files containing atomic coordinates of protein structures and explicitly handles multi-chain proteins. The PDB file must include valid ATOM records with the following fields:

- Columns 6–11: Atom serial number (integer)
- Columns 12–16: Atom name (e.g., CA for alpha-carbon)
- Columns 17–20: Residue name (e.g., ALA for alanine)
- Column 21: Chain identifier (single character)
- Columns 22–26: Residue sequence number (integer)
- Columns 30–54: X, Y, Z coordinates (floating-point values)

Structural Requirements:

- Ensure the PDB file contains no missing atoms in the region of interest
- Hydrogens are optional but will be included if present
- Non-standard residues (e.g., ligands) are ignored in this version

Output Directory Customization The optional `output_directory` argument allows users to:

- Specify a custom path for results storage (default: current working directory)
- Use absolute or relative directory paths

System Requirements:

- Output_directory is an optional parameter

4.1.2 Output Description

The program generates the following results:

Terminal Output

- Summary of detected pockets:
  - Number of pockets.
  - Properties for each pocket:
    - Volume (Å^3), surface area (Å^2), depth (Å), and center coordinates.
    - List of surrounding residues (e.g., THR31, TYR34).

Text Files

1. **pockets_report.txt**
   - **Detailed report including:**
     - Parameters used (voxel size, probe radius).
     - Van der Waals radii values.
     - Pocket-specific data (voxel count, residues, atoms).
   - **Residue Format:**
     - Residues now include chain identifiers (e.g., L-THR31, H-ARG350).
   - **Atom-Level Data:**
     - Specific atoms (e.g., OG1, NE1) are listed under each residue.
   - **Example entry:**
     - ```
       Parameters used for the prediction of pockets
       Voxel Size: 0.5 Å
       Probe Radius: 1.4 Å
       Minimum protein-solvent-protein events (PSP): 3
       Minimum voxel count per pocket: 30
       Van der Waals radii values per element:
       C: 1.7 Å
       H: 1.17 Å
       O: 1.4 Å
       N: 1.5 Å
       S: 1.85 Å
       Number of Detected Pockets: 10

       ==== POCKET 1 ====
       ```
       ```

```
Number of voxels in the pocket: 1083
Properties of the pocket:
Center: (8.15, 16.17, -0.82) Å
Surface area: 173.75 Å²
Volume: 135.38 Å³
Depth: 1.12 Å
Residues in the surface of the pocket (17): L-THR31...H-SER405
Atoms in the surface of the pocket:
THR31: Atom 202: O (13.9900,19.1780,1.9550)
...
SER405: Atom 1615: OG (6.8100,19.6610,-0.2770)
```

2. **pocket_residues.txt**

   Tabular data of atoms/residues adjacent to each pocket:

```
Pocket 1
 Chain   Residue   ResID  AtomID   Atom Name         X         Y         Z
-------------------------------------------------------------------------
     L       THR      31     202           O    13.990    19.178     1.955
```

Visualization Files

- Chimera-compatible scripts (located in the `chimera/` subdirectory).
- Color Coding: Each pocket is assigned a unique color (Pocket 1 = red, Pocket 2 = green, etc.). Labels are positioned vertically to avoid overlap.
  - Ball-and-stick visualization:
    - `bs_residues_pockets_visualization.cmd`: Highlights entire residues.
    - `bs_atoms_pockets_visualization.cmd`: Highlights individual atoms.
  - Surface visualization: The `surface_*.cmd` files add molecular surfaces to highlighted residues/atoms.
    - `surface_residues_pockets_visualization.cmd`: Surface representation of residues.
    - `surface_atoms_pockets_visualization.cmd`: Surface representation of atoms.

4.2 Example: PDB 1A6U

4.2.1 Command Execution

```
python find_pockets.py 1a6u.pdb ./results
```

4.2.2 Terminal Output Summary

```
=== Results Summary ===

Pocket 1
  Center: (8.153036934441458, 16.17219113573394, -0.8206334256694161) Å
  Surface area: 173.75 Å²
  Volume: 135.375 Å³
  Depth:  1.118033988749895 Å
  Nearby residues: 17
  Residues: THR31, TYR34, GLY52, THR53, ASN54, ASN55, GLY66, TRP93, SER95,
  ↪  ASN96, TRP98, ARG350, TYR399, TYR401, TYR402, GLY403, SER405
Pocket 2
  Center: (-3.059708333333357, 28.49579166666655, 3.306750000000046) Å
  Surface area: 77.0 Å²
  Volume: 60.0 Å³
  Depth:  1.0 Å
  Nearby residues: 9
  Residues: THR47, GLY48, PRO58, TYR332, ARG398, ASP400, TYR406, ASP408,
  ↪  TYR409
Pocket 3
  Center: (6.547331210191044, 8.423611464968117, 14.8144331210191) Å
  Surface area: 48.5 Å²
  Volume: 39.25 Å³
  Depth:  1.118033988749895 Å
  Nearby residues: 11
  Residues: ASN96, TRP98, PHE100, ARG343, LEU345, GLU346, TRP347, TYR360,
  ↪  ASN361, GLU362, LYS363
...
```

Key Observations

- Pocket 1 is the largest detected cavity, with a shallow depth (1.12 Å), likely a surface cleft.
- Hydrophilic residues (e.g., ASN54, SER95) dominate the surface, suggesting a polar binding site.
- Smaller pockets (e.g., Pocket 10: 5.25 Å^3) may represent transient solvent pockets.

4.3 Visualizations

4.3.1 Screenshot 1: Balls-and-Stick Residues

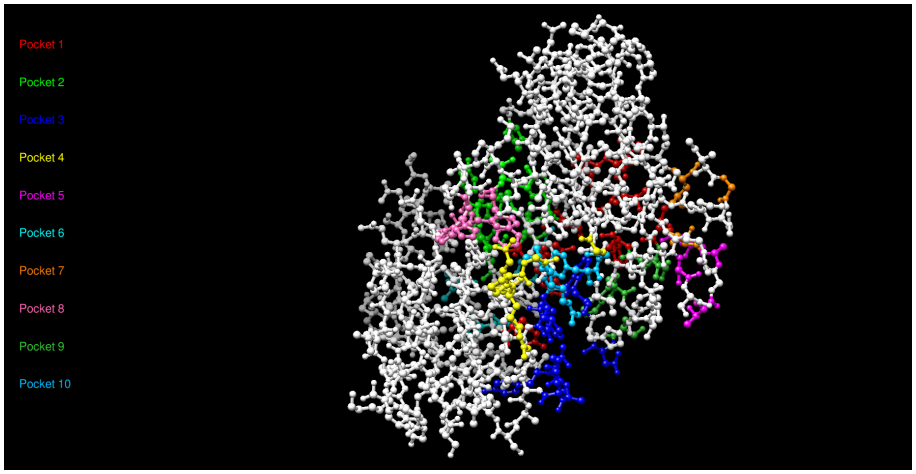

Figure 6: The Balls-and-Stick residues from the 1a6u.pdb file.

- Figure 6 shows highlighted residues (colored sticks) form the pocket surface. The protein backbone is shown in white.

4.3.2 Screenshot 2: Balls-and-Stick Atoms

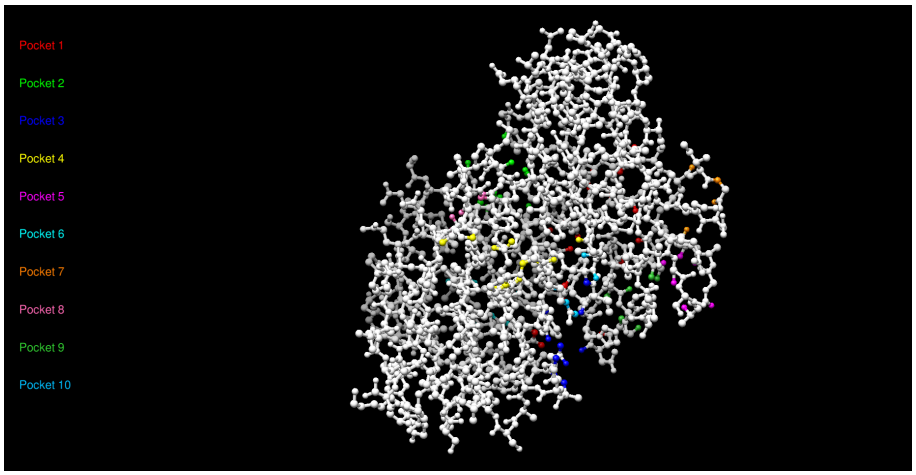

Figure 7: The Balls-and-Stick atoms from the 1a6u.pdb file.

- Figure 7 shows specific atoms (e.g., CA, CB) within residues are colored to show proximity to the pocket.

4.3.3 Screenshot 3: Surface Residues

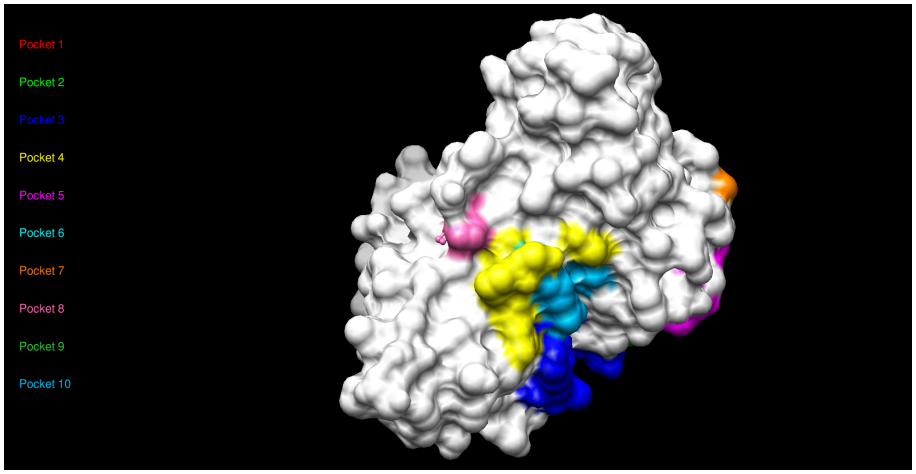

Figure 8: The surface residues from the 1a6u.pdb file.

- Figure 8 shows surface rendering of pocket-lining residues. Colors differentiate adjacent regions.

4.3.4 Screenshot 4: Surface Atoms

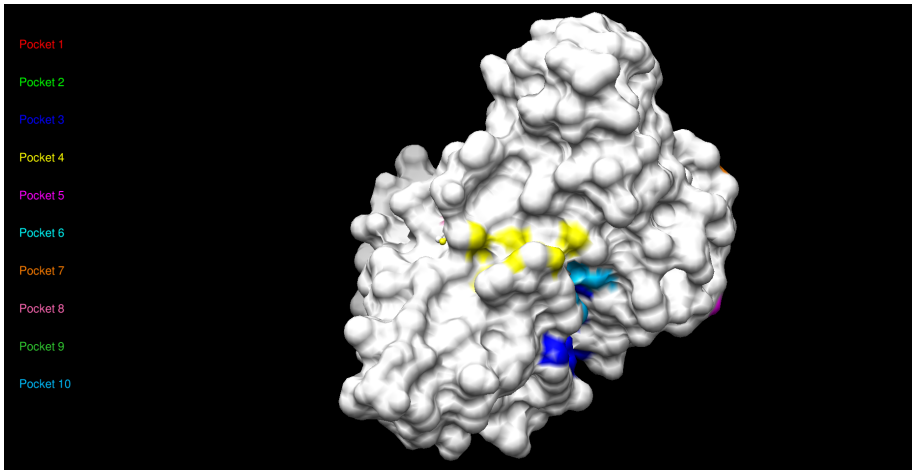

Figure 9: The surface atoms from the 1a6u.pdb file.

- Figure 9 shows atomic surface visualization showing pocket topology and depth.

4.4 How to Visualize Results in Chimera

4.4.1 Running Through the Application

1. Open Chimera.

2. Run `File > Open` and select one of the generated `.cmd` scripts.

3. Adjust the view using the toolbar (rotate, zoom).

4. Use `Tools > Depiction > 2D Labels` to toggle pocket labels.

4.4.2 Running Through the Terminal

```
chimera path/to/your_script.cmd
```

- Replace `path/to/your_script.cmd` with the actual path to your .cmd file.
- Example:

  ```
  chimera ./1a6u_output/chimera/bs_residues_pockets_visualization.cmd
  ```

4.5 Advanced Options

To detect smaller/shallower pockets, adjust these parameters in `run_complete_workflow()`, for example:

```
run_complete_workflow(
    file_path="1a6u.pdb",
    output_dir="./output"
    voxel_size=0.4,       # Higher resolution (slower)
    MIN_PSP=2,            # Lower threshold for noisy data
    probe_radius=1.2      # Smaller solvent probe
)
```

5 GitHub repository

The Python code, additional results table, scripts for the evaluation, R script for the statistical analysis and this report can be found at: https://github.com/mariatc025/find_pockets_jmx.

Bibliography

Connolly, M. L. (1983). Analytical molecular surface calculation. *Applied Crystallography*, *16*(5), 548–558.

Hendlich, M., Rippmann, F., & Barnickel, G. (1997). Ligsite: Automatic and efficient detection of potential small molecule-binding sites in proteins. *Journal of Molecular Graphics and Modelling*, *15*(6), 359–363.

Huang, B., & Schroeder, M. (2006). Ligsite csc: Predicting ligand binding sites using the connolly surface and degree of conservation. *BMC structural biology*, *6*, 1–11.

Levitt, D. G., & Banaszak, L. J. (1992). Pocket: A computer graphies method for identifying and displaying protein cavities and their surrounding amino acids. *Journal of molecular graphics*, *10*(4), 229–234.

Saberi Fathi, S. M., & Tuszynski, J. A. (2014). A simple method for finding a protein's ligand-binding pockets. *BMC Structural Biology*, *14*, 1–9.