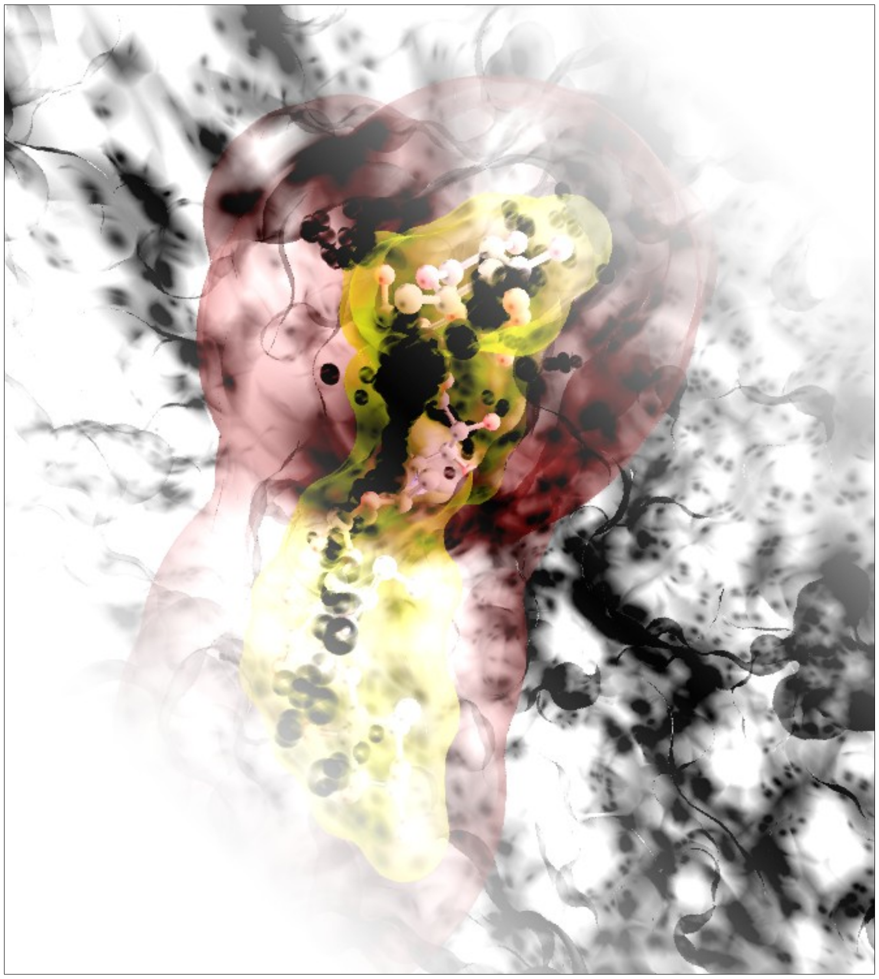
fpocket Users' Manual

version 1.0 March 24, 2008

authors : Vincent Le Guilloux[1] & Peter Schmidtke[2], supervisor : Pierre Tufféry[3]

fpocket is a protein pocket prediction algorithm. Given a PDB protein structure it enables the user to identify potent binding sites. Based on Voronoi tessellation, this algorithm is very fast and particularly well suited for large scale protein binding pocket screenings and development of scoring functions for binding pocket characterization.


acarbose binding site on alpha amylase (7taa).picture generated using VMD and tachyon rendering and GIMP post-processing.

1 ICOA – Chemoinformatics and Molecular modeling division – University of 'Orleans
2 MMB - Dept. Physical Chemistry – University of Barcelona
3 MTI - Inserm U973 - Université Paris Diderot

Notes

1. *This program uses output coming from Qhull. Qhull is currently not shipped with fpocket and has to be installed seperately. More information about Qhull can be found in the paper : Barber, C.B., Dobkin, D.P., and Huhdanpaa, H.T., "The Quickhull algorithm for convex hulls," ACM Trans. on Mathematical Software, 22(4):469-483, Dec 1996, http://www.qhull.org*

2. *Part of this software includes code based on external code developed by the Theoretical and Computational Biophysics Group in the Beckman Institute for Advanced Science and Technology at the University of Illinois at Urbana-Champaign. The PDB parser of the Molfile Plugin of VMD were modified for the purposes of fpocket's PDB parsing.*

3. *Within the whole documentation code and output from computer programs are represented and formatted in the following way :* **`ls -1 > out.txt`**

Contents

Introduction

Thanks for taking the time to read this official users' guide of *fpocket*. In this guide are presented general functionalities of the *fpocket* program and its derivatives, *dpocket* and *tpocket*. Yes, indeed fpocket is a package of three distinct programs, mentioned here before. fpocket is an acronym for “finding” pocket; dpocket is an acronym for “describing” pockets as it is for extraction of physico-chemical descriptors of pockets; tpocket is an acronym for “testing” pockets, as it is used for testing on a large scale scoring function for ranking protein cavities developped with fpocket, among each other.

This is not a usual guide. You can find here elements you can find in usual user guides, but we included several examples in the getting started section, which should enhance fast understanding of how to work with fpocket. The getting started guide can be understood like a mini tutorial of basic functionalities of this software.

License & Copyright

Contributions

This software was developed, validated, documented and distributed by Vincent Le Guilloux & Peter Schmidtke. Both, contributed equally to this project. The work was initiated and supervised by Pierre Tufféry.

Publication

The methods paper about this software was published in BMC Bioinformatics. In order to cite fpocket in the future, please cite this paper :

Vincent Le Guilloux, Peter Schmidtke and Pierre Tuffery, “Fpocket: An open source platform for ligand pocket detection”, BMC Bioinformatics 2009, 10:168

Installation

Prerequisites

Currently fpocket proposes two different ways for visualization of binding pockets. Both are based on commonly used molecular visualization tools : VMD[REF] and PyMol[REF]. In order to use visualization you need to install at least one of both softwares. Currently, visualization using VMD has better rendering and performances and visualization using PyMol better handling of binding pockets. You can download VMD for free from http://www.ks.uiuc.edu/Research/vmd/. PyMol can be freely downloaded from http://delsci.com/rel/099/.

Dependencies

fpocket relies on Qhull. In the officially released version fpocket ships Qhull with it and qhull compilation is automatically done when compiling and installing fpocket. Thus fpocket has no depencies that one should previously install in order to run the program.

System Requirements

fpocket is available for Linux/Unix type systems only. Thus fpocket works on Linux/Unix as well as Mac OSX workstations.

In order to run fpocket, you should have at minimum a Pentium III 500 Mhz with 128Mb of RAM. This program was co-developed and tested under the following Linux distributions : openSuse 10.3, Centos 5.2, Fedora Core 7, Ubuntu 8.10 as well as Mac OS X.

How to install fpocket

To install the full package, download the latest fpocket release from http://sourceforge.net/projects/fpocket. This should usually provide you a file like fpocket-src-1.0.tgz.

In order to install fpocket now, use the following series of commands in a command line.

```
tar -xzf fpocket-src-1.0.tgz
```

```
cd fpocket-src-1.0/
make
make test
```

If the make and make test command yield no errors, your installation can be completed by typing :

```
sudo make install
```

This last command only works if you have administrator rights.

For installing the supplementary data release, please refer to the INSTALL.txt file in you fpocket-src-1.0 directory.

Known Bugs

No known bugs exist for now. If you encounter any strange behavior, difficulty to install fpocket or a system specific bug, please contact the developers of this software and provide a bug report. You can find a template for a bug report in the main directory of the distribution. This template is called bugreport.txt.

For any question for support on fpocket please use the mailing list of fpocket : fpocket-support@lists.sourceforge.net

Getting Started

fpocket

To run the following examples, we use several sample input files provided with the package you have downloaded (situated in the (...)**`fpocket-1.0/sample/`** directory). Consequently, we suppose that the current directory is set to (...)**`fpocket-1.0/`**, or any other directory that would include this **`sample/`** directory and its content.

Example

Here is shown a very simple and straightforward example of how to run fpocket on a single PDB file downloaded from the RCSB PDB[REF]. The following command line will execute fpocket on the **`3LKF.pdb`** file situated in the **`sample`** directory.

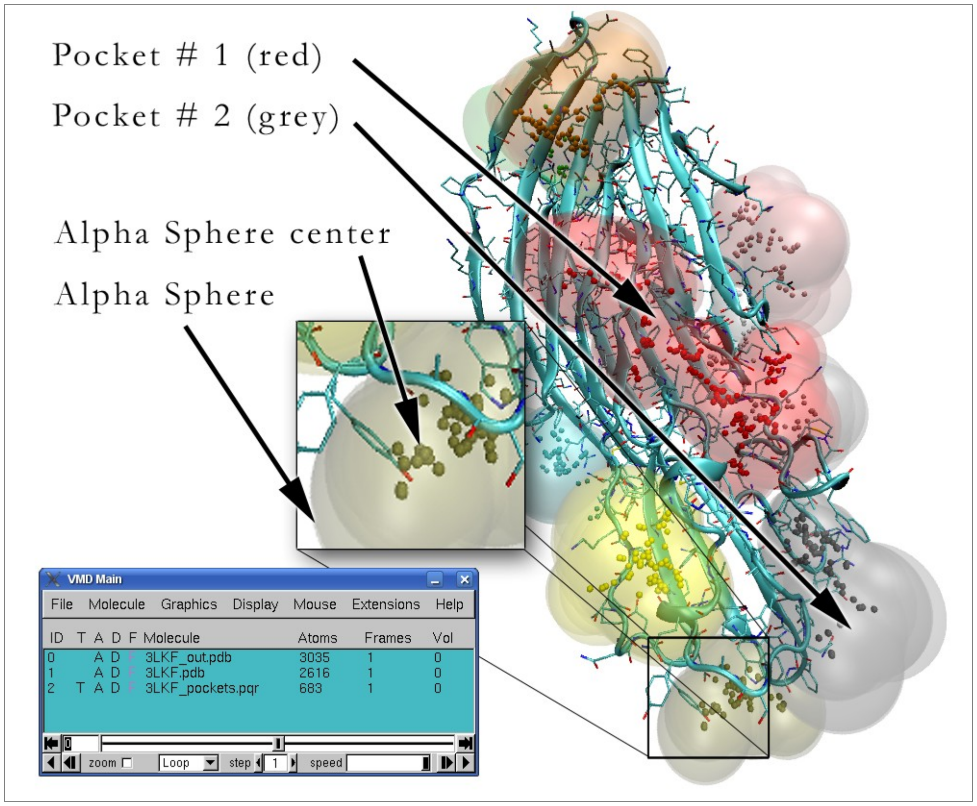
```
fpocket -f sample/3LKF.pdb
```

It is mandatory to give a PDB input file using the **`-f`** flag in command line. If nothing is given, fpocket prints the fpocket usage/help to the screen. fpocket will use standard parameters for the detection of cavities. Fore more information about these parameters see the Advanded features chapter – fpocket section (page 17).

If fpocket works properly the output on the screen should look like this :

```
=========== Pocket hunting begins ==========
=========== Pocket hunting ends ============
```

If you have a look now in the **`sample`** directory, you will notice that fpocket created a folder named **`3LKF_out/`**. This folder contains all the output from fpocket, so what you are actually interested in. If you just want to see rapidly the results, go to the **`3LKF_out`** directory and launch the **`3LKF_VMD.sh`** script. This script will launch the VMD molecular visualizer and charge the protein with binding site information coming from fpocket.

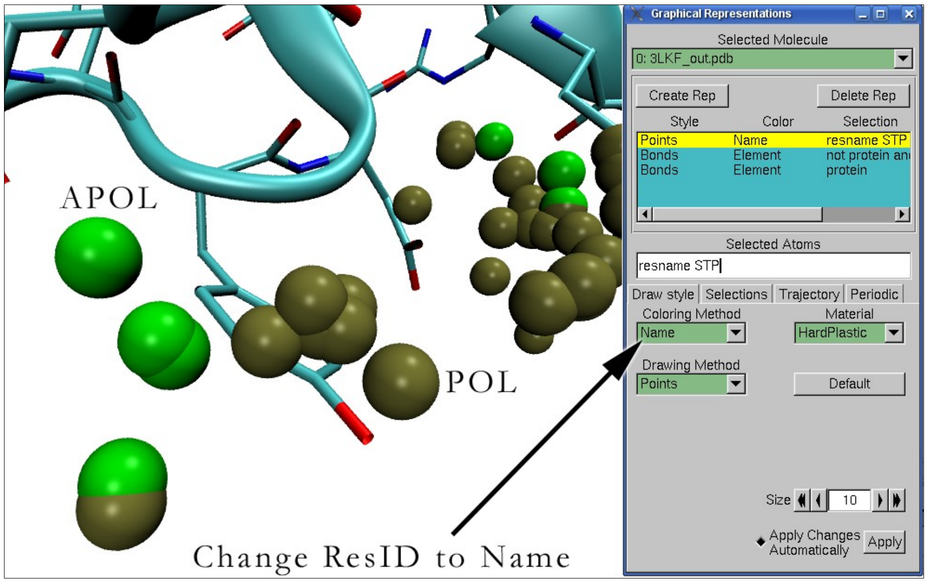


```
ID  T A D F Molecule            Atoms   Frames  Vol
0     A D   3LKF_out.pdb        3035    1       0
1     A D   3LKF.pdb            2616    1       0
2   T A D   3LKF_pockets.pqr    683     1       0
```

Illustration 1: Explanation of the fpocket VMD output

The illustration above is somehow what you will see if you launch the VMD script. Well, you will see this in less beautiful, but let us oversee the eye candy we have prepared here for you. VMD is well suited for representing both information, the volume of alpha spheres and their respective centers. Usually the visual volume information is not of primordial importance, as the larger alpha spheres tend to reach far out of the protein and smaller alpha spheres are not visible because they are recovered by larger ones. As it can be seen within the Main VMD window, the visualization script charges 3 structures, all of them are explained in more detail in the output section of this chapter.

If you had a closer look before on the methodological aspects of this algorithm (we invite you to read the paper) a natural question would be how to represent apolar and polar alpha spheres. Currently the color code represents only the residue ID (rank of the cavity). If you want to see characteristics of alpha spheres we invite you to change the representation of alpha spheres. This can be found by clicking Graphics -> Representations. Another window will show up. There you select the first molecule (3LKF_out.pdb), like represented on the figure below.



Selected Molecule: 0: 3LKF_out.pdb

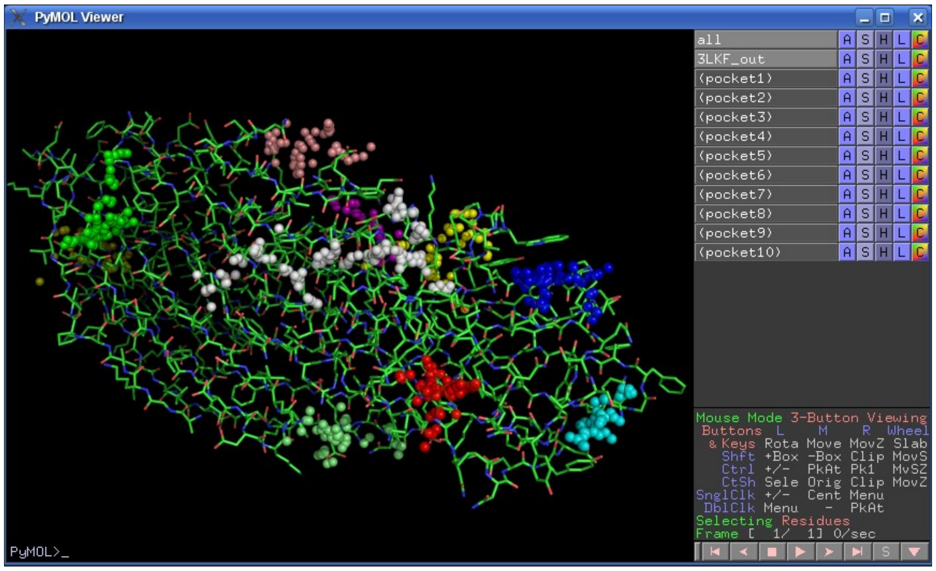
Style	Color	Selection
Points	Name	resname STP
Bonds	Element	not protein an
Bonds	Element	protein

Selected Atoms: resname STP

Coloring Method: Name
Material: HardPlastic
Drawing Method: Points
Size 10
Apply Changes Automatically

Illustration 2: Showing alpha sphere characteristics using VMD

A script for fast visualization using PyMOL is also provided. PyMOL provides nice features browsing and selecting different pockets, using the predefined selection patterns on the right side of the main window. However, PyMOL does not interpret well the pqr file format, so alpha sphere volumes are not accurate and only alpha sphere centers can be shown.



```
Mouse Mode 3-Button Viewing
Buttons  L    M    R  Wheel
 & Keys Rota Move MovZ Slab
   Shft +Box -Box Clip MovS
   Ctrl +/-  PkAt Pk1  MvSZ
   CtSh Sele Orig Clip MovZ
SnglClk +/-  Cent Menu
 DblClk Menu  -   PkAt
Selecting Residues
Frame [  1/   1] 0/sec
```

```
PyMOL>_
```

Illustration 3: fpocket PyMOL output

Basic input

Mandatory (1 OR 2):

1: flag -f : one standard PDB file name.

2: flag -F : one text file containing a simple list of pdb path

Optional:

For this see Advanced features chapter – fpocket section (page 17).

Output

Fpocket output is made of many files. To have a detailed overview of those files, see Advanced features chapter – fpocket section (page 17).

Is there something else? No, you have done. Congratulations, you have successfully performed your first cavity prediction with fpocket...without any accidents we hope. As you might have seen, usage of fpocket is rather simple, although it is command line based software (for now). Furthermore you should have seen that fpocket is very fast, well, lets say if you do not run a P1 100Mhz.

As mentioned before, fpocket provides much more possibilities especially for filtering out unwanted pockets, clustering of alpha spheres. For all these issues and usage of these more advanced features, refer to chapter Advanced features, section fpocket (page 17) of this manual.

dpocket

Until now you have seen what the majority of cavity detection algorithms can do. So a part from speed and hopefully prediction results, nothing distinguishes fpocket from other algorithms like ligsite, sitemap, sitefinder, pocketpicker, pass ...

This is just partially true, because the fpocket package contains dpocket. D is an acronym for describing. One purpose a cavity detection algorithm can be used for is the extraction of descriptors of the physico-chemical environment of the cavity. dpocket allows to do this in a very simple and straightforward way. As extracting binding pocket descriptors on only one protein would be somehow meaningless for studying pocket characteristics, dpocket enables analysis of multiple structures. So now, no longer scripting and automation is necessary to do these kind of things. But lets have a closer look using again a very simple example you can try on your workstation.

Example

Here we go. dpocket requires one single input file. This input file must be a text file containing the following information : 1 – the PDB file of the protein you want to analyze and 2 – the ID of the ligand you would like to have as reference in order to define an explicitly defined binding pocket. The file used in this example (**`sample/test_dpocket.txt)`** looks like this :

```
data/3LKF.pdb   pc1
data/1ATP.pdb   atp
data/7TAA.pdb   abc
```

Here we analyze three pdb files. Note that the ligand name should be separated by a tabulation from the pdb file name. You can launch dpocket on this sample file using the following command :

```
dpocket -f sample/test_dpocket.txt
```

dpocket will yield 3 results files in the current directory. These files will be by default :

```
dpout_explicitp.txt
dpout_fpocketnp.txt
dpout_fpocketp.txt
```

If you want to change naming of these files, use the -o flag in command line to define a new prefix for the fpocket output files, for example **`my_test`** as prefix would yield **`my_test_explicitp.txt`**. The three output files contain the in fpocket implemented pocket descriptors for each binding pocket found by fpocket :

`*_fpocketp.txt`, describes all binding pockets found by fpocket that match one of the detection criteria. In other word, fpocket found several pocket in the protein, and this file will contain descriptors of pocket that are considered to be the binding pocket using some detection criteria.

`*_fpocketnp.txt`, describes on the contrary all pockets found by fpocket that are not found to be the actual pocket using the detection criteria.

`*_explicitp.txt,` describes the pockets explicitely defined. By explicitely defined here, we mean that the pocket will be defined as all vertices/atoms situated at a given distance of the ligand (4A by default), regardless of what fpocket found during the algorithm.

The ouput files are tab separated ASCII text files that are easy to parse using statistical software such as R. Thus statistical analysis of pocket descriptors becomes a very straightforward and easy process. Basically, the two first files might be used to establish a new scoring function as they describes what fpocket finds, while the last file could be used for a more detailed and accurate analysis of the exact part of the protein that interact with the ligand.

For more details of the output refer to the output section below, or to dpockets Advanced features section (page 21).

Basic input

Mandatory:

1: flag -f : a dpocket input file, this file has to contain the path to the PDB file, as well as the residuename of the reference ligand, separated by tabulation.

Optional:

1 : flag -o : the prefix you want to give to dpocket output files

dpocket offers much more optional parameters in order to guide the pocket detection. For this see Advanced features chapter – dpocket section (page 21).

Output

For more details of the output refer to the output section below, or to dpockets Advanced features section (page 21).

In conclusion of this first very easy dpocket run, you can see that you have a very fast and reliable tool to extract pocket descriptors, of binding pockets and “non binding pockets” on a large scale level. These descriptor files provide an excellent tool for further statistical analysis and model building, which leads immediately to your wish to write a new scoring function for ranking cavities using the different descriptors. Well, fpocket, dpocket and tpocket are very useful tools to do exactly this! So go ahead. Lets suppose you have passed several thousands of PDB files and analyzed statistically the significance of all descriptors. You have set up a new scoring function. Now you have an external test set of PDB files you haven't tested. How can you evaluate your scoring function? This is actually also a very easy task, using tpocket.

tpocket

As already mentioned in the preceding paragraph, tpocket can be used in order to evaluate rapidly cavity scoring functions. If you are for example in the pharmaceutical industry and you want to set up the ultimate drugability prediction score, you might be able to do this with fpocket and dpocket. Afterwards you can actually test your method using tpocket. T is an acronym for testing, here.

Something fancy we did not tell you about before is, that you can also test your scoring function on apo structures using tpocket. The only requirement is the need to align holo and apo structure to obtain a space correspondance between apo and holo pocket. But lets explain this with an example. Of course, testing a holo dataset is even more easy, you just need to provide the resname of the ligand and tpocket will do the rest.

Example – tpocket on apo structures

If you had a look in the publication of fpocket, you might have seen that the algorithm was validated on a dataset of 48 proteins previously used to evaluate several pocket detection algorithms. As fpocket programmers are, by definition, very nice people, they have included this data set (holo and aligned apo structures) in the distribution of fpocket, released as fpocket-1.0-data. So let us use this set as example here. When you extract the dataset in your folder you should have a data folder containing among others two files, **`pp_apo-t.txt`** and **`pp_cplx-t.txt`**. The first file is a tpocket input file in order to assess the capacity of the scoring function to rank correctly known binding sites on apo structures. The second file is also a tpocket inputfile, but this time for known binding sites on holo structures. Here is a part of **`pp_apo-t.txt`** :

```
data/pp_data/unbound/1QIF-1ACJ.pdb      data/pp_data/complex/1ACJ.pdb   tha
data/pp_data/unbound/3APP-1APU.pdb      data/pp_data/complex/1APU.pdb   iva
data/pp_data/unbound/1HSI-1IDA.pdb      data/pp_data/complex/1IDA.pdb   qnd
data/pp_data/unbound/1PSN-1PSO.pdb      data/pp_data/complex/1PSO.pdb   iva
```

```
data/pp_data/unbound/1L3F-2TMN.pdb      data/pp_data/complex/2TMN.pdb   po3
data/pp_data/unbound/3TMS-1BID.pdb      data/pp_data/complex/1BID.pdb   UMP
data/pp_data/unbound/8ADH-1CDO.pdb      data/pp_data/complex/1CDO.pdb   NAD
data/pp_data/unbound/1HXF-1DWD.pdb      data/pp_data/complex/1DWD.pdb   MID
```

Here the first column contains the path to the apo structure, aligned to the holo structure, which is given in the second column. Using a holo dataset, the first and the second column would be the same. The third column indicates the PDB HETATM code of the ligand in the holo structure that is situated in the binding site.

You can use this file to run tpocket using the following command line :

```
tpocket -L data/pp_apo-t.txt
```

Let us continue with the more interesting case, the first example, with a lot of structures. After some time of calculation, tpocket will provide two standard output files. The moment has come, you will finally know if you discovered the ultimate method of drugability prediction, or sugar binding site prediction or whatever. The first file is called by default **stats_g.txt**. It contains global statistics about the prediction using all evaluation criterias available in tpocket, so for example how many binding sites you found among the 3 first ranked cavities. For representational purposes only the first of the six tables available in this file is depicted hereafter :

```
     Ratio of good predictions (dist = 4A)
------------------------------------------
Rank <=  1  :           0.69
Rank <=  2  :           0.83
Rank <=  3  :           0.94
Rank <=  4  :           0.94
Rank <=  5  :           0.94
Rank <=  6  :           0.94
Rank <=  7  :           0.94
Rank <=  8  :           0.94
Rank <=  9  :           0.94
Rank <= 10  :           0.94
-------------------------------------
Mean distance          : 2.924573
Mean relative overlap  : 39.373226
```

This table schedules the capacity of your scoring function to identify the binding sites of the 48 apo structures using the criteria published within [REF]. Not represented here, tpocket provides two other, maybe more accurate, measures for a correctly identified binding site. These measures are explained in more detail in the Advanced features – tpocket - correctly identified binding site section (page), as they can be a bit more tricky.

The second output file provides more accurate statistics about each structure analyzed. This file, called **stats_p.txt** enables the user to analyze more closely why scoring might not work well on a specific structure. Here is an extract of the first columns and lines of this file :

```
LIG | COMPLEXE | APO | NB_PCK | OVLP1 | OVLP2 | DIST_CM | POS1 | POS2 | POS3
```

```
THA 1ACJ.pdb 1QIF-1ACJ.pdb    22   79.31   78.33    0.00    1    1    0
IVA 1APU.pdb 3APP-1APU.pdb     4    0.00    0.00    3.43    0    0    1
QND 1IDA.pdb 1HSI-1IDA.pdb     4   82.69   81.65    3.19    1    1    1
IVA 1PSO.pdb 1PSN-1PSO.pdb     9   80.00   51.38    3.49    1    1    1
PO3 2TMN.pdb 1L3F-2TMN.pdb    10   58.33   72.00    2.69    1    1    1
UMP 1BID.pdb 3TMS-1BID.pdb    15   63.64   60.78    3.52    1    1    1
NAD 1CDO.pdb 8ADH-1CDO.pdb    18    0.00    0.00    3.41    0    0    1
MID 1DWD.pdb 1HXF-1DWD.pdb    10   93.48   81.37    3.86    1    1    1
```

Using this output you have a detailed view of what worked and what did not worked for all criteria. For instance, in this example, fpocket detects well all apo binding sites a part from the first one using the PocketPicker criterion for binding site identification (DIST_CM). POS3 corresponds to the rank of the cavity using the scoring function of fpocket. You have further information about the number of pockets per protein and the exact overlap with the actual pocket.

Now if you want to assess your scoring function on holo structures, you also can use tpocket. This time you only have to provide the pp_cplx.txt, also provided within the distribution. As you can see, this file is very similar to pp_apo.txt. Only the first column repeats the path to the complex structure like this :

```
data/pp_data/complex/1acj.pdb   data/pp_data/complex/1acj.pdb   tha
data/pp_data/complex/1apu.pdb   data/pp_data/complex/1apu.pdb   iva
data/pp_data/complex/1ida.pdb   data/pp_data/complex/1ida.pdb   qnd
data/pp_data/complex/1pso.pdb   data/pp_data/complex/1pso.pdb   iva
data/pp_data/complex/2tmn.pdb   data/pp_data/complex/2tmn.pdb   po3
data/pp_data/complex/1bid.pdb   data/pp_data/complex/1bid.pdb   ump
data/pp_data/complex/1cdo.pdb   data/pp_data/complex/1cdo.pdb   nad
```

Input

Mandatory:

1: flag -L : a tpocket input file, this file has to contain the paths to the PDB files (apo, holo or holo,holo if you want to test fpocket only on holo structures), as well as the residuename of the reference ligand, separated by tabulation.

Optional:

1 : flag -o : the prefix you want to give to tpocket detailed statistics

2 : flag -e : the prefix you want to give to tpocket general statistics

tpocket offers much more optional parameters in order to guide the pocket detection. For this see Advanced features chapter – tpocket section (page).

Output

Using standard parameters on the example tpocket list given in the example paragraph above, tpocket returns two output files :

- **`stats_p.txt`** : This file contains the detailed statistics of tpocket. The name and the ligand of the analyzed PDB structure are repeated, as well as the exact overlap of the fpocket identified binding pocket with the actual binding pocket (identified with the help of the ligand, called OVLP here). You will see two different overlaps in the output. For further informations refer to advanced features on page . Furthermore, the distance criterion used in the Chemistry Central Journal paper for publication of PocketPicker was used (DIST_CM). Next, you can also have exact information about the rank of the cavity using the fpocket scoring function.
- **`sats_g.txt`** : Second, tpocket provides more general statistics about pocket identification on the dataset provided. For both overlap criterions the ranking performance (the capacity of the fpocket scoring to rank correctly a binding site having a certain minimum overlap with the actual binding site) is printed into this file. Furthermore the distance criterion published in [REF] is also evaluated. Thus, statistics in this file gives you a rapid overview over the global performance of your method.

Summarizing features of tpocket, one could retain, that tpocket is a very fast way to test fpockets performance on your own dataset and test your own scoring functions for ranking purposes of identified binding sites.

You have finished the Getting started section. We hope that you notice the usefulness of this package of programs for the research of new features, descriptors and scoring functions in the binding site identification field. Well, this was only a very fast overview over the very basic features of fpocket, dpocket and tpocket. If you want to dive into development of your own pocket descriptors and scoring functions, or if you want to change the pocket detection parameters for your purposes, continue with the Advanced features section, next.

Advanced features

You want to know more about fpocket? This is the section for you, here we tried to compile in a (we hope) comprehensive manner the most important details of fpocket, dpocket and tpocket, to which you have access by command line. It is primordial to know, that fpockets performance was assessed and scoring function was established for standard parameters. The performance of pocket detection and scoring is highly dependent on these parameters, so keep in mind that you might have to adapt scoring to your specific problem.

Note that this section does not provide too much information about the theoretical background of the way fpocket works. In order to learn more about this read the Materials & Methods of the freely available paper [REF] on the BMC Bioinformatics website. Nevertheless, we tried to keep it as clear as possible, using some application examples.

fpocket

Input command line arguments

Mandatory:

The simplest way to run fpocket is either by providing a single pdb file, or by providing a list of pdb file, stored in a simple text file. You will need one of these two input to run fpocket:

- **`-f`** : one standard PDB file that you want to analyze with fpocket
- **`-F`** : a simple list of pdb files.

Optional:

- **`-m`** : (default 3Å) This flag enables the user to modify the minimum radius an alpha sphere might have in a binding pocket. An alpha sphere is a contact sphere, that touches 4 atoms in 3D space without having any internal atoms. Here 3Å allow filtering of too small (protein internal) alpha spheres. I you want to analyze internal interstices, lower this parameter. In the contrary, if you want to analyze more solvent exposed cavities, you can raise this parameter in order to filter out too buried cavities.
- **`-M`** : (default 6Å) Here you can modify the maximum radius of alpha spheres in a pocket. An alpha sphere is a contact sphere, that touches 4 atoms in 3D space without having any internal atoms. Here 7Å allow to filter out too large contact spheres, that are lying on the protein surface. If you want to analyze very flat and solvent exposed surface depressions, raise this parameter. For analysis of buried parts of the protein you can lower this parameter. Higher radii might be more interesting for identification of protein protein binding sites or

polysaccharide binding sites. Smaller radii enable detection of buried cavities for small organic molecules (drugs, for instance).

- **`-i`** : (default 35) This flag indicates how many alpha spheres a pocket must contain at least in order to figure in the results provided by fpocket. This parameter enables filtering of too small cavities. Thus, if you want to analyze smaller cavities also, lower this parameter, if you are only interested in huge cavities, like NADP binding sites, you can raise it in order to retain only very few pockets in the end. To give you an idea, a rather big cavity, like a NADP binding site, can have hundreds of alpha spheres. Thus, 30 as standard parameter enables also to keep smaller binding sites.
- **`-A`** : (default 3) Fpocket distinguishes between two types of alpha spheres. Polar alpha spheres and apolar alpha spheres. This flag ranges from 0 to 4 and modifies the definition of the alpha sphere type. By default, an alpha sphere contacting at least 3 apolar atoms (having an electronegativity below 2.8) is considered as apolar. If this is not the case it is considered as polar.
- **`-D`** : (default 1.73Å) fpocket is based on Qhull. Basically fpocket submits a set of points (atom positions of the protein) to Qhull and Qhull returns a set of voronoi vertices and edges and connectivity information. fpocket performs, after a first filtering of dumb alpha spheres, a first alpha sphere clustering step. Later clustered alpha spheres will define a pocket. This parameter here enables the user to modify the first clustering step. fpocket seeks alpha spheres that are at most at 1.73Å distance from the current alpha sphere and connected to it by a Voronoi edge. This clustering step will create small, very local clusters. If you want to decrease the size of these clusters of alpha spheres, decrease this parameter. You will have as result a lot of small pockets (do not forget to modify -i in order to see very small pockets). If you want to cluster more generously to already larger pockets in the first step, increase this parameter.
- **`-r`** : (default 4.5Å) This parameter influences the second clustering step of small pockets to larger pockets. For each small initial pocket (alpha sphere cluster) the center of mass is calculated. Next, all pockets that have their centers of mass at most at a distance of 4.5Å are clustered together and form a bigger pocket. Similarly to -D, if you want to decrease the size of the pocket, decrease this parameter. In the contrary if you want to have larger cavities as result, increase this parameter.
- **`-s`** : (default 2.5Å) The last clustering step is a multiple linkage clustering. Here every pocket is checked for having at least n alpha spheres at a maximum of 2.5Å from alpha spheres of another pocket. Increasing this distance will yield huger pockets and descreasing smaller pockets. The parameter n can also be modified using the following flag.
- **`-n`** : (default 2) The number of alpha spheres a pocket has to have close to alpha spheres of another pocket in order to be clustered together in the last clustering step. Be careful, this clustering depends also on the distance criterion (-s flag). If you want to well distinguish surface cavities separated by some small barriers of the protein surface you can increase this parameter or leave it like that. In the contrary if you want to detect larger binding site, that might even bridge of surface protrusions set this parameter to 1 or 2.
- **`-p`** : (default 0.0) This is another parameter for filtering unwanted pockets. It defines the maximum ratio of apolar alpha spheres and the number of alpha spheres in a pocket in order to keep the pocket in the results list. That is to say, by default every pocket is kept

(0.0). Now, if you would like to filter rather hydrophobic pockets, raise this parameter and very polar cavities will be filtered out. This parameter is a ratio, not a percentage, thus it ranges from 0 to 1.

- **`-v`** : (default 2500) By default, pockets volume are calculated using a monte-carlo algorithm. Basically, the algorithm pick a random point in the space and check if it is included in any alpha sphere, and store this status. This is repeated N times, and we estimate the volume of the pocket using ratio between the number of hit and the number of iteration, scaled by the size of the box. This parameter defines the number of iteration to perform. Of course, the higher the value is, the greater the accuracy will be, but the performance will be slowed down.
- **`-b`** : (NOT USED BY DEFAULT) This option allows the user to chose a discrete algorithm to calculate the volume of each pocket instead of the Monte Carlo method. This algorithm put each pocket into a grid of dimention (1/N*X ; 1/N*Y ; 1/N*Z), N being the value given using this option, and X, Y and Z being the box dimensions, determined using coordinates of vertices. Then, a triple iteration on each dimensions is used to estimate the volume, checking if each points given by the iteration is in one of the pocket's vertices. This parameter defines the grid discretization. If this parameter is used, this algorithm will be used instead of the Monte Carlo algorithm.

  Warning: Although this algorithm could be more accurate, a high value might dramatically slow down the program, as this algorithm has a maximum complexity of N*N*N*nb_vertices, and a minimum of N*N*N !!!

Output files description

fpocket yields output directly in the directory of the data file, creating a directory using the name of the PDB file followed bu the _out extension. Here, the command **`ll sample/3LKF_out o`**f the current sample run would look something like this:

```
total 332
-rw-r--r-- 1 peter users    769 Nov 29 00:14 3LKF.pml
-rw-r--r-- 1 peter users    698 Nov 29 00:14 3LKF.tcl
-rwxr-xr-x 1 peter users     30 Nov 29 00:14 3LKF_PYMOL.sh
-rwxr-xr-x 1 peter users     41 Nov 29 00:14 3LKF_VMD.sh
-rw-r--r-- 1 peter users 245835 Nov 29 00:14 3LKF_out.pdb
-rw-r--r-- 1 peter users   6725 Nov 29 00:14 3LKF_pockets.info
-rw-r--r-- 1 peter users  49355 Nov 29 00:14 3LKF_pockets.pqr
drwxr-xr-x 2 peter users   4096 Nov 29 00:14 pockets
```

As you can see, fpocket provides a lot of files and another subdirectory. However, majority of these files are necessary for easy visualization of binding pockets. Lets explain the content and utility of each file :

- **`3LKF.pml`** : this is a PyMOL script for visualization of binding pockets using PyMOL
- **`3LKF.tcl`** : this a tcl script for visualization of binding pockets using VMD
- **`3LKF_PYMOL.sh`** : this is the executable script to launch fast visualization using PYMOL

- **`3LKF_VMD.sh`** : this is the executable script to launch fast visualization using VMD
- **`3LKF_out.pdb`** : this is the most important file, it contains the initial PDB structure given as input. Non cofactor HETATM occurrences will be stripped off in this file compared to the original PDB input file. The PDB file contains centers of alpha spheres using the HETATM definition as dummy atoms. These alpha sphere centers are attached in the end of the PDB file, using the STP residue name (for site point). Apolar alpha spheres carry the atom name APOL, polar alpha spheres the atom name POL. Pockets are sets of alpha spheres. They can be distinguished by residue number. Thus residue STP 1 would be the first binding pocket according to fpocket. To show this more clearly here is an extract of the **`3LKF_out.pdb`** :

```
ATOM   2349   CD LYS A 299       9.679  16.827 105.636  1.00 19.91           C
ATOM   2350   CE LYS A 299      10.371  16.314 104.370  1.00 25.17           C
ATOM   2351   NZ LYS A 299      11.749  15.794 104.597  1.00 32.36           N
ATOM   2352  OXT LYS A 299       5.240  20.009 107.670  1.00 16.06           O
HETATM 2736  POL STP C   1      18.291  37.420  83.622  0.00  0.00          Ve
HETATM 2756  POL STP C   1      18.445  37.638  83.606  0.00  0.00          Ve
HETATM 3208  POL STP C   1      18.325  37.403  83.631  0.00  0.00          Ve
HETATM 3208  POL STP C   1      18.450  37.618  83.610  0.00  0.00          Ve
```

- **`3LKF_pockets.pqr`** : This file contains all alpha sphere centers, as the 3LKF_out.pdb file, but contains no information about the protein structure. Furthermore using the pqr format enables writing of the van der Waals radius of atoms explicitely in this file. Here this possibility was used to output the radii of alpha spheres of a pocket. Charging this pqr file, one can analyze more precisely the volume recognized by fpocket. Note that, currently only VMD supports reading this format correctly. PyMOL is able to read pqr file, but does not interpret van der Waals radii.
- **`pockets/`** : Well, again a subdirectory. But I promise, it's the last one. For development purposes or easy analysis, fpocket proposes this directory which contains according to the current example :

```
pocket0_atm.pdb   pocket2_vert.pqr  pocket5_atm.pdb   pocket7_vert.pqr
pocket0_vert.pqr  pocket3_atm.pdb   pocket5_vert.pqr  pocket8_atm.pdb
pocket1_atm.pdb   pocket3_vert.pqr  pocket6_atm.pdb   pocket8_vert.pqr
pocket1_vert.pqr  pocket4_atm.pdb   pocket6_vert.pqr  pocket9_atm.pdb
pocket2_atm.pdb   pocket4_vert.pqr  pocket7_atm.pdb   pocket9_vert.pqr
```

The **`*_atm.pdb`** files contain only the atoms contacted by alpha spheres in the given pocket. Complementary to this information, **`*_vert.pqr`** files contain only the centers and radii of alpha spheres within the respective pocket. As extensions mention, atoms are output in the PDB file format and alpha sphere centers in the PQR file format.

dpocket

Input command line arguments

Mandatory:

- **`-f :`** a dpocket input file, this file has to contain the path to the PDB file, as well as the residuename (PDB HET residue tag, like "**`hem`**", for heme) of the reference ligand, separated by a tabulation. See the Getting started section for an example of such a file (page 11).

Optional:

- **`-o :`** (default **`dpout`**) the prefix you want to give to dpocket output files. The standard will produce three output files named dpout_fpocketnp.txt, dpout_fpocketp.txt, dpout_explicitp.txt.
- **`-e`** : Use the first explicit interface definition (default): we define the explicit pocket as being all atoms contacted by alpha spheres situated at a distance of d A° from any ligand atom.
- **`-E`** : Use the second explicit interface definition: we define the explicit pocket as being all atoms situated at a distance of d A° from any ligand atom.
- **`-d`** : The distance criteria used for the explicit pocket definition.

Last, all optional parameters used by fpocket are also accessible on command line through dpocket. Refer to the preceding paragraph to see details about fpocket parameters.

Output files description

As shown in the example, dpocket creates 3 output files. Lets describe them a bit more in detail here :

- **`dpout_explicitp.txt`** : This file contains all pocket descriptors implemented in fpocket of the explicitly defined binding pocket. What does this mean, explicitly? In the input you have associated a ligand identification to each PDB file. This ligand is used by fpocket in order to identify the actual binding pocket. If you want to know more about this process, refer to the Advanced features section of dpocket (page 21). Now let us have a look was is actually in this file.

```
pdb ligand overlap lig_vol pocket_vol nb_alpha_spheres mean_asph_ray
data/3LKF.pdb PC 100.00   132.90  1678.64    29  3.94
data/1ATP.pdb ATP 100.00   322.62  2127.53    65  3.59
data/7TAA.pdb ABC 100.00   608.66  4977.48    97  4.20
```

Note that this is only an extract of this file. It contains a lot of columns (descriptors) that are not represented here. The first line describes the nature of the entry. The next line recapitulates the pdb structure analyzed (**`data/3LKF.pdb`**), the ligand used as reference (**`PC`**). Next the overlap between the actual and found binding pocket is shown, here 100% as this is an explicitly defined binding pocket. The next entries can be used as descriptors, like the ligand volume, the pocket volume, the number of alpha spheres in the binding pocket, the mean alpha sphere radius ... For a complete list of all implemented descriptors in fpocket, refer to the Advanced features – Pocket descriptors section (page 24).

The volumes calculated here are not accurate at all. If you want to calculate accurate volumes you have to change parameters for volume calculation. As volume calculations are generally over-estimated using alpha sphere approaches, especially for open binding pockets, this calculation is made available, but uses the minimum sampling for the calculation. For more accurate calculation significantly more calculation time would be necessary.

- **`dpout_fpocketnp.txt`** : This file contains the same kind of descriptors as the preceding one, but this time for pockets identified by fpocket, that are "non binding pockets". Non binding pockets means here, that the pockets do not correspond to the pocket where the reference ligand binds. Be careful, this does not necessarily mean that other pockets do not bind anything.
- **`dpout_fpocketp.txt`** : The last file is also formated the same way as the preceding both. This file contains the binding pocket, this time identified by fpocket and not explicitly by the ligand.

tpocket

This program of the fpocket package is certainly very useful for testing new scoring methods rapidly on a large dataset of protein ligand complexes. However one might encounter difficulties to understand results, interest, advantages and drawbacks of this methodology. In order to facilitate your understanding of this package we provide some more fundamental information first, before treating more practical questions about tpocket.

Input command line arguments

Mandatory:

- **`-L`** : a tpocket input file. This file has to contain the paths to the PDB files (apo, holo or holo,holo if you want to test fpocket only on holo structures), as well as the residuename (PDB HET residue tag, like "**`hem`**" for heme) of the reference ligand, separated by tabulations.

Optional:

- **`-o`** : (default ./**`stats_p.txt`**) The filename you want to give to tpocket detailed statistics.
- **`-e`** : (default ./**`stats_g.txt`**) The filename you want to give to tpocket global

statistics.

- `-d` : Distance criteria used for one of the 3 definition of a pocket: All atoms situated at a distance lower of equal that d will be considered as part of the actual pocket.
- `-k` : Keep fpocket output for each pdb test.

Last, all optional parameters used by fpocket are also accessible on command line through tpocket. Refer to page 17 to see details about fpocket parameters.

Actual pocket definition for evaluation of fpocket

Delimiting, and more generally defining what is the exact binding pocket of a protein in an automated way is not that easy. Finding a criteria that evaluate correctly the ability of fpocket to detect the actual binding site of a protein is consequently even more difficult.

Tpocket makes use of 6 different ways to determine if a pocket found by fpocket could be considered as the actual binding pocket, with respect to a given ligand:

- `1` The actual binding site is reduced to a single point, the barycenter of the pocket (calculated using alpha spheres). The binding pocket is defined as the pocket which barycenter is situated at a distance of 4A of any ligand atom. It corresponds to the Ppc discussed in the paper.
- `2` The actual binding pocket is defined by the set of atoms that are in contact with alpha sphere that are nearby (< 3A) the actual ligand. This set of atoms is then compared to all atoms contacted by all voronoi vertices included in each pocket found by fpocket. WARNING: this is currently not safely usable for an holo/apo dataset.
- `3` The actual binding pocket is defined by the set of atoms that are nearby (4A) the actual ligand. The same procedure as for the first definition is then applied to say whether a pocket can be considered as the actual pocket or not. WARNING: this is currently not safely usable for an holo/apo dataset.
- `4` The actual binding pocket is defined by the set of alpha sphere nearby (< 3A) the actual ligand. Then, for a given pocket, we calculate the correspondence between alpha sphere in the pocket, and alpha sphere in the actual binding pocket. If this ratio exceed a certain value (25%), we consider this pocket as being the actual pocket.
- `5` For a given pocket, we calculate the proportion of ligand atom that are nearby (< 3A) at least one alpha sphere of pocket. If this proportion exceed a certain value (50%), we consider this pocket as being the actual pocket.
- `6` A combination of both 5th and 6th criteria described above. If both 4th and 5th criterion are satisfied, then this criteria is. This corresponds to the MOc (Mutual Overlap criterion) discussed in the paper.

The reason why we choose 3A for the criteria 2, 4 and 5 is quite simple: as in the current algorithm, the minimum radius of an alpha sphere is 3A, a ligand atom situated at a distance lower or equal than this value can be considered as included in this alpha sphere, and therefore detected. Of course, this applies to alpha sphere with higher radius too.

All of these criteria have their strengths and witnesses, that's why we choose to implement all of them.

Pocket descriptors

In order to discriminate an interesting pocket from a lot of uninteresting ones, fpocket uses descriptors for each pocket. A scoring function, using these descriptors, was trained to well identify what we generally call “binding site”. Here are set together all descriptors implemented in fpocket. The ones that are actually used for scoring are marked with a *, and the one having the tag normalized associated with have a normalized (ie. scaled to a [0, 1] range, the highest (resp the lowest) value of a given descriptor being set to 1 (resp 0)) equivalent descriptor.

Number of alpha spheres (normalized) *

As the title says, this is surely the most simple descriptor. The number of alpha spheres reflects generally more or less proportionally the size of the cavity.

Density of the cavity (normalized) *

This descriptor tends to measure the density and “buriedness” of a pocket. It is nothing else than the mean value of all alpha sphere pair to pair distances in the binding pocket. Thus, a small value indicates a rather big compactness of the binding pocket and thus a rather burried pocket. Larger values give indication about more extended and exposed cavities.

Polarity Score (normalized) *

In the contrary to hydrophobicity this descriptor tries to measure the hydrophilicity character of a binding pocket. To each residue of the binding pocket a polarity score is assigned (as published on http://www.info.univ-angers.fr/~gh/Idas/proprietes.htm). The final polarity score is the mean of all polarity scores of all residues in the binding pocket. This is extremely approximative, so should not be overestimated. Each residue is evaluated only once.

Mean local hydrophobic density (normalized)*

This descriptor tries to identify if the binding pocket contains local parts that are rather hydrophobic. For each apolar alpha sphere the number of apolar alpha sphere neighbors is detected by seeking for overlapping apolar alpha spheres. The sum of all apolar alpha sphere neighbors is divided by the total number of apolar alpha spheres in the pocket. Last this score is normalized compared to other binding pockets.

Proportion of apolar alpha spheres (normalized) *

This descriptor, returned as percentage, reflects the proportion of apolar alpha spheres among all alpha spheres of one pocket identified by fpocket. This can reflect somehow the hydrophobic/-philic character of a binding pocket.

Maximum distance between two alpha sphere (normalized)

This descriptor store the maximum distance found between two alpha sphere in a given pocket.

Hydrophobicity Score

This descriptor is based on a residue based hydrophobicity scale published by Monera & al. in the Journal of Protein Science 1, 319-329 (1995). For all residues implicated in the binding site the mean hydrophobicity score is calculated and is used as descriptor for the whole pocket. Each residue is evaluated only once.

Charge Score

According to (http://www.info.univ-angers.fr/~gh/Idas/proprietes.htm) the charge of each amino acid in the binding site is tracked. The mean charge for all amino acids in contact with at least one alpha sphere of the pocket is calculated to form this charge score. Each residue is evaluated only once.

Volume Score

Similarly to other descriptors, this one is based on data published on (http://www.info.univ-angers.fr/~gh/Idas/proprietes.htm). This data resumes relative volume of different amino acids. In order to calculate this descriptor the mean volume score of all amino acids in contact with at least one alpha sphere of the pocket is calculated. Each residue is evaluated only once.

Composition of amino acids

As the name indicates, fpocket tracks the composition in amino acids of binding pockets. If at least one atom of a residue is in contact with at least one alpha sphere of a binding pocket it is accounted to be part of the binding site. This descriptor is returned as cumulative list, for instance you can find 2 valines, 3 glutamates etc... in the binding site.

Occurences of amino acids in different descriptor outputs are given in the following order : Ala, Cys, Asp, Glu, Phe, Gly, His, Ile, Lys, Leu, Met, Asn, Pro, Gln, Arg, Ser, Thr, Val, Trp, Tyr.

Pocket volume

As indicated by the name, this descriptor tries to evaluate the volume of a binding pocket using a Monte-Carlo algorithm that calculates full volume occupied by all alpha sphere in a given pocket. The number of iteration of this algorithm can be controlled using fpocket input parameters.

B-factor score (normalized)

Please handle with a lot of care this score with native crystal structures. This score is based on the mean B-factor of all atoms of the binding pocket (atoms that are contacted by at least one alpha sphere). As the B factor does not necessarily reflect flexibility in crystal structures, this score is somehow abusive. However, one could imagine performing molecular dynamics or other in order to determine relative flexibility of atoms and store this information in the B-factor column of the PDB file format.

This descriptor is normalized with other pockets of the same protein.

Cofactor definition

fpocket, dpocket and tpocket contain in the current release (1.0) a fixed set of cofactors. So far so good, but what for? Cofactors are often structurally necessary or must be present in the protein structure for ligand binding. The PDB nomenclature, however, treats them as usual hetero atoms, using the HETATM tag. This is the tag that fpocket uses to identify and eliminate crystallographic waters and possible ligands of holo protein structures. In order to force fpocket to keep the cofactor you are interested in, that is to say, to consider it as entire part of the protein structure for binding pocket detection, a list list of HETATM names is defined in the beginning of the **`rpdb.c`** file under the name **`static const char *ST_keep_hetatm[]`**. The next line of code defines the number of cofactors defined in this list : **`static const int ST_nb_keep_hetatm = 111 ;`**

If you would like to add a new cofactor, you have to modifiy this code. First you add the whished HETATM tag to **`ST_keep_hetatm`** in the end of the list. Thus for example, **`MSE`** will become **`MSE , PTE`** If your cofactor has the HETATM tag **`PTE`**. Do not forget to increment the **`ST_nb_keep_hetatm`** variable to **`112`**, else this cofactor will not be taken into account.

Next you have to recompile the program, before being able to use this new definition.

In future releases this cofactor definition will be done dynamically with an external list.

The following list resumes the cofactors fpocket considers as recurrent in the PDB and useful to keep in protein structures in a systematic manner.

HETATM tag	name	HETATM tag	name	HETATM tag	name
hea	Heme-a	hbi	7,8-dihydrobiopterin	bio	Biopterin
cfm	Fe-mo-S cluster	clp	Fe-S cluster	fes	Fe2/s2 (inorganic) cluster
f3s	Fe3-s4 cluster	fs4	Iron/sulfur cluster	bph	Bacteriopheophytin a

bpb	Bacteriopheophytin B	bcl	Bacteriochlorophyll a	bcb	Bacteriochlorophyll B
cob	Co-methylcobalamin	zn	Zinc ion	fea	Monoazido-mu-oxo-diiron
feo	Mu-oxo-diiron	h4b	5,6,7,8-tetrahydrobiopterin	bh4	(6r,1'R,2's)-5,6,7,8 tetrahydrobiopterin
bhs	6s-5,6,7,8-tetrahydrobiopterin	hbl	7,8- Dihydro- L-Biopterin	thb	Tetrahydrobiopterin
ddh	Diacetyldeuteroheme	dhe	Heme D	has	Heme-as
hdd	Cis-heme D hydroxychlorin gamma-spirolactone	hdm	Dimethyl propionate ester heme	heb	Heme B/C
hec	Heme C	heo	Heme O	hes	Zinc substituted heme C
hev	1,3-Dedimethyl-1,3-Divinyl Heme	mhm	[7,12- DIETHYL-3,8,13,17-TETRAMETHYL-21H,23H- PORPHINE-2,18- DIPORPANOTO-(2)-N21,N22,N23,N24,] IRON	srm	Siroheme
ver	Iron-octaethylporphyrin	1fh	12-phenylheme	2fh	2-phenylheme
hc0	2 iron/2 sulfur/6 carbonyl/1 water inorganic cluste	hc1	2 iron/2 sulfur/5 carbonyl/2 water inorganic cluster	hf3	Smallest hf-oxo-phosphate cluster hf3
hf5	Hf oxo cluster hf5	nfs	Fe(4)-ni(1)-S(5) cluster	omo	Mo(vi)(=O)(oh)2 cluster
phf	Hf-oxo-phosphate cluster phf	sf3	Fe4-s3 cluster	sf4	Iron/sulfur cluster
cfm	Fe-mo-S cluster	cfn	Fe(7)-mo-S(9)-N cluster	clf	Fe(8)-S(7) cluster
clp	Fe-S cluster	cn1	Oxo-iron cluster 2	cnb	Oxo-iron cluster 1
cnf	Oxo-iron cluster 3	cub	Cu(i)-S-mo(iv)(=O)o-nbic cluster	cum	Cu(i)-S-mo(vi)(=O)oh cluster
cun	Cu(i)-S-mo(iv)(=O)oh cluster	cuo	Cu2-o2 cluster	fs2	Fe-S-O hybrid cluster
fso	Iron/sulfur/oxygen hybrid cluster	fsx	FE4-S3-O3 Cluster	pho	Pheophytin a
bhi	4-bromo-3-hydroxy-3-methyl butyl diphosphate	chl	Chlorophyll B	cl1	Alpha chlorophyll a
cl2	Beta chlorophyll a	cla	Chlorophyll a	cch	Clorocruoro HEM
cfo	Chloro diiron-oxo moiety	fe2	Fe (ii) ion	fci	Ferricrocin-iron
fco	Carbonmonoxide-(dicyano) iron	fdc	Iron(iii) dicitrate	fea	Monoazido-mu-oxo-diiron
feo	Mu-oxo-diiron	fne	(mu-sulphido)-bis(mu-	hif	Fe(iii)-(4-

			cys,S)-[tricarbonyliron-di-(cys, S)nickel(ii)] (Fe-ni)		mesoporphyrinone)
ofo	Hydroxy diiron-oxo moiety	pfc	Phenylferricrocin-iron	he5	ZND-DME
baz	Bis(5-amidino-benzimidazolyl)methane zinc	boz	Bis(5-amidino-benzimidazolyl)methanone zinc	fe	Fe (iii) ion
hem	Protoporphyrin ix containing Fe	hco	2-acetyl-protoporphyrin ix	1cp	Coproporphyrin i
cln	Sulfur substituted protoporphyrin ix	coh	Protoporphyrin ix containing co	cp3	Coproporphyrin iii
deu	Co(iii)-(deuteroporphyrin ix)	fdd	Fe(iii) 2,4-dimethyl deuteroporphyrin ix	fde	Fe(iii) deuteroporphyrin ix
fec	FE-Coproporphyrin III	fmi	Fe-(4-mesoporphyrinone)-R-isomer	heg	Protoporphyrin ix containing Mg
heg	Protoporphyrin ix containing Mg	hni	Protoporphyrin ix containing ni(ii)	mmp	N-methylmesoporphyrin
mnh	Manganese protoporphyrin ix	mnr	Protoporphyrin ix containing Mn	mp1	N-methylmesoporphyrin containing copper
pc3	Coproporphyrin i containing co(iii)	pcu	Cu(ii)meso(4-N-tetramethylpyridyl)porphyrin	pni	Tetra[N-methyl-pyridyl] porphyrin-nickel
por	Porphyrin Fe(iii)	pp9	Protoporphyrin ix	mse	Selenomethionine

Customizing fpocket

This section will introduce several ways of customizing fpocket by modifying the source code. We will first gives all instructions needed to recompile and rebuild the full package when any modification of the source code has to be taken into account. Then, we will describe how to write a new scoring function, and how to write your own descriptors and include it to dpocket output. We will not show the full content of the function to modify as we want to stay as concise as possible. The newly added code for these examples will be highlighted in blue.

How to rebuild the package

After any modification to the fpocket source code, you will logically need to rebuild the package so the modification could be taken into account. Here is the current procedure to do so:

```
$ cd PATH/fpocket-src-1.0

$ make uninstall
```

```
        $ make clean
```

Then, you will have to perform the installation process again to rebuild the package.

Writing your own scoring function

Writing your own scoring function using currently implemented descriptors is a simple task, provided that you are not afraid to write one line of C code. Currently, the fpocket algorithm sort pockets using each pocket score. Each score is calculated by a single function. The source file src/pscoring.c contains the definition of this function that have the following prototype:

```
   float score_pocket(s_desc *pdesc) ;
```

The function takes as argument a pointer to a structure that contains all descriptors currently available in fpocket, and is called for each pocket to be scored. All descriptors available have been described previously, and you can check the exact name given to each of them in the source file headers/descriptors.h that defines the s_desc structure shown here.

Lets say that you just want to score pockets according to the number of alpha sphere of each pocket. To do so, you just have to change the content of `score_pocket` function and return the right value:

```
        float score_pocket(s_desc *pdesc)

        {

                float score = (float) pdesc->nb_asph ;

                return score ;

        }
```

Although this example is really simple, you may now understand that you can write any kind of scoring function, like a linear or non-linear combination of descriptors derived from a regression model or any other method. The only limitation is the use of available descriptors implemented in fpocket.

Of course, although the current scoring function has very satisfying performances using only 4 of the available descriptors, you may want to implement your own set. The next section will give you the basics to do so.

Writing your own descriptor

So what if you want to write your own descriptors? Well this will be a little more difficult that writing your own scoring function, but nothing is impossible!

Suppose that we want to add a new (and very simple) descriptor: the maximum alpha sphere radius in a given pocket.

First of all, you have to add the variable that will store your descriptor to the structure containing all descriptors. This has to be done in the descriptor.h source file, in the definition of the structure **`s_desc`**. We will add the following line:

```
        typedef struct s_desc

        {       ...
```

```
            float as_max_r ;
    } s_desc ;
```

After adding our variable, we need to give a default value when no calculation have been performed, lets say -1. This is done in the function **`reset_desc`** located in the same file:

```
    void reset_desc(s_desc *desc)
    {       ...
            desc->as_max_r = -1.0 ;
    }
```

Let's now implement our descriptor. Go to the *src/descriptor.c* source file. In this file, you fill find the main function that calculate descriptors based on a list of atoms and a list of alpha sphere. Here is the prototype of this function:

```
    void set_descriptors( s_atm **tatoms, int natoms,
                          s_vvertice **tvert, int nvert,
                          s_desc *desc) ;
```

As you can see, the function takes in argument a list of atoms, a list of vertices, and an input/output descriptor structure that will actually store all descriptors calculated. When descriptors has to be calculated on a given pocket, we first get all atoms and vertices of the pocket, and we call this function using those atoms and vertices as arguments. The calculation then use information on atoms and vertices to calculate descriptors.

Based on those parameters, you will have to write your own code in this function, and update in consequent the **`desc`** variable given in argument so the descriptor value could be stored. Lets do this. You will probably notice that the current code is not fully modular. This is because of computational optimization: a fully modular code sometimes requires additional loop and treatment compared to an optimized code. Anyway, the task is still very simple. Lets go into the part of the code that will do the job.

```
    void set_descriptors( s_atm **tatoms, int natoms,
                          s_vvertice **tvert, int nvert,
                          s_desc *desc)
    {       ...
            float as_max_r = -1.0 ; /* Declare and initialize the descriptor */
            ...

            for(i = 0 ; i < nvert ; i++) {
                    /* Loop through all vertices and update descriptors */
                    vcur = tvert[i] ;
                    if(vcur->ray > as_max_r) as_max_r = vcur->ray ;
                    ...
            }
            ...
            desc->as_max_r = as_max_r ;      /* Store the descriptor */
```

```
        }
```

That's it, your descriptor is implemented, as each pocket descriptors is automatically calculated using this function at the end of the fpocket algorithm. Thus, it can now be used in the scoring function described previously, after rebuilding the package of course.

Normalizing your descriptors

An advantage of normalization is that two descriptors generated from pockets of two different proteins can be compared to each other at a certain degree, depending on the normalization process. For example, if we normalize the number of alpha sphere between 0 and 1 (well here it's more a scaling than a normalization), the largest pocket of any protein will always have 1 as value for the normalized descriptor.

To do so, we can't use the exact same process as adding a given descriptor, because all descriptors of all pockets need to be calculated before the normalization step. Consequently, the calculation of all normalized descriptors is currently performed in the *src/pocket.c* source file. In this file, the function **`set_normalized_descriptors`** does the job, and have the following prototype:

```
   void set_normalized_descriptors(c_lst_pockets *pockets)
```

As you can see, it simply takes in argument a list of pockets, in fact a simple chained list, e.g. all pockets found in a given protein. Of course each pocket contained in this structure have a descriptor structure associated with.

Lets now enter more deeply into the code, and implement a normalized version of the new descriptors so it ranges between 0 and 1. The first step is similar to the first step needed to implement a new descriptors: you need to add a variable that will store this normalized descriptor in the structures pdesc:

```
        typedef struct s_desc
        {       ...
                float as_max_r ;
                float as_max_r_norm ;
        } s_desc ;
```

You can now add the default initialization of this descriptor:

```
        void reset_desc(s_desc *desc)
        {       ...
                desc->as_max_r = -1.0 ;
                desc->as_max_r_norm = -1.0 ;
        }
```

Lets implement the descriptor now. Go to the *src/pocket.c* source file, **`set_normalized_descriptor`** function. To calculate the normalized descriptor, we need the min and max value of the non-normalized descriptors. Next, we have to loop on the pocket list, update the min and max if necessary, and perform the normalization at the end of the loop. So easy:

```
void set_normalized_descriptors(c_lst_pockets *pockets)
{       ...
        /* Declare min and max */
        float  as_max_r_m = 1000,         /* Initialize to a large value*/
               as_max_r_M = -1.0 ;        /* Initialize to a small value */
        ...
        cur = pockets->first ;
        /* Perform a first processing step, e.g. to set min and max */
        while(cur) {
               dcur = pcur->pdesc ;
               ...
               if(cur == pockets->first) {
                      ...
                      /* If it is the first pocket, min = max = pocket */
                      as_max_r_m =  as_max_r_M = dcur->as_max_r ;
               }
               else {
                      ...
                      /* If it is the Nth != 1 pocket, check and update
                         min and max if necessary*/
                      if(dcur->as_max_r > as_max_r_M)
                             as_max_r_M = dcur-> as_max_r ;
                      else if(dcur->as_max_r < as_max_m)
                             as_max_r_m = dcur->as_max_r ;
               }
               cur = cur->next ;
        }

        /* Perform a second loop to do the actual normalisation */
        cur = pockets->first ;
        while(cur) {
               dcur = cur->pocket->pdesc ;
               ...
               dcur->as_max_r_norm = (dcur->as_max_r - as_max_r_m)
                                        / (as_max_r_M - as_max_r_m) ;
        }
}
```

And that's it. There is a little bit more effort to provide here to normalize the descriptor, but we believe it's not that much to do.

Unfortunately, we haven't taken the time to automatically add any new descriptor to the dpocket input. So basically here, your descriptors is implemented and can be used by a scoring function, but is not written to the dpocket output. The next paragraph will learn you how to so, it's very easy.

Including your descriptor in dpocket

Although it would be possible, we haven't taken the time to construct a system that would detect and add automatically any new descriptor to the dpocket output.

So let's do this manually. The dpocket output format is defined by 3 macro in the dpocket.h header file:

```
#define M_DP_OUTP_HEADER "pdb lig ..."
#define M_DP_OUTP_FORMAT "%s %s ..."
#define M_DP_OUTP_VAR (fc, l, ovlp, status, dst, lv, d) fc, l, ...
```

The first macro define the header of the output file. The second macro correspond to the format of each value to output given to the fprintf function. Finally, the last macro are the list of variable, with **d** being the pointer to the descriptor structure defined previously. Basically, writing the dpocket output for each pocket requires two main processes: write the header, and loop to write each pocket descriptors.

To include our descriptor into the dpocket output, we just need to add the header label of the descriptor, add the output format of the descriptor, and add the descriptor itself. Those three steps will modify the first, the second, and the third macro defined previously, respectively. The only difficulty is to keep the correspondence between of all 3 positions (header, format and variable) in the line: column number (position) of the header corresponding to the number of alpha sphere must correspond the that of the format and variable. For example, if we want to add our normalized variable at the first position of dpocket output, it would give:

```
#define M_DP_OUTP_HEADER "as_max_r pdb lig ..."
#define M_DP_OUTP_FORMAT "%3.5f %s %s ..."
#define M_DP_OUTP_VAR (fc, l, ovlp, status, dst, lv, d) d->as_max_r,
                                                        fc, l, ovlp, ...
```

That's pretty all. Remember to be careful on this step: adding a new descriptor to dpocket is really easy in theory, but losing the correspondence between header, format and variable position columns is easy too, in which case interpretation, visualization and analysis of dpocket output become somehow difficult or even meaningless.