Fragment Hotspot Maps Documentation

Release 1.0.0

Chris Radoux, Peter Curran, Mihaela Smilova

Feb 06, 2019

CONTENTS:

The Hotspot API has been created to work alongside the CCDC's CSD Python API, allowing you to run Fragment Hotspot Map calculations and process the results within Python.

CHAPTER

ONE

TUTORIAL

This section will introduce the main functionality of the Hotspots API

1.1 Getting Started

Although the Hotspots API is publicly available, it is dependant on the CSD python API - a commercial package. If you are an academic user, it's likely your institution will have a license. If you are unsure if you have a license or would like to enquire about purchasing one, please contact support@ccdc.cam.ac.uk

Please note, this is an academic project and we would therefore welcome feedback, contributions and collaborations. If you have any queries regarding this package please contact us (pcurran@ccdc.cam.ac.uk)!

NB: We recommend installing on a Linux machine

1.1.1 Installation

Step 1: Install CSDS 2019

Available from CCDC downloads page here.

You will need a valid site number and confirmation code, this will have been emailed to you when you bought your CSDS 2019 license

You may need to set the following environment variables:

```
export CSDHOME=<path_to_CSDS_installation>/CSD_2019
```

Step 2: Install Ghecom

Available from Ghecom download page here.

"The source code of the ghecom is written in C, and developed and executed on the linux environment (actually on the Fedora Core). For the installation, you need the gcc compiler. If you do not want to use it, please change the "Makefile" in the "src" directory."

Download the file "ghecom-src-[date].tar.gz" file.

```
tar zxvf ghecom-src-[date].tar.gz
cd src
make
export GHECOM_EXE="<download_directory>"
```

Step 3: Create a conda environment (recommended)

```
conda create -n hotspots_env python=2.7
```

Step 4: Install RDKit and CSD python API

Download the standalone CSD python API package from here.

```
conda install -c rdkit -n hotspots_env rdkit
conda install csd-python-api-2.x.x-linux-py2.7-conda.tar.bz2
```

Step 5: Install Hotspots API

```
source activate hotspots_env
pip install hotspots
```

. . . and you're ready to go!

1.2 Running a Calculation

1.2.1 Protein Preparation

The first step is to make sure your protein is correctly prepared for the calculation. The structures should be protonated with small molecules and waters removed. Any waters or small molecules left in the structure will be included in the calculation.

One way to do this is to use the CSD Python API:

```
from ccdc.protein import Protein

prot = Protein.from_file('protein.pdb')
prot.remove_all_waters()
prot.add_hydrogens()
for l in prot.ligands:
    prot.remove_ligand(l.identifier)
```

For best results, manually check proteins before submitting them for calculation.

1.2.2 Calculating Fragment Hotspot Maps

Once the protein is prepared, the *hotspots.calculation.Runner* object can be used to perform the calculation:

```
from hotspots.calculation import Runner

r = Runner()
results = Runner.from_protein(prot)
```

Alternatively, for a quick calculation, you can supply a PDB code and we will prepare the protein as described above:

```
r = Runner()
results = Runner.from_pdb("1hcl")
```

1.2.3 Reading and Writing Hotspots

Writing

The *hotspots.hs_io* module handles the reading and writing of both `hotspots.calculation.results` and `hotspots.best_volume.Extractor` objects. The output *.grd* files can become quite large, but are highly compressible, therefore the results are written to a *.zip* archive by default, along with a PyMOL run script to visualise the output.

```
from hotspots import hs_io

out_dir = "results/pdb1"

# Creates "results/pdb1/out.zip"
with HotspotWriter(out_dir, grid_extension=".grd", zip_results=True) as w:
    w.write(results)
```

Reading

If you want to revisit the results of a previous calculation, you can load the *out.zip* archive directly into a `hotspots.calculation.results` instance:

```
from hotspots import hs_io

results = hs_io.HotspotReader('results/pdb1/out.zip').read()
```

1.3 Using the Output

While Fragment Hotspot Maps provide a useful visual guide, the grid-based data can be used in other SBDD analysis.

1.3.1 Scoring

One example is scoring atoms of either proteins or small molecules.

This can be done as follows:

```
from ccdc.protein import Protein
from ccdc.io import MoleculeReader, MoleculeWriter
from hotspots.calculation import Runner

    r = Runner()
    prot = Protein.from_file("1hcl.pdb")     # prepared protein
    hs = r.from_protein(prot)

    # score molecule
    mol = MoleculeReader("mol.mol2")
    scored_mol = hs.score(mol)
```

(continues on next page)

(continued from previous page)

```
    with MoleculeWriter("score_mol.mol2") as w:
        w.write(scored_mol)

    # score protein
    scored_prot = hs.score(hs.prot)
    with MoleculeWriter("scored_prot.mol2") as w:
        w.write(scored_prot)
```

To learn about other ways you can use the Hotspots API please read our API documentation.

CHAPTER

TWO

ATOMIC HOTSPOT CALCULATION API

Atomic Hotspot detection is the first step in the Fragment Hotspot Maps algorithm and is implemented through SuperStar.

The main class of the *hotspots.atomic_hotspot_calculation* module are:

- *hotspots.atomic_hotspot_calculation.AtomicHotspot*

More information about the SuperStar method is available:

- SuperStar: A knowledge-based approach for identifying interaction sites in proteins M. L. Verdonk, J. C. Cole and R. Taylor, J. Mol. Biol., 289, 1093-1108, 1999 [DOI: 10.1006/jmbi.1999.2809]

class hotspots.atomic_hotspot_calculation.**AtomicHotspot**(*settings=None*)

A class for handling the calculation of Atomic Hotspots using SuperStar

class InstructionFile(*jobname*, *probename*, *settings*, *cavity=None*)

handles the instruction files for the commandline call of the Atomic Hotspot calculation

Parameters

- **jobname** (*str*) – general format "<probename>.ins"
- **probename** (*str*) – identifier for the atomic probe
- **settings** (*hotspots.AtomicHotspot.Settings*) – supplied if settings are adjusted from default
- **cavity** (*tup*) – (float(x), float(y), float(z)) describing the centre of a cavity

write(*fname*)

writes out the instruction file, enables calculation to be repeated

Parameters **fname** (*str*) – path of the output file

class Settings(*database='CSD'*, *map_background_value=1*, *box_border=10*, *min_propensity=0*, *superstar_sigma=0.5*)

handles the adjustable settings of Atomic Hotspot calculation.

NB: not all settings are exposed here, for other settings please look at *hotspots.atomic_hotspot_calculation._atomic_hotspot_ins()*

Parameters

- **database** – database from which the underlying interaction libraries are extracted
- **map_background_value** – the default value on the output grid
- **box_border** – padding on the output grid
- **min_propensity** – the minimum propensity value that can be assigned to a grid. value = 1 is random

- **superstar_sigma** – the sigma value for the gaussian smoothing function

atomic_probes
The probe atoms to be used by the Atomic Hotspot calculation.
Available atomic probes:
for the CSD:
- "Alcohol Oxygen",
- "Water Oxygen",
- "Carbonyl Oxygen",
- "Oxygen Atom",
- "Uncharged NH Nitrogen",
- "Charged NH Nitrogen",
- "RNH3 Nitrogen",
- "Methyl Carbon",
- "Aromatic CH Carbon",
- "C-Cl Chlorine"
- "C-F Fluorine",
- "Cyano Nitrogen",
- "Sulphur Atom",
- "Chloride Anion",
- "Iodide Anion"

for the PDB:
- "Alcohol Oxygen",
- "Water Oxygen",
- "Carbonyl Oxygen",
- "Amino Nitrogen",
- "Aliphatic CH Carbon"
- "Aromatic CH Carbon"

Return dict key= "identifier", value="SuperStar identifier"

calculate(*protein*, *nthreads=None*, *cavity_origins=None*)
Calculates the Atomic Hotspot

This function executes the Atomic Hotspot Calculation for a given input protein.

Parameters

- **protein** (*ccdc.protein.Protein*) – The input protein for which the Atomic Hotspot is to be calculated
- **nthreads** (*int*) – The number of processor to be used in the calculation. NB: do not exceed the number of available CPU's
- **cavity_origins** (*list*) – The list of cavity origins, if supplied the Atomic Hotspot detection will run on a cavity mode. This increase the speed of the calculation however small cavity can be missed in some cases.

Returns list of `hotspots._AtomicHotspotResults` instances

```
>>> from pdb_python_api import PDBResult
>>> from ccdc.protein import Protein
>>> from hotspots.atomic_hotspot_calculation import AtomicHotspot
```

```
>>> if __name__ == "__main__":
>>>     # NB: main guard required for multiprocessing on windows!
>>>     PDBResult("1mtx").download(out_dir="./1mtx")
>>>     protein = Protein.from_file("1mtx.pdb")
```

(continues on next page)

(continued from previous page)

```
>>>     protein.add_hydrogens()
>>>     protein.remove_all_waters()
```

```
>>>     a = AtomicHotspot()
>>>     a.settings.atomic_probes = {"apolar": "AROMATIC CH CARBON",
>>>                                 "donor": "UNCHARGED NH NITROGEN",
>>>                                 "acceptor": "CARBONYL OXYGEN"}
>>>     results = a.calculate(protein=protein, nthreads=3)
[_AtomicHotspotResult(donor), _AtomicHotspotResult(apolar), _
↪AtomicHotspotResult(acceptor)]
```

CHAPTER

THREE

HOTSPOT CALCULATION API

The *hotspots.calculation* handles the main Fragment Hotspot Maps algorithm. In addition, an alternative pocket burial method, Ghecom, is provided.

The main classes of the *hotspots.calculation* module are:

- *hotspots.calculation.Buriedness*
- *hotspots.calculation.Runner*

More information about the Fragment Hotspot Maps method is available from:

- Radoux, C.J. et. al., Identifying the Interactions that Determine Fragment Binding at Protein Hotspots J. Med. Chem. 2016, 59 (9), 4314-4325 [dx.doi.org/10.1021/acs.jmedchem.5b01980]

More information about the Ghecom method is available from:

- Kawabata T, Go N. Detection of pockets on protein surfaces using small and large probe spheres to find putative ligand binding sites. Proteins 2007; 68: 516-529

class hotspots.calculation.**Buriedness**(*protein*, *out_grid*, *settings=None*)

Bases: object

A class to handle the calculation of pocket burial

This provides a python interface for the command-line tool. Ghecom is available for download here!

NB: Currently this method is only available to linux users

Please ensure you have set the following environment variable:

```
>>> export GHECOM_EXE=<path_to_ghecom>
```

Parameters

- **protein** (*ccdc.protein.Protein*) – protein to submit for calculation
- **out_grid** (*ccdc.utilities.Grid*) – the output grid NB: must be initialised so that the bounding box covers the whole protein
- **settings** (*hotspots.hotspot_calculation.Buriedness.Settings*) –

class Settings(*ghecom_executable=None*, *grid_spacing=0.5*, *radius_min_large_sphere=2.5*, *radius_max_large_sphere=9.5*, *mode='M'*)

Bases: object

A class to handle the buriedness calculation settings using ghecom

Parameters

- **ghecom_executable** (*str*) – path to ghecom executable NB: should now be set as environment variable
- **grid_spacing** (*float*) – spacing of the results grid. default = 0.5
- **radius_min_large_sphere** (*float*) – radius of the smallest sphere
- **radius_max_large_sphere** (*float*) – radius of the largest sphere
- **mode** (*str*) – options
  - 'D'ilation 'E'rosion, 'C'losing(molecular surface), 'O'pening.
  - 'P'ocket(masuya_doi),'p'ocket(kawabata_go),'V':ca'V'ity, 'e'roded pocket.
  - 'M'ultiscale_closing/pocket,'I'nterface_pocket_bwn_two_chains.
  - 'G'rid_comparison_binary 'g'rid_comparison_mutiscale.
  - 'R'ay-based lig site PSP/visibility calculation.
  - 'L'igand-grid comparison (-ilg and -igA are required)[P]

calculate()
runs the buriedness calculation

Returns *hotspots.calculation._BuriednessResult*: a class with a `ccdc.utilities.Grid` attribute

class hotspots.calculation.**Runner**(*settings=None*)
Bases: object

A class for running the Fragment Hotspot Map calculation

class Settings(*nrotations=3000*, *apolar_translation_threshold=15*, *polar_translation_threshold=15*, *polar_contributions=False*, *return_probes=False*, *sphere_maps=False*)
Bases: object

adjusts the default settings for the calculation

Parameters

- **nrotations** (*int*) – number of rotations (keep it below 10**6)
- **apolar_translation_threshold** (*float*) – translate probe to grid points above this threshold. Give lower values for greater sampling. Default 15
- **polar_translation_threshold** (*float*) – translate probe to grid points above this threshold. Give lower values for greater sampling. Default 15
- **polar_contributions** (*bool*) – allow carbon atoms of probes with polar atoms to contribute to the apolar output map.
- **return_probes** (*bool*) – Generate a sorted list of molecule objects, corresponding to probe poses
- **sphere_maps** (*bool*) – When setting the probe score on the output maps, set it for a sphere (radius 1.5) instead of a single point.

from_pdb(*pdb_code*, *charged_probes=False*, *probe_size=7*, *buriedness_method='ghecom'*, *nprocesses=3*, *cavities=False*, *settings=None*)
generates a result from a pdb code

Parameters

- **pdb_code** (*str*) – PDB code

- **charged_probes** (*bool*) – If True include positive and negative probes
- **probe_size** (*int*) – Size of probe in number of heavy atoms (3-8 atoms)
- **buriedness_method** (*str*) – Either 'ghecom' or 'ligsite'
- **nprocesses** (*int*) – number of CPU's used
- **settings** (hotspots.calculation.Runner.Settings) – holds the calculation settings

Returns a hotspots.result.Result instance

```
>>> from hotspots.calculation import Runner
```

```
>>> runner = Runner()
>>> runner.from_pdb("1hcl")
Result()
```

from_protein(*protein*, *charged_probes=False*, *probe_size=7*, *buriedness_method='ghecom'*, *cavities=None*, *nprocesses=1*, *settings=None*, *buriedness_grid=None*)
generates a result from a protein

Parameters

- **protein** – a ccdc.protein.Protein instance
- **charged_probes** (*bool*) – If True include positive and negative probes
- **probe_size** (*int*) – Size of probe in number of heavy atoms (3-8 atoms)
- **buriedness_method** (*str*) – Either 'ghecom' or 'ligsite'
- **cavities** – Coordinate or *ccdc.cavity.Cavity* or *ccdc.molecule.Molecule* or list specifying the cavity or cavities on which the calculation should be run
- **nprocesses** (*int*) – number of CPU's used
- **settings** (hotspots.calculation.Runner.Settings) – holds the sampler settings
- **buriedness_grid** (*ccdc.utilities.Grid*) – pre-calculated buriedness grid

Returns a *hotspots.result.Results* instance

```
>>> from ccdc.protein import Protein
>>> from hotspots.calculation import Runner
```

```
>>> protein = Protein.from_file(<path_to_protein>)
```

```
>>> runner = Runner()
>>> settings = Runner.Settings()
>>> settings.nrotations = 1000  # fewer rotations increase speed at the␣
↪expense of accuracy
>>> runner.from_protein(protein, nprocesses=3, settings=settings)
Result()
```

CHAPTER

FOUR

HOTSPOT IO API

The *hotspots.hs_io* module was created to facilitate easy reading and writing of Fragment Hotspot Map results.

There are multiple components to a `hotspots.result.Result` including, the protein, interaction grids and buriedness grid. It is therefore tedious to manually read/write using the various class readers/writers. The Hotspots I/O organises this for the user and can handle single `hotspots.result.Result` or lists of `hotspots.result.Result`.

The main classes of the `hotspots.io` module are:

- `hotspots.io.HotspotWriter`
- `hotspots.io.HotspotReader`

class hotspots.hs_io.**HotspotReader**(*path*)

A class to organise the reading of a `hotspots.result.Result`

Parameters **path** (*str*) – path to the result directory (can be .zip directory)

read(*identifier=None*)

creates a single or list of `hotspots.result.Result` instance(s)

Parameters **identifier** (*str*) – for directories containing multiple Fragment Hotspot Map results,

identifier is the subdirectory for which a `hotspots.result.Result` is requried

Returns *hotspots.result.Result* a Fragment Hotspot Map result

```
>>> from hotspots.hs_io import HotspotReader
```

```
>>> path = "<path_to_results_directory>"
>>> result = HotspotReader(path).read()
```

class hotspots.hs_io.**HotspotWriter**(*path*, *visualisation='pymol'*, *grid_extension='.grd'*, *zip_results=True*, *settings=None*)

A class to handle the writing of a :class‘hotspots.result.Result‘. Additionally, creation of the PyMol visualisation scripts are handled here.

Parameters

- **path** (*str*) – path to output directory
- **visualisation** (*str*) – “pymol” or “ngl” currently only PyMOL available
- **grid_extension** (*str*) – “.grd”, “.ccp4” and “.acnt” supported
- **zip_results** (*bool*) – If True, the result directory will be compressed. (recommended)
- **settings** (`hotspots.hs_io.HotspotWriter.Settings`) – settings

class Settings
A class to hold the *hotspots.hs_io.HotspotWriter* settings

compress(*archive_name*, *delete_directory=True*)
compresses the output directory created for this `hotspots.HotspotResults` instance, and removes the directory by default. The zipped file can be loaded directly into a new `hotspots.HotspotResults` instance using the `from_zip_dir()` function

Parameters

- **archive_name** (*str*) – file path
- **delete_directory** (*bool*) – remove the out directory once it has been zipped

write(*hr*)
writes the Fragment Hotspot Maps result to the output directory and create the pymol visualisation file

Parameters **hr** (*hotspots.result.Result*) – a Fragment Hotspot Maps result or list of results

```
>>> from hotspots.calculation import Runner
>>> from hotspots.hs_io import HotspotWriter
```

```
>>> r = Runner
>>> result = r.from_pdb("1hcl")
>>> out_dir = <path_to_out>
>>> with HotspotWriter(out_dir) as w:
>>>     w.write(result)
```

CHAPTER

FIVE

RESULT API

The *hotspots.result* contains classes to extract valuable information from the calculated Fragment Hotspot Maps.

The main classes of the *hotspots.result* module are:

- *hotspots.result.Results*
- *hotspots.result.Extractor*

hotspots.result.Results can be generated using the *hotspots.calculation* module

```
>>> from hotspots.calculation import Runner
>>>
>>> r = Runner()
```

either

```
>>> r.from_pdb("pdb_code")
```

or

```
>>> from ccdc.protein import Protein
>>> protein = Protein.from_file("path_to_protein")
>>> result = r.from_protein(protein)
```

The *hotspots.result.Results* is the central class for the entire API. Every module either feeds into creating a *hotspots.result.Results* instance or uses it to generate derived data structures.

The *hotspots.result.Extractor* enables the main result to be broken down based on molecular volumes. This produces molecule sized descriptions of the cavity and aids tractibility analysis and pharmacophoric generation.

class hotspots.result.**Extractor**(*hr*, *settings=None*)

A class to handle the extraction of molecular volumes from a Fragment Hotspot Map result

Parameters

- **hr** (*hotspots.HotspotResults*) – A Fragment Hotspot Maps result
- **settings** (*hotspots.Extractor.Settings*) – Extractor settings

class Settings(*volume=150*, *cutoff=14*, *spacing=0.5*, *min_feature_gp=5*, *max_features=10*, *min_distance=6*, *island_max_size=100*, *pharmacophore=True*)

Default settings for hotspot extraction

Parameters

- **volume** (*float*) – required volume (default = 150)
- **cutoff** (*float*) – only features above this value are considered (default = 14)

- **spacing** (*float*) – grid spacing, (default = 0.5)
- **min_feature_gp** (*int*) – the minimum number of grid points required to create a feature (default = 5)
- **max_features** (*int*) – the maximum number of features in a extracted volume (default = 10)(not recommended, control at pharmacophore)
- **min_distance** (*float*) – the minimum distance between two apolar interaction peaks (default = 6)
- **island_max_size** (*int*) – the maximum number of grid points a feature can take. (default = 100)(stops overinflation of polar features)
- **pharmacophore** (*bool*) – if True, generate a Pharmacophore Model (default = True)

extract_all_volumes(*volume='125'*, *pharmacophores=True*)

from the main Fragment Hotspot Map result, the best continuous volume is calculated using peaks in the apolar maps as a seed point.

Parameters

- **volume** (*float*) – volume in Angstrom^3
- **pharmacophores** (*bool*) – if True, generates pharmacophores

Returns a *hotspots.result.Results* instance

```
>>> result
<hotspots.result.Results object at 0x000000001B657940>
```

```
>>> from hotspots.result import Extractor
```

```
>>> extractor = Extractor(result)
>>> all_vols = extractor.extract_all_volumes(volume=150)
[<hotspots.result.Results object at 0x000000002963A438>,
 <hotspots.result.Results object at 0x0000000029655240>,
 <hotspots.result.Results object at 0x000000002963D2B0>,
 <hotspots.result.Results object at 0x000000002964FDD8>,
 <hotspots.result.Results object at 0x0000000029651D68>,
 <hotspots.result.Results object at 0x00000000296387F0>]
```

extract_best_volume(*volume='125'*, *pharmacophores=True*)

from the main Fragment Hotspot Map result, the best continuous volume is returned

Parameters

- **volume** (*float*) – volume in Angstrom^3
- **pharmacophores** (*bool*) – if True, generates pharmacophores

Returns a *hotspots.result.Results* instance

```
>>> result
<hotspots.result.Results object at 0x000000001B657940>
```

```
>>> from hotspots.result import Extractor
```

```
>>> extractor = Extractor(result)
>>> best = extractor.extract_best_volume(volume=400)
[<hotspots.result.Results object at 0x0000000028E201D0>]
```

class hotspots.result.**Results**(*super_grids*, *protein*, *buriedness=None*, *pharmacophore=None*)
A class to handle the results of the Fragment Hotspot Map calcation and to organise subsequent analysis

Parameters

- **super_grids** (*dict*) – key = probe identifier and value = grid
- **protein** (*ccdc.protein.Protein*) – target protein
- **buriedness** (*ccdc.utilities.Grid*) – the buriedness grid
- **pharmacophore** (*bool*) – if True, a pharmacophore will be generated

static from_grid_ensembles(*res_list*, *prot_name*, *charged=False*)
Experimental feature

Creates ensemble map from a list of Results. Structures in the ensemble have to aligned by the binding site of interest prior to the hotspots calculation.

TODO: Move to the calculation module?

Parameters

- **res_list** – list of *hotspots.result.Results*
- **prot_name** (*str*) – str
- **out_dir** (*str*) – path to output directory

Returns a *hotspots.result.Results* instance

get_difference_map(*other*, *tolerance*)
Experimental feature.

Generates maps to highlight selectivity for a target over an off target cavity. Proteins should be aligned by the binding site of interest prior to calculation. High scoring regions of a map represent areas of favourable interaction in the target binding site, not present in off target binding site

Parameters

- **other** – a *hotspots.result.Results* instance
- **tolerance** (*int*) – how many grid points away to apply filter to

Returns a *hotspots.result.Results* instance

get_pharmacophore_model(*identifier='id_01'*, *threshold=5*)
Generates a hotspots.hotspot_pharmacophore.PharmacophoreModel instance from peaks in the hotspot maps

TODO: investigate using feature recognition to go from grids to features.

Parameters

- **identifier** (*str*) – Identifier for displaying multiple models at once
- **cutoff** (*float*) – The score cutoff used to identify islands in the maps. One peak will be identified per island

Returns a hotspots.hotspot_pharmacophore.PharmacophoreModel instance

histogram(*fpath='histogram.png'*)
get histogram of zero grid points for the Fragment Hotspot Result

Parameters **fpath** – path to output file

Returns data, plot

```
>>> result
<hotspots.result.Results object at 0x000000001B657940>
>>> plt = result.histogram()
>>> plt.show()
```

map_values()
get the number zero grid points for the Fragment Hotspot Result

Returns dict of str(probe type) by a `numpy.array` (non-zero grid point scores)

score(*obj=None*, *tolerance=2*)
annotate protein, molecule or self with Fragment Hotspot scores

Parameters

- **obj** – *ccdc.protein.Protein*, *ccdc.molecule.Molecule* or *hotsptos.result.Results* (find the median)
- **tolerance** (*int*) – the search radius around each point

Returns scored obj, either `ccdc.protein.Protein`, `ccdc.molecule.Molecule` or `hotspot.result.Results`

```
>>> result          # example "1hcl"
<hotspots.result.Results object at 0x000000001B657940>
```

```
>>> from numpy import np
>>> p = result.score(result.protein)     # scored protein
>>> np.median([a.partial_charge for a in p.atoms if a.partial_charge > 0])
8.852499961853027
```

CHAPTER

SIX

HOTSPOT PHARMACOPHORE API

The *hotspots.hs_pharmacophore* module contains classes for the conversion of Grid objects to pharmacophore models.

The main class of the *hotspots.hs_pharmacophore* module is:

- *hotspots.hs_pharmacophore.PharmacophoreModel*

A Pharmacophore Model can be generated directly from a `hotspots.result.Result` :

```
>>> from hotspots.calculation import Runner
```

```
>>> r = Runner()
>>> result = r.from_pdb("1hcl")
>>> result.get_pharmacophore_model(identifier="MyFirstPharmacophore")
```

The Pharmacophore Model can be used in Pharmit or CrossMiner

```
>>> result.pharmacophore.write("example.cm")    # CrossMiner
>>> result.pharmacophore.write("example.json")     # Pharmit
```

More information about CrossMiner is available:

- Korb O, Kuhn B, hert J, Taylor N, Cole J, Groom C, Stahl M "Interactive and Versatile Navigation of Structural Databases" J Med Chem, 2016, 59(9):4257, [DOI: 10.1021/acs.jmedchem.5b01756]

More information about Pharmit is available:

- Jocelyn Sunseri, David Ryan Koes; Pharmit: interactive exploration of chemical space, Nucleic Acids Research, Volume 44, Issue W1, 8 July 2016, Pages W442-W448 [DIO: 10.1093/nar/gkw287]

class hotspots.hs_pharmacophore.**PharmacophoreModel**(*settings*, *identifier=None*, *features=None*, *protein=None*, *dic=None*)

A class to handle a Pharmacophore Model

Parameters

- **settings** (`hotspots.hs_pharmacophore.PharmacophoreModel.Settings`) – Pharmacophore Model settings
- **identifier** (*str*) – Model identifier
- **features** (*list*) – list of :class:hotspots.hs_pharmacophore._PharmacophoreFeatures
- **protein** (*ccdc.protein.Protein*) – a protein
- **dic** (*dict*) – key = grid identifier(interaction type), value = `ccdc.utilities.Grid`

class Settings(*feature_boundary_cutoff=5*, *max_hbond_dist=5*, *radius=1.0*, *vector_on=False*, *transparency=0.6*, *excluded_volume=True*, *binding_site_radius=12*)

settings available for adjustment

Parameters

- **feature_boundary_cutoff** (*float*) – The map score cutoff used to generate islands
- **max_hbond_dist** (*float*) – Furthest acceptable distance for a hydrogen bonding partner (from polar feature)
- **radius** (*float*) – Sphere radius
- **vector_on** (*bool*) – Include interaction vector
- **transparency** (*float*) – Set transparency of sphere
- **excluded_volume** (*bool*) – If True, the CrossMiner pharmacophore will contain excluded volume spheres
- **binding_site_radius** (*float*) – Radius of search for binding site calculation, used for excluded volume

static from_hotspot(*result*, *identifier='id_01'*, *threshold=5*, *settings=None*)

creates a pharmacophore model from a Fragment Hotspot Map result

(included for completeness, equivalent to *hotspots.result.Result.get_pharmacophore()*)

Parameters

- **result** (*hotspots.result.Result*) – a Fragment Hotspot Maps result (or equivalent)
- **identifier** (*str*) – Pharmacophore Model identifier
- **threshold** (*float*) – values above this value
- **settings** (hotspots.hs_pharmacophore.PharmacophoreModel.Settings) – settings

Returns *hotspots.hs_pharmacophore.PharmacophoreModel*

```
>>> from hotspots.calculation import Runner
>>> from hotspots.hs_pharmacophore import PharmacophoreModel
```

```
>>> r = Runner()
>>> result = r.from_pdb("1hcl")
>>> model = PharmacophoreModel(result, identifier="pharmacophore")
```

static from_ligands(*ligands*, *identifier*, *protein=None*, *settings=None*)

creates a Pharmacophore Model from a collection of overlaid ligands

Parameters

- **ligands** (*ccdc,molecule.Molecule*) – ligands from which the Model is created
- **identifier** (*str*) – identifier for the Pharmacophore Model
- **protein** (*ccdc.protein.Protein*) – target system that the model has been created for
- **settings** (hotspots.hs_pharmacophore.PharmacophoreModel.Settings) – Pharmacophore Model settings

Returns *hotspots.hs_pharmacophore.PharmacophoreModel*

```
>>> from ccdc.io import MoleculeReader
>>> from hotspots.hs_pharmacophore import PharmacophoreModel
```

```
>>> mols = MoleculeReader("ligand_overlay_model.mol2")
>>> model = PharmacophoreModel.from_ligands(mols, "ligand_overlay_
↪pharmacophore")
>>> # write to .json and search in pharmit
>>> model.write("model.json")
```

static from_pdb(*pdb_code*, *chain*, *out_dir=None*, *representatives=None*, *identifier='LigandBasedPharmacophore'*)

creates a Pharmacophore Model from a PDB code.

This method is used for the creation of Ligand-Based pharmacophores. The PDB is searched for protein-ligand complexes of the same UniProt code as the input. These PDB's are align, the ligands are clustered and density of atom types a given point is assigned to a grid.

Parameters

- **pdb_code** (*str*) – single PDB code from the target system
- **chain** (*str*) – chain of interest
- **out_dir** (*str*) – path to output directory
- **representatives** – path to .dat file containing previously clustered data (time saver)
- **identifier** (*str*) – identifier for the Pharmacophore Model

Returns *hotspots.hs_pharmacophore.PharmacophoreModel*

```
>>> from hotspots.hs_pharmacophore import PharmacophoreModel
>>> from hotspots.result import Results
>>> from hotspots.hs_io import HotspotWriter
>>> from ccdc.protein import Protein
>>> from pdb_python_api import PDBResult
```

```
>>> # get the PDB ligand-based Pharmacophore for CDK2
>>> model = PharmacophoreModel.from_pdb("1hcl")
```

```
>>> # the models grid data is stored as PharmacophoreModel.dic
>>> # download the PDB file and create a Results
>>> PDBResult("1hcl").download(<output_directory>)
>>> result = Result(protein=Protein.from_file("<output_directory>/1hcl.pdb"),␣
↪super_grids=model.dic)
>>> with HotspotWriter("<output_directory>") as w:
>>>     w.write(result)
```

rank_features(*max_features=4*, *feature_threshold=0*, *force_apolar=True*)

orders features by score

Parameters

- **max_features** (*int*) – maximum number of features returned
- **feature_threshold** (*float*) – only features above this value are considered
- **force_apolar** – ensures at least one point is apolar

Returns list of features

```
>>> from hotspots.hs_io import HotspotReader
```

```
>>> result = HotspotReader("out.zip").read()
>>> model = result.get_pharmacophore_model()
>>> print(len(model.features))
38
>>> model.rank_features(max_features=5)
>>> print(len(model.features))
5
```

write (*fname*)

writes out pharmacophore. Supported formats:

- ".cm" (*CrossMiner*),
- ".json" (Pharmit),
- ".py" (*PyMOL*),
- ".csv",
- ".mol2"

Parameters **fname** (*str*) – path to output file

CHAPTER

SEVEN

HOTSPOT DOCKING API

The *hotspots.hs_docking* module contains functionality which faciliates the **automatic** application of insights from Fragment Hotspot Maps to docking.

This module is designed to extend the existing CSD python API

More information about the CSD python API is available:

- The Cambridge Structural Database C.R. Groom, I. J. Bruno, M. P. Lightfoot and S. C. Ward, Acta Crystallographica Section B, B72, 171-179, 2016 [DOI: 10.1107/S2052520616003954]
- CSD python API 2.0.0 documentation

More information about the GOLD method is available:

- Development and Validation of a Genetic Algorithm for Flexible Docking G. Jones, P. Willett, R. C. Glen, A. R. Leach and R. Taylor, J. Mol. Biol., 267, 727-748, 1997 [DOI: 10.1006/jmbi.1996.0897]

class hotspots.hs_docking.**DockerSettings**(*_settings=None*)

A class to handle the integration of Fragment Hotspot Map data with GOLD

This class is designed to mirror the existing CSD python API for smooth integration. For use, import this class as the docking settings rather than directly from the Docking API.

```
>>> from ccdc.docking import Docker
>>> from ccdc.protein import Protein
>>> from hotspots.calculation import Runner
>>> from hotspots.hs_docking import DockerSettings
```

```
>>> protein = Protein.from_file("1hcl.pdb")
```

```
>>> runner = Runner()
>>> hs = runner.from_protein(protein)
```

```
>>> docker.settings.add_protein_file("1hcl.pdb")
>>> docker.settings.add_ligand_file("dock_me.mol2", ndocks=25)
>>> constraints = docker.settings.HotspotHBondConstraint.from_
↪hotspot(protein=docker.settings.proteins[0], hr=hs)
>>> docker.settings.add_constraint(constraints)
>>> docker.dock()
```

```
>>> docker.Results(docker.settings).ligands
```

class HotspotHBondConstraint(*atoms*, *weight=5.0*, *min_hbond_score=0.001*, *_constraint=None*)

A protein HBond constraint constructed from a hotspot Assign Protein Hbond constraints based on the highest scoring interactions.

Parameters

- **atoms** (*list*) – list of `ccdc.molecule.Atom` instances from the protein. *NB: The atoms should be donatable hydrogens or acceptor atoms.*
- **weight** – the penalty to be applied for no atom of the list forming an HBond.
- **min_hbond_score** – the minimal score of an HBond to be considered a valid HBond.

static create(*protein*, *hr*, *max_constraints=2*, *weight=5.0*, *min_hbond_score=0.001*, *cutoff=14*)

creates a `hotspots.hs_docking.HotspotHBondConstraint`

Parameters

- **protein** (*ccdc.protein.Protein*) – the protein to be used for docking
- **hr** (*hotspots.calculation.Result*) – a result from Fragment Hotspot Maps
- **max_constraints** (*int*) – max number of constraints
- **weight** (*float*) – the factor by which the atoms Fragment Hotspot Map score will be multiplied
- **min_hbond_score** (*float*) – float between 0.0 (bad) and 1.0 (good) determining the minimum hydrogen bond quality in the solutions.
- **cutoff** – minimum score required to assign the constraint

Return list list of `hotspots.hs_docking.HotspotHBondConstraint`

add_fitting_points(*hr*, *volume=400*, *threshold=17*, *mode='threshold'*)

uses the Fragment Hotspot Maps to generate GOLD fitting points.

GOLD fitting points are used to help place the molecules into the protein cavity. Pre-generating these fitting points using the Fragment Hotspot Maps helps to biast results towards making Hotspot interactions.

Parameters

- **hr** (*hotspots.result.Result*) – a Fragment Hotspot Maps result
- **volume** (*int*) – volume of the occupied by fitting points in Angstroms ^ 3
- **threshold** (*float*) – points above this value will be included in the fitting points
- **mode** (*str*) – 'threshold'- assigns fitting points based on a score cutoff or 'bcv'- assigns fitting points from best continuous volume analysis (recommended)

CHAPTER

EIGHT

HOTSPOT UTILITIES API

The `hotspots.utilities` module contains classes to for general functionality.

The main classes of the `hotspots.extraction` module are:

- *`hotspots.hs_utilities.Helper`*
- *`hotspots.hs_utilities.Figures`*

class hotspots.hs_utilities.**Coordinates**(*x*, *y*, *z*)

x
Alias for field number 0

y
Alias for field number 1

z
Alias for field number 2

class hotspots.hs_utilities.**Figures**
Class to handle the generation of hotspot related figures

TO DO: is there a better place for this to live?

static histogram(*hr*)
creates a histogram from the hotspot scores

Parameters **hr** (`hotspots.result.Results`) – a Fragment Hotspot Map result

Returns data, plot

class hotspots.hs_utilities.**Helper**
A class to handle miscellaneous functionality

static cavity_centroid(*obj*)
returns the centre of a cavity

Parameters **obj** – can be a *ccdc.cavity.Cavity* or

Returns Coordinate

static cavity_from_protein(*prot*)
currently the Protein API doesn't support the generation of cavities directly from the Protein instance this method handles the tedious writing / reading

Parameters **prot** (*`ccdc.protein.Protein`*) – protein

Returns *ccdc.cavity.Cavity*

static get_distance(*coords1*, *coords2*)
given two coordinates, calculates the distance

Parameters

- **coords1** (*tup*) – float(x), float(y), float(z), coordinates of point 1
- **coords2** (*tup*) – float(x), float(y), float(z), coordinates of point 2

Returns float, distance

static get_label(*input*, *threshold=None*)
creates a value labels from an input grid dictionary

Parameters **input** (*dic*) – key = "probe identifier" and value = *ccdc.utilities.Grid*

Return ccdc.molecule.Molecule*ccdc.molecule.Molecule* pseduomolecule which contains score labels

static get_lines_from_file(*fname*)
gets lines from text file, used in Ghecom calculation

Returns list, list of str

static get_out_dir(*path*)
checks if directory exists, if not, it create the directory

Parameters **path** (*str*) – path to directory

Return str path to output directory

CHAPTER

NINE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

h