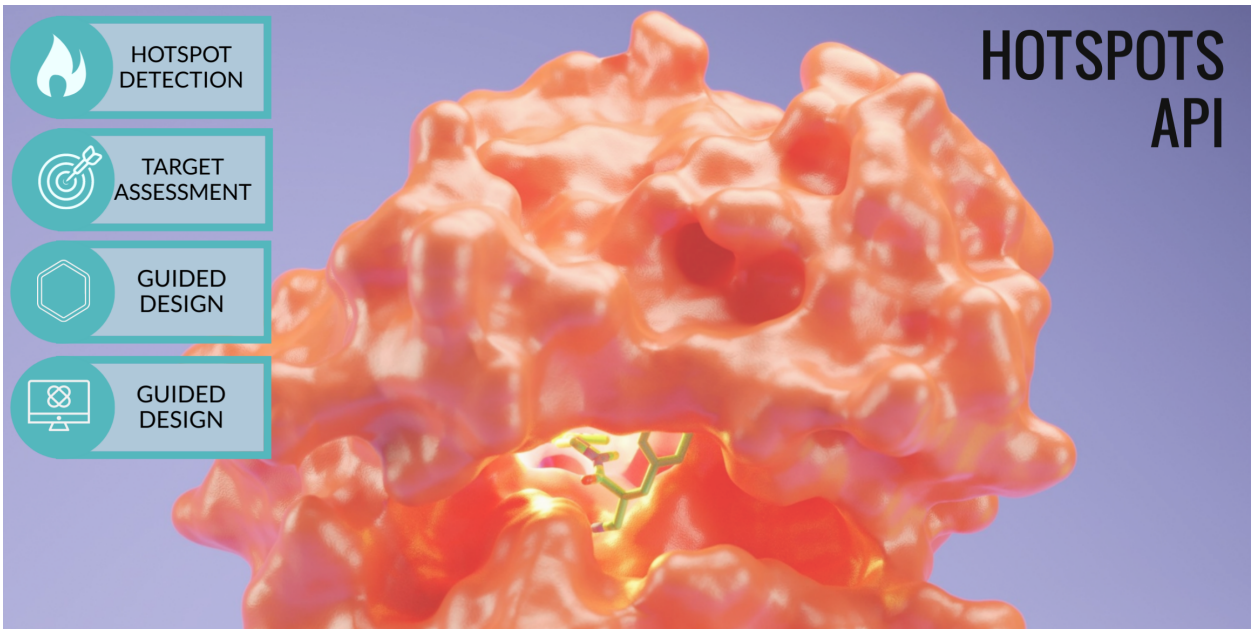
# Fragment Hotspot Maps Documentation

***Release 1.0.2***

**Jan 31, 2020**

# CONTENTS:




This 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. If this isn't possible, be aware that only the LIGSITE method will be available for the buriedness calculation.

CHAPTER

ONE

# INTRODUCTION

Fragment hotspot maps predicts the location and key interaction features of small molecule binding "hotspots" and provides valuable insights for several stages of early drug and drug target discovery. Built upon the vast quantity of interaction data in the CSD, fragment hotspot maps is able to rapidly detect hotspots from a global search of a protein.

The probability of forming common intermolecular interactions (hydrogen-bonding, charged, apolar) is estimated using Superstar. Superstar fragments a protein and uses interaction libraries, abstracted from the CSD, to predict the likelihood of finding a probe atom at a given point. The following probes are used: "apolar": Aromatic CH Carbon, "acceptor": Carbonyl oxygen, "donor": Uncharged NH Nitrogen, "negative": Carboxylate, "positive": Charged NH Nitrogen. Although SuperStar does have some hydrophobic correction, the local protein environment is not fully considered. Consequently, large regions of the protein are scored highly. Hotspots arise from enclosed, hydrophobic pockets that can form directional, polar interactions. Therefore, this method incorporates these physical characteristics into the detection of hotspots. This is done in two ways; weighting the SuperStar Maps by the degree of burial and sampling the weighted maps using hydrophobic molecular probes. This method was validated on a set of 21 fragment-to-lead progression. The median fragment atom scores were in the top 98% of all grid point scores.

CHAPTER

TWO

# INSTALLATION NOTES

## 2.1 Step 1: Install CSDS 2019

The CSDS is available from the link below:

https://www.ccdc.cam.ac.uk/support-and-resources/csdsdownloads/

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
```

## 2.2 Step 2: Install Ghecom

Ghecom is available from the link below:

http://strcomp.protein.osaka-u.ac.jp/ghecom/download_src.html

"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>"
```

## 2.3 Step 3: Create a conda environment (recommended)

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

## 2.4 Step 4: Install RDKit and CSD python API

Install RDKit:

```
conda install -n hotspots -c rdkit rdkit
```

The latest standalone CSD-Python-API installer from is available from the link below:

https://www.ccdc.cam.ac.uk/forum/csd_python_api/General/06004d0d-0bec-e811-a889-005056977c87

Install the Python CSD API:

```
unzip csd-python-api-2.1.0-linux-64-py2.7-conda
conda install -n hotspots -c <path to ccdc_conda_channel> csd-python-api
```

## 2.5 Step 5: Install Hotspots API

Install Hotspots v1.x.x:

a) Latest stable release (recommended for most users):

```
conda activate hotspots
pip install hotspots
```

or

```
pip install https://github.com/prcurran/hotspots/archive/v1.x.x.zip
```

b) Very latest code

```
mkdir ./hotspots_code
git clone git@github.com:prcurran/hotspots.git
conda activate hotspots
cd hotspots_code
pip install hotspots_code
```

NB: dependencies should install automatically. If they do not, please see setup.py for the package requirements!

. . . and you're ready to go!

CHAPTER

THREE

# COOKBOOK DOCUMENTATION

## 3.1 Running a Calculation

### 3.1.1 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.

### 3.1.2 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")
```

### 3.1.3 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)
```

### 3.1.4 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()
```

## 3.2 Tractability Assessment

Not all pockets provide a suitable environment for binding drug-like molecules. Therefore, good predictions of target tractability can save time and effort in early hit identification. Fragment Hotspot Maps annotates a set of grids which span the entire volume of pockets within proteins. The grids are scores represent the likelihood of making a particular intermolecular interaction and therefore they can be used to differientate between pockets and help researchers select a pocket with the highest chance of being tractable.

This cookbook example provides a very simple workflow to generate a target tractability model.

### 3.2.1 Tractability workflow

Firstly, the fragment hotspots calculation is performed. This is done by initialising a *hotspots.calculation.Runner* class object, and generate a *hotspots.result.Result* object, in this case we used the *hotspots.calculation.Runner.from_pdb* method which generates a result from a PDB code.

Next, A *hotspots.result.Result* object is returned. Not all the points within the grids are relevant - the entire pocket may not be involved in binding. Therefore, a subset of the cavity grid points are selected. The Best Continous Volume method is used to return a sub-pocket which corresponds to a user defined volume, and the algorithm selects a continous area which maximises the total score of the fragment hotspot maps grid points. In this case an approximate drug-like volume of 500 A^3 is used. This is carried out by using the *hotspots.Extractor.extract_volume()* class method.

Finally, the score distribution for the best continuous volume are used to discriminate between different pockets. For this simple cookbook example, we use the median value to rank the different pockets. These are returned as a dataframe. The code block below contains the complete workflow:

```
import numpy as np
import pandas as pd
from hotspots.calculation import Runner
from hotspots.result import Extractor


def tractability_workflow(protein, tag):
    """
    A very simple tractability workflow.

    :param str protein: PDB identification code
    :param str tag: Tractability tag: either 'druggable' or 'less-druggable'
    :return: `pandas.DataFrame`
    """
    # 1) calculate Fragment Hotspot Result
    runner = Runner()
    result = runner.from_pdb(pdb_code=protein,
                             nprocesses=1,
                             buriedness_method='ghecom')

    # 2) calculate Best Continuous Volume
    extractor = Extractor(hr=result)
    bcv_result = extractor.extract_volume(volume=500)

    # 3) find the median score
    for probe, grid in bcv_result.super_grids.items():
        values = grid.grid_values(threshold=5)
        median = np.median(values)

    # 4) return the data
    return pd.DataFrame({'scores': values,
                         'pdb': [protein] * len(values),
                         'median': [median] * len(values),
                         'tractability': [tag] * len(values),
                         })
```

### 3.2.2 Ranking Pockets

For this tutorial example, we simply rank the pockets by the median value of the best continuous volume score. Of course, for more complex ranking or classification methods could be used if desired. For this example, we take a random selection of 65 (43 = 'druggable', 22 = 'less druggable'). More information on this dataset

Krasowski, A.; Muthas, D.; Sarkar, A.; Schmitt, S.; Brenk, R. DrugPred: A Structure-Based Approach To Predict Protein Druggability Developed Using an Extensive Nonredundant Data Set. J. Chem. Inf. Model. 2011, 2829–2842. https://doi.org/10.1021/ci200266d.

```
def main():
    subset = {'1e9x': 'd', '1udt': 'd', '2bxr': 'd',
                '1r9o': 'd', '3d4s': 'd', '1k8q': 'd',
                '1xm6': 'd', '1rwq': 'd', '1yvf': 'd',
                '2hiw': 'd', '1gwr': 'd', '2g24': 'd',
                '1c14': 'd', '1ywn': 'd', '1hvy': 'd',
                '1f9g': 'n', '1ai2': 'n', '2ivu': 'd',
                '2dq7': 'd', '1m2z': 'd', '2fb8': 'd',
                '1o5r': 'd', '2gh5': 'd', '1ke6': 'd',
                '1k7f': 'd', '1ucn': 'n', '1hw8': 'd',
                '2br1': 'd', '2i0e': 'd', '1js3': 'd',
                '1yqy': 'd', '1u4d': 'd', '1sqi': 'd',
                '2gsu': 'n', '1kvo': 'd', '1gpu': 'n',
                '1qpe': 'd', '1hvr': 'd', '1ig3': 'd',
                '1g7v': 'n', '1qmf': 'n', '1r58': 'd',
                '1v4s': 'd', '1fth': 'n', '1rsz': 'd',
                '1n2v': 'd', '1m17': 'd', '1kts': 'n',
                '1ywr': 'd', '2gyi': 'n', '1cg0': 'n',
                '5yas': 'n', '1icj': 'n', '1gkc': 'd',
                '1hqg': 'n', '1u30': 'd', '1nnc': 'n',
                '1c9y': 'n', '1j4i': 'd', '1qxo': 'n',
                '1o8b': 'n', '1nlj': 'n', '1rnt': 'n',
                '1d09': 'n', '1olq': 'n'}

    pdbs, tags = zip(*[[str(pdb), str(tag)] for pdb, tag in training_set.
↪items()])

    with concurrent.futures.ProcessPoolExecutor(max_workers=2) as executor:
        dfs = executor.map(tractability_workflow, pdbs, tags)

    df = pd.concat(dfs, ignore_index=True)
    df = df.sort_values(by='median', ascending=False)

    df.to_csv('scores.csv')

if __name__ == '__main__':
    main()
```

We can visualise the score distributions using the seaborn plotting library and use the *ccdc.descriptors* module for statical analysis on the ranked pockets.

### JoyPlot

```
# adapted from the seaborn documentation.
#
import matplotlib.patches as patches
import matplotlib.pyplot as plt
import seaborn as sns

def joyplot(df, fname='test.png'):
    """
```

(continues on next page)

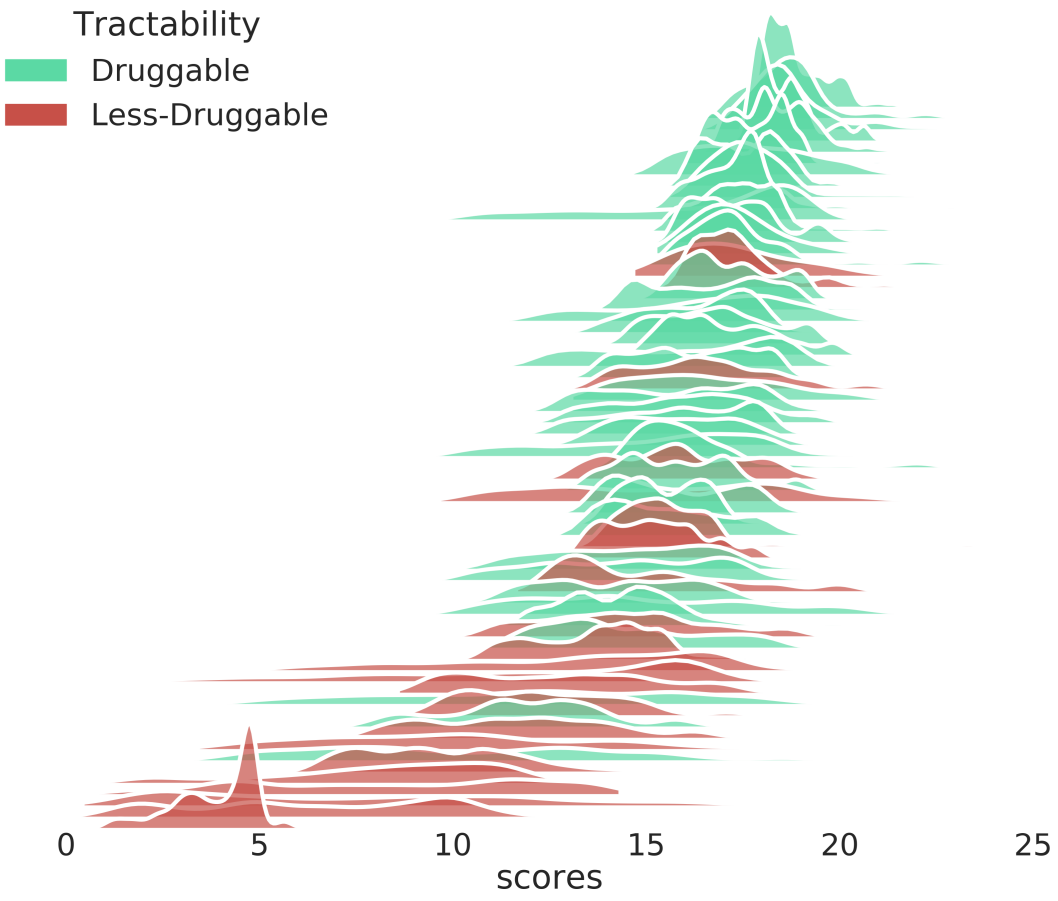
(continued from previous page)

```
    Visualises the Fragment Hotspot Maps score distributions as
    a series of stacked kernal density estimates ordered by
    median value.

    Adapted from the seaborn gallery:
        https://seaborn.pydata.org/examples/kde_ridgeplot.html

    :param `pandas.DataFrame` df: Fragment Hotspot scores data
    :return None
    """
    sns.set(style="white",
            rc={"axes.facecolor": (0, 0, 0, 0)},
            font_scale=6)

    palette = ["#c75048",
               "#5bd9a4"
               ]

    # Initialize the FacetGrid object
    ax = sns.FacetGrid(df,
                       row="pdb",
                       hue="tractability",
                       height=4,
                       aspect=75,
                       palette=palette)

    # Draw the densities in a few steps
    ax.map(sns.kdeplot, "scores", clip_on=False, shade=True, alpha=.7, lw=3,␣
↪bw=.2)
    ax.map(sns.kdeplot, "scores", clip_on=False, color="w", lw=9, bw=.2)

    # Format the plots
    ax.set(xlim=(0, 25))  #
    ax.fig.subplots_adjust(hspace=-.9)
    ax.set_titles("")
    ax.set(yticks=[])
    ax.despine(bottom=True, left=True)

    # Create legend, and position
    tag = {"d": "Druggable", "n": "Less-Druggable"}
    labels = [tag[s] for s in set(df["tractability"])]
    handles = [patches.Patch(color=col, label=lab)
               for col, lab in zip(palette, labels)]

    legend = plt.legend(handles=handles,
                        title='Tractability',
                        loc="upper right",
                        bbox_to_anchor=(0.3, 7.5))
    frame = legend.get_frame()
    frame.set_facecolor('white')
    frame.set_edgecolor('white')
```

(continues on next page)

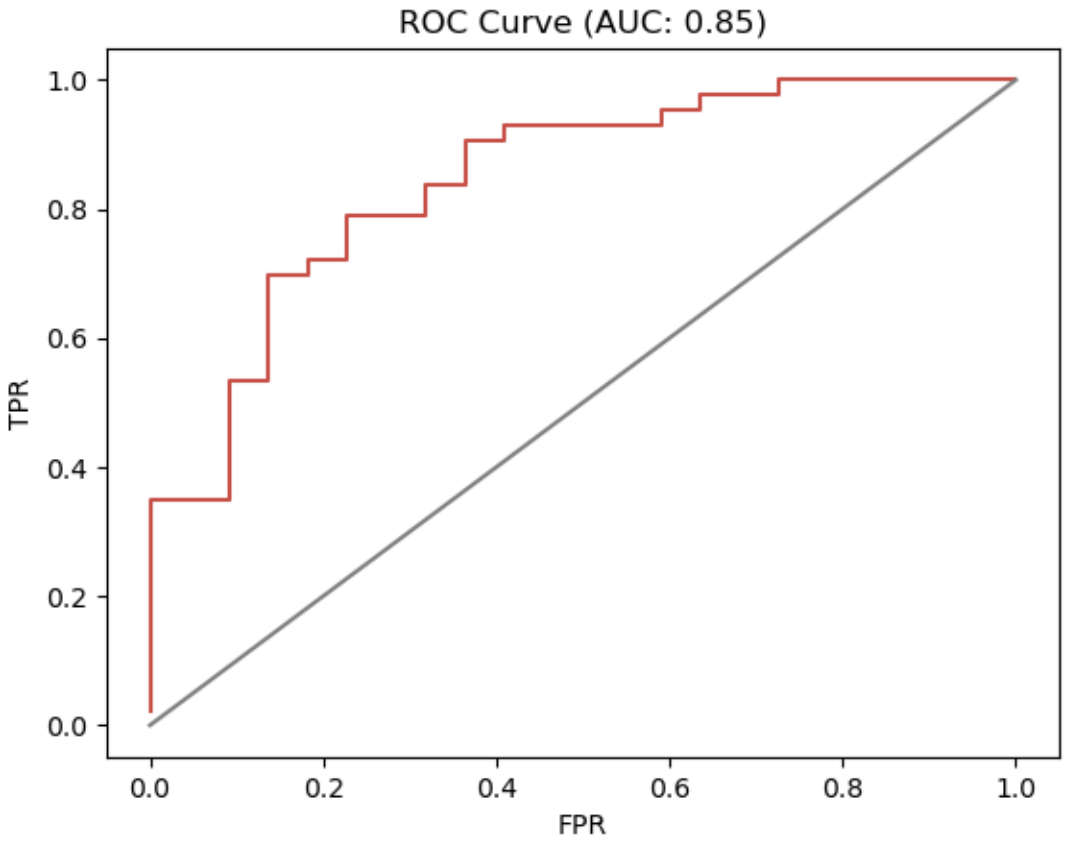
(continued from previous page)

```
    plt.savefig(fname)
    plt.close()

df = pd.read_csv('scores.csv')
df = df.sort_values(by='median', ascending=False)
joyplot(df, 'druggable_joy.png')
```



### Rank Statistics

```
import operator
import seaborn as sns

from ccdc.descriptors import StatisticalDescriptors as sd


def rocplot(data, fname='roc.png'):
    """
    Create a ROC Curve using seaborn

    :param lists data: supply ranked data as list of list
    :return: None
    """
    rs = sd.RankStatistics(scores=data, activity_column=operator.
↪itemgetter(2))
    tpr, fpr = rs.ROC()
    ax = sns.lineplot(x=fpr, y=tpr, estimator=None, color="#c75048")
    ax = sns.lineplot(x=[0,1], y=[0,1], color="grey")
    ax.set(xlabel='FPR', ylabel='TPR', title=f"ROC Curve (AUC: {rs.AUC():.2f})
↪")

    plt.savefig(fname)
    plt.close()

df = pd.read_csv('scores.csv')

t = []
m = []
letter_to_number = {"d": 1, "n": 0}
for p in set(df['pdb']):
    a = df.loc[df['pdb'] == p]
    t.append(letter_to_number[list(a['tractability'])[0]])
    m.append(list(a['median'])[0])

df = pd.DataFrame({'tractability': t,  'median':m})
df = df.sort_values(by='median', ascending=False)
data = list(zip(list(df["median"]), list(df["tractability"])))

rocplot(data, fname='druggable_roc.png')
```




## 3.3 Hotspot-Guided Docking

Molecular docking is a staple of early-stage hit identification. When active small molecules are known, key interactions can be selected to steer the scoring of molecular pose to favour those molecules making the selected interaction. Fragment Hotspot maps can predict critical interactions in the absence of binding data. Using these predictions as constraints will likely improve docking enrichment. A preliminary study was conducted by (Radoux, 2018) and a full validation is currently underway. Protein Kinase B (AKT1) has been chosen for this tutorial example and was used in the prelimenary docking study.

Radoux, C. J. The Automatic Detection of Small Molecule Binding Hotspots on Proteins Applying Hotspots to Structure-Based Drug Design. (2018). doi:10.17863/CAM.22314

### 3.3.1 A Hotspot Constraint

To begin, a hotspot calculation is performed on AKT1 (PDB: 3cqw). For this example, the protein was protonated using X, all waters, ligands and metal centres were removed.

*hotspots.hsdocking.DockerSettings* inherits from *ccdc.docking.Docker.Settings* to allow smooth integration with the CCDC python API. The following code snippet demonstrates how a constraint is generated and added to the *hotspots.hsdocking.DockerSettings* class.

```
from hotspots.hs_docking import DockerSettings
from hotspots.hs_io import HotspotReader


result = HotspotReader(<pathtohotspot>).read()

docker = Docker()
docker.settings = DockerSettings()
docker.settings.add_protein_file(<pathtoprotein>)

constraints =
docker.settings.HotspotHBondConstraint.create(protein=docker.settings.
↪proteins[0],
                                                                hr=result,
                                                                weight=10,
                                                                min_hbond_score=0.
↪05,
                                                                max_constraints=1)
for constraint in constraints:
    docker.settings.add_constraint(constraint)
```

### 3.3.2 View the Constraints

The automatic hotspot constraints are designed to be used unsupervised, as part of large scale docking studies where it is not practical to assess every protein manually. However, if you are studying a small number of proteins, you may want to view the suggested hydrogen bond constraints before running GOLD docking.

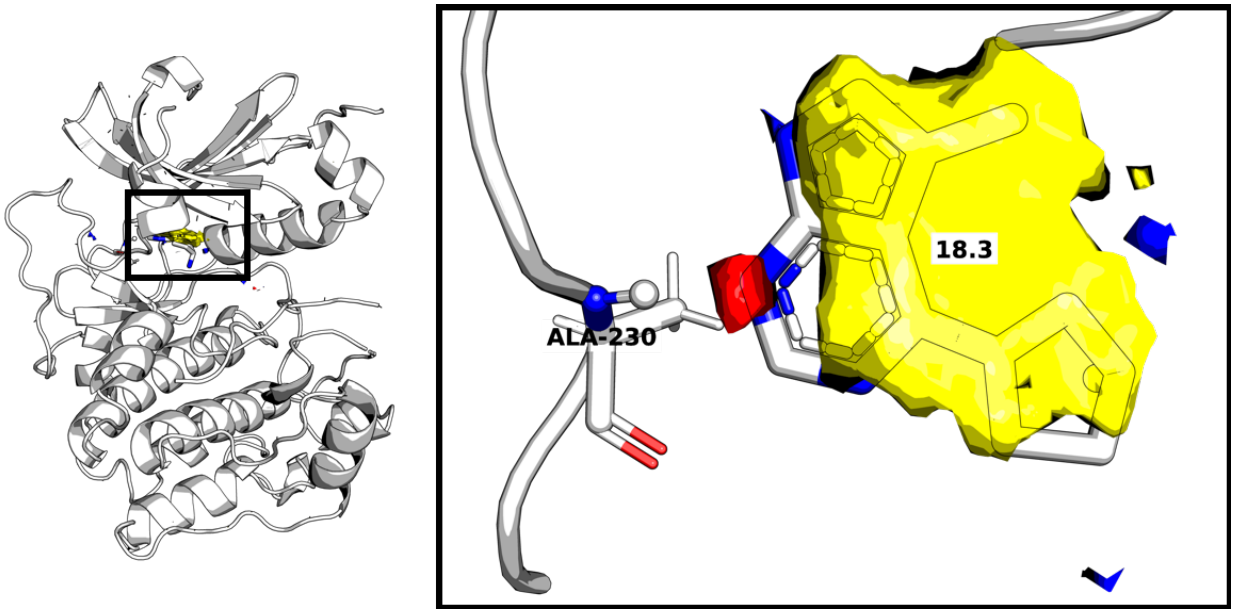
```
hotspot = "<path to hotspot out.zip>"
hotspot = HotspotReader(hotspot).read()

for p, g in hotspot.super_grids.items():
    hotspot.super_grids[p] = g.max_value_of_neighbours()

    # grayscale dilation used for noise reduction

constraints = hotspot._docking_constraint_atoms(max_constraints=5,
                                                 max_distance=4,
                                                 threshold=17,
                                                 min_size=8
                                                 )
```

(continues on next page)

(continued from previous page)

```
# max_constraints: limits the number of constraints selected
# max_distance: island cenrtroid must be within max_distance to be selected
# threshold: hotspots contoured a threshold score
# min_size: island must have > min_size grid points to be selected

mol = constraints.to_molecule()

with MoleculeWriter("constraints.mol2") as w:
    w.write(mol)
```

The docking constraints generated from the hotspots can be converted into a *ccdc.molecule.Molecule* object and then easily visualised in a molecular visualing program. We use PyMOL. In this case, 1 hydrogen bond donor constraint is selected and therefore there is no selection to be made.




### 3.3.3 GOLD Docking

With the modified docking settings class, the rest of the docking calculation is carried out in a similar manner to any other GOLD API docking. A full run script is provided below, which has been adapted from the CCDC API documentation.

https://downloads.ccdc.cam.ac.uk/documentation/API/cookbook_examples/docking_examples.html

```
def dock(inputs):
    """
    submit a GOLD API docking calculation using
    docking constraints automatically generated
    from the Hotspot API

    :param ligand_path:
    :param out_path:
```

(continues on next page)

(continued from previous page)

```
    :param hotspot:
    :param weight:
    :return:
    """
    def add_ligands(docker, ligand_path):

        with gzip.open(os.path.join(ligand_path,
                                    "actives_final.mol2.gz"), 'rb') as f_in:

            with open(os.path.join(docker.settings.output_directory,
                                    "actives_final.mol2"), 'wb') as f_out:
                shutil.copyfileobj(f_in, f_out)

        with gzip.open(os.path.join(ligand_path,
                                    "decoys_final.mol2.gz"), 'rb') as f_in:
            with open(os.path.join(docker.settings.output_directory,
                                   "decoys_final.mol2"), 'wb') as f_out:
                shutil.copyfileobj(f_in, f_out)

        docker.settings.add_ligand_file(os.path.join(docker.settings.output_
↪directory,
                                                     "actives_final.mol2"),
                                        ndocks=5)

        docker.settings.add_ligand_file(os.path.join(docker.settings.output_
↪directory,
                                                     "decoys_final.mol2"),
                                        ndocks=5)

    def add_protein(docker, hotspot, junk):

        pfile = os.path.join(junk, "protein.mol2")
        with MoleculeWriter(pfile) as w:
            w.write(hotspot.protein)

        docker.settings.add_protein_file(pfile)

    def define_binding_site(docker, ligand_path):

        crystal_ligand = MoleculeReader(os.path.join(ligand_path,
                                                     "crystal_ligand.mol2")
                                        )[0]
        docker.settings.binding_site =
        docker.settings.BindingSiteFromLigand(protein=docker.settings.
↪proteins[0],
                                              ligand=crystal_ligand)

    def add_hotspot_constraint(docker, hotspot, weight):

        if int(weight) != 0:
            constraints =
            docker.settings.HotspotHBondConstraint.create(protein=docker.
↪settings.proteins[0],
```

(continues on next page)

(continued from previous page)

```
                                                        hr=hotspot,
                                                        weight=int(weight),
                                                        min_hbond_score=0.
↪05,
                                                        max_constraints=1)

            for constraint in constraints:
                docker.settings.add_constraint(constraint)

    def write(docker, out_path):

        results = Docker.Results(docker.settings)

        # write ligands
        with MoleculeWriter(os.path.join(out_path, "docked_ligand.mol2")) as␣
↪w:
            for d in results.ligands:
                w.write(d.molecule)

        # copy ranking file
        #
        # in this example, this is the only file we use for analysis.
        # However, other output files can be useful.

        copyfile(os.path.join(junk, "bestranking.lst"),
                 os.path.join(out_path, "bestranking.lst"))

    # GOLD docking routine
    ligand_path, out_path, hotspot, weight, search_efficiency = inputs
    docker = Docker()

    # GOLD settings
    docker.settings = DockerSettings()
    docker.settings.fitness_function = 'plp'
    docker.settings.autoscale = search_efficiency
    junk= os.path.join(out_path, "all")
    docker.settings.output_directory = junk

    # GOLD write lots of files we don't need in this example
    if not os.path.exists(junk):
        os.mkdir(junk)
    docker.settings.output_file = os.path.join(junk, "docked_ligands.mol2")

    # read the hotspot
    hotspot = HotspotReader(hotspot).read()
    for p, g in hotspot.super_grids.items():
        hotspot.super_grids[p] = g.max_value_of_neighbours()
        # dilation to reduce noise

    add_ligands(docker,ligand_path)
    add_protein(docker, hotspot, junk)
    define_binding_site(docker, ligand_path)
```

(continues on next page)

(continued from previous page)

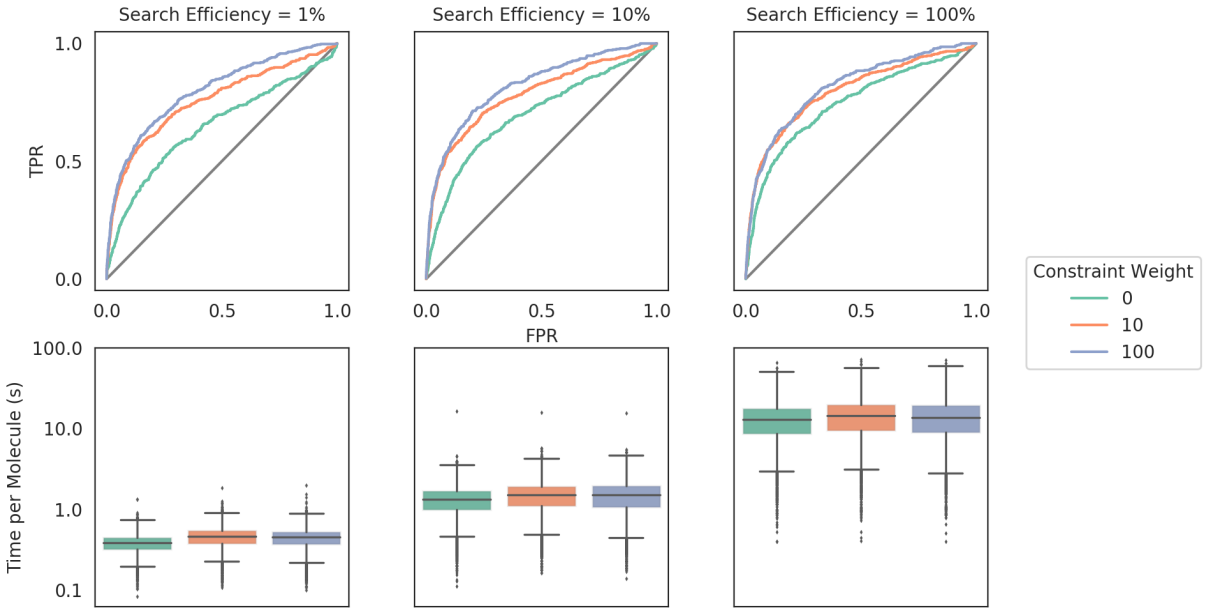
```
    add_hotspot_constraint(docker, hotspot, weight)

    docker.dock(file_name=os.path.join(out_path, "hs_gold.conf"))
    write(docker, out_path)

    # Clean out unwanted files
    shutil.rmtree(junk)


def create_dir(path):
    if not os.path.exists(path):
        os.mkdir(path)
    return path


def main():

    ligand_path = '<path to input directory>'
    output_path =  '<path to output directory>'
    hotspot_path = '<path to out.zip>'
    constraint_weight = 10
    search_efficiency = 100

    dock(inputs=(ligand_path,
                 output_path,
                 hotspot_path,
                 constraint_weight,
                 search_efficiency))
```

### 3.3.4 Performance Demonstration

Using the code above, the AKT1 DUD-e set was docked against AKT1 (PDB: 3cqw), using the hotspot selected constraint (the amide hydrogen of ALA230). The docking calculations were run varying the weight of the protein hydrogen bond constraint [0, 10, 100] and the search efficiency [1, 10, 100].

The most significant effect is on retrieval speed. GOLD can be run using different search efficiencies which control the degree of sampling in the genetic algorithm. By using automated constraints, one can outperform 100% search efficiency results in 1% search efficiency settings; a speed improvement of more than an order of magnitude. While this work showcases this use case, we will undertake further work in future to evaluate the benefit more generally across a wider range of targets.

The code to generate the figure and statistics is given below.




```
import os

import numpy as np
import operator
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from ccdc.descriptors import StatisticalDescriptors as sd

def rank_stats(parent, s, w):
    """
    Create two `pandas.DataFrame` from the "bestranking.lst"
    GOLD output

    :param str parent: path to parent directory
    :param str s: name of subdirectory
    :param str w: name of subsubdirectory
    :return:
    """
    # read data and process data from output file
    fname = os.path.join(parent,
                         f"search_efficiency_{s}",
                         str(w),
                         "bestranking.lst")
    lines = [l.strip("\n") for l in open(fname, "r").readlines()]
    header = lines[5]
    header = [b.strip() for b in
              [a for a in header.split("  ") if a != '' and a != '#']]
    all = lines[7:]

    cat = list(zip(*[[a for a in entry.split(" ") if a != '']
```

(continues on next page)

(continued from previous page)

```
                    for entry in all]))

    # generate a dataframe and alter datatypes
    df = pd.DataFrame({h: cat[i] for i, h in enumerate(header)})
    df["actives"] = np.array(
        list(map(lambda x: 'CHEMBL' in x, list(df['Ligand name'])))
    ).astype(int)
    df["search_efficiency"] = [int(s)] * len(df)
    df["weight_int"] = [int(w)] * len(df)
    df["weight_str"] = [str(w)] * len(df)
    df["Score"] = df["Score"].astype(float)
    df["time"] = df["time"].astype(float)
    df["log_time"] = np.log10(list(df["time"]))
    df = df[['Score',
             'log_time',
             'actives',
             'search_efficiency',
             'weight_int',
             'weight_str']]
    df = df.sort_values(by=['Score'], ascending=False)

    # Use CCDC's descriptors API
    rs = sd.RankStatistics(scores=list(zip(list(df['Score']),
                                           list(df['actives']))),
                           activity_column=operator.itemgetter(1))

    # ROC
    tpr, fpr = rs.ROC()
    df["tpr"] = tpr
    df["fpr"] = fpr

    # Enrichment Metrics
    metric_df = pd.DataFrame({"search efficiency": [s],
                              "weight": [w],
                              "AUC": [rs.AUC()],
                              "EF1": [rs.EF(fraction=0.01)],
                              "EF5": [rs.EF(fraction=0.05)],
                              "EF10": [rs.EF(fraction=0.1)],
                              "BEDROC16": [rs.BEDROC(alpha=16.1)],
                              "BEDROC8": [rs.BEDROC(alpha=8)]
                              })
    return df, metric_df


def roc_plot(df, search_efficiency, ax, palette='Set2'):
    """
    Plot a ROC plot for the docking data

    :param `pands.DataFrame` df: data
    :param int search_efficiency: data is split by search efficiency
    :param `matplotlib.axes.Axes` ax: Matplotlib axis to plot data onto
    :param list palette: list of RGB tuples
```

(continues on next page)

(continued from previous page)

```
    :return:
    """

    selected = df.loc[df['search_efficiency'] == search_efficiency]
    # random
    sns.lineplot(x=[0, 1], y=[0, 1], color="grey", ax=ax)
    # docking rank

    d = {"0": {"color": sns.color_palette(palette)[0],"linestyle": "-"},
         "10": {"color": sns.color_palette(palette)[1],  "linestyle": "-"},
         "100": {"color": sns.color_palette(palette)[2], "linestyle": "-"}}

    lines = [ax.plot(grp.fpr, grp.tpr, label=n, **d[n])[0]
             for n, grp in selected.groupby("weight_str")]


    ax.set_ylabel("")
    ax.set_xlabel("")
    ax.set_xticks([0, 0.5, 1])
    ax.set_title(label=f"Search Efficiency = {search_efficiency}%",
                 fontdict={'fontsize':10})

    return lines


def box_plot(df, search_efficiency, ax, palette='Set2'):
    """
    Plot a boxplot for the docking time data

    :param `pands.DataFrame` df: data
    :param int search_efficiency: data is split by search efficiency
    :param `matplotlib.axes.Axes` ax: Matplotlib axis to plot data onto
    :param list palette: list of RGB tuples
    :return:
    """

    selected = df.loc[df['search_efficiency'] == search_efficiency]
    sns.boxplot(x="weight_int", y="log_time", order=[0,10,100],data=selected,
                palette=palette,linewidth=1.2, fliersize=0.5, ax=ax)
    ax.set_ylabel("")
    ax.set_xlabel("")
    ax.set_xticks([])

    # just for asthetics
    for patch in ax.artists:
        r,g,b,a = patch.get_edgecolor()
        patch.set_edgecolor((r,g,b,.1))


def _asthetics(fig, axs, lines):
    """
    Extra formatting tasks
```

(continues on next page)

(continued from previous page)

```
    :param `matplotlib.figure.Figure` fig: mpl figure
    :param list axs: list of  `matplotlib.axes.Axes`
    :param `matplotlib.lines.Line2D` lines: ROC lines
    :return:
    """
    # format axes
    axs[0][0].set_yticks([0, 0.5, 1])
    yticks = [-1, 0, 1, 2]
    axs[1][0].set_yticks(yticks)
    axs[1][0].set_yticklabels([str(10 ** float(l)) for l in yticks])
    axs[0][1].set_xlabel("FPR")
    axs[0][0].set_ylabel("TPR")
    axs[1][0].set_ylabel("Time per Molecule (s)")
    # format legend
    fig.legend(lines,
               [f"{w}" for w in [0,10,100]],
               (.83, .42),
               title="Constraint Weight")
    # format canvas
    plt.subplots_adjust(left=0.1, right=0.8, top=0.86)

def main():
    # read and format the data
    search_effiencies = [1, 10, 100]
    weights = [0, 10, 100]
    parent = "/vagrant/github_pkgs/hotspots/examples/" \
             "2_docking/virtual_screening/akt1/"

    df1, df2 = zip(*[rank_stats(parent, s, w)
                     for s in search_effiencies for w in weights])

    # Plotted Data (ROC and Box plots)
    df1 = pd.concat(df1, ignore_index=True)

    # Table Data (Rank Stats: AUC, EF, BEDROC)
    df2 = pd.concat(df2, ignore_index=True)
    df2.to_csv('rankstats.csv')

    # Plot the ROC and box plots
    sns.set_style('white')
    fig, axs = plt.subplots(nrows=2,
                            ncols=3,
                            sharey='row',
                            gridspec_kw={'wspace':0.26,
                                         'hspace':0.22},
                            figsize=(10,6), dpi=200)

    for i, row in enumerate(axs):
        for j, ax in enumerate(row):
            if i == 0:
                lines = roc_plot(df1, search_effiencies[j], ax)
```

(continues on next page)

(continued from previous page)

```
            else:
                box_plot(df1, search_effiencies[j], ax)

    _asthetics(fig, axs, lines)
    plt.savefig("new_grid.png")
    plt.close()

if __name__ == "__main__":
    main()
```

## 3.4 Pharmacophores

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
```

The CSD Python API's documentation details how the output a ".cm" file can be used for Pharmacophore searching in CrossMiner. See the link below for details.

https://downloads.ccdc.cam.ac.uk/documentation/API/descriptive_docs/pharmacophore.html

CHAPTER

FOUR

# 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*, *clear_tmp=False*)
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*, *clear_tmp=False*)
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()
```

**from_superstar**(*protein*, *superstar_grids*, *buriedness*, *charged_probes=False*, *probe_size=7*, *settings=None*, *clear_tmp=False*)

calculate hotspot maps from precalculated superstar maps. This enables more effective parallelisation and reuse of object such as the Buriedness grids

**Parameters**

- **protein** – a ccdc.protein.Protein instance
- **superstar_grids** – a hotspots.atomic_hotspot_calculation._AtomicHotspotResult instance
- **buriedness** – a hotspots.grid_extension.Grid 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)
- **settings** – *hotspots.calculation.Runner.Settings* settings: holds the sampler settings
- **clear_tmp** (*bool*) – If True, clear the temporary directory

**Returns**

CHAPTER

FIVE

# 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

SIX

# 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*, *mvon=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)
- **mvon** (*bool*) – Run Max value of neighbours (default = True)

**extract_volume**(*volume='125'*)
Returns a HotspotResult with a restricted volume

**Parameters** **volume** (*int*) – target map volume

**Return hotspots.result.Results*hotspots.result.Results*** A fresh result object

**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

**atomic_volume_overlap**(*mol*)
for a given mol, return a dictionary of dictionaries containing the percentage overlap of each atoms VDW radius with the Hotspot Grids.

{“donor”: {“atomic_label”: percentage_overlap}

**Parameters** **mol** –

**Returns**

**static from_grid_ensembles**(*res_list*, *prot_name*, *charged=False*, *mode='max'*)
*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

**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)

**percentage_matched_atoms**(*mol*, *threshold*, *match_atom_types=True*)
for a given molecule, the 'percentage match' is given by the percentage of atoms which overlap with the hotspot result (over a given overlap threshol)

**Parameters**

- **mol** –
- **threshold** –
- **match_atom_types** –

**Returns**

**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

SEVEN

# 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*, *min_island_size=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*, *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

hotspots.hs_pharmacophore.**tanimoto_dist**(*a*, *b*)

calculate the tanimoto distance between two fingerprint arrays :param a: :param b: :return:

CHAPTER

EIGHT

# 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=10*)

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 constraint weight (default to be determined)
- **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`

**generate_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

NINE

# 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*)

**property x**
Alias for field number 0

**property y**
Alias for field number 1

**property 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_atom_type**(*atom*)
return the atom classification

**Parameters** **atom** –

**Returns**

**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

# PYTHON MODULE INDEX

## h