Chemoinformatics Software Documentation

Denmark Laboratory

Contents

Introduction

Vision

The major goal of the cheminformatics project within the Denmark lab is to provide a framework for catalyst optimization, using quantitative and statistical methods to systematically describe and characterize catalytic systems towards the ability to accurately predict selectivity trends from catalyst structures.

Current catalyst optimization methods are, we believe, inefficient, requiring many rounds of either exhaustive screening or underwhelming "chemical intuition" to guide the selection of catalysts for further rounds of optimization. We seek to remove the exhaustive screening methods and serendipitous discovery of robust, selective, and active catalysts. We want to remove the synthetic overhead through exploration of useful chemical space, and reduce the amount of unnecessary screening.

The research program established within the Denmark lab seeks not to further modeling techniques already perfected by the machine learning community, but to develop a workflow that uses informatics in conjunction with computational chemistry methods to guide the selection of catalyst screening subsets as well as further catalyst optimization.

Cheminformatics

Cheminformatics is the marriage of computational chemistry and informatics techniques. The simplest way to put it is that cheminformatics seeks to use calculated or empirical quantities associated with a population of molecules, and determine what relationship exists between the properties/quantities.

Calculated chemical properties (either fully calculated in the computer or derived from empirical data) are called chemical descriptors. Chemical descriptors can be used to categorize compounds, find relationships in chemical properties that are unobservable to the human eye, and be used to correlate structure/properties with function. It is this latter task that we seek to exploit. Chemical descriptors come in many flavors, but can generally be divided into three categories: One dimensional (1D) descriptors are "whole molecule" descriptors, describing a property of the molecule (e.g., ClogD, number of rotatable bonds); Two dimensional (2D) descriptors, describing structural features of the molecule in terms of connectivity and fragments that comprise the molecule; and finally three

dimensional (3D) descriptors that are calculated using a 3D representation of the molecule species under observation. For enantioselectivity, we feel that 3D descriptors are important for understanding both the origin of stereoselectivity and predicting the outcome of any particular catalyst once a model has been developed, despite the difficulty in calculating such descriptors.

In the past, cheminformatics has mainly been used within the pharmaceutical realm to attempt to correlate calculated properties with activity data. The models generated typically focused on 1D and 2D descriptors due to the overwhelming amount of data generated for each model. In the case of cheminformatics, developing models on less accurate or information-sparse descriptors is easier as the number of samples increases.

Cheminformatics in the Denmark Lab

It is traditional at this point to say that the cheminformatics project in the Denmark lab is based on Quantitative Structure Activity Relationship (QSAR) models. However, the scope with which we use cheminformatics is beyond QSAR, and it is time to recognize that fact. We use cheminformatics for chemical space analysis, diversity analysis and clustering, training set selection, etc. The extent with which cheminformatics is integrated into the research program for the subgroup is much higher than simple addition of QSAR modeling following selectivity/activity screens.

The lab now focuses on exploiting chemical diversity, expanding the chemical space coverage of novel catalyst and ligand scaffolds with the expectation that increasing diversity in chemical properties will allow us to solve problems previously unsolvable with these ligands/catalysts. (See Figure 1).

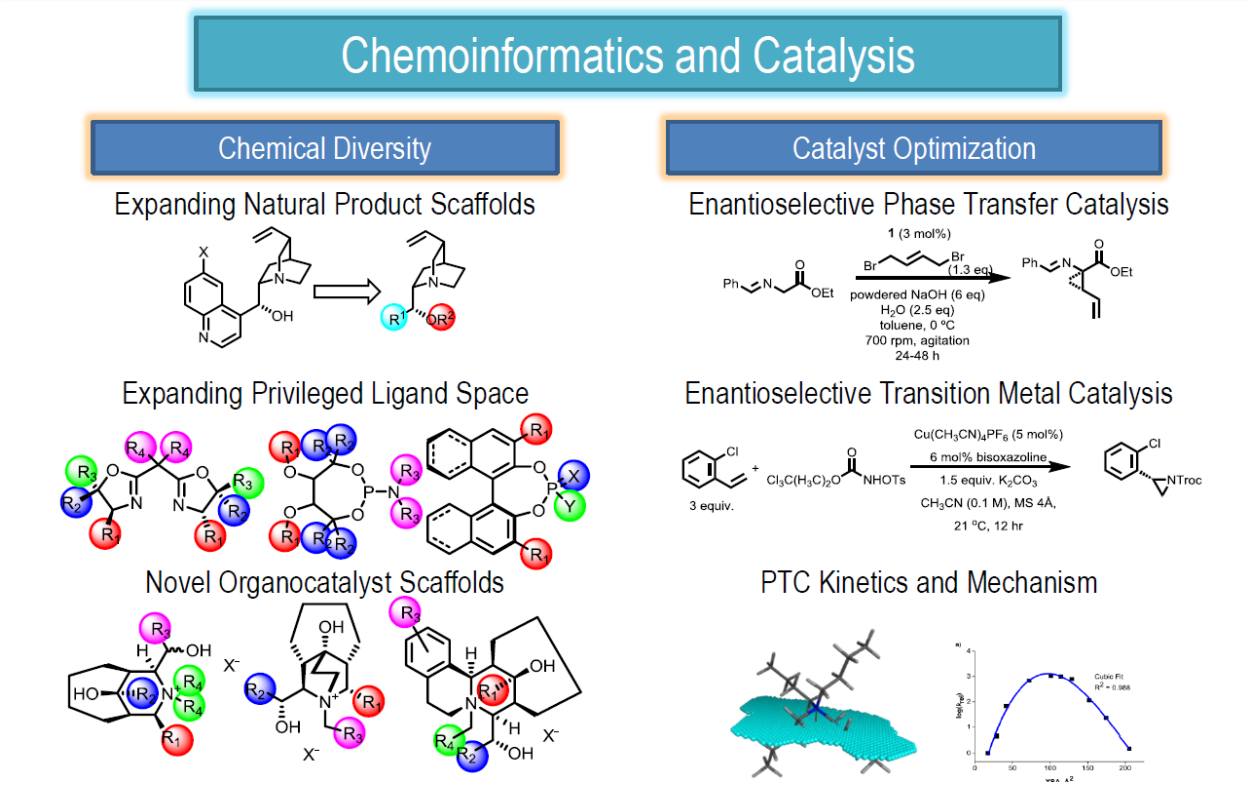


Figure 1. Using chemical diversity to drive catalyst optimization cycles.

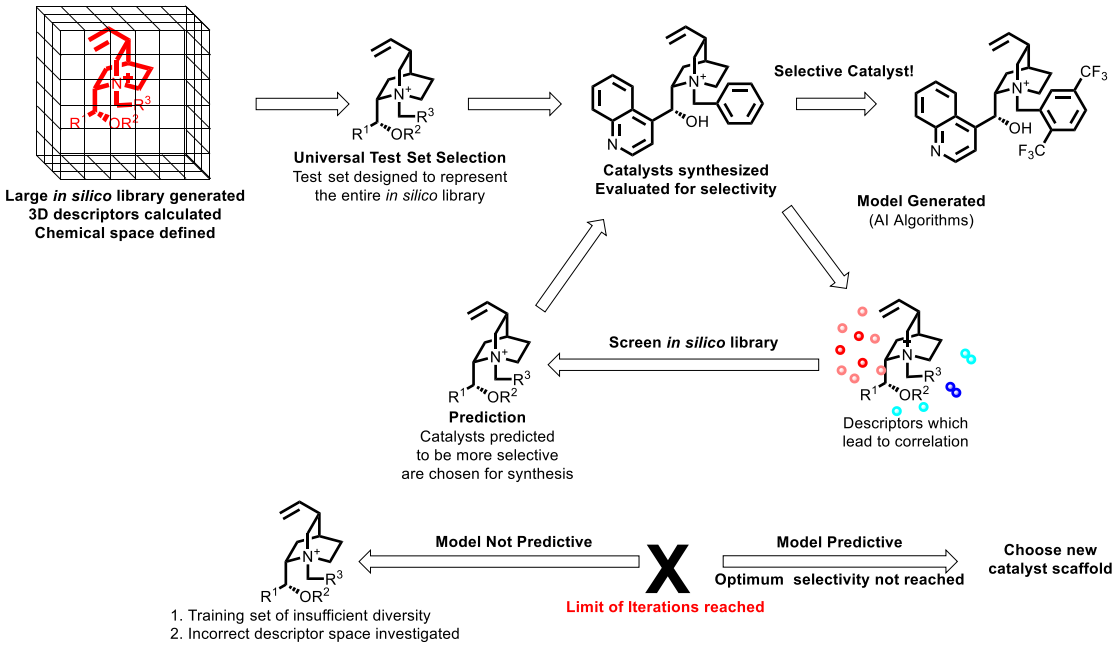
Computational Workflow – Current

The software framework and overall cheminformatics computational workflow, current as of this writing, will be overviewed in this section. The purpose of this section is not to go into detail as to how each piece of the workflow is designed or operated, but merely to provide the big picture of the process.

Generation of an *in silico* Library

The first stage of the workflow is the generation of a library of compounds within the computer. This includes optimizing the geometry of said structures and preparing them for descriptor calculations. This operation is generally performed using in-house developed software, and proceeds through a combinatorial process of attaching different substituent groups at specific attachment points. The *in silico* libraries we generate are thought to contain every possible, **synthesizable** catalyst.

Scheme 1. Cheminformatics Workflow



Descriptor Calculation

Once the *in silico* library is constructed, 3D descriptor calculation commences. Currently, we use DFT to calculate electrostatic potential energy (between a positive point charge and the molecule) and MMFF Van der Waals steric potential energy (between an sp3 carbon and the molecule) at various points in space around the catalyst structure. The grid of points can either be manually specified or a surface-fitted contour grid can be generated using interfaced DFT program NWCHEM. The descriptors are then further processed to grid independent descriptors (GRINDs). These processes and algorithms are discussed in the descriptor calculation and processing chapter.

Chemical Space Analysis – Selection of Universal Training Sets (UTSs)

The calculated descriptors are then used to define a chemical space for the library. As we define which descriptors go into the chemical space and how the defining descriptors are manipulated, we must take

care to ensure we have a meaningful chemical space. This is especially important, as the chemical space is used to determine the training sets of catalysts selected from the full *in silico* library for testing.

This introduces the concept of Universal Training Sets or UTSs. A UTS is a subset of catalysts from any given catalyst *in silico* library that represents the variance and diversity of that library. Essentially, we define a UTS as a representative subset of the full library. The beauty of the concept is that this test set, once synthesized, can be used as an initial optimization starting point for ANY reaction that the training set's base scaffold is competent in. And, it is mathematically, quantitatively guaranteed to have the full breadth of chemical properties that are included in the library.

The idea is not dissimilar from the concept of Design of Experiments. However, we cannot necessarily use the same kind of approach as DoE due to the fact that our variables are not necessarily continuous; we cannot guarantee that any chosen set of descriptors exists within the library. Thus, we use a defined chemical space and distance based metrics to choose our training subset. This will be further discussed in the Universal Training Set selection chapter.

Synthesis, Screening, and Modeling

Once the training set is selected, it is synthesized and readied for testing. Screening is done on appropriate reactions; generally reactions that are known to work with a given ligand/catalyst scaffold and are already optimized in a racemic variant.

Modeling is done using scikit-learn, a python2 machine learning and statistics library. Actual techniques, including cross-validation and estimator choice will be discussed in the Modeling chapter.

Summary

At this point, we have covered the general overview of the cheminformatic workflow and the idea behind this project. The following chapters will go over both the theory and the codebase developed in the Denmark lab, including examples and tutorials. All of the code presented in this manuscript is housed on the group server for anyone's use.

ccheminfolib Documentation

Introduction

This chapter serves as the low-level documentation for features within the ccheminfolib cheminformatics package. The philosophy behind the development of this software package was to develop a modular backend system that can easily and quickly be used in prototype scripting situations as the methodology is developed. Previous incarnations of the Denmark lab cheminformatics software were monolithic, single class conglomerations of descriptor calculation functions that were hard to modify, extend, or reuse. The development of ccheminfolib seeks to fix this problem.

Structure of ccheminfolib

The package ccheminfolib is broken down into three branches: cchemlib, which contains the classes and associated functions for the datatypes used in the package (Points, Atoms, Molecules, etc.) as well as the parser class definitions (used to transfigure computational chemistry files to our datatypes); cdesclib which contains the calculator and controller classes for calculating descriptors and interfacing with computational chemistry programs, respectively; and cqsarlib, which contains centralized code for generating library, database, and modeling objects.

Generally, cchemlib is responsible for constructing everything up to the Molecule datatype level. The next branch, cdesclib calculates the descriptors, and cqsarlib does the modeling. It's that simple.

Dependences

The ccheminfolib package requires python 2.7, and uses NumPy/SciPy and RDKit, as well as scikit-learn. We suggest using Anaconda to install python2.7, as conda and pip together make the installation of these extensions much easier, and in many cases are already installed.

cchemlib Documentation

Datatypes (ccheminfolib.cchemlib.datatypes)

class ccheminfolib.cchemlib.Datatype(self, type=999)

Base abstract class for the datatypes in ccheminfolib.

Parameters:

type : int, type of datatype in use

These are defined as constants. Available values are BASE (999), ATOM (0), BOND (1), MOLECULE (2), DESCRIPTOR (3), GRIDPOINT (4), GRID (5), and POINT (6).

Methods:

None

class ccheminfolib.cchemlib.Point(ID, label, x, y, z)

Defines a single geometric point in 3D space.

Parameters

Label	Type	Description
ID	int	Unique identifier, user controlled
label	string	Unique identifier used in lieu of serial ID "Person friendly"
x	float	Location of the point on X-axis
y	float	Location of the point on Y-axis
z	float	Location of the point on Z-axis

Methods

get_distance_to_point(other) : returns the distance between another Point object and this point object

__sub__(other) : Overloaded subtraction operator, returns distance to other Point object (uses get_distance_to_point())

__eq__(other) : Overloaded '=', returns true if other Point object is at the same location (uses get_distance_to_point())

__init__(ID, label, x, y, z) *See description above*

get_distance_to_point(other)

Returns distance between this point object to another point object.

Parameters

Label	Type	Description
other	ccheminfolib.cchemlib.Point object	Point object to calculate distance from

Returns: (float) Distance between the points, or NotImplemented if 'other' is not a Point object

class ccheminfolib.cchemlib.Atom(ID, label, point, atom_type, formal_charge = 0, mpa_charge = 0.0)

Atom datatype. Molecules are made up of Atoms.

Parameters

Label	Type	Description
ID	int	Unique identifier, user controlled
label	string	Unique identifier used in lieu of serial ID "Person friendly"
point	Point	cchemlib.Point object; defines 3D coordinates of the atom
type	string	Atom type based on Tripos atom types; defined constants in cchemlib.atomtypes
formal_charge	int	Charge based on VSPER rules; not used in many cases
mpa_charge	float	Charge calculated using MPA; not used in many cases

Methods

get_distance_from_atom(other) : returns the distance from another Atom object and this Atom object

__sub__(other) : Overloaded subtraction operator, returns distance to other Point object (uses get_distance_from_Atom())

__init__(ID, label, point,atom_type,formal_charge=0,mpa_charge=0) *See description above*

get_distance_from_atom(other)

Returns distance between this Atom object to another Atom object.

Parameters

Label	Type	Description
other	ccheminfolib.cchemlib.Atom object	Atom object to calculate distance from

Returns: (float) Distance between the points, or NotImplemented if 'other' is not an Atom object

class ccheminfolib.cchemlib.Bond(ID, label, start_atom,end_atom, bond_type)

Bond datatype. Bonds form between Atoms!

Parameters

Label	Type	Description
ID	int	Unique identifier, user controlled
label	string	Unique identifier used in lieu of serial ID "Person friendly"
start_atom	int	ID of the Atom object at the start of the bond
end_atom	int	ID of the Atom object at the end of the bond
bond_type	string	Bond type based on tripos mol2 bond type definitions. Defined in cchemlib.bondtypes

Methods

set_start_atom(start_atom_ID) : Sets the atom ID for the start atom

set_end_atom(end_atom_ID) : Sets the atom ID for the end atom

__init__(ID, label, start_atom, end_atom, bond_type) *See description above*

set_start_atom(start_atom_ID)

Sets the Bond.start_atom variable to a new ID

Parameters

Label	Type	Description
start_atom_ID	int	New ID of the starting atom

Returns: Nothing.

set_end_atom(end_atom_ID)

Sets the Bond.end_atom variable to a new ID

Parameters

Label	Type	Description
end_atom_ID	int	New ID of the ending atom

Returns: Nothing.

class ccheminfolib.cchemlib.Descriptor(type, value)

Descriptor datatype.

Each descriptor type serves a purpose. Grid point descriptors have an ELE and VDW potential energy associated with them. Eventually, GRINDs are calculated from these later on. Not used directly with grids.

Parameters

Label	Type	Description
type	int	One of several defined descriptor types; constants located in cchemlib.datatypes Available values are ELE, VDW, ELE_M, VDW_P, CO. Only ELE and VDW are used for gridpoint descriptors, but full listing used for GRINDs.
value	float	Value of the descriptor (converted automatically to floating point value)

Methods

None.

__init__(type, value) *See description above*

Notes:

class ccheminfolib.cchemlib.Gridpoint(ID, point, descriptors={})

Gridpoint class. A gridpoint consists of a location in 3D space (Point object) and a set of calculated values (descriptors).

Parameters

Label	Type	Description
ID	int	Unique identifier, user controlled
point	Point	cchemlib.Point object; defines 3D coordinates of the atom
descriptors	dict	Dictionary of descriptors. Keys are descriptor types. One descriptor per type.

Methods

add_descriptor(type, descriptor) : Sets the current descriptor for a given type.
get_distance_from_gridpoint(other) : Returns the distance from another Gridpoint object
get_distance_from_point(other) : Returns the distance from a Point object to this Gridpoint
__sub__(other) : Overloaded subtraction operator, returns distance to other Gridpoint or Point object (uses get_distance_from_gridpoint() and get_distance_from_point ())
__eq__(other) : Overloaded equality operator, returns True if other is a Gridpoint object and the distance between this Gridpoint and other is <0.0001 units.
__str__() : Overloaded string operator. Returns a string comprised of the ELE and VDW type descriptors and the x,y,z coordinates. If any value is missing, it is skipped in the generation of the string.

__init__(ID, point, descriptors={}) *See description above*

add_descriptor(type,descriptor)

Returns distance between this Atom object to another Atom object.

Parameters

Label	Type	Description
type	int	Defined type of descriptor as described in cchemlib.Descriptor
descriptor	cchemlib.Descriptor	cchemlib.Descriptor object, instantiated with values.

Returns: Nothing.

get_distance_from_gridpoint(other)

Returns distance between this Gridpoint object to another Gridpoint object.

Parameters

Label	Type	Description
other	ccheminfolib.cchemlib.Gridpoint	Gridpoint object to calculate distance from

Returns: (float) Distance between the Gridpoints, or NotImplemented if 'other' is not a Gridpoint object

get_distance_from_point(other)

Returns distance between this Gridpoint object to another Gridpoint object.

Parameters

Label	Type	Description
other	ccheminfolib.cchemlib.Point	Point object to calculate distance from

Returns: (float) Distance between the points, or NotImplemented if 'other' is not a Point object

class ccheminfolib.cchemlib.Molecule(label, charge=0, multiplicity=1)

Molecule datatype.

In the real world, molecules are entities comprised of groups of atoms tethered by bonds. In the computer, the Molecule datatype contains the atoms and bonds that comprise the molecule maintained in a wrapper that the computer can manipulate. The molecule datatype also contains the gridpoints (grid), which contains the descriptors for the molecule. From the fields of the grid, the GRIND can be calculated further on. We put placeholders for everything.

Parameters

Label	Type	Description
label	string	Unique identifier, user controlled
charge	int	Overall formal charge of the molecule. No charge is the default
multiplicity	int	Multiplicity of the molecule. Singlet state is the default.

Methods

add_atom(atom) : adds an Atom object to the Molecule

add_bond(bond) : adds a Bond object to the Molecule

add_raw_bond(start_atom_id, end_atom_id, bond_type) : Constructs a Bond object and adds to Molecule.

remove_atom(atom_ID) : removes Atom from the molecule, cleans up any bonds involved with the atom

change_atom_id(old_id, new _id) : attempts to change the ID of a specified Atom object.

change_atom_coord(atom_ID, x, y, z) : attempts to change the 3D coordinates of a specified Atom

generate_grid(spacing=1.0, force_origin=False, forced_origin=[0.,0.,0.], grid_overwrite=False) : Generates a spherical grid around the molecule with a specified point spacing.

set_formal_charge(charge) : Sets the formal charge of the molecule

get_formal_charge() : Returns the current formal charge setting. Default is 0 (integer)

set_gridpoint_descriptor(gridpoint_ID, descriptor_type, descriptor) : Attempts to set a descriptor value for the given gridpoint and descriptor type

set_gridpoint_descriptors(gridpoint_ID, descriptors) : Attempts to attach a dict of descriptors to a specific Gridpoint

get_gridpoint_descriptors(gridpoint_ID) : Attempts to return the descriptors of a particular Gridpoint

set_atom_MPA_charge(atom_ID, charge) : Sets the charge based on Mulliken Population Analysis (MPA)

write_mol2(filename) : Writes the Molecule to a file as a Tripos mol2 filetype (text file).

determine_if_intersect(ring_atoms, bond) : Determines if the given Bond object intersects the atoms defined by the set ring_atoms. Note: Not generally used as a member function, should probably be defined private.

check() : Checks the integrity of the molecule (mainly looking for ring/bond intersections).

__init__(label, charge=0, multiplicity=1) *See description above*

add_atom(atom)

Adds an Atom object to the molecule. Requires a fully instantiated Atom object to the molecule. Note: Overrides the Atom.ID value to match the next ID available in the Molecule object for bookkeeping purposes.

Parameters

Label	Type	Description
atom	cchemlib.Atom	Fully instantiated Atom object

Returns : cchemlib.datatypes.SUCCESS if successful, cchemlib.datatypes.FAIL if the addition is unsuccessful. Note: Should not fail EVER!

add_bond(bond)

Adds a Bond object to the molecule. Requires a fully instantiated Bond object to the molecule. Note: Overrides the Bond.ID value to match the next ID available in the Molecule object for bookkeeping purposes.

Parameters

Label	Type	Description
bond	cchemlib.Bond	Fully instantiated Bond object

Returns : cchemlib.datatypes.SUCCESS if successful, cchemlib.datatypes.FAIL if the addition is unsuccessful.

add_raw_bond(start_atom_id, end_atom_id, bond_type)

Adds a Bond object to the molecule. Builds a Bond object using data provided. Note: Sets the Bond.ID value to match the next ID available in the Molecule object for bookkeeping purposes.

Parameters

Label	Type	Description
start_atom_id	int	Atom.ID of the starting atom
end_atom_id	int	Atom.ID of the ending atom
bond_type	string	Type of bond. Defined in bondtypes.py

Returns : cchemlib.datatypes.SUCCESS if successful, cchemlib.datatypes.FAIL if the addition is unsuccessful.

remove_atom(atom_ID)

Removes an Atom from the Molecule. This function also fixes any changes to the Atom ID and Bond ID based on the atom's removal from the order; this is necessary for bookkeeping.

Parameters

Label	Type	Description
atom_ID	int	Atom.ID of the atom to remove

Returns : cchemlib.datatypes.SUCCESS if successful, cchemlib.datatypes.FAIL if the addition is unsuccessful.

change_atom_id(old_id, new_id)

Changes an atom's Atom.ID and fixes the Bonds involved to match that ID.

Parameters

Label	Type	Description
old_id	int	Current Atom.ID to be changed
new_id	int	New Atom.ID

Returns : cchemlib.datatypes.SUCCESS if successful, cchemlib.datatypes.FAIL if the addition is unsuccessful.

change_atom_coord(atom_ID, x,y,z)

Changes the x,y,z coordinates of a specific atom

Parameters

Label	Type	Description
atom_ID	int	ID of the Atom to have the coordinates changed
x	float	New coordinate in the X-dimension
y	float	New coordinate in the Y-dimension
z	float	New coordinate in the Z-dimension

Returns : cchemlib.datatypes.SUCCESS if successful, cchemlib.datatypes.FAIL if the addition is unsuccessful.

generate_grid(spacing=1.0, force_origin=False, forced_origin = [0.,0.,0.], grid_overwrite=False)

Creates a spherical grid of Gridpoints based on location of the molecule in 3D space. Can either generate the origin automatically or have an origin set by the user using the appropriate flags.

Parameters

Label	Type	Description
spacing	float	Distance between gridpoints; default is 1.0 Å
force_origin	boolean	Flag for designating an origin manually or automatically
forced_origin	list of floats	If force_origin=True, a set of coordinates must be given. Defaults to [0.,0.,0.]
grid_overwrite	boolean	If set to True, will not overwrite a grid that is already defined. Default value is False.

Returns : cchemlib.datatypes.SUCCESS if successful, cchemlib.datatypes.FAIL if the addition is unsuccessful.

set_formal_charge(charge)

Sets the formal charge of the Molecule object. Cannot fail.

Parameters

Label	Type	Description
charge	int	Charge of the molecule. Must be an integer.

Returns : Nothing.

get_formal_charge()

Returns the formal charge of the Molecule object. Cannot fail.

Parameters

None

Returns : (int) Formal charge of the Molecule.

set_gridpoint_descriptor(gridpoint_ID, descriptor_type, descriptor)

Adds a Descriptor to a Gridpoint. If the descriptor type is already set for this point, it will override the original Descriptor object of that type.

Parameters

Label	Type	Description
gridpoint_ID	int	ID of the Gridpoint to set the Descriptor for
descriptor_type	int	Descriptor type as described in cchemlib.datatypes
descriptor	cchemlib.Descriptor	Instantiated Descriptor type to add to the Gridpoint

Returns : cchemlib.datatypes.SUCCESS if successful, cchemlib.datatypes.FAIL if unsuccessful.

set_gridpoint_descriptors(gridpoint_ID, descriptors)

Overwrites the Molecules.descriptors dictionary. Does not check if the passed descriptors variable is a dictionary of Descriptor objects. Generally not used by the user, but as a helper function in the Calculator/Controller/Parser classes.

Parameters

Label	Type	Description
gridpoint_ID	int	ID of the Gridpoint to set the Descriptor for
descriptors	dict of cchemlib.Descriptor objects	Dictionary object of Descriptor objects. Does not check, assumes they are given .

Returns : Nothing

set_atom_MPA_charge(atom_ID, charge)

Sets an atoms MPA charge

Parameters

Label	Type	Description
atom_ID	int	ID of the Atom to change the charge for
charge	float	Charge of the atom based on Mulliken Population Analysis

Returns : cchemlib.datatypes.SUCCESS if successful, cchemlib.datatypes.FAIL if unsuccessful.

write_mol2(filename)

Writes a tripos mol2 file (text) for the Molecule object.

Parameters

Label	Type	Description
filename	string	Full path to the file. Assumes '.mol2' is part of the string.

Returns : cchemlib.datatypes.SUCCESS if successful, cchemlib.datatypes.FAIL if unsuccessful.

determine_if_intersect(ring_atoms, bond)

Determines if the given bond intersects a ring defined by the atoms in the list of Atom.ID supplied.

Parameters

Label	Type	Description
ring_atoms	list of int	Atom.ID of atoms comprising a ring
bond	cchemlib.Bond	Bond object

Returns : (boolean) returns True if there is an intersect, False if none is detected.

check()

Molecular integrity check. Determines if there are any ring/bond intersections.

Parameters

None

Returns : cchemlib.datatypes.SUCCESS if Molecule passes the check, cchemlib.datatypes.FAIL if it does not pass the check. Note: Does not FIX problems.

Notes on the Molecule Object:

Parsers (ccheminfolib.cchemlib.parse)

class ccheminfolib.cchemlib.Parser(filename)

Parser superclass. All parsers inherit from this class. Contains some basic conversion functions for unit conversions.

Parameters

Label	Type	Description
filename	string	Full path to the file that's going to be parsed by the Parser

Methods

hartrees_to_kcal (energy) : converts energy from hartrees to kcal/mol
bohr_to_angstrom(coord) : converts a bohr coordinate to angstrom
print_file_open_error() : Prints an error message if the file cannot be opened.

__init__(filename) *See description above*

hartress_to_kcal(energy)

Converts the given energy to kcal/mol from hartrees.

Parameters

Label	Type	Description
energy	float	Energy value given in hartrees

Returns : (float) Energy in kcal/mol

bohr_to_angstrom(coord)

Converts given coord from bohr to angstrom.

Parameters

Label	Type	Description
coord	float	Coordinate in a single dimension in bohr units

Returns : (float) Coordinate in angstroms

print_file_open_error()

Prints "ERROR: Could not open file " + self.filename

Parameters

None

Returns : None

class ccheminfolib.cchemlib.mol2Parser(filename, label)

Class for parsing tripos mol2 file format. Constructs a Molecule from a mol2 file.

Parameters

Label	Type	Description
filename	string	Full path to the file that's going to be parsed by the Parser
label	string	Label for the molecule. Generally used for naming its file later on. Use serial number!!!!!!!

Methods

parse() : Parses the mol2 file, separating file lines into different categories for further parsing

molecule() : Creates a Molecule object based on the file that has been parsed

__init__(filename) *See description above*

parse()

Reads the given mol2 file and categorizes the file lines into different components: 1) Header, 2) Atom lines, and 3) Bond lines. This is the first pass for interpreting the file, and is the parsing step.

Parameters

None

Returns : (int) cchemlib.datatypes.SUCCESS if the MOL2 file structure was correct and properly parsed, cchemlib.datatypes.FAIL if parsing fails. Always be sure to check for SUCCESS.

molecule()

Takes the parsed MOL2 file and creates a Molecule object based on the information contained in the MOL2 file. Note: Many ccheminfolib classes that construct Molecule objects have molecule() functions that return the Molecule object associated with the class.

Parameters

None

Returns : (cchemlib.Molecule) Instantiated Molecule object on success, cchemlib.datatypes.FAIL if a Molecule object could not be constructed.

class ccheminfolib.cchemlib.respParser(filename)

This parser works on RESP files output by Jaguar during ESP grid calculations. The ultimate goal of this class is to return a dictionary of Gridpoint objects with ESP descriptors instantiated.

Parameters

Label	Type	Description
filename	string	Full path to the file that's going to be parsed by the Parser

Methods

convert_single_line(line, ID) : converts a single line of the file to a Gridpoint with ESP descriptor; returns the Gridpoint object.

parse() : Opens the RESP file and converts the data to properly labeled and converted Gridpoint objects

__init__(filename) *See description above*

convert_single_line(line, ID)

Takes a single line of the RESP file and converts it to an instantiated Gridpoint object. Returns the Gridpoint object on success. Note: Generally not called by the user.

Parameters

Label	Type	Description
line	string	Line from RESP file. '\n' terminated.
ID	int	Gridpoint.ID supplied by the caller. Generally controlled in parse().

Returns : (cchemlib.Gridpoint) Instantiated Gridpoint object with the ESP descriptor populated.

parse()

Parses the RESP file, converting all data to a dictionary of Gridpoints intended to populate the Molecules descriptor field.

Parameters

None

Returns : (dict of cchemlib.Gridpoints) Dictionary of instantiated Gridpoint objects (dict keys match Gridpoint.ID) on success, cchemlib.datatypes.FAIL if the parser fails.

class ccheminfolib.cchemlib.nwXYZParser(filename, mol)

This parser works on XYZ files output by NWCHEM during the ESP calculation. For whatever reason, NWCHEM gently massages the location of the molecule in 3D space, and so we have to adjust our Molecule objects to match so the descriptors are in the right location, relative to the Molecule. This parser first parses the file, then applies the correction to the Molecule object.

Parameters

Label	Type	Description
filename	string	Full path to the file that's going to be parsed by the Parser
mol	cchemlib.Molecule	Instantiated Molecule object that needs coordinate conversion.

Methods

convert_single_line(line, ID) : Takes a single parsed line of the XYZ file and modifies the specific Atom of the Molecule that the line pertains to.

parse() : Opens the XYZ file and converts the data to properly labeled strings for use in molecule() and convert_single_line()

molecule() : Modifies the associated Molecule object (mol) to the correct coordinates and returns a deep copy of the Molecule object.

__init__(filename) *See description above*

convert_single_line(line, ID)

Takes a single parsed line of the XYZ file and modifies the specific Atom of the Molecule that the line pertains to.

Parameters

Label	Type	Description
line	string	Line from XYZ file. '\n' terminated.
ID	int	Atom.ID supplied by the caller. Controlled by molecule()

Returns : (int) cchemlib.datatypes.SUCCESS if the Atom coordinates are successfully changed, cchemlib.datatypes.FAIL if not.

parse()

Parses the XYZ file. Sorts the lines into an internal list of lines, ensures proper ordering.

Parameters

None

Returns : : (int) cchemlib.datatypes.SUCCESS if the file is properly read and parsed, cchemlib.datatypes.FAIL if not.

molecule()

Modifies the associated Molecule object (mol) to the correct coordinates and returns a deep copy of the Molecule object.

Parameters

None

Returns : (cchemlib.Molecule) Deep copy of the Molecule object with modified Atom coordinates if parsing/conversion is successful. Returns cchemlib.datatypes.FAIL if not successful.

class ccheminfolib.cchemlib.nwGridParser(filename, leveling=True)

This parser works on GRID files output from NWCHEM after an ESP calculation. Similar to respParser, this class constructs a dictionary of Gridpoint objects with the ESP descriptor populated. Note: MUST BE USED AFTER nwXYZParser has modified the parent Molecule object, or else the Gridpoint dictionary coordinates will not match with the Atom coordinates!

Parameters

Label	Type	Description
filename	string	Full path to the file that's going to be parsed by the Parser
leveling	boolean	Flags whether or not to force leveling of gridpoint energies to 30 kcal/mol. Defaults to True.

Methods

convert_single_line(line, ID) : converts a single line of the file to a Gridpoint with ESP descriptor; returns the Gridpoint object.

parse() : Opens the GRID file and converts the data to properly labeled and converted Gridpoint objects

__init__(filename) *See description above*

convert_single_line(line, ID)

Takes a single line of the GRID file and converts it to an instantiated Gridpoint object. Returns the Gridpoint object on success. Note: Generally not called by the user.

Parameters

Label	Type	Description
line	string	Line from GRID file. '\n' terminated.
ID	int	Gridpoint.ID supplied by the caller. Generally controlled in parse().

Returns : (cchemlib.Gridpoint) Instantiated Gridpoint object with the ESP descriptor populated.

parse()

Parses the GRID file, converting all data to a dictionary of Gridpoints intended to populate the Molecules descriptor field.

Parameters

None

Returns : (dict of cchemlib.Gridpoints) Dictionary of instantiated Gridpoint objects (dict keys match Gridpoint.ID) on success, cchemlib.datatypes.FAIL if the parser fails.

cdesclib Documentation

Calculators (ccheminfolib.cdesclib.Calculator)

class ccheminfolib.cdesclib.Calculator(type, molecule)

Calculator superclass. These classes are responsible for calculating descriptors based on given data.

Parameters

Label	Type	Description
type	int	Type of descriptor calculated. Currently use cchemlib.datatypes.VDW or ELE
molecule	cchemlib.Molecule	Instantiated Molecule object.

Methods

molecule () : Returns the Molecule object associated with the calculator in its current state.

__init__(type, molecule) *See description above*

molecule ()

Returns the associated Molecule object in its current state. Can be called before or after a calculation is performed

Parameters

None

Returns : (cchemlib.Molecule) Returns the Molecule object. Cannot fail unless associated Molecule object is explicitly deleted.

class ccheminfolib.cdesclib.vdWCalculator(molecule,probe_atom_type="C.2", leveling=True)

This calculator calculates the MMFF94x approximation steric potential energy between a probe atom and the molecule at each location in the currently defined grid. The provided Molecule object must have a set of Gridpoints already defined for it prior to the calculation.

Parameters

Label	Type	Description
molecule	cchemlib.Molecule	Instantiated Molecule object with defined set of Gridpoints
probe_atom_type	const string	Atom type of the probe atom. Constants defined in cchemlib.atomtypes. Defaults to atomtypes.C_SP2 ("C.2")
leveling	boolean	Flag for descriptor energy leveling. Defaults to True

Methods

calculate_vdw_between_atoms(atom1, atom2) : Calculates the VDW PE between two Atom objects
calculate_for_single_gridpoint(gridpoint_ID) : Calculates the VDW PE between all Atoms and a point.
calculate() : Calculates the molecular VDW PE interaction between all Gridpoints and Atoms.
molecule() : Returns a deep copy of the associated Molecule object. Implemented in the inherited class.

__init__(molecule,probe_atom_type="C.2", leveling=True) *See description above*

calculate_vdw_between_atoms(atom1, atom2)

Calculates the VDW approximation PE between two Atom objects. Generally, one atom is the probe atom located at a Gridpoint object location, and the other is an Atom object of the Molecule. Not generally called by the user (it is a helper function) but left non-private for testing/research purposes.

Parameters

Label	Type	Description
atom1	cchemlib.Atom	Instantiated Atom object
atom2	cchemlib.Atom	Instantiated Atom object

Returns : (float) Calculated VDW PE between the Atom objects. Based on MMFF94x force field.

calculate_for_single_gridpoint(gridpoint_ID)

Calculates the summed VDW PE between a probe atom located at the Gridpoint.Point and the Atoms of the Molecule.

Parameters

Label	Type	Description

gridpoint_ID	int	ID of the Gridpoint object.

Returns : (float) Summed VDW PE descriptor for the given Gridpoint location.

calculate()

Calculates the VDW PE descriptors for the entire Molecule based on the defined Gridpoint set.

Parameters

None

Returns : (int) cchemlib.datatypes.SUCCESS if the calculation succeeds for all Gridpoints, cchemlib.datatypes.FAIL if not.

class ccheminfolib.cdesclib.GRINDCalculator(molecule, bin_min=2.0, num_nodes=4, num_points=100, bins=20, bin_size=1.5, num_stddev=2.0, stddev = 6.0, get_stddev=False)

This calculator uses the VDW and ESP descriptors at the Gridpoints to calculate Grid INdependent Descriptors (GRINDs). Note: For the method listing in GRIND type calculators, while all methods are included for development/research understanding, many are helper functions and SHOULD NOT be called in a main script. These are marked as such with "Generally not called by the user." messages. This calculator is the basic GRIND calculation class; specialized GRIND calculators inherit from this class.

Parameters

Label	Type	Description
molecule	cchemlib.Molecule	Instantiated Molecule object with defined set of Gridpoints
bin_min	float	Starting point for binning distances. Default to 2.0 Å
num_nodes	int	Number of anchor points to use when determining the GRIND point population (GPP). Default set to 4.
num_points	int	Number of GPP to populate for each field type. Default set to 100
bins	int	Number of bins to use. bin_min+bin_size*bins = max distance. Default set to 20 bins.
bin_size	float	"Width" of each bin. Default set to 1.5 Å, but smaller values generally used.
num_stddev	float	Number of std. deviations of distance to spread out anchor nodes. Default set to 2.0
stddev	float	Std. deviation of point-point pair distances. Usually approximated from a random sample of a library. Default set to 6.0 Å
get_stddev	boolean	Flag that signals whether or not the std. deviation of PPP distances should be calculated. Default set to False

Methods

parse_fields() : Parses the Gridpoint descriptor dictionary into 4 separate fields, ordering based on descriptor values. Creates four separate fields, two each for ELE and VDW.

parse_nodes(field_type) : Determines the anchor nodes for a specific field type.

get_grind_points(field_type): Determines the GRIND Point Population (GPP) for a given field type. Requires that parse_fields() and parse_nodes() have been run.

calculate_grind_for_bin(field_type, bin_num) : Calculates the GRIND descriptor for a given bin for a given field type.

calculate_grind_for_field(field_type, abs_flag=False) : Calculates the GRIND descriptors for a given field

use_full_field_points(field_type) : Calculates GRIND for a field using all descriptors in said field. Very computationally intensive, does not require anchor nodes be defined.
calculate(use_anchors=True) : Calculates the full GRIND descriptor set for the given Molecule object. Flag specifies whether or not anchor nodes are required to determine GPP.
molecule() : Returns a deep copy of the associated Molecule object. Implemented in the base Calculator class.

__init__(molecule, bin_min=2.0, num_nodes=4, num_points=100, bins=20, bin_size=1.5, num_stddev=2.0, stddev = 6.0, get_stddev=False) *See description above*

parse_fields()

Parses the Gridpoint descriptor dictionary into 4 separate fields, ordering based on descriptor values. Creates four separate fields, two each for ELE and VDW. The fields differ in the ordering of the points being from the most negative to most positive, or vice versa. **Not generally called by the user (it is a helper function) but left non-private for testing/research purposes.**
Parameters
None
Returns : (int) cchemlib.datatypes.SUCCESS if the field parsing succeeds, cchemlib.datatypes.FAIL if not.

parse_nodes(field_type)

Determines the anchor nodes for a specific field type. **Not generally called by the user (it is a helper function) but left non-private for testing/research purposes.**
Parameters

Label	Type	Description
field_type	int	Field type; types defined cchemlib.datatypes (ELE, ELE_M, VDW, VDW_P)

Returns : (int) cchemlib.datatypes.SUCCESS if the parsing succeeds, cchemlib.datatypes.FAIL if not.

get_grind_points(field_type)

Determines the GRIND Point Population (GPP) for a given field type. Requires that parse_fields() and parse_nodes() have been run. **Not generally called by the user (it is a helper function) but left non-private for testing/research purposes.**
Parameters

Label	Type	Description
field_type	int	Field type; types defined cchemlib.datatypes (ELE, ELE_M, VDW, VDW_P)

Returns : (int) cchemlib.datatypes.SUCCESS if the parsing succeeds, cchemlib.datatypes.FAIL if not.

calculate_grind_for_bin (field_type, bin_num)

Calculates the GRIND descriptor for a given bin for a given field type. **Not generally called by the user (it is a helper function) but left non-private for testing/research purposes.**
Parameters

Label	Type	Description
field_type	int	Field type; types defined cchemlib.datatypes (ELE, ELE_M, VDW, VDW_P)
bin_num	int	Bin ID for which to calculate the GRIND for

Returns : (int) cchemlib.datatypes.SUCCESS if the calculation succeeds, cchemlib.datatypes.FAIL if not.

calculate_grind_for_field (field_type, abs_flag=False)

Calculates the GRIND descriptors for a given field. Can use the signed values or absolute values of descriptors for this calculation. **Not generally called by the user (it is a helper function) but left non-private for testing/research purposes.**

Parameters

Label	Type	Description
field_type	int	Field type; types defined cchemlib.datatypes (ELE, ELE_M, VDW, VDW_P)
abs_flag	boolean	Flag for using absolute values instead of signed values of descriptors. Defaults to False.

Returns : (int) cchemlib.datatypes.SUCCESS if the parsing succeeds, cchemlib.datatypes.FAIL if not.

use_full_field_points(field_type)

Calculates the GRIND descriptors for a given field using all the points of the field. For large molecules, this takes a LONG time to calculate, and it is not parallelized. Not recommended for anything other than substituent investigations. No abs_flag. **Not generally called by the user (it is a helper function) but left non-private for testing/research purposes.**

Parameters

Label	Type	Description
field_type	int	Field type; types defined cchemlib.datatypes (ELE, ELE_M, VDW, VDW_P)

Returns : (int) cchemlib.datatypes.SUCCESS if the calculation succeeds, cchemlib.datatypes.FAIL if not.

calculate(use_anchors=True)

Calculates the GRIND descriptors for the associated Molecule.

Parameters

Label	Type	Description
use_anchors	boolean	Flag for using anchors or full field points. Defaults to True (recommended!)

Returns : (int) cchemlib.datatypes.SUCCESS if the calculation succeeds, cchemlib.datatypes.FAIL if not.

Controllers (ccheminfolib.cdesclib.Controller)

These classes are responsible for interfacing with computational chemistry programs. Generally this involves code to write input files, but most classes are provided functions to actually launch jobs given the proper permissions and programs installed. In most cases, job files are written by the controller and the user is directed to create scripts for launching jobs on the cluster as of this writing. There are initial plans on writing either a web backend or some other script set for doing the job launch directly using these Controllers, but at the time of this writing we rely on outside scripts for that.

class ccheminfolib.cdesclib.Controller(type)

Controller superclass.

Parameters

Label	Type	Description
type	int	Signals the program this class interfaces with.

Methods

None

__init__(type) *See description above*

class ccheminfolib.cdesclib.NWChemController(molecule, functional='b3lyp', basis='6-311G**', molchg=0, geomopt=False)

This controller interfaces with the NWCHEM program on the cluster or on a private workstation to run ESP calculation jobs. Currently only writes the nwchem input file (.nw file). Note: For the method listing in Controllers, while all methods are included for development/research understanding, many are helper functions and SHOULD NOT be called in a main script. These are marked as such with "Generally not called by the user." messages. This calculator is the basic GRIND calculation class; specialized GRIND calculators inherit from this class.

Parameters

Label	Type	Description
molecule	cchemlib.Molecule	Instantiated Molecule object with defined set of Gridpoints
functional	string	DFT/HF functional to use in the SCF calculation Default: b3lyp
basis	string	Basis set to use in the SCF calculation. Default: 6-311G**
molchg	int	Formal charge of the input molecule. Default: 0
geomopt	boolean	Flag for geometry optimization. Generally not used (long calc time). Default: False

Methods

write_nw_file() : Writes the .nw file for the ESP calculation.

__init__(molecule, functional='b3lyp', basis='6-311G', molchg=0, geomopt=False)** *See description above*

write_nw_file(filename, scratch_dir, permanent_dir)

Writes the .nw file given the parameters supplied in the initialization of the Controller.

Parameters

Label	Type	Description
filename	string	Full path filename to where the NW file should be written.
scratch_dir	string	Full path to the scratch directory for the job for NWCHEM. Note: Take into account this is defined based on what system the jobs are run on.
permanent_dir	string	Location of the input files on the system the jobs will be run on.

Returns: (int) cchemlib.datatypes.SUCCESS if file write succeeds, cchemlib.datatypes.FAIL if not.

class ccheminfolib.cdesclib.JaguarController(molecule, operator='B3LYP', basis='3-21G*', molchg=0)

This controller interfaces with the Jaguar program on the cluster or on a private workstation to run ESP calculation jobs. Currently writes the jaguar input file (.in file) and the grid specification file (generally .dat file). This controller has functions for launching Jaguar but these are rarely used due to desired batch running. Note: For the method listing in Controllers, while all methods are included for development/research understanding, many are helper functions and SHOULD NOT be called in a main script. These are marked as such with "Generally not called by the user." messages. This calculator is the basic GRIND calculation class; specialized GRIND calculators inherit from this class.

Parameters

Label	Type	Description
molecule	cchemlib.Molecule	Instantiated Molecule object with defined set of Gridpoints
operator	string	DFT/HF functional to use in the SCF calculation Default: B3LYP
basis	string	Basis set to use in the SCF calculation. Default: 3-21G*
molchg	int	Formal charge of the input molecule. Default: 0

Methods

write_grid_file(grid_directory=os.getcwd()+'\\', write_directory=os.getcwd()+'\\') : Writes the custom grid coordinates to a text file for the ESP calculation.

write_in_file(write_directory=os.getcwd()+'\\', directory=os.getcwd()+'/') : Writes the jaguar input file (.in file) for the ESP calculation.

run_job(command_options, command_string='/share/apps/Schrodinger2014-4/jaguar', wait=True) : Launches jobs based on the command_string and command_options provided.

__init__(molecule, operator='B3LYP', basis='3-21G*', molchg=0) *See description above*

write_grid_file(grid_directory=os.getcwd()+'\\', write_directory=os.getcwd()+'\\')

Writes the custom grid coordinates to a text file for the ESP calculation. Default parameters assume this is being run on a Windows workstation.

Parameters

Label	Type	Description
grid_directory	string	Full path directory where the text grid file will be located for the job (on the system that RUNs the job)
write_directory	string	Full path directory to where the text file should be written on the system running this method.

Returns: (int) cchemlib.datatypes.SUCCESS if file write succeeds, cchemlib.datatypes.FAIL if not.

write_in_file(write_directory=os.getcwd()+'\\', directory=os.getcwd()+'/')

Writes the jaguar input file (.in file) for the ESP calculation.

Parameters

Label	Type	Description
write_directory	string	Full path directory to where the text file should be written on the system running this method.
directory	string	Full path directory where the file will be located on job launch.

Returns: (int) cchemlib.datatypes.SUCCESS if file write succeeds, cchemlib.datatypes.FAIL if not

job_run(command_options, command_string='/share/apps/Schrodinger2014-4/jaguar', wait=True)

Launches jaguar jobs based on the command_string and command options

Parameters

Label	Type	Description
command_options	list	List of strings that comprise command options for Jaguar. Should be in the order they'd be placed in the launch command. Should contain run command, host lists, etc.
command_string	string	Full path to the jaguar executable. Should terminate with 'jaguar'. Default should be updated when the Schrodinger suite is updated on the cluster.
wait	boolean	Flag to indicate whether the job should be launched in the background (wait=False) or not (wait=True). Defaults to True.

Returns: (int) cchemlib.datatypes.SUCCESS if the job submission (wait=False) succeeds or the job itself succeeds (wait=True), cchemlib.datatypes.FAIL if not

cqsarlib Documentation

Libraries (ccheminfolib.cqsarlib.library)

class ccheminfolib.cqsarlib.Library(name)

Base class for the Library objects. These objects organize Molecule objects and their associated observable and descriptor data. All Library classes inherit from this base class. So far, only DescriptorLibrary is fully implemented and thus is the only class listed other than this one

Parameters

Label	Type	Description
name	string	Name for the library

Methods

None

__init__(name) *See description above*

class ccheminfolib.cqsarlib.DescriptorLibrary(name, n_descriptors_per_mol)

This class defines an object that organizes Molecule objects with their associated descriptors and observable data. This is the object that is generally loaded by modeling scripts that use outside packages for modeling for ease of access to calculated descriptors and observables. Note: This does a lot of collating and the run times can be long for large *in silico* libraries.

This class was originally built to construct itself from supplied Molecule objects, but it was determined that a faster and more flexible way to do this is to use Molecule objects and process the descriptor data separately (see examples), and then use a file of descriptor data + observable data to construct this loadable serializable datatype for further use.

Parameters

Label	Type	Description
name	string	Name of the library. Optional, but used for file writing.
n_descriptors_per_mol	int	Total number of operational descriptors per Molecule object.

Methods

addObservable(mol_label, observable) : Attempts to adds an observable for a given Molecule.
getObservable(mol_label) : Attempts to return the observable value associated with a Molecule.
inTrainingSet(mol_label) : Returns True or False for if the supplied mol_label is in the training set.
addDescriptor(mol_label, descriptor_id, descriptor) : Attempts to add a given Descriptor object to the Molecule object referenced.
addMolDescriptors(mol_label, descriptors) : Attempts to add a full descriptor set to the labeled Molecule object, if it exists.
getMolDescriptor(mol_label, descriptor_id) : Attempts to return the specified descriptor from the specified Molecule
getMolDescriptors(mol_label) : Attempts to return the full descriptors for the specified Molecule
getDescriptorIDs() : Attempts to return a list of descriptor ID numbers.

__init__(name, n_descriptors_per_mol) *See description above*

addObservable(mol_label, observable)

Attempts to add the observable to a dictionary of observables, replacing the current observable specified for this molecule. This function will print a warning if no descriptors are defined for the specified Molecule, but the observable will be appended.

Parameters

Label	Type	Description
mol_label	string	Specific molecule label, generally the serial num filename.
observable	float	Currently the desired observable value to model.

Returns : (int) cchemlib.datatypes.SUCCESS if the observable is added to a known Molecule. Will return datatypes.FAIL if the molecule has no specified descriptors already added.

getObservable(mol_label)

Attempts to return the observable of the specified Molecule.

Parameters

Label	Type	Description
mol_label	string	Specific molecule label, generally the serial num filename.

Returns : (float) Observable for the specified Molecule. cchemlib.datatypes.FAIL if the observable value doesn't exist for the specified Molecule.

inTrainingSet(mol_label)

Returns True if the specified molecule has an associated observable value, False if not.

Parameters

Label	Type	Description
mol_label	string	Specific molecule label, generally the serial num filename.

Returns : (boolean True if the specified molecule has an associated observable value, False if not. Molecule labels are sorted into a training set list if an observable value has been provided for that Molecule.

addDescriptor(mol_label, descriptor_id, descriptor)

Attempts to add a descriptor for the specified Molecule. Note: To save space, the Molecule objects themselves have been removed from the DescriptorLibrary. To save space, the descriptors of each Molecule object is added to this as a raw value instead of an object.

Parameters

Label	Type	Description
mol_label	string	Specific molecule label, generally the serial num filename.
descriptor_id	int	ID of the descriptor. IMPORTANT FOR ORDERING THE DESCRIPTOR SET
descriptor	float	Value of the descriptor being added. THIS IS NOT A cchemlib.Descriptor object

Returns : (int) cchemlib.datatypes.SUCCESS if addition succeeds, cchemlib.datatypes.FAIL if not.

addMolDescriptors(mol_label, descriptors)

Attempts to add a dictionary of descriptors for the specified Molecule. Note: To save space, the Molecule objects themselves have been removed from the DescriptorLibrary. To save space, the descriptors of each Molecule object is added to this as a raw value instead of an object.

Parameters

Label	Type	Description
mol_label	string	Specific molecule label, generally the serial num filename.

descriptors	dict	Dictionary of the style {descriptor_id:descriptor_value}

Returns : (int) cchemlib.datatypes.SUCCESS if addition succeeds, cchemlib.datatypes.FAIL if not.

getMolDescriptor(mol_label, descriptor_id)

Attempts to retrieve the descriptor value of the specified descriptor for the specified Molecule. Note: To save space, the Molecule objects themselves have been removed from the DescriptorLibrary. To save space, the descriptors of each Molecule object is added to this as a raw value instead of an object.

Parameters

Label	Type	Description
mol_label	string	Specific molecule label, generally the serial num filename.
descriptor_id	int	ID of the descriptor of interest

Returns : (float) Specified descriptor value on success, (int) cchemlib.datatypes.FAIL if not.

getMolDescriptors(mol_label)

Attempts to return the dictionary of descriptors for the specified Molecule. Note: To save space, the Molecule objects themselves have been removed from the DescriptorLibrary. To save space, the descriptors of each Molecule object is added to this as a raw value instead of an object.

Parameters

Label	Type	Description
mol_label	string	Specific molecule label, generally the serial num filename.
descriptors	dict	Dictionary of the style {descriptor_id:descriptor_value}

Returns : (dictionary of float values) Specified descriptors on success, (int) cchemlib.datatypes.FAIL if not.

getDescriptorIDs()

Attempts to return the list of descriptor ID numbers

Parameters

None

Returns : (list of int) Current list of descriptor IDs that are consistent across the Library; (int) cchemlib.datatypes.FAIL if unsuccessful.

Code Recipes and Examples

Introduction

This chapter serves as a full resource of code listings demonstrating different tasks that can be performed with ccheminfolib, including full sets of scripts used in the Denmark lab to accomplish library generation, descriptor calculation and processing, and modeling. This chapter combines Python2 code (both for individual workstation work as well as working on the cluster), shell code and scripts for running jobs on odyssey, and other assorted utilities and considerations.

General Information

The scripts here are designed to be as general as possible, with extensive commentary on how to modify the code to change parameters. This documentation assumes a significant amount of understanding of the underlying design of ccheminfolib, and how the individual parts work together.

The user should expect a significant amount of modification required to the given scripts for the proper usage within specific contexts. Many scripts require directory names, file locations and names, and things of that nature and these will be pointed out in the example scripts.

Dependences

The ccheminfolib package requires python 2.7, and uses NumPy/SciPy and RDKit, as well as scikit-learn. We suggest using Anaconda to install python2.7, as conda and pip together make the installation of these extensions much easier, and in many cases are already installed.

Constructing Molecules and Libraries

The Molecule datatype is essential to many of the workings of ccheminfolib. While the object themselves can be saved (with Pickle), oftentimes more common file formats are desired and the Molecule datatype is used to do calculations, but then the pertinent data is saved to preserve permanent filesystem space. To that end, typical usage is to use text files (such as mol2 files) to save structural data, and text-delimited files (such as Comma Separated Variable (csv) files) to save calculated descriptor data.

It should be noted that ccheminfolib provides for both types of usage, with the Molecule datatype able to store all descriptor data and be saved as a binary file, but thus far the most common usage is as a run-time construct.

Converting SYBYL Mol2 Files to Molecule Datatype

Converting a text file with Sybyl's mol2 format to the Molecule datatype is a trivial process and handled using the cchemlib.parser.mol2Parser object. The following script demonstrates the proper usage:

Code Listing 1: parsing_mol2_into_Molecule.py

```
# This file is part of ccheminfolib, a library for the generation of
# chemical descriptors and modeling of these descriptors (QSAR/QSSR/QSPR)
#
# Copyright (C) 2011-2017 Jeremy Henle and the Denmark Lab
#
# This library is confidential and should not be distributed until
# the time deemed appropriate (Jeremy H., 2017)
"""
    This script uses the ccheminfolib.cchemlib.mol2Parser object
    to parse a single mol2 file into a Molecule Datatype
"""
# standard imports
import sys
import os
```

```
# ccheminfolib imports
from ccheminfolib.cchemlib import datatypes as dt
from ccheminfolib.cchemlib import mol2Parser

if __name__ == '__main__':
    # check to see that a mol2 filename was passed
    if len(sys.argv) < 2:
        # print a helpful usage statement
        print "Usage: parsing_mol2_into_Molecule.py /path/to/mol2/file"
        sys.exit()
    elif '.mol2' not in sys.argv[1]:
        # you need to give us the proper file format!
        print "File format not supported!"
        sys.ext()
    else:
        # do nothing
        pass
    # initialize the parser
    # we send the filename to the parser, along with a label
    # note: here, the label is constructed from the filename
    ## we assume filename structure /path/to/mol2/file
    label = sys.argv[1].split('/')[-1].split('.mol2')[0]
    p = mol2Parser(sys.argv[1], label)

    # now we parse the file and check for success
    if p.parse() != dt.SUCCESS:
        # print a helpful message
        print "Parsing failed on file: " + sys.argv[1]
        sys.exit()

    # now, return the molecule object
    mol = p.molecule()

    print "Parsing complete!"
    print mol
    print str(mol)
```

Oftentimes it can be beneficial to deepcopy the p.molecule() statement to ensure the Molecule object is not modified by improper reuse of the mol2Parser object, but careful construction of the script and usage should make it unnecessary.

Constructing a Molecule from Cores and R-Groups

The combinatorial method of generating catalyst/ligand libraries generally requires the specification of the core parts of the compounds of interest, and the substituents that are appended to that core. To that end, a MoleculeConstructor class was developed. This class takes both molecule cores with attachment points defined and R-groups with attachment assignments, and constructs the final molecules from them.

Core molecules are molecules that have specific attachment points labeled. These points are usually hydrogen atoms that are given the label A1, A2, A3,…, An; the label matches an assignment for the R group molecule. **Note: the attachment points must begin at A1. A0 is reserved for attachment point**

assignment on the R-group. The program maintains the configuration of the stereogenic center as set with the attachment label.

R-group molecules are essentially fragments of the compound of interest that are exchanged in a combinatorial fashion. Each R-group has a single attachment point denoted by a hydrogen atom labeled A0.

The **MoleculeConstructor** object requires a core Molecule object, and a dictionary of R-group Molecule objects with the keys set to the assignment of the R-group Molecule object. An example dictionary is

r_groups = {“A1”: r_group_obj_1, “A2”:r_group_obj_2}

for a core Molecule that has 2 attachment points. The constructor of the class also takes a molecule label as an argument.

The MoleculeConstructor class can be used to quickly generate a library of compounds by iterating over many cores and R_group configurations. An example of this is shown in code listing 2.

It should be noted that the structures created by the MoleculeConstructor are not minimized structures, and the structural checks ONLY check for through-ring bonds (“Olympic ringing”). The structures created should be properly minimized using MOE (for MMFF), MOPAC (PM6), or NWCHEM (HF/DFT). The authors suggest MMFF minimization first, prior to higher levels of theory.

Note: Oftentimes numpy/scipy throw matrix math warnings for the vector work. These are numpy/scipy errors and DO NOT impact the final result in any meaningful way.

Code Listing 2: molecule_constructor.py

```
# This file is part of ccheminfolib, a library for the generation of
# chemical descriptors and modeling of these descriptors (QSAR/QSSR/QSPR)
#
# Copyright (C) 2011-2017 Jeremy Henle and the Denmark Lab
#
# This library is confidential and should not be distributed until
# the time deemed appropriate (Jeremy H., 2017)
"""
    This script uses the ccheminfolib.cdesclib.MoleculeConstructor object
    to create a library of BINOL-phosphoramide catalysts and save them as
    unoptimized structures in the Tripos mol2 format.
"""
# standard imports
import os
import sys
from copy import deepcopy
# ccheminfolib --> don't need to import numpy
from ccheminfolib.cchemlib import datatypes as dt
from ccheminfolib.cchemlib import mol2Parser
from ccheminfolib.cdesclib import MoleculeConstructor

# file directories -- need modification for customized jobs
mol2_dir = 'mol2/'
r1_dir = 'r1/'      # additional R-group directories can be added.
core_dir = 'cores/'


if __name__ == '__main__':
```

```
# get the cores
cores = {}
# walk through the directory containing mol2 files of core molecules
for root,dir,files in os.walk(core_dir):
    for file in files:
        if '.mol2' in file:
            # load file and initialize the parser
            p = mol2Parser(core_dir+file, file.split('.')[0])
            # parse the file
            if p.parse() != dt.SUCCESS:
                print "Core parse failed: " + file
                sys.exit()
            # assign the molecule to a dict with the core label as the key
            # note the use of deepcopy
            cores[file.split('.')[0]] = deepcopy(p.molecule())
print "Found " + str(len(cores)) + " cores!"

# get the r_groups
# note: for additional r groups, simply duplicate this code for r2, r3, r4, etc.
r1 = {}
for root,dir,files in os.walk(r1_dir):
    for file in files:
        if '.mol2' in file:
            p = mol2Parser(r1_dir+file, file.split('.')[0])
            if p.parse() != dt.SUCCESS:
                print "Core parse failed: " + file
                sys.exit()
            r1[file.split('.')[0]] = deepcopy(p.molecule())
print "Found " + str(len(r1)) + " R1 groups!"

for core in cores:
    # additional r groups should be nested in this loop!
    for r in r1:
        # create the r_group dictionary
        r_groups = {'A1': r1[r]}
        # initialize the MoleculeConstructor, passing the core molecule
        # and the dict of r_groups and the label for our new molecule
        constructor = MoleculeConstructor(cores[core], r_groups, core+'_'+r)
        # make the molecule -- this function will return an error
        # if the molecule is determined to be defective
        if constructor.construct() != dt.SUCCESS:
            print "Could not construct molecule " + core+'_'+r
            sys.exit()
        mol = deepcopy(constructor.molecule())
        mol.write_mol2(mol2_dir+mol.label+'.mol2')
        print "Wrote: " + mol.label
```

Minimizing a Library of Compounds: Controlling MOE with Python

The next step in library preparation is the minimization of the compounds. For the initial optimization, it is highly recommended that the user use MOE to minimize the structures using molecular mechanics. This section will demonstrate the method with which to use python to write and launch batch files to run MOE from the command line. Note: This is written for windows computers.

The code here writes a number of temporary batch files to a particular directory, then executes the files. Due to a quirk in the python library subprocess, the python script must be in the same directory as the bat files.

Code Listing 3: minimize_molecules.py

```
# This file is part of ccheminfolib, a library for the generation of
# chemical descriptors and modeling of these descriptors (QSAR/QSSR/QSPR)
#
# Copyright (C) 2011-2017 Jeremy Henle and the Denmark Lab
#
# This library is confidential and should not be distributed until
# the time deemed appropriate (Jeremy H., 2017)
"""
    This script demonstrates the usage of Python2 to
    run MOE from the command line.
"""
# standard library imports
import os
from subprocess import Popen

if __name__ == '__main__':

    # list of files to minimize
    mol2_file_list = []
    # directories
    # directory with unoptimized files
    unopt_dir =
'D:/Dropbox2/Dropbox/research/computation/PROJECTS/j_pad/bpad/conformers/rdkit_conform
ers_unopt_mol2/'
    # directory where optimized files should be written
    opt_dir =
'D:/Dropbox2/Dropbox/research/computation/PROJECTS/j_pad/bpad/conformers/rdkit_conform
ers_FULL_mmff_mol2/'


    # MOE control strings -- every MOE control script
    # should have a series of SVL statements to
    # describe the desired job

    # string1 - '/path/to/moebatch -exec \"ReadTriposMOL2 \'' -- ensure path to
moebatch correct
    string1 = 'D:/Dropbox2/Dropbox/moe2011/bin/moebatch -exec \"ReadTriposMOL2 \''

    # string2 - modify path to the ff file!
    string2='\'; pot_Load  \'D:/Dropbox2/Dropbox/moe2011/lib/mmff94x.ff\'; MM [
pot_charge:0, keep_chirality:\'geometry\' ]; WriteTriposMOL2 \''

    # dont modify string3
    string3='\'; Close [];\"\n'

    # helpful for later
    count = 1

    # this is not necessary, but should we have to terminate the script
    # this allows us to not rerun optimizations that are done

    already_done = []
    for root, dir, files in os.walk(opt_dir):
        for file in files:
            if '.mol2' in file:
                already_done.append(file)
    # populate the list of jobs to do

    for root,dir,files in os.walk(unopt_dir):
        for file in files:
            # check for proper file format
            if '.mol2' not in file:
```

```
                continue
            if file in already_done:
                continue

            svl_file = open(str(count)+'.bat','w')
            svl_file.write(string1)
            svl_file.write(unopt_dir+file)
            svl_file.write(string2)
            #new_file = file.split('iv')[0] +'iv_'+file.split('iv')[1]
            svl_file.write(opt_dir+file)
            svl_file.write(string3)
            print count
            count += 1
            svl_file.close()
    #svl_file.write ('\"')
    # this code launches each bat file. Modify cwd= to the path to the bat_files, the
directory this file is currently in
    for x in range(1,count+1):
        p = Popen(str(x)+'.bat',
cwd=r"D:\\Dropbox2\\Dropbox\\research\\computation\\PROJECTS\\j_pad\\bpad\\conformers\
\bat_files")
        stdout, stderr = p.communicate()
        print stdout
```

Generating a Conformer Library from a Single Conformer

The latest descriptors developed by the Denmark Lab involve calculating data from a set of conformers derived from a single structure instead of the single structure itself. To generate the conformer library, one must calculate a number of conformers for each molecule. To do this, we use Python in conjunction with MOE. Similar to the minimization script, this python script simply writes the svl for conformer library generation. The conformers are identified, written to a temp database, and then saved into a single mol2 file for further processing.

The second code listing here demonstrates using ccheminfolib and python to separate the single files into multiple mol2 files. This is a general parsing algorithm and can be used in any case for separating multi-molecule mol2 files.

Note: To parallelize this method, modify Code Listing 4 to write the bat files, and then use a separate python scripts to launch the bat files in parallel. See "make_bat_run_lists.py" and "run_moe_conf_search.py" in the code listings folder provided with this documentation.

Code Listing 4: generate_conformer_library.py

```
# This file is part of ccheminfolib, a library for the generation of
# chemical descriptors and modeling of these descriptors (QSAR/QSSR/QSPR)
#
# Copyright (C) 2011-2017 Jeremy Henle and the Denmark Lab
#
# This library is confidential and should not be distributed until
# the time deemed appropriate (Jeremy H., 2017)
"""
    This script demonstrates the usage of Python2 to
    run MOE from the command line to generate a
    conformer library.
"""
# standard library imports
```

```
import os
from subprocess import Popen

if __name__ == '__main__':
    # files to conformerize
    mol2_file_list = []
    # directories -- UPDATE THIS!
    unopt_dir =
'D:/Dropbox2/Dropbox/research/computation/PROJECTS/j_pad/bpad/mol2_unopt/'
    mdb_dir = 'D:/Dropbox2/Dropbox/research/computation/PROJECTS/j_pad/bpad/moe_conf/'
    opt_dir =
'D:/Dropbox2/Dropbox/research/computation/PROJECTS/j_pad/bpad/mmff_mol2/'


    # SVL strings. Modify directories where appropriate!
    string1 = 'D:/Dropbox2/Dropbox/moe2011/bin/moebatch -exec \"ReadTriposMOL2 \''
    string2='\'; pot_Load  \'D:/Dropbox2/Dropbox/moe2011/lib/mmff94x.ff\'; ConfSearch
[ infile:\'\', outfile:\''
    # string3 contains the parameters for the conformer search. These can be modified
on a per system basis
    string3 = '\', method:\'LowModeMD\', cutoff:7, maxconf:50, maxfail:100, maxit:1000
]; db_ExportTriposMOL2 [ \''
    string4 = '\', \''
    string5 = '\' ]; Close [];\"\n'
    #mol2_write_file=temp.mol2
    count = 1

    # prevent redundant jobs
    already_done = []
    for root, dir, files in os.walk(mdb_dir):
        for file in files:
            if '.mol2' in file:
                already_done.append(file)

    for root,dir,files in os.walk(opt_dir):
        for file in files:
            if file in already_done:
                continue
            # construct the svl string in a bat file
            svl_file = open(str(count)+'.bat','w')
            svl_file.write(string1)
            svl_file.write(opt_dir+file)
            svl_file.write(string2)
            svl_file.write(mdb_dir+file.split('.')[0]+'.mdb')
            svl_file.write(string3)
            svl_file.write(mdb_dir+file.split('.')[0]+'.mdb')
            svl_file.write(string4+mdb_dir+file+string5)

            print count
            count += 1
            svl_file.close()

    # run the bat files in order
    # remember to modify the cwd variable in Popen()
    for x in range(1,count+1):
        p = Popen(str(x)+'.bat',
cwd=r"D:\\Dropbox2\\Dropbox\\research\\computation\\PROJECTS\\j_pad\\bpad\\moe_conf")
        stdout, stderr = p.communicate()
        print stdout
```

Code Listing 5: separate_mol2_files.py

```
# This file is part of ccheminfolib, a library for the generation of
# chemical descriptors and modeling of these descriptors (QSAR/QSSR/QSPR)
#
# Copyright (C) 2011-2017 Jeremy Henle and the Denmark Lab
#
# This library is confidential and should not be distributed until
# the time deemed appropriate (Jeremy H., 2017)
"""
    This script demonstrates how to separate a
    multi-molecule mol2 file into individual
    mol2 files.
"""
# standard library imports
import os
import sys

# directory where we should write our new mol2 files too
mol2_dir = 'separated_mol2/'

# iterate through the directory with files to separate
for root, dir, files in os.walk('mmff_mol2/'):
    for file in files:
        if '.mol2' in file:
            # new file, reset counter
            count = 0
            # open the file
            f = None
            try:
                f = open('mmff_mol2/'+file,'r')
            except IOError:
                print "Could not open file " + file +', continuing to next file!'
                continue
            # new file handle
            nf = None
            for line in f:
                # this is the first line in any mol2 file, so we know when a new
                # molecule starts!
                if '@<TRIPOS>MOLECULE' in line:
                    # if it exists, close the old writing file handle
                    if nf != None:
                        nf.close()
                    # open the next mol2 file. We append an ascending number to the
end
                    # of the original file name to keep track of the conformers.
                    nf = open(mol2_dir+file.split('.')[0]+'_'+str(count)+'.mol2','w')
                    count += 1
                    nf.write(line)
                else:
                    # just write the line...
                    nf.write(line)
```

class ccheminfolib.cchemlib.Point(ID, label, x, y, z)

Defines a single geometric point in 3D space.

Parameters

Label	Type	Description
ID	int	Unique identifier, user controlled
label	string	Unique identifier used in lieu of serial ID “Person friendly”
x	float	Location of the point on X-axis
y	float	Location of the point on Y-axis
z	float	Location of the point on Z-axis

Methods

get_distance_to_point(other) : returns the distance between another Point object and this point object

__sub__(other) : Overloaded subtraction operator, returns distance to other Point object (uses get_distance_to_point())

__eq__(other) : Overloaded ‘=’, returns true if other Point object is at the same location (uses get_distance_to_point())

__init__(ID, label, x, y, z) *See description above*

get_distance_to_point(other)

Returns distance between this point object to another point object.

Parameters

Label	Type	Description
other	ccheminfolib.cchemlib.Point object	Point object to calculate distance from

Returns: (float) Distance between the points, or NotImplemented if ‘other’ is not a Point object

class ccheminfolib.cchemlib.Atom(ID, label, point, atom_type, formal_charge = 0, mpa_charge = 0.0)

Atom datatype. Molecules are made up of Atoms.

Parameters

Label	Type	Description

ID	int	Unique identifier, user controlled
label	string	Unique identifier used in lieu of serial ID "Person friendly"
point	Point	cchemlib.Point object; defines 3D coordinates of the atom
type	string	Atom type based on Tripos atom types; defined constants in cchemlib.atomtypes
formal_charge	int	Charge based on VSPER rules; not used in many cases
mpa_charge	float	Charge calculated using MPA; not used in many cases

Methods

get_distance_from_atom(other) : returns the distance from another Atom object and this Atom object

__sub__(other) : Overloaded subtraction operator, returns distance to other Point object (uses get_distance_from_Atom())

__init__(ID, label, point,atom_type,formal_charge=0,mpa_charge=0) *See description above*

get_distance_from_atom(other)

Returns distance between this Atom object to another Atom object.

Parameters

Label	Type	Description
other	ccheminfolib.cchemlib.Atom object	Atom object to calculate distance from

Returns: (float) Distance between the points, or NotImplemented if 'other' is not an Atom object

class ccheminfolib.cchemlib.Bond(ID, label, start_atom,end_atom, bond_type)

Bond datatype. Bonds form between Atoms!

Parameters

Label	Type	Description
ID	int	Unique identifier, user controlled

label	string	Unique identifier used in lieu of serial ID “Person friendly”
start_atom	int	ID of the Atom object at the start of the bond
end_atom	int	ID of the Atom object at the end of the bond
bond_type	string	Bond type based on tripos mol2 bond type definitions. Defined in cchemlib.bondtypes

Methods

set_start_atom(start_atom_ID) : Sets the atom ID for the start atom

set_end_atom(end_atom_ID) : Sets the atom ID for the end atom

__init__(ID, label, start_atom, end_atom, bond_type) *See description above*

set_start_atom(start_atom_ID)

Sets the Bond.start_atom variable to a new ID

Parameters

Label	Type	Description
start_atom_ID	int	New ID of the starting atom

Returns: Nothing.

set_end_atom(end_atom_ID)

Sets the Bond.end_atom variable to a new ID

Parameters

Label	Type	Description
end_atom_ID	int	New ID of the ending atom

Returns: Nothing.

class ccheminfolib.cchemlib.Descriptor(type, value)

Descriptor datatype.

Each descriptor type serves a purpose. Grid point descriptors have an ELE and VDW potential energy associated with them. Eventually, GRINDs are calculated from these later on. Not used directly with grids.

Parameters

Label	Type	Description
type	int	One of several defined descriptor types; constants located in cchemlib.datatypes Available values are ELE, VDW, ELE_M, VDW_P, CO. Only ELE and VDW are used for gridpoint descriptors, but full listing used for GRINDs.
value	float	Value of the descriptor (converted automatically to floating point value)

Methods
None.

__init__(type, value) *See description above*

Notes:

class ccheminfolib.cchemlib.Gridpoint(ID, point, descriptors={})
Gridpoint class. A gridpoint consists of a location in 3D space (Point object) and a set of calculated values (descriptors).

Parameters

Label	Type	Description
ID	int	Unique identifier, user controlled
point	Point	cchemlib.Point object; defines 3D coordinates of the atom
descriptors	dict	Dictionary of descriptors. Keys are descriptor types. One descriptor per type.

Methods

add_descriptor(type, descriptor) : Sets the current descriptor for a given type.
get_distance_from_gridpoint(other) : Returns the distance from another Gridpoint object
get_distance_from_point(other) : Returns the distance from a Point object to this Gridpoint
__sub__(other) : Overloaded subtraction operator, returns distance to other Gridpoint or Point object (uses get_distance_from_gridpoint() and get_distance_from_point ())
__eq__(other) : Overloaded equality operator, returns True if other is a Gridpoint object and the distance between this Gridpoint and other is <0.0001 units.
__str__() : Overloaded string operator. Returns a string comprised of the ELE and VDW type descriptors and the x,y,z coordinates. If any value is missing, it is skipped in the generation of the string.

__init__(ID, point, descriptors={}) *See description above*

add_descriptor(type,descriptor)

Returns distance between this Atom object to another Atom object.

Parameters

Label	Type	Description
type	int	Defined type of descriptor as described in cchemlib.Descriptor
descriptor	cchemlib.Descriptor	cchemlib.Descriptor object, instantiated with values.

Returns: Nothing.

get_distance_from_gridpoint(other)

Returns distance between this Gridpoint object to another Gridpoint object.

Parameters

Label	Type	Description
other	ccheminfolib.cchemlib.Gridpoint	Gridpoint object to calculate distance from

Returns: (float) Distance between the Gridpoints, or NotImplemented if 'other' is not a Gridpoint object

get_distance_from_point(other)

Returns distance between this Gridpoint object to another Gridpoint object.

Parameters

Label	Type	Description
other	ccheminfolib.cchemlib.Point	Point object to calculate distance from

Returns: (float) Distance between the points, or NotImplemented if 'other' is not a Point object

class ccheminfolib.cchemlib.Molecule(label, charge=0, multiplicity=1)

Molecule datatype.

In the real world, molecules are entities comprised of groups of atoms tethered by bonds. In the computer, the Molecule datatype contains the atoms and bonds that comprise the molecule maintained in a wrapper that the computer can manipulate. The molecule datatype also contains the gridpoints (grid), which contains the descriptors for the molecule. From the fields of the grid, the GRIND can be calculated further on. We put placeholders for everything.

Parameters

Label	Type	Description
label	string	Unique identifier, user controlled
charge	int	Overall formal charge of the molecule. No charge is the default
multiplicity	int	Multiplicity of the molecule. Singlet state is the default.

Methods

add_atom(atom) : adds an Atom object to the Molecule
add_bond(bond) : adds a Bond object to the Molecule
add_raw_bond(start_atom_id, end_atom_id, bond_type) : Constructs a Bond object and adds to Molecule.
remove_atom(atom_ID) : removes Atom from the molecule, cleans up any bonds involved with the atom
change_atom_id(old_id, new _id) : attempts to change the ID of a specified Atom object.
change_atom_coord(atom_ID, x, y, z) : attempts to change the 3D coordinates of a specified Atom
generate_grid(spacing=1.0, force_origin=False, forced_origin=[0.,0.,0.], grid_overwrite=False) : Generates a spherical grid around the molecule with a specified point spacing.
set_formal_charge(charge) : Sets the formal charge of the molecule
get_formal_charge() : Returns the current formal charge setting. Default is 0 (integer)
set_gridpoint_descriptor(gridpoint_ID, descriptor_type, descriptor) : Attempts to set a descriptor value for the given gridpoint and descriptor type
set_gridpoint_descriptors(gridpoint_ID, descriptors) : Attempts to attach a dict of descriptors to a specific Gridpoint
get_gridpoint_descriptors(gridpoint_ID) : Attempts to return the descriptors of a particular Gridpoint
set_atom_MPA_charge(atom_ID, charge) : Sets the charge based on Mulliken Population Analysis (MPA)
write_mol2(filename) : Writes the Molecule to a file as a Tripos mol2 filetype (text file).
determine_if_intersect(ring_atoms, bond) : Determines if the given Bond object intersects the atoms defined by the set ring_atoms. Note: Not generally used as a member function, should probably be defined private.
check() : Checks the integrity of the molecule (mainly looking for ring/bond intersections).

__init__(label, charge=0, multiplicity=1) *See description above*

add_atom(atom)

Adds an Atom object to the molecule. Requires a fully instantiated Atom object to the molecule. Note: Overrides the Atom.ID value to match the next ID available in the Molecule object for bookkeeping purposes.

Parameters

Label	Type	Description
atom	cchemlib.Atom	Fully instantiated Atom object

Returns : cchemlib.datatypes.SUCCESS if successful, cchemlib.datatypes.FAIL if the addition is unsuccessful. Note: Should not fail EVER!

add_bond(bond)

Adds a Bond object to the molecule. Requires a fully instantiated Bond object to the molecule. Note: Overrides the Bond.ID value to match the next ID available in the Molecule object for bookkeeping purposes.

Parameters

Label	Type	Description
bond	cchemlib.Bond	Fully instantiated Bond object

Returns : cchemlib.datatypes.SUCCESS if successful, cchemlib.datatypes.FAIL if the addition is unsuccessful.

add_raw_bond(start_atom_id, end_atom_id, bond_type)

Adds a Bond object to the molecule. Builds a Bond object using data provided. Note: Sets the Bond.ID value to match the next ID available in the Molecule object for bookkeeping purposes.

Parameters

Label	Type	Description
start_atom_id	int	Atom.ID of the starting atom
end_atom_id	int	Atom.ID of the ending atom
bond_type	string	Type of bond. Defined in bondtypes.py

Returns : cchemlib.datatypes.SUCCESS if successful, cchemlib.datatypes.FAIL if the addition is unsuccessful.

remove_atom(atom_ID)

Removes an Atom from the Molecule. This function also fixes any changes to the Atom ID and Bond ID based on the atom's removal from the order; this is necessary for bookkeeping.

Parameters

Label	Type	Description
atom_ID	int	Atom.ID of the atom to remove

Returns : cchemlib.datatypes.SUCCESS if successful, cchemlib.datatypes.FAIL if the addition is unsuccessful.

change_atom_id(old_id, new_id)

Changes an atom's Atom.ID and fixes the Bonds involved to match that ID.

Parameters

Label	Type	Description
old_id	int	Current Atom.ID to be changed
new_id	int	New Atom.ID

Returns : cchemlib.datatypes.SUCCESS if successful, cchemlib.datatypes.FAIL if the addition is unsuccessful.

change_atom_coord(atom_ID, x,y,z)

Changes the x,y,z coordinates of a specific atom

Parameters

Label	Type	Description
atom_ID	int	ID of the Atom to have the coordinates changed
x	float	New coordinate in the X-dimension
y	float	New coordinate in the Y-dimension
z	float	New coordinate in the Z-dimension

Returns : cchemlib.datatypes.SUCCESS if successful, cchemlib.datatypes.FAIL if the addition is unsuccessful.

generate_grid(spacing=1.0, force_origin=False, forced_origin = [0.,0.,0.], grid_overwrite=False)

Creates a spherical grid of Gridpoints based on location of the molecule in 3D space. Can either generate the origin automatically or have an origin set by the user using the appropriate flags.

Parameters

Label	Type	Description
spacing	float	Distance between gridpoints; default is 1.0 Å
force_origin	boolean	Flag for designating an origin manually or automatically
forced_origin	list of floats	If force_origin=True, a set of coordinates must be given. Defaults to [0.,0.,0.]
grid_overwrite	boolean	If set to True, will not overwrite a grid that is already defined. Default value is False.

Returns : cchemlib.datatypes.SUCCESS if successful, cchemlib.datatypes.FAIL if the addition is unsuccessful.

set_formal_charge(charge)

Sets the formal charge of the Molecule object. Cannot fail.

Parameters

Label	Type	Description
charge	int	Charge of the molecule. Must be an integer.

Returns : Nothing.

get_formal_charge()

Returns the formal charge of the Molecule object. Cannot fail.

Parameters

None

Returns : (int) Formal charge of the Molecule.

set_gridpoint_descriptor(gridpoint_ID, descriptor_type, descriptor)

Adds a Descriptor to a Gridpoint. If the descriptor type is already set for this point, it will override the original Descriptor object of that type.

Parameters

Label	Type	Description
gridpoint_ID	int	ID of the Gridpoint to set the Descriptor for
descriptor_type	int	Descriptor type as described in cchemlib.datatypes

descriptor	cchemlib.Descriptor	Instantiated Descriptor type to add to the Gridpoint

Returns : cchemlib.datatypes.SUCCESS if successful, cchemlib.datatypes.FAIL if unsuccessful.

set_gridpoint_descriptors(gridpoint_ID, descriptors)

Overwrites the Molecules.descriptors dictionary. Does not check if the passed descriptors variable is a dictionary of Descriptor objects. Generally not used by the user, but as a helper function in the Calculator/Controller/Parser classes.

Parameters

Label	Type	Description
gridpoint_ID	int	ID of the Gridpoint to set the Descriptor for
descriptors	dict of cchemlib.Descriptor objects	Dictionary object of Descriptor objects. Does not check, assumes they are given .

Returns : Nothing

set_atom_MPA_charge(atom_ID, charge)

Sets an atoms MPA charge

Parameters

Label	Type	Description
atom_ID	int	ID of the Atom to change the charge for
charge	float	Charge of the atom based on Mulliken Population Analysis

Returns : cchemlib.datatypes.SUCCESS if successful, cchemlib.datatypes.FAIL if unsuccessful.

write_mol2(filename)

Writes a tripos mol2 file (text) for the Molecule object.

Parameters

Label	Type	Description
filename	string	Full path to the file. Assumes '.mol2' is part of the string.

Returns : cchemlib.datatypes.SUCCESS if successful, cchemlib.datatypes.FAIL if unsuccessful.

determine_if_intersect(ring_atoms, bond)

Determines if the given bond intersects a ring defined by the atoms in the list of Atom.ID supplied.

Parameters

Label	Type	Description
ring_atoms	list of int	Atom.ID of atoms comprising a ring
bond	cchemlib.Bond	Bond object

Returns : (boolean) returns True if there is an intersect, False if none is detected.

check()

Molecular integrity check. Determines if there are any ring/bond intersections.
Parameters
None
Returns : cchemlib.datatypes.SUCCESS if Molecule passes the check, cchemlib.datatypes.FAIL if it does not pass the check. Note: Does not FIX problems.

Notes on the Molecule Object:

Parsers (ccheminfolib.cchemlib.parse)

class ccheminfolib.cchemlib.Parser(filename)

Parser superclass. All parsers inherit from this class. Contains some basic conversion functions for unit conversions.

Parameters

Label	Type	Description
filename	string	Full path to the file that's going to be parsed by the Parser

Methods

hartrees_to_kcal (energy) : converts energy from hartrees to kcal/mol
bohr_to_angstrom(coord) : converts a bohr coordinate to angstrom
print_file_open_error() : Prints an error message if the file cannot be opened.

__init__(filename) *See description above*

hartress_to_kcal(energy)

Converts the given energy to kcal/mol from hartrees.

Parameters

Label	Type	Description
energy	float	Energy value given in hartrees

Returns : (float) Energy in kcal/mol

bohr_to_angstrom(coord)

Converts given coord from bohr to angstrom.

Parameters

Label	Type	Description
coord	float	Coordinate in a single dimension in bohr units

Returns : (float) Coordinate in angstroms

print_file_open_error()

Prints "ERROR: Could not open file " + self.filename

Parameters

None

Returns : None

class ccheminfolib.cchemlib.mol2Parser(filename, label)

Class for parsing tripos mol2 file format. Constructs a Molecule from a mol2 file.

Parameters

Label	Type	Description
filename	string	Full path to the file that's going to be parsed by the Parser
label	string	Label for the molecule. Generally used for naming its file later on. Use serial number!!!!!!!

Methods

parse() : Parses the mol2 file, separating file lines into different categories for further parsing

molecule() : Creates a Molecule object based on the file that has been parsed

__init__(filename) *See description above*

parse()

Reads the given mol2 file and categorizes the file lines into different components: 1) Header, 2) Atom lines, and 3) Bond lines. This is the first pass for interpreting the file, and is the parsing step.

Parameters

None

Returns : (int) cchemlib.datatypes.SUCCESS if the MOL2 file structure was correct and properly parsed, cchemlib.datatypes.FAIL if parsing fails. Always be sure to check for SUCCESS.

molecule()

Takes the parsed MOL2 file and creates a Molecule object based on the information contained in the MOL2 file. Note: Many ccheminfolib classes that construct Molecule objects have molecule() functions that return the Molecule object associated with the class.

Parameters

None

Returns : (cchemlib.Molecule) Instantiated Molecule object on success, cchemlib.datatypes.FAIL if a Molecule object could not be constructed.

class ccheminfolib.cchemlib.respParser(filename)

This parser works on RESP files output by Jaguar during ESP grid calculations. The ultimate goal of this class is to return a dictionary of Gridpoint objects with ESP descriptors instantiated.

Parameters

Label	Type	Description
filename	string	Full path to the file that's going to be parsed by the Parser

Methods

convert_single_line(line, ID) : converts a single line of the file to a Gridpoint with ESP descriptor; returns the Gridpoint object.

parse() : Opens the RESP file and converts the data to properly labeled and converted Gridpoint objects

__init__(filename) *See description above*

convert_single_line(line, ID)

Takes a single line of the RESP file and converts it to an instantiated Gridpoint object. Returns the Gridpoint object on success. Note: Generally not called by the user.

Parameters

Label	Type	Description
line	string	Line from RESP file. '\n' terminated.
ID	int	Gridpoint.ID supplied by the caller. Generally controlled in parse().

Returns : (cchemlib.Gridpoint) Instantiated Gridpoint object with the ESP descriptor populated.

parse()

Parses the RESP file, converting all data to a dictionary of Gridpoints intended to populate the Molecules descriptor field.

Parameters

None

Returns : (dict of cchemlib.Gridpoints) Dictionary of instantiated Gridpoint objects (dict keys match Gridpoint.ID) on success, cchemlib.datatypes.FAIL if the parser fails.

class ccheminfolib.cchemlib.nwXYZParser(filename, mol)

This parser works on XYZ files output by NWCHEM during the ESP calculation. For whatever reason, NWCHEM gently massages the location of the molecule in 3D space, and so we have to adjust our Molecule objects to match so the descriptors are in the right location, relative to the Molecule. This parser first parses the file, then applies the correction to the Molecule object.

Parameters

Label	Type	Description
filename	string	Full path to the file that's going to be parsed by the Parser
mol	cchemlib.Molecule	Instantiated Molecule object that needs coordinate conversion.

Methods

convert_single_line(line, ID) : Takes a single parsed line of the XYZ file and modifies the specific Atom of the Molecule that the line pertains to.

parse() : Opens the XYZ file and converts the data to properly labeled strings for use in molecule() and convert_single_line()

molecule() : Modifies the associated Molecule object (mol) to the correct coordinates and returns a deep copy of the Molecule object.

__init__(filename) *See description above*

convert_single_line(line, ID)

Takes a single parsed line of the XYZ file and modifies the specific Atom of the Molecule that the line pertains to.

Parameters

Label	Type	Description
line	string	Line from XYZ file. '\n' terminated.
ID	int	Atom.ID supplied by the caller. Controlled by molecule()

Returns : (int) cchemlib.datatypes.SUCCESS if the Atom coordinates are successfully changed, cchemlib.datatypes.FAIL if not.

parse()

Parses the XYZ file. Sorts the lines into an internal list of lines, ensures proper ordering.

Parameters

None

Returns : : (int) cchemlib.datatypes.SUCCESS if the file is properly read and parsed, cchemlib.datatypes.FAIL if not.

molecule()

Modifies the associated Molecule object (mol) to the correct coordinates and returns a deep copy of the Molecule object.

Parameters

None

Returns : (cchemlib.Molecule) Deep copy of the Molecule object with modified Atom coordinates if parsing/conversion is successful. Returns cchemlib.datatypes.FAIL if not successful.

class ccheminfolib.cchemlib.nwGridParser(filename, leveling=True)

This parser works on GRID files output from NWCHEM after an ESP calculation. Similar to respParser, this class constructs a dictionary of Gridpoint objects with the ESP descriptor populated. Note: MUST BE USED AFTER nwXYZParser has modified the parent Molecule object, or else the Gridpoint dictionary coordinates will not match with the Atom coordinates!

Parameters

Label	Type	Description
filename	string	Full path to the file that's going to be parsed by the Parser
leveling	boolean	Flags whether or not to force leveling of gridpoint energies to 30 kcal/mol. Defaults to True.

Methods

convert_single_line(line, ID) : converts a single line of the file to a Gridpoint with ESP descriptor; returns the Gridpoint object.

parse() : Opens the GRID file and converts the data to properly labeled and converted Gridpoint objects

__init__(filename) *See description above*

convert_single_line(line, ID)

Takes a single line of the GRID file and converts it to an instantiated Gridpoint object. Returns the Gridpoint object on success. Note: Generally not called by the user.

Parameters

Label	Type	Description
line	string	Line from GRID file. '\n' terminated.
ID	int	Gridpoint.ID supplied by the caller. Generally controlled in parse().

Returns : (cchemlib.Gridpoint) Instantiated Gridpoint object with the ESP descriptor populated.

parse()

Parses the GRID file, converting all data to a dictionary of Gridpoints intended to populate the Molecules descriptor field.

Parameters

None

Returns : (dict of cchemlib.Gridpoints) Dictionary of instantiated Gridpoint objects (dict keys match Gridpoint.ID) on success, cchemlib.datatypes.FAIL if the parser fails.

cdesclib Documentation

Calculators (ccheminfolib.cdesclib.Calculator)

class ccheminfolib.cdesclib.Calculator(type, molecule)

Calculator superclass. These classes are responsible for calculating descriptors based on given data.

Parameters

Label	Type	Description
type	int	Type of descriptor calculated. Currently use cchemlib.datatypes.VDW or ELE
molecule	cchemlib.Molecule	Instantiated Molecule object.

Methods

molecule () : Returns the Molecule object associated with the calculator in its current state.

__init__(type, molecule) *See description above*

molecule ()

Returns the associated Molecule object in its current state. Can be called before or after a calculation is performed

Parameters

None

Returns : (cchemlib.Molecule) Returns the Molecule object. Cannot fail unless associated Molecule object is explicitly deleted.

class ccheminfolib.cdesclib.vdWCalculator(molecule,probe_atom_type="C.2", leveling=True)

This calculator calculates the MMFF94x approximation steric potential energy between a probe atom and the molecule at each location in the currently defined grid. The provided Molecule object must have a set of Gridpoints already defined for it prior to the calculation.

Parameters

Label	Type	Description
molecule	cchemlib.Molecule	Instantiated Molecule object with defined set of Gridpoints
probe_atom_type	const string	Atom type of the probe atom. Constants defined in cchemlib.atomtypes. Defaults to atomtypes.C_SP2 ("C.2")
leveling	boolean	Flag for descriptor energy leveling. Defaults to True

Methods

calculate_vdw_between_atoms(atom1, atom2) : Calculates the VDW PE between two Atom objects
calculate_for_single_gridpoint(gridpoint_ID) : Calculates the VDW PE between all Atoms and a point.
calculate() : Calculates the molecular VDW PE interaction between all Gridpoints and Atoms.
molecule() : Returns a deep copy of the associated Molecule object. Implemented in the inherited class.

__init__(molecule,probe_atom_type="C.2", leveling=True) *See description above*

calculate_vdw_between_atoms(atom1, atom2)

Calculates the VDW approximation PE between two Atom objects. Generally, one atom is the probe atom located at a Gridpoint object location, and the other is an Atom object of the Molecule. Not generally called by the user (it is a helper function) but left non-private for testing/research purposes.

Parameters

Label	Type	Description
atom1	cchemlib.Atom	Instantiated Atom object
atom2	cchemlib.Atom	Instantiated Atom object

Returns : (float) Calculated VDW PE between the Atom objects. Based on MMFF94x force field.

calculate_for_single_gridpoint(gridpoint_ID)

Calculates the summed VDW PE between a probe atom located at the Gridpoint.Point and the Atoms of the Molecule.

Parameters

Label	Type	Description
gridpoint_ID	int	ID of the Gridpoint object.

Returns : (float) Summed VDW PE descriptor for the given Gridpoint location.

calculate()

Calculates the VDW PE descriptors for the entire Molecule based on the defined Gridpoint set.

Parameters

None

Returns : (int) cchemlib.datatypes.SUCCESS if the calculation succeeds for all Gridpoints, cchemlib.datatypes.FAIL if not.

class ccheminfolib.cdesclib.GRINDCalculator(molecule, bin_min=2.0, num_nodes=4, num_points=100, bins=20, bin_size=1.5, num_stddev=2.0, stddev = 6.0, get_stddev=False)

This calculator uses the VDW and ESP descriptors at the Gridpoints to calculate Grid INdependent Descriptors (GRINDs). Note: For the method listing in GRIND type calculators, while all methods are included for development/research understanding, many are helper functions and SHOULD NOT be called in a main script. These are marked as such with "Generally not called by the user." messages. This calculator is the basic GRIND calculation class; specialized GRIND calculators inherit from this class.

Parameters

Label	Type	Description
molecule	cchemlib.Molecule	Instantiated Molecule object with defined set of Gridpoints
bin_min	float	Starting point for binning distances. Default to 2.0 Å
num_nodes	int	Number of anchor points to use when determining the GRIND point population (GPP). Default set to 4.
num_points	int	Number of GPP to populate for each field type. Default set to 100
bins	int	Number of bins to use. bin_min+bin_size*bins = max distance. Default set to 20 bins.
bin_size	float	"Width" of each bin. Default set to 1.5 Å, but smaller values generally used.
num_stddev	float	Number of std. deviations of distance to spread out anchor nodes. Default set to 2.0
stddev	float	Std. deviation of point-point pair distances. Usually approximated from a random sample of a library. Default set to 6.0 Å
get_stddev	boolean	Flag that signals whether or not the std. deviation of PPP distances should be calculated. Default set to False

Methods

parse_fields() : Parses the Gridpoint descriptor dictionary into 4 separate fields, ordering based on descriptor values. Creates four separate fields, two each for ELE and VDW.

parse_nodes(field_type) : Determines the anchor nodes for a specific field type.

get_grind_points(field_type): Determines the GRIND Point Population (GPP) for a given field type. Requires that parse_fields() and parse_nodes() have been run.

calculate_grind_for_bin(field_type, bin_num) : Calculates the GRIND descriptor for a given bin for a given field type.

calculate_grind_for_field(field_type, abs_flag=False) : Calculates the GRIND descriptors for a given field

use_full_field_points(field_type) : Calculates GRIND for a field using all descriptors in said field. Very computationally intensive, does not require anchor nodes be defined.

calculate(use_anchors=True) : Calculates the full GRIND descriptor set for the given Molecule object. Flag specifies whether or not anchor nodes are required to determine GPP.

molecule() : Returns a deep copy of the associated Molecule object. Implemented in the base Calculator class.

__init__(molecule, bin_min=2.0, num_nodes=4, num_points=100, bins=20, bin_size=1.5, num_stddev=2.0, stddev = 6.0, get_stddev=False) *See description above*

parse_fields()

Parses the Gridpoint descriptor dictionary into 4 separate fields, ordering based on descriptor values. Creates four separate fields, two each for ELE and VDW. The fields differ in the ordering of the points being from the most negative to most positive, or vice versa. **Not generally called by the user (it is a helper function) but left non-private for testing/research purposes.**

Parameters

None

Returns : (int) cchemlib.datatypes.SUCCESS if the field parsing succeeds, cchemlib.datatypes.FAIL if not.

parse_nodes(field_type)

Determines the anchor nodes for a specific field type. **Not generally called by the user (it is a helper function) but left non-private for testing/research purposes.**

Parameters

Label	Type	Description
field_type	int	Field type; types defined cchemlib.datatypes (ELE, ELE_M, VDW, VDW_P)

Returns : (int) cchemlib.datatypes.SUCCESS if the parsing succeeds, cchemlib.datatypes.FAIL if not.

get_grind_points(field_type)

Determines the GRIND Point Population (GPP) for a given field type. Requires that parse_fields() and parse_nodes() have been run. **Not generally called by the user (it is a helper function) but left non-private for testing/research purposes.**

Parameters

Label	Type	Description
field_type	int	Field type; types defined cchemlib.datatypes (ELE, ELE_M, VDW, VDW_P)

Returns : (int) cchemlib.datatypes.SUCCESS if the parsing succeeds, cchemlib.datatypes.FAIL if not.

calculate_grind_for_bin (field_type, bin_num)

Calculates the GRIND descriptor for a given bin for a given field type. **Not generally called by the user (it is a helper function) but left non-private for testing/research purposes.**

Parameters

Label	Type	Description
field_type	int	Field type; types defined cchemlib.datatypes (ELE, ELE_M, VDW, VDW_P)
bin_num	int	Bin ID for which to calculate the GRIND for

Returns : (int) cchemlib.datatypes.SUCCESS if the calculation succeeds, cchemlib.datatypes.FAIL if not.

calculate_grind_for_field (field_type, abs_flag=False)

Calculates the GRIND descriptors for a given field. Can use the signed values or absolute values of descriptors for this calculation. **Not generally called by the user (it is a helper function) but left non-private for testing/research purposes.**

Parameters

Label	Type	Description
field_type	int	Field type; types defined cchemlib.datatypes (ELE, ELE_M, VDW, VDW_P)
abs_flag	boolean	Flag for using absolute values instead of signed values of descriptors. Defaults to False.

Returns : (int) cchemlib.datatypes.SUCCESS if the parsing succeeds, cchemlib.datatypes.FAIL if not.

use_full_field_points(field_type)

Calculates the GRIND descriptors for a given field using all the points of the field. For large molecules, this takes a LONG time to calculate, and it is not parallelized. Not recommended for anything other than substituent investigations. No abs_flag. **Not generally called by the user (it is a helper function) but left non-private for testing/research purposes.**

Parameters

Label	Type	Description
field_type	int	Field type; types defined cchemlib.datatypes (ELE, ELE_M, VDW, VDW_P)

Returns : (int) cchemlib.datatypes.SUCCESS if the calculation succeeds, cchemlib.datatypes.FAIL if not.

calculate(use_anchors=True)

Calculates the GRIND descriptors for the associated Molecule.

Parameters

Label	Type	Description
use_anchors	boolean	Flag for using anchors or full field points. Defaults to True (recommended!)

Returns : (int) cchemlib.datatypes.SUCCESS if the calculation succeeds, cchemlib.datatypes.FAIL if not.

Controllers (ccheminfolib.cdesclib.Controller)

These classes are responsible for interfacing with computational chemistry programs. Generally this involves code to write input files, but most classes are provided functions to actually launch jobs given the proper permissions and programs installed. In most cases, job files are written by the controller and the user is directed to create scripts for launching jobs on the cluster as of this writing. There are initial plans on writing either a web backend or some other script set for doing the job launch directly using these Controllers, but at the time of this writing we rely on outside scripts for that.

class ccheminfolib.cdesclib.Controller(type)

Controller superclass.

Parameters

Label	Type	Description
type	int	Signals the program this class interfaces with.

Methods

None

__init__(type) *See description above*

class ccheminfolib.cdesclib.NWChemController(molecule, functional='b3lyp', basis='6-311G**', molchg=0, geomopt=False)

This controller interfaces with the NWCHEM program on the cluster or on a private workstation to run ESP calculation jobs. Currently only writes the nwchem input file (.nw file). Note: For the method listing in Controllers, while all methods are included for development/research understanding, many are helper functions and SHOULD NOT be called in a main script. These are marked as such with "Generally not called by the user." messages. This calculator is the basic GRIND calculation class; specialized GRIND calculators inherit from this class.

Parameters

Label	Type	Description
molecule	cchemlib.Molecule	Instantiated Molecule object with defined set of Gridpoints
functional	string	DFT/HF functional to use in the SCF calculation Default: b3lyp
basis	string	Basis set to use in the SCF calculation. Default: 6-311G**
molchg	int	Formal charge of the input molecule. Default: 0
geomopt	boolean	Flag for geometry optimization. Generally not used (long calc time). Default: False

Methods

write_nw_file() : Writes the .nw file for the ESP calculation.

__init__(molecule, functional='b3lyp', basis='6-311G', molchg=0, geomopt=False)** *See description above*

write_nw_file(filename, scratch_dir, permanent_dir)

Writes the .nw file given the parameters supplied in the initialization of the Controller.

Parameters

Label	Type	Description
filename	string	Full path filename to where the NW file should be written.
scratch_dir	string	Full path to the scratch directory for the job for NWCHEM. Note: Take into account this is defined based on what system the jobs are run on.
permanent_dir	string	Location of the input files on the system the jobs will be run on.

Returns: (int) cchemlib.datatypes.SUCCESS if file write succeeds, cchemlib.datatypes.FAIL if not.

class ccheminfolib.cdesclib.JaguarController(molecule, operator='B3LYP', basis='3-21G*', molchg=0)

This controller interfaces with the Jaguar program on the cluster or on a private workstation to run ESP calculation jobs. Currently writes the jaguar input file (.in file) and the grid specification file (generally .dat file). This controller has functions for launching Jaguar but these are rarely used due to desired batch running. Note: For the method listing in Controllers, while all methods are included for development/research understanding, many are helper functions and SHOULD NOT be called in a main script. These are marked as such with "Generally not called by the user." messages. This calculator is the basic GRIND calculation class; specialized GRIND calculators inherit from this class.

Parameters

Label	Type	Description
molecule	cchemlib.Molecule	Instantiated Molecule object with defined set of Gridpoints
operator	string	DFT/HF functional to use in the SCF calculation Default: B3LYP
basis	string	Basis set to use in the SCF calculation. Default: 3-21G*
molchg	int	Formal charge of the input molecule. Default: 0

Methods

write_grid_file(grid_directory=os.getcwd()+'\\', write_directory=os.getcwd()+'\\') : Writes the custom grid coordinates to a text file for the ESP calculation.

write_in_file(write_directory=os.getcwd()+'\\', directory=os.getcwd()+'/') : Writes the jaguar input file (.in file) for the ESP calculation.

run_job(command_options, command_string='/share/apps/Schrodinger2014-4/jaguar', wait=True) : Launches jobs based on the command_string and command_options provided.

__init__(molecule, operator='B3LYP', basis='3-21G*', molchg=0) *See description above*

write_grid_file(grid_directory=os.getcwd()+'\\', write_directory=os.getcwd()+'\\')
Writes the custom grid coordinates to a text file for the ESP calculation. Default parameters assume this is being run on a Windows workstation.

Parameters

Label	Type	Description
grid_directory	string	Full path directory where the text grid file will be located for the job (on the system that RUNs the job)
write_directory	string	Full path directory to where the text file should be written on the system running this method.

Returns: (int) cchemlib.datatypes.SUCCESS if file write succeeds, cchemlib.datatypes.FAIL if not.

write_in_file(write_directory=os.getcwd()+'\\', directory=os.getcwd()+'/')
Writes the jaguar input file (.in file) for the ESP calculation.

Parameters

Label	Type	Description
write_directory	string	Full path directory to where the text file should be written on the system running this method.
directory	string	Full path directory where the file will be located on job launch.

Returns: (int) cchemlib.datatypes.SUCCESS if file write succeeds, cchemlib.datatypes.FAIL if not

job_run(command_options, command_string='/share/apps/Schrodinger2014-4/jaguar', wait=True)
Launches jaguar jobs based on the command_string and command options

Parameters

Label	Type	Description
command_options	list	List of strings that comprise command options for Jaguar. Should be in the order they'd be placed in the launch command. Should contain run command, host lists, etc.
command_string	string	Full path to the jaguar executable. Should terminate with 'jaguar'. Default should be updated when the Schrodinger suite is updated on the cluster.
wait	boolean	Flag to indicate whether the job should be launched in the background (wait=False) or not (wait=True). Defaults to True.

Returns: (int) cchemlib.datatypes.SUCCESS if the job submission (wait=False) succeeds or the job itself succeeds (wait=True), cchemlib.datatypes.FAIL if not

cqsarlib Documentation

Libraries (ccheminfolib.cqsarlib.library)

class ccheminfolib.cqsarlib.Library(name)

Base class for the Library objects. These objects organize Molecule objects and their associated observable and descriptor data. All Library classes inherit from this base class. So far, only DescriptorLibrary is fully implemented and thus is the only class listed other than this one

Parameters

Label	Type	Description
name	string	Name for the library

Methods

None

__init__(name) *See description above*

class ccheminfolib.cqsarlib.DescriptorLibrary(name, n_descriptors_per_mol)

This class defines an object that organizes Molecule objects with their associated descriptors and observable data. This is the object that is generally loaded by modeling scripts that use outside packages for modeling for ease of access to calculated descriptors and observables. Note: This does a lot of collating and the run times can be long for large *in silico* libraries.

This class was originally built to construct itself from supplied Molecule objects, but it was determined that a faster and more flexible way to do this is to use Molecule objects and process the descriptor data separately (see examples), and then use a file of descriptor data + observable data to construct this loadable serializable datatype for further use.

Parameters

Label	Type	Description
name	string	Name of the library. Optional, but used for file writing.
n_descriptors_per_mol	int	Total number of operational descriptors per Molecule object.

Methods

addObservable(mol_label, observable) : Attempts to adds an observable for a given Molecule.
getObservable(mol_label) : Attempts to return the observable value associated with a Molecule.
inTrainingSet(mol_label) : Returns True or False for if the supplied mol_label is in the training set.
addDescriptor(mol_label, descriptor_id, descriptor) : Attempts to add a given Descriptor object to the Molecule object referenced.
addMolDescriptors(mol_label, descriptors) : Attempts to add a full descriptor set to the labeled Molecule object, if it exists.
getMolDescriptor(mol_label, descriptor_id) : Attempts to return the specified descriptor from the specified Molecule
getMolDescriptors(mol_label) : Attempts to return the full descriptors for the specified Molecule
getDescriptorIDs() : Attempts to return a list of descriptor ID numbers.

__init__(name, n_descriptors_per_mol) *See description above*

addObservable(mol_label, observable)

Attempts to add the observable to a dictionary of observables, replacing the current observable specified for this molecule. This function will print a warning if no descriptors are defined for the specified Molecule, but the observable will be appended.

Parameters

Label	Type	Description
mol_label	string	Specific molecule label, generally the serial num filename.
observable	float	Currently the desired observable value to model.

Returns : (int) cchemlib.datatypes.SUCCESS if the observable is added to a known Molecule. Will return datatypes.FAIL if the molecule has no specified descriptors already added.

getObservable(mol_label)

Attempts to return the observable of the specified Molecule.

Parameters

Label	Type	Description
mol_label	string	Specific molecule label, generally the serial num filename.

Returns : (float) Observable for the specified Molecule. cchemlib.datatypes.FAIL if the observable value doesn't exist for the specified Molecule.

inTrainingSet(mol_label)

Returns True if the specified molecule has an associated observable value, False if not.

Parameters

Label	Type	Description
mol_label	string	Specific molecule label, generally the serial num filename.

Returns : (boolean True if the specified molecule has an associated observable value, False if not. Molecule labels are sorted into a training set list if an observable value has been provided for that Molecule.

addDescriptor(mol_label, descriptor_id, descriptor)

Attempts to add a descriptor for the specified Molecule. Note: To save space, the Molecule objects themselves have been removed from the DescriptorLibrary. To save space, the descriptors of each Molecule object is added to this as a raw value instead of an object.

Parameters

Label	Type	Description
mol_label	string	Specific molecule label, generally the serial num filename.
descriptor_id	int	ID of the descriptor. IMPORTANT FOR ORDERING THE DESCRIPTOR SET
descriptor	float	Value of the descriptor being added. THIS IS NOT A cchemlib.Descriptor object

Returns : (int) cchemlib.datatypes.SUCCESS if addition succeeds, cchemlib.datatypes.FAIL if not.

addMolDescriptors(mol_label, descriptors)

Attempts to add a dictionary of descriptors for the specified Molecule. Note: To save space, the Molecule objects themselves have been removed from the DescriptorLibrary. To save space, the descriptors of each Molecule object is added to this as a raw value instead of an object.

Parameters

Label	Type	Description
mol_label	string	Specific molecule label, generally the serial num filename.
descriptors	dict	Dictionary of the style {descriptor_id:descriptor_value}

Returns : (int) cchemlib.datatypes.SUCCESS if addition succeeds, cchemlib.datatypes.FAIL if not.

getMolDescriptor(mol_label, descriptor_id)

Attempts to retrieve the descriptor value of the specified descriptor for the specified Molecule. Note: To save space, the Molecule objects themselves have been removed from the DescriptorLibrary. To save space, the descriptors of each Molecule object is added to this as a raw value instead of an object.

Parameters

Label	Type	Description
mol_label	string	Specific molecule label, generally the serial num filename.
descriptor_id	int	ID of the descriptor of interest

Returns : (float) Specified descriptor value on success, (int) cchemlib.datatypes.FAIL if not.

getMolDescriptors(mol_label)

Attempts to return the dictionary of descriptors for the specified Molecule. Note: To save space, the Molecule objects themselves have been removed from the DescriptorLibrary. To save space, the descriptors of each Molecule object is added to this as a raw value instead of an object.

Parameters

Label	Type	Description
mol_label	string	Specific molecule label, generally the serial num filename.
descriptors	dict	Dictionary of the style {descriptor_id:descriptor_value}

Returns : (dictionary of float values) Specified descriptors on success, (int) cchemlib.datatypes.FAIL if not.

getDescriptorIDs()

Attempts to return the list of descriptor ID numbers

Parameters

None

Returns : (list of int) Current list of descriptor IDs that are consistent across the Library; (int) cchemlib.datatypes.FAIL if unsuccessful.