Developer reference

`molSimplify` version 1.0

by

E. Ioannidis

KULiK GROUP

March, 2016

Contents

1 Directory tree

The files that comprise this project are shown in the following directory tree:

```
molSimplify
├── main.py # start of the program
├── pybel.py # modified pybel
├── Bind/ # local extra molecules folder
│   ├── *.mol # extra molecules mol files
│   └── bind.dict # dictionary of extra molecules
├── Cores/ # local cores folder
│   ├── *.mol # local cores mol files
│   └── cores.dict # local cores dictionary
├── Classes/ # classes definitions
│   ├── atom3D.py # atom3D class definition
│   ├── globalvars.py # global variables
│   ├── mGUI.py # main GUI class definition
│   ├── mol3D.py # mol3D class definition
│   └── mWidgets.py # custom GUI widgets class
├── Data/ # folder with useful data
│   ├── *.dat # backbones for common geometries
│   └── ML.dat # metal-ligand bond lengths database
├── icons/ # icons for GUI folder
│   └── *.png # icons for GUI
├── Ligands/ # local ligands folder
│   ├── *.mol # local ligands mol files
│   ├── ligands.dict # local ligands dictionary
│   └── simple_ligands.dict # simple ligands alternative names
└── Scripts/ # folder with main scripts
    ├── addtodb.py # local database interaction
    ├── dbinteract.py # external database interaction
    ├── enerator.py # initial driver called by GUI
    ├── geometry.py # geometric operations
    ├── grabguivars.py # gathers data from GUI to input file
    ├── inparse.py # parses input files
    ├── io.py # loads cores/ligands/extra molecules
    ├── jobgen.py # generates jobscripts
    ├── postmold.py # parses molden files for MO info
    ├── postmwfn.py # uses Multiwfn for post processing
    ├── postparse.py # generates runs summary and nbo
    ├── postproc.py # driver for post processing
    ├── qcgen.py # generates quantum chemistry input
    ├── rungen.py # coordinates file generation
    └── structgen.py # builds/modifies structures
```

The routines included in each of the aforementioned files are explained in detail next.

2 Classes

2.1 atom3D.py

This file defines a custom atom class named atom3D. Each object has the following attributes: mass, atomic number, covalent radius, 3D coordinates and symbol.

2.1.1 __init__()

This is the constructor of the class the takes as arguments the symbol and the 3D coordinates of the atom. It pulls the atomic mass from a database included in `globalvars.py` and if it doesn't find the corresponding atomic symbol in the database it assumes a carbon atom.

2.1.2 coords()

Returns a list with the 3D coordinates of the atom.

2.1.3 distance()

Calculates the distance with another atom in 3D space.

2.1.4 distancev()

Returns the difference vector with respect to another atom in 3D space.

2.1.5 ismetal()

Returns true or false if the atom is a metal or not. It compares the symbol with a database from `globalvars.py`.

2.1.6 symbol()

Returns the symbol of the atom.

2.1.7 translate()

Translates the atom in 3D space by δx, δy, δz specified as input.

2.1.8 __repr__()

Prints the various methods available in the class.

2.2 globalvars.py

This class offers data and various global functions useful throughout the program. In the files there is a big dictionary with properties of atoms, a list of metals and a list of element symbols.

2.2.1 mybash()

This function uses the subprocess module in python to run a shell command and catch the output by redirecting `stdout`. It returns the output of the shell command.

2.2.2 __init__()

The constructor of the globalvars class is mainly responsible for finding the appropriate paths that are required for the program to run. It looks up the system type (linux or OSX) and then tries to read the configuration file `/.molSimplify` in order to assign 3 paths. The attribute `installdir` of the Class contains the installation directory of the program needed to read the local molecule database, icons and data, the `cemdbdir` attribute contains the folder where the external chemical databases are located and the attribute `multiwfn` contains the path of the Multiwfn executable needed for post processing. It also has information about the path of the home directory of the user, jobs running directory and some global counters such as the number of structures generated.

2.2.3 amass()

Returns the dictionary with the atomic properties.

2.2.4 metals()

Returns a list of metals.

2.2.5 elementsbynum()

Returns a list of elements sorted by atomic number.

2.3 mGUI.py

This file defines the main GUI class.

2.3.1 __init__()

The constructor of the GUI is responsible for creating all objects that will be visible when the user starts the program and interacts with it. It uses many different widgets provided

by the PyQt5 library and contains windows, push buttons, check buttons, dropdown boxes, sliders, editable text boxes and menu bars. The user can interact with many of these widgets through corresponding callback functions that are described next.

2.3.2 qDBload()

Function that loads a molecule from a file to be added in the local molecule database.

2.3.3 enableDB()

Callback for button that enables addition/removal from local database and brings up the corresponding window.

2.3.4 qaddDB()

Adds the specified molecule to the local database.

2.3.5 qdelDB()

Deletes the specified molecule from the local database.

2.3.6 dbchange()

Enables and disables input options according to whether a core, ligand or extra molecule are added or removed.

2.3.7 postprocGUI()

Initiates the post processing by calling for the input file generation and then passing control to the post processing routines.

2.3.8 qcDBload()

Loads a molecule from a file to be used for interaction with external databases.

2.3.9 searchDBW()

Enables interface for interaction with external databases and loads the corresponding window. It also checks for existing databases and prompts the user to specify a different folder if no database is found or if no database folder is specified.

2.3.10 qaddcDB()

Initiates the screening or similarity search with external databases by constructing the appropriate input file and passing control to the corresponding module.

2.3.11 cdbchange()

Enables and disables input in the database.

2.3.12 runGUI()

Initiates structure generation by creating the appropriate input file and passing control to the corresponding module.

2.3.13 drawligs()

This module loads the ligands specified in the GUI, creates a .svg representation using babel and then converts this representation to a .png file using imagemagick. The final png file is then loaded and visualized.

2.3.14 enableffinput()

Enables input for the force field optimization.

2.3.15 dirload()

The user browses for the running directory that is defined in this routine.

2.3.16 qdumpS()

Grabs all data from the GUI and creates corresponding input files.

2.3.17 getscreensize()

Returns the screen size in pixels.

2.3.18 sliderChanged()

Changes the text in the distortion slider.

2.3.19 matchgeomcoord()

Matches the corresponding geometries with the coordination number specified.

2.3.20 enableemol()

Enables input for the specification of extra molecule placement.

2.3.21 qcinput()

Brings up the corresponding window for specifying the input for quantum chemistry input file generation.

2.3.22 jobdef()

Creates a file with the default options for jobscript generation.

2.3.23 qctdef()

Creates a file with the default options for terachem input file generation.

2.3.24 qcgdef()

Creates a file with the default options for GAMESS input file generation.

2.3.25 qcqdef()

Creates a file with the default options for QChem input file generation.

2.3.26 enableqeinput()

Enables input for quantum chemistry input file generation.

2.3.27 enablejinput()

Enables input for jobscript generation.

2.3.28 jobenable()

Loads the window for jobscript generation.

2.3.29 setupp()

Loads the window for post processing. It also checks for a valid Multiwfn installation and prompts the user to specify a valid path if no functioning Multiwfn code is found.

2.3.30 pdload()

Loads directory with results to be used for post processing.

2.4 mol3D.py

This class defines a 3D molecule class. Each mol3D object has the following attributes: a list of atom3D objects (atoms), the number of atoms comprising the molecule, the total mass, the total size in Angstrom, charge, an openbabel molecule object (OBmol), a list of connection atoms, denticity, identifier used in naming folders (ident) and a globalvars object (globs).

2.4.1 __init__()

The constructor of the class initializes the various attributes of a mol3D object.

2.4.2 addatom()

Adds an atom3D object to the molecule.

2.4.3 alignmol()

Aligns the molecule so that an atom of the molecule matches another atom (not from the same molecule) specified in the input.

2.4.4 centermass()

Calculates the center of mass of the molecule.

2.4.5 centersym()

Calculates a center of symmetry treating all atoms, heavy and light as equivalent.

2.4.6 convert2mol3D()

Converts an openbabel molecule object to a mol3D atom by extracting the corresponding 3D coordinates.

2.4.7 combine()

Combines two molecules into one.

2.4.8 coordsvec()

Returns a vector with the xyz coordinates of all the atoms in the molecule.

2.4.9 copymol3D()

Copies all the attributes of one mol3D to another.

2.4.10 deleteatom()

Deletes an atom from the molecule specified with its index.

2.4.11 deleteatoms()

Deletes a list of atoms.

2.4.12 deleteHs()

Deletes all Hydrogens from the molecule.

2.4.13 distance()

Calculates the distance between two molecules.

2.4.14 findcloseMetal()

Returns the index of the metal closest to a specified atom.

2.4.15 findAtomsbySymbol()

Returns a list with the indices of all atoms with the specified symbol.

2.4.16 findsubMol()

This module runs a connectivity graph search and finds a submolecule within the current molecule. The user specifies the two atoms that would be disconnected if the submolecule was separated from the current molecule. It loops over all the atomic connections starting from the first reference atom and eliminating atoms that don't belong to the submolecule.

2.4.17 getAtom()

Returns the atom3 object that corresponds to the specified index.

2.4.18 getAtoms()

Returns the list of atom3D objects in the molecule.

2.4.19 getBondedAtoms()

Returns list of all the atoms bonded to a specified atom in the molecule. It uses the covalent radii of the atoms to detect bonding.

2.4.20 getBondedAtomsnotH()

Returns atoms that are bonded to the reference atom and that are not Hydrogens.

2.4.21 getHs()

Returns all the Hydrogens in the molecule.

2.4.22 getHsbyAtom()

Returns the list of Hydrogens bonded to a specific atom in the molecule.

2.4.23 getClosestAtom()

Returns the index of the closest atom with respect to a specified atom in the molecule.

2.4.24 getMask()

Returns the center of mass of the atoms specified by an atomic mask. This mask can be a list of atomic indices, an expression such as 1-5, an atomic symbol or any combination of those.

2.4.25 getClosestAtomnoHs()

Returns the closest non-Hydrogen atom with respect to a specified atom in the molecule.

2.4.26 getOBmol()

Loads an openbabel molecule from a file using pybel and saves it to the OBmol attribute of the corresponding mol3D object.

2.4.27 initialize()

Reinitializes the molecule.

2.4.28 maxdist()

Calculates the maximum distance between the atoms in 2 molecules.

2.4.29 mindist()

Calculates the minimum distance between the atoms in 2 molecules.

2.4.30 mindistmol()

Calculates the minimum distance between all atoms in the molecule.

2.4.31 mindistnoH()

Calculates the minimum distance between the non-Hydrogen atoms in 2 molecules.

2.4.32 molsize()

Calculates the molecular size based on the maximum distance of an atom from the center of mass of the molecule.

2.4.33 overlapcheck()

Checks for overlap between two molecules by looping over all atoms and comparing their distance to the sum of their covalent radii.

2.4.34 printxyz()

Prints the xyz coordinates of the molecule.

2.4.35 readfromxyz()

Creates a mol3D object from xyz file.

2.4.36 rmsd()

Calculates the RMSD of two molecules.

2.4.37 sanitycheck()

Checks if atoms within a molecule overlap.

2.4.38 translate()

Translates the molecule by δx,δy,δz.

2.4.39 writegxyz()

Writes a xyz-like file for GAMESS input.

2.4.40 writexyz()

Writes an xyz file with the coordinates of the complex.

2.4.41 writemxyz()

Writes an xyz file combinding 2 molecules.

2.4.42 __repr__()

Prints all the available methods within the class.

2.5 mWidgets.py

This class contains various widgets that are redefined for using them in the GUI and is basically an extension of the existing Widgets provided by PyQt5.

3 Scripts

3.1 addtodb.py

This script contains modules for interacting with the local molecule database.

3.1.1 addtolb()

Adds new molecule to local ligand database. It checks if the new ligand already exists and also if the input is correct.

3.1.2 addtocdb()

Adds new molecule to local cores database. It checks if the new core already exists and also if the input is correct.

3.1.3 addtobdb()

Adds new molecule to local extra molecules database. It checks if the new core already exists and also if the input is correct.

3.1.4 removefromDB()

Removes molecule from database, either core, ligand or extra molecule.

3.2 dbinteract.py

Script for interacting with external molecular databases.

3.2.1 setupdb()

Looks for sdf and fs files in the directory specified in globs.chemdbdir and prints an error if no file exists.

3.2.2 obfilters()

Prints list of available keywords for filtering.

3.2.3 checkscr()

Checks if filtering options exists in the input file. Parses the input and passes it to the screening routine.

3.2.4 getsimllar()

Performs similarity search based on SMILES string or molecule from file using babel.

3.2.5 stripsalts()

Remove salts from SMILES strings.

3.2.6 matchsmarts()

Based on the initial SMARTS string in the similarity search or screening define correct connection atom for the results.

3.2.7 dbsearch()

Main driver for the interaction. Parses the input file and coordinates the various routines. Initially similarity search is performed to get an initial set of SMILES strings and then its filtered based on the filtering options. Finally salts are stripped and unique strings are isolated and printed in a multimolecular SMILES file.

3.3 generator.py

3.3.1 startgen()

Main driver for starting the three modules in the program. It reads the input file and initiates one of the three corresponding modules. Either structure generation, database search or post processing. It also loops over multiple cores if more than one are specified.

3.4 geometry.py

This script contains the geometric manipulation routines that are used during the structure generation.

3.4.1 norm()

Calculates the Euclidean norm of a vector.

3.4.2 distance()

Distance between 2 points in 3D space.

3.4.3 vecdiff()

List with the difference of two vectors.

3.4.4 checkcolinear()

Checks if 3 points in 3D space are colinear.

3.4.5 vecangle()

Calculates the angle between 2 vectors in 3D space.

3.4.6 rotation_params()

Based on 3 points R_0, R_1, R_2 in space it calculates the angle between R_{21} and R_{10} and the unit vector perpendicular to the plane spanned by R_{21} and R_{10}.

3.4.7 kabsch()

Aligns one molecule with respect to another to minimize the RMSD value between them using the Kabsch algorithm.

3.4.8 ReflectPlane()

Contains roto-translation matrix for reflection of a point through a plane with given normal vector and a point on plane.

3.4.9 PointRotateAxis()

Rotates a point by a specific angle about a given axis.

3.4.10 PointTranslateSph()

Converts to spherical coordinates and places a point given an angle, reference point and distance.

3.4.11 PointRotateSph()

Rotates a point about x,y,z axes in spherical coordinates given 3 angles θ_x, θ_y, θ_z.

3.4.12 reflect_through_plance()

Reflects a molecule through a plane with given normal vector and point.

3.4.13 rotate_around_axis()

Rotates molecule around specified axis by a given angle.

3.4.14 setPdistance()

Translates molecule to set the distance of an atom in its molecule with respect to a reference point.

3.4.15 setcmdistance()

Translates molecule to set the distance of its center of mass with respect to a reference point.

3.4.16 protate()

Rotates molecule in spherical coordinates with respect to a specified point given angle and distance.

3.4.17 cmrotate()

Rotates molecule about its center of mass by angles θ_x, θ_y, θ_z.

3.4.18 rotateRef()

Rotates molecule about a reference point by angles θ_x, θ_y, θ_z.

3.4.19 aligntoaxis()

Translates molecule and aligns its center of mass to a given axis.

3.5 grabguivars.py

3.5.1 writeinputc()

Writes input to input file.

3.5.2 writeinputp()

Writes input for post processing to input file.

3.5.3 writeinputf()

Writes input for structure generation to input file.

3.5.4 grabguivars()

Collects input parameters from the GUI and writes them to input file.

3.5.5 grabguivarstc()

Grabs GUI options for terachem and writes them to the default file.

3.5.6 grabguivarsgam()

Grabs GUI options for GAMESS and writes them to default file.

3.5.7 grabguivarsqch()

Grabs GUI options for QChem and writes them to default file.

3.5.8 grabguivarsjob()

Grabs GUI options for jobscript generation and writes them to default file.

3.5.9 grabdbguivars()

Grabs GUI options for interaction with external databases and writes them to input file.

3.5.10 grabguivarsP()

Grabs GUI options for post processing and writes them to input file.

3.5.11 loadfrominputtc()

Loads input file for terachem input generation.

3.5.12 loadfrominputgam()

Loads input file for GAMESS input generation.

3.5.13 loadfrominputqch()

Loads input file for QChem input generation.

3.5.14 loadfrominputjob()

Loads input file for jobscript generation.

3.5.15 loadfrominputfile()

Loads general GUI data from input file.

3.6 inparse.py

3.6.1 checkinput()

Checks if core and ligands are specified and prints warnings.

3.6.2 cleaninput()

Cleans and consolidates input for various keywords.

3.6.3 parseinput()

Parses input files, collects variables and assigns them to global structure args.

3.6.4 parsecommandline()

Creates the parser args which holds all the input parameters used in modules of the program.

3.7 io.py

3.7.1 getgeoms()

Gives geometries available within the program for simple coordination complexes.

3.7.2 readdict()

Reads python dictionaries from file.

3.7.3 getligs()

Reads dictionary with local ligand molecules.

3.7.4 checkTMsmiles()

Checks and corrects SMILES strings for transition metals.

3.7.5 getbinds()

Reads dictionary with local extra molecules.

3.7.6 loaddata()

Loads data for the available backbones.

3.7.7 core_load()

Loads a core either from the local database, a SMILES string or a file and saves it as an OBmol molecule.

3.7.8 lig_load()

Loads a ligand either from the local database, a SMILES string or a file and saves it as an OBmol molecule.

3.7.9 bind_load()

Loads an extra molecule either from the local database, a SMILES string or a file and saves it as an OBmol molecule.

3.8 jobgen.py

This script generates the jobscripts.

3.8.1 sgejobgen()

Generates jobscript for the SGE scheduler. Default parameters are specified and overwritten based on the user's input.

3.8.2 slurmjobgen()

Generates jobscript for the SGE scheduler. Default parameters are specified and overwritten based on the user's input.

3.9 postmold.py

This scripts parses molden files and collects information about the molecular orbitals.

3.9.1 getrange()

This module locates the indices of the atomic S,P,D orbitals that correspond to the metal in the structure.

parse()

This module parses a module file and collects information about the contributions of the metal atomic orbitals to the MOs. It prints this information to a *_orbs.txt file.

3.9.2 parsed()

This module parses the generated *_orbs.txt files and calculates properties such as homo, lumo, fermi energy and d-band center.

3.9.3 getresd()

This module collects the information from parsed() and prints them into the summary file avorbs.txt.

3.9.4 moldpost()

Loops over the molden files and coordinates the parsing.

3.10 postmwfn.py

This script uses Multiwfn to perform post processing including population analysis, delocalization indices and cube file generation.

3.10.1 calculate_integral()

Calculates an integral by summation.

3.10.2 spreadcalc()

Calculates probabilistic modes that correspond to mean, variance and skewness of a distribution for electron density, ELF and laplacian.

3.10.3 calcHELP()

Calculates the HELP value by integration.

3.10.4 parsecube()

Reads and parses a Gaussian cube file.

3.10.5 wfncalc()

Coordinates the calculation of wavefunction properties such as density, ELF and HELP values.

3.10.6 cubespin()

Generates the alpha and beta electron density cube files based on the density and spindensity cube files.

3.10.7 getcubes()

Generates cube files for the density, spindensity, ELF and alpha and beta spin densities.

3.10.8 getwfnprops()

Coordinates the calculation of average properties from the cube files and prints a summary in wfnprops.txt.

3.10.9 getcharges()

Performs population analysis using VDD, Hirshfeld and Mulliken printing the summary in charges.txt.

3.10.10 deloc()

Calculates the delocalization and the delocalization index for the metal printing the summary in deloc_res.txt.

3.11 postparse.py

This script gives a summary of the calculation and parses nbo output.

3.11.1 nbo_parser()

Parses nbo output and extracts information about charges, contributions of orbitals to NBO etc.

3.11.2 nbopost()

Coocrdinates parsing of nbo output and writes the summary to nbo.txt.

3.11.3 gampost()

Parses output from GAMESS and prints the summary in gam-results.txt.

3.11.4 terapost()

Parses output from terachem and prints the summary in tera-results.txt.

3.12 postproc.py

This module coordinates the post processing by calling the various routines according to the specified user input.

3.13 qcgen.py

This script generates the input files for quantum chemistry calculations.

3.13.1 multitcgen()

Loops over multiple DFT functionals and generates the corresponding input files for terachem.

3.13.2 tcgen()

Generates the input file for terachem.

3.13.3 xyz2gxyz()

Converts an xyz file to a GAMESS format that will be included in the input file.

3.13.4 tcgen()

Generates the input file for GAMESS.

3.13.5 multigamgen()

Loops over multiple DFT functionals and generates the corresponding input files for GAMESS.

3.14 qcgen.py

3.14.1 tcgen()

Generates the input file for QChem.

3.14.2 multitcgen()

Loops over multiple DFT functionals and generates the corresponding input files for QChem.

3.15 rungen.py

This script coordinates the generation of the various files in the program including structures, input files and jobscripts.

3.15.1 randomgen()

Total random generation by selecting ligands with no constraints.

3.15.2 getconstsample()

Generates all possible combinations of ligands from the database and collects a sample that satisfies the constraints imposed by the user.

3.15.3 constrgen()

The constrained random generation module parses the constraints imposed by the user, generates a sample that satisfies those constraints and then runs the structure generation routines.

3.15.4 multigenruns()

Runs the structure generation for multiple charges and spin states.

3.15.5 checkmultilig()

This module checks if a specified moelcular file contains more than one ligand (from a database search for example). If there are many ligands, it loops over all of them generating all possible complexes.

3.15.6 rungen()

Main driver for file generation. It builds a folder name for the complex, creates the corresponding directory and then calls the routines for the structure generation, the input file and the jobscript generation.

3.16 structgen.py

This is the script that performs the structure generation.

3.16.1 setdiff()

Gets the difference between two sets of data (lists).

3.16.2 getbackcombs()

Lists all possible combinations of connection points in the backbone for a specified geometry.

3.16.3 getnupdateb()

Returns a combination of connection points that satisfies the denticity of the ligand and also updates the list of available connection points.

3.16.4 getsmilescat()

Either reads or calculates the connection atoms for a given SMILES string.

3.16.5 getsmident()

Returns the denticity of a SMILES ligand.

3.16.6 modifybackbone()

If ligand angles are specified, the backbone is modified to accompany the user input.

3.16.7 modifybackbonep()

If angles of the backbone points are specified, the backbone is modified to accompany the user input.

3.16.8 distortbackbone()

Distorts the backbone according to the degree of distortion specified in the input.

3.16.9 getbondlength()

Returns the approximated metal-ligand bond length. Initially the program looks for the M-L bond length in the database and if doesn't find an exact match it tries to find a similar one. If no similar exists then it approximates the M-L bond length with the sum of the corresponding covalent radii.

3.16.10 ffopt()

Force field optimization routine. The program converts a mol3D object to an OBmol molecule, sets up a forcefield with appropriate constraints and performs the optimization.

3.16.11 ffoptd()

Similar to ffopt but for custom cores (different constraints).

3.16.12 getconnection()

Uses manual search and force field optimization to select the optimum position for adding a ligand to a custom core. The program tries to both minimize steric repulsion between adjacent atoms and to align the center of mass of the new ligand correctly.

3.16.13 mcomplex()

This module generates metal coordination complexes by gradually adding ligands to a simple metal core. The procedure depends on the denticity of the ligand with various geometric

routines performed at each step. Force field optimization can be included here as well.

3.16.14 customcore()

This module generates complexes by either functionalizing a custom core with additional ligands or by replacing existing ligands with new ones. Force field optimization can be included here as well.

3.16.15 structgen()

This module builds the core complexes by calling customcore() or mcomplex() and then places additional molecules around the core. It finally generates the corresponding xyz files and returns the paths to the xyz files.

4 Data files

4.1 Core dictionary

The file cores.dict contains all the cores existing in the local database. Each entry includes a key corresponding to the core name and values corresponding to the mol file of the core, the connection atom(s) index in the structure and the maximum denticity allowed for ligands that are going to functionalize that core.

4.2 Ligand dictionary

The file ligands.dict contains all the ligands existing in the local database. Each entry includes a key corresponding to the core name and values corresponding to the ligand mol file, a short unique identifier used in folder naming followed by the indices of the connection atoms in the ligand.

4.3 Extra molecule dictionary

The file bind.dict contains all the extra molecules existing in the local database. Each entry includes a key corresponding to the extra molecule's name and the corresponding mol or xyz file.

4.4 Simple ligands dictionary

The simple_ligands.dict file contains pairs of popular ligand names with the corresponding SMILES string such as NH3:N.

4.5 Metal-ligand bond lengths

The file ML.dat contains the DFT calculated values for metal-ligand bond lengths. Each entry includes keys such as: metal, oxidation state, spin multiplicity, element directly connected to the metal and ligand.

4.6 Backbone files

The backbone files contain the xyz coordinates of the points that comprise the backbone.