IDPConformerGenerator

Release 0.7.23

Julie Forman-Kay Lab

Mar 27, 2024

CONTENTS

IDPConformerGenerator is a flexible, modular platform for generating ensembles of disordered protein states that builds conformers by sampling backbone torsion angles of relevant sequence fragments extracted from protein structures in the RCSB Protein Data Bank.

IDPConformerGenerator can efficiently build large and diverse conformer pools of disordered proteins, with user defined options enabling variable fractional population of secondary structures, including matching those assigned based on NMR chemical shift data. These conformer pools are intended to be utilized as input for further approaches to match experimental data, such as re-weighting or sub-setting algorithms.

Note to users: IDR conformers generated with `ldrs`, specifically processed using the `align_coords()` function in `ldrs_helper.py` prior to v0.7.17 may have the wrong stereochemistry. This bug has since been fixed. Thank you for your understanding and we apologize for any inconvenience this has caused.

CHAPTER

ONE

CONTENTS

1.1 Installation

IDPConformerGenerator uses only Python-based APIs for which we expect it to run native on any system Python can run, as long as the third-party installation requirements are met.

We tested IDPConfGen on Ubuntu 18.04 LTS and 20.04 LTS as well as on WSL2.0 and the Graham cluster, an HPC resource of the Digital Research Alliance of Canada (DRAC).

Follow the steps below to install IDPConformerGenerator (`idpconfgen` or `IDPConfGen` for short) on your local machine:

1.1.1 From source

Clone from the official repository:

```
git clone https://github.com/julie-forman-kay-lab/IDPConformerGenerator
```

And navigate to the new `IDPConformerGenerator` folder:

```
cd IDPConformerGenerator
```

Run the following commands to install `idpconfgen` dependencies if you use Anaconda as your Python package manager:

```
conda env create -f requirements.yml
conda activate idpconfgen
```

Note: If you don't use Anaconda to manage your Python installations, you can use `virtualenv` and the `requirements.txt` file following the commands:

```
virtualenv idpcgenv --python=3.9
source venv/bin/activate
pip install -r requirements.txt
```

If you have difficulties installing `idpconfgen`, raise an Issue in the main GitHub repository, and we will help you.

Install `idpconfgen` in development mode in order for your installation to be always up-to-date with the repository:

```
python setup.py develop --no-deps
```

Note: The above applies also if you used `virtualenv` instead of `conda`.

Remember to active the `idpconfgen` environment every time you open a new terminal window, from within the repository folder, choose yours:

```
# Installation with Anaconda
conda activate idpconfgen

# Installation with virtualenv
source idpcgenv/bin/activate
```

To update to the latest version, navigate to the repository folder, activate the `idpconfgen` python environment as described above, and run the commands:

```
git pull

# if you used anaconda to create the python environment, run:
conda env update -f requirements.yml

# if you used venv to create the python environment, run:
pip install -r requirements.txt  --upgrade

python setup.py develop --no-deps
```

Your installation will become up to date with the latest developments.

1.1.2 From source on the Graham Cluster (DRAC)

Log-in and make sure you are in your user home directory:

```
cd
```

Load the required python packages and modules on Graham's servers:

```
module load scipy-stack dssp boost
```

Create and activate a `virtualenv` as DRAC recommends `anaconda3` not be installed in the home folder:

```
virtualenv --no-download idpcgenv
source idpcgenv/bin/activate
```

For the first time installation, install dependencies manually using `pip`. Please note that the `--no-index` searches through DRAC's available packages. If they're not available, it will install from the web:

```
pip install --no-index --upgrade pip
pip install numba --no-index
pip install pybind11 --no-index
pip install tox
```

We are ready to clone from source and installation from here will be similar to local:

```
git clone https://github.com/julie-forman-kay-lab/IDPConformerGenerator
cd IDPConformerGenerator
```

Make sure you're in the `idpcgenv` virtual environment before installing. Install with:

```
python setup.py develop --no-deps
```

When you login again to your cluster account remember to reactive the `idpconfgen` environment before using `idpconfgen`:

```
cd
source idpcgenv/bin/activate
```

1.1.3 Installing third-party software

Some functionalities of `idpconfgen` require third-party software. These are not mandatory to install unless you want to use such operations.

DSSP

IDPConfGen uses DSSP to calculate secondary structures. However, you only need DSSP if you are generated the database from scratch. If you use a prepared database JSON file you don't need to install DSSP.

Please note we are only compatible with DSSP versions 2 and 3. If you have installed DSSP version 4 (check by using the command `mkdssp --version`) please refer to this issue for a proper re-installation after removing DSSP version 4.

Install MC-SCE

IDPConformerGenerator can integrate MC-SCE to generate sidechains on top of the backbone conformers it generates, on the fly. For that you need to install MC-SCE on top of the *idpconfgen* Python environment. First, install IDPConfGen as described above. Next, follow these steps:

```
# ensure you are in the parent IDPConformerGenerator GitHub folder
# clone and enter the MC-SCE GitHub repository
git clone https://github.com/THGLab/MCSCE
cd MCSCE

# Make sure you're in the idpconfgen environment then
# install the additional dependencies using pip
pip install tensorflow
pip install tqdm
pip install pathos

# Install MC-SCE on top of IDPConformerGenerator
python setup.py develop --no-deps

# Navigate back to the IDPConformerGenerator GitHub folder and install
# `idpconfgen` again if needed
cd ../IDPConformerGenerator
python setup.py develop --no-deps
```

Now, if you choose the flag `-scm mcsce` in `idpconfgen build` command, IDPConfGen will use MC-SCE to build sidechains as backbone conformers are generated. You will see `idpconfgen build -h` has a specific group of parameters dedicated to MC-SCE, you can explore those as well.

For installation on a cluster via virtualenv, dependencies need to be manually installed as the following for MC-SCE:

```
# ensure you're in the idpcgenv and the IDPConformerGenerator GitHub folder
git clone https://github.com/THGLab/MCSCE

# MC-SCE also requires numba and tox but that's already handled in previous steps
pip install tensorflow --no-index
pip install keras --no-index
pip install tqdm --no-index
pip install pathos --no-index

# cd into the MCSCE GitHub folder and install MC-SCE
cd MCSCE
python setup.py develop --no-deps

# cd back into the IDPConformerGenerator GitHub folder and install idpconfgen on top of␣
↪MC-SCE
cd ..
python setup.py develop --no-deps
```

Install Int2Cart

IDPConformerGenerator can use Int2Cart on the fly to optimize bond geometries of the backbones calculated. For this feature, you must have a CUDA compatible GPU as well as install Int2Cart on top of the `idpconfgen` Python environment. First, install IDPConfGen as described above. Next, follow these steps. Please note that these steps are the same if you have installed idpconfgen through `virtualenv`:

```
# ensure you are in the IDPConformerGenerator GitHub folder

# Install a pre-requisite of Int2Cart: sidechainnet
git clone https://github.com/THGLab/sidechainnet
cd sidechainnet
pip install -e .
cd ..

# Install Int2Cart
git clone https://github.com/THGLab/int2cart
cd int2cart
pip install -e .
pip install pyyaml
cd ..

# you should be back in the IDPConformerGenerator GitHub folder
```

Running Int2Cart on the Graham cluster requires GPU allocations and `module load cuda`. Otherwise, installation is the same within the `idpconfgen` virtualenv.

Troubleshooting Int2Cart installation

If IDPConfGen is still giving you an error that Int2Cart is not installed, please test this import in the `idpconfgen` environment:

```
python
>>> from modelling.models.builder import BackboneBuilder
```

If you receieve this error: `ImportError: TensorBoard logging requires TensorBoard version 1.15 or above`, do the following:

```
pip install tensorboard==1.15.0
```

CheSPI

To use CSSS via the `idpconfgen cssconv` command you need CheSPI. Please refer to https://github.com/protein-nmr/CheSPI to install CheSPI.

2D

The use 2D via the `idpconfgen cssconv` command you need 2D. Please refer to https://github.com/carlocamilloni/d2D.

1.2 Usage

IDPConformerGenerator runs entirely through command-lines. Follow the explanations in this page plus the documentation on the command-line themselves.

1.2.1 Command-lines

To execute `idpconfgen` command-line, run `idpconfgen` in your terminal window, after *installation*:

```
idpconfgen
```

or:

```
idpconfgen -h
```

Both will output the help menu.

Note: All subclients have the `-h` option to show help information.

`idpconfgen` has several interfaces that perform different functions. However, there is a sequence of interfaces that need to be used to prepare the local torsion angle database and the files needed to build conformers. After these operations executed, you will end up with a single `json` file that you can use to feed the build calculations. The other files are safe to be removed.

1.2.2 IDPConfGen Small Peptide Example

The `example/` folder contains instructions to setup IDPConformerGenerator database from scratch and generate conformers for a small peptide.

Note: The purpose of this module is to test the installation of IDPConformerGenerator on a small example peptide. The database you will procure here will not be used further cases, much less in practice due to the database housing only 100 PDB IDs.

To build the torsion angle database you need to provide IDPConfGen with a list of PDB/mmCIF files. Our advice is that you use a culled list of your choice from the Dunbrack PISCES database.

The PDB chain id list should have the format of the `cull100` file in this folder, which emulates the format provided by PISCES. Actually, the only column that is read by IDPConfGen is the first column. No header lines are allowed.

```
12E8H       221  XRAY       1.900    0.22    0.27
16PKA       415  XRAY       1.600    0.19    0.23
1A05A       358  XRAY       2.000    0.20    0.28
1A0JA       223  XRAY       1.700    0.17    0.21
1A12A       413  XRAY       1.700    0.19    0.22
1A1XA       108  XRAY       2.000    0.21    0.25
```

The first three alphanumeric characters are the PDBID codes. The forth (or more) are the PDB chain identifier. mmCIF files have chain IDs of several characters.

Run the following command to download the PDB files:

```
idpconfgen pdbdl cull100 -u -n -d pdbs.tar
```

You can inspect all options of this (and any other) subclient with `-h`:

```
idpconfgen pdbdl -h
```

This execution will create a `pdbs.tar` file with the parser PDBs. Those PDBs contain only the information needed for IDPConfGen. Unnecessary chains or residues were removed.

We now proceed to the identification of secondary structure elements. For this you need to have DSSP program installed. IDPConfGen is built to be modular. You can also use any other program to calculate secondary structure but you would need to implement the respective parser. The DSSP parser is implemented. To install DSSP follow these instructions: https://github.com/julie-forman-kay-lab/IDPConformerGenerator/issues/48.

The following command will operate on the `pdbs.tar` file and will create temporary files and a result file with the DSSP information:

```
idpconfgen sscalc pdbs.tar -rd -n
```

You will see that the files `sscalc.json` and `sscalc_splittled.tar` were created. `sscalc.json` matches the sequence information with that of the secondary structure identity. `sscalc_splitted.tar` contains the PDB chains split into continuous chains.

Now we need to calculate the torsion angles. There are several options available in the command line but these are good defaults:

```
idpconfgen torsions sscalc_splitted.tar -sc sscalc.json -o idpconfgen_database.json -n
```

This will create the final file `idpconfgen_database.json`. This is the file IDPConfGen needs to generate conformers. This is the torsion angle database file. If you open it, you will see it is a regular human-readable `json` file.

Finally, to generate conformers you will use the `build` interface. The build interface has several parameters that can be use to fine-tune the conformer construction protocol. You can read deeper instructions in the documentation and client help. The following is a good default that uses the FASPR method for adding side chains:

```
idpconfgen build -db idpconfgen_database.json -seq EGAAGAASS -nc 10 --dloop-off --dany -n
```

After some time you will see 10 conformers in the folder. Please note that searching for loops is enabled by default for `--dloop`. Appending `--dhelix --dstrand` will extend sampling to alpha-helicies and beta-strands in addition to loops. For more information on usage, please view `idpconfgen build -h`.

If you would like to use a program to assign secondary structure bias based on NMR chemical shifts (e.g. CheSPI or 2D) to employ probabilistic custom secondary structure sampling (CSSS), the `probs8_[ID].txt` output from CheSPI or .TXT output from 2D would have to be standardized into a user-editable text file indicating the probability of secondary structures (based on DSSP codes) on a per residue basis. The following example will process CheSPI output and assign probabilities to L/H/E based on H/G/I/E/ /T/S/B structures on a per residue basis:

```
idpconfgen csssconv -p8 probs8_ex.txt -o csss_ex.json
```

For simplicity, secondary structures from CheSPI and 2D are grouped into L/H/E as defined by idpconfgen. If you do not want this grouping feature, please build the database above without `-rd` and run `csssconv` with `--full` to avoid grouping. To build with the CSSS file, `-csss` would have to point to the CSSS.JSON file:

```
idpconfgen build -db idpconfgen_database.json -seq EGAAGAASS -nc 10 -csss csss_ex.json --
↪dloop-off -n
```

After some time you will see 10 conformers in the folder with the probabilistic CSSS.

All IDPConfGen operations can be distributed over multiple cores. Use the flag `-n` to indicate the number of cores you wish to use. Appending only `-n` will use all available CPU threads except for one.

This is all you need to know for a basic usage of IDPConfGen. Now you can use a larger PDB database to feed the conformational sampling.

1.2.3 Building With Variable Bond-Geometry Strategies

To increase user flexibility and build parameterization,.IDPConformerGenerator has 4 bond geometry strategies to choose from: sampling (default), fixed, exact, and int2cart. These strategies could be selected from using the `--bgeo-strategy` flag during the `build` process. Please note that a new database needs to be generated with the `bgeodb` subclient to use the `exact` strategy, and Int2Cart will need to be installed to use the `int2cart` strategy (see below).

The default `sampling` strategy aims to overcome limitations of having fixed bond angles and bond lengths to increase the diversity of conformations sampled. The `fixed` strategy uses average bond geometries on a per-residue basis derived from an extended Dunbrack PISCES database `cull_d200611/200611/cullpdb_pc90_res1.6_R0.25_d200611_chains8807`. The `exact` method uses exact bond/bend angles and bond lengths for each residue in a fragment sampled from the database. To initialize the new, backwards-compatible, database use the `bgeodb` module on the `idpconfgen_database.json` previously generated using `torsions`. Reminder, this requires the `sscalc_splitted.tar` file generated by `sscalc` in the early stages of creating the database:

```
idpconfgen bgeodb sscalc_splitted.tar -sc idpconfgen_database.json -o idpconfgen_
↪extended_database.json -n
```

1.2.4 A Real Case Scenario

Note: The purpose of this module is to use IDPConformerGenerator as you would with a real protein of interest (unfolded drkN SH3 is presented here). The database procured from this module can be re-usable for future cases.

The example with a small peptide in the `example` folder is a good way to get introduced to IDPConfGen. Although building other IDP conformer ensembles use the same workflow as the previous example, we will go over more detailed usage examples with a well studied IDP, the unfolded state of the drkN SH3 domain.

Chemical shift data for the unfolded state of the drkN SH3 domain (BMRB ID: 25501) has been already processed with 2D and CheSPI and secondary structure propensity calculations can be found in `example/drksh3_example` as `drk_d2D.txt` and `probs8_25501_unfolded.txt` respectively for 2D and CheSPI output.

An extensive culled list is in `cull.tar`. Unpack it with:

```
tar -xf cull.tar
```

This is the same culled list used in the IDPConfGen main publication. However feel free to choose your own from the Dunbrack PISCES database.

Steps from now on will assume you're in the working directory of `example/drksh3_examples`.

To initialize the database if you do not already have one, we must download the PDB files from our culled list (can be found in the supplemental package in the IDPConformerGenerator paper):

```
idpconfgen pdbdl cullpdb_pc90_res2.0_R0.25_d201015_chains24003 -u -n -d pdbs.tar
```

Next we will create temporary files storing the secondary structure information for each PDB file downloaded. Later to be processed for their torsion angles:

```
idpconfgen sscalc pdbs.tar -rd -n -cmd <DSSP EXEC>
```

Please note that since IDPConfGen is a toolkit, many of these modules can be used with custom folders or `.tar` files.

Finally, torsion angles are extracted and the database we will use for future calculations can be created with the `torsions` subclient:

```
idpconfgen torsions sscalc_splitted.tar -sc sscalc.json -n -o idpconfgen_database.json
```

Now we're ready to construct multiple conformer ensembles for the unfolded states of the drkN SH3 domain. To build 100 conformers, sampling only the loop region, the default limits to the backbone and side chain L-J energy potentials are be 100 kJ and 250 kJ respectively, using default fragment sizes, no substitutions, and to have side chains added with FASPR:

```
idpconfgen build \
    -db idpconfgen_database.json \
    -seq drksh3.fasta \
    -nc 100 \
    -of ./drk_L+_nosub_faspr \
    -n
```

`idpconfgen` is deterministic. Therefore, the random seed defines the sampling progression - read *here* for more information.

To switch the side chain building algorithm to MC-SCE (recommended), you would first have to install MC-SCE. Please re-visit the *installation* page to get MC-SCE set up. Here's the following example:

```
idpconfgen build \
    -db idpconfgen_database.json \
    -seq drksh3.fasta \
    -nc 100 \
    -scm mcsce \
    -of ./drk_L+_nosub_mcsce \
    -n
```

Note: Running MC-SCE within IDPConformerGenerator can be memory (RAM) intensive. Consider running with a lower number of CPU threads using the *-n* flag if necessary.

The defaults for `--mcsce-n_trials` is 16 while using the `--mcsce-mode exhaustive`, however we recommend trials larger or equal to 100 for smaller conformer pools. In this exercise, we will be using the default MC-SCE side chain building mode `simple`.

If you're encountering an error with MC-SCE running interally through IDPConformerGenerator, we recommend you to generate backbones first, then pack sidechains after. For example, these would be the commands, required to generate backbones first and then sidechains.:

```
idpconfgen build \
    -db idpconfgen_database.json \
    -seq drksh3.fasta \
    -nc 100 \
    -dsd \
    -of ./drk_L+_nosub_bb \
    -n

# Make the output folder for MC-SCE
mkdir ./drk_L+_nosub_mcsce

# Run MC-SCE in the idpconfgen environment because it's already installed
mcsce ./drk_L+_nosub_bb 100 \
    -w \
    -o ./drk_L+_nosub_mcsce \
    -s \
    -m simple \
    -l drk_L+_nosub_mcsce_log
```

As stated in the `idpconfgen build -h`, sampling using other secondary structure parameters required `--dloop` to be turned off `--dloop-off`. For example, if we'd like to sample only helices and extended strands:

```
idpconfgen build \
    -db idpconfgen_database.json \
    -seq drksh3.fasta \
    -nc 100 \
    -et 'pairs' \
    --dstrand \
    --dhelix \
    --dloop-off \
    -of ./drk_H+E+_nosub \
    -n
```

For sampling loops, helices, and strands, we would specify `--dhelix --dstrand` where `--dloop` is turned on by

default. However, sampling without biasing for secondary structure can be done with `--dany --dloop-off`.

To sample using custom secondary structure sampling (CSSS) a CSSS database (.JSON) file needs to be created specifying the secondary structure probabilities for each residue. This can be done using the `makecsss` module if chemical shift data is not readily available, if you'd like to edit a pre-existing CSSS.JSON, or create a new file. Here's an example for making a custom CSSS.JSON file that samples only helices for residues 15-25 of the unfolded state of the drkN SH3 domain and loops for everything else:

```
idpconfgen makecsss -cp 1-14 L 1.0|15-25 H 1.0|26-59 L 1.0 -o cust_csss_drk.json
```

If chemical shift files are readily available, consider using CheSPI or 2D to generate the CSSS.JSON. 2D predictions have been included in the `example/drksh3_ex_resources` folder as `drk_d2D.txt`. CheSPI `probs8_*` predictions have been included in the `example/drksh3_ex_resources` folder as `probs8_25501_unfolded.txt`.

To convert output from 2D to CSSS, use the `csssconv` subclient with flag `-d2D`:

```
idpconfgen csssconv -d2D drk_d2D.txt -o csss_drk_d2D.json
```

To convert output from CheSPI to CSSS, use the `csssconv` subclient with flag `-p8`:

```
idpconfgen csssconv -p8 probs8_25501_unfolded.txt -o csss_drk_chespi.json
```

The outputted `csss_*.json` files will be used for the `-csss` flag in the `build` subclient. For example, constructing 100 conformers for the unfolded state of the drkN SH3 domain using the 2D predictions and the same settings for energy and MC-SCE as above:

```
idpconfgen build \
    -db idpconfgen_database.json \
    -seq drksh3.fasta \
    -nc 100 \
    -csss csss_drk_d2D.json \
    --dloop-off \
    -et 'pairs' \
    -of ./drk_CSSSd2D_nosub \
    -n
```

The default fragment size probabilities for building are (1, 1, 3, 3, 2) for fragment sizes of (1, 2, 3, 4, 5) respectively. To change this, we would have to create a `.TXT` file with two columns, the first specifying what fragment sizes from lowest to highest, the second specifying their relative probabilities. We have provided an example in `example/drksh3_ex_resources` as `customFragments.txt`. To use these custom fragment size probabilities with CSSS:

```
idpconfgen build \
    -db idpconfgen_database.json \
    -seq drksh3.fasta \
    -nc 100 \
    -xp customFragments.txt \
    -csss csss_drk_d2D.txt \
    --dloop-off \
    -et 'pairs' \
    -of ./drk_fragN_CSSSd2D_nosub \
    -n
```

Finally, to expand torsion angle sampling beyond the residue identity, we can provide a residue tolerance map using the `-urestol` flag in the `build` subclient. For this example, we will be using columns 5, 3, and 2 from the EDSSMat50 substitution matrix:

```
idpconfgen build \
    -db idpconfgen_database.json \
    -seq drksh3.fasta \
    -nc 100 \
    --dany \
    --dloop-off \
    -urestol '{"R":"RK","D":"DE","C":"CY","C":"CW","Q":"QH","E":"ED","H":"HYQ","I":"IVM",
↪"I":"IL","K":"KR","M":"MI","M":"MVL","F":"FY","F":"FWL","W":"WYFC","Y":"YF","Y":"YC","Y
↪":"YWH"}' \
    -et 'pairs' \
    -of ./drk_ANY_sub532 \
    -n
```

Please note for the above run, we are sampling the torsion angle database disregarding secondary structure with the `--dany` flag.

Hopefully this more in-depth realistic example with the unfolded state of the drkN SH3 domain has provided you with the utilities and usage examples to explore IPDConfGen more with your custom protein systems.

1.2.5 Modeling Disordered Region Tails on a Folded Domain

Note: When modeling multi-chain complexes with the `ldrs` subclient, the FASTA file format for the `-seq` parameter must be as follows with no blank spaces.

`>A Sequence for chain A >B Sequence for chain B`

If you would like to skip a chain while modeling multi-chain complexes, you must have the identical sequence in the `.fasta` file to the chains in the template you would like to skip.

Clash-checking and will be done with skipped-chains in consideration.

The following example will walk you through building N-terminal and C-terminal IDR tails on folded regions using the Local Disordered Region Sampling (LDRS `ldrs` subclient).

For this exercise, we will be constructing the tails on the human CNOT7 deadenylase protein. Please enter the example `example/cnot7_example` folder where you will find the complete CNOT7 sequence: `cnot7.fasta`, and a PDB of the folded region from PDB ID 4GMJ: `4GMJ_CNOT7.pdb`.

Note: If your input PDB has phosphorylated residues such as phosphorylated threonine and serine, please change the three-letter code in the PDB file indicating the residue label to the non-modified version. For example: `TPO` phosphorylated threonine will become `THR` and `SEP` phosphorylated serine will become `SER`.

Using the `resre` subclient can help you with this.

Steps from now will assume you're in the `example/cnot7_example` directory and have already created your preferred reusable IDPConformerGenerator *database*. For instructions on the database, please visit the previous exercise "A Real Case Scenario".

To generate the disordered terminal tails on CNOT7 run the following command:

```
idpconfgen ldrs \
    -db <PATH TO DATABASE.JSON> \
    -seq cnot7.fasta \
```

(continues on next page)

(continued from previous page)

```
    -etbb 100 \
    -etss 250 \
    -nc 100 \
    -fld 4GMJ_CNOT7.pdb \
    -of ./cnot7_ldrs_L+_faspr \
    -n
```

The `ldrs` subclient with automatically detect the N-IDR and C-IDR tail based on mismatches in the primary sequence of the `.fasta` file (or input sequence from `-seq`) and the PDB file of the folded domain. This command took approximately 3 minutes on a single workstation with 64 GB DDR4 RAM and 50 CPU threads (`-n 50`) clocked at 3.0 GHz.

To check your outputs against what is to be expected for this tutorial section. Please click here and download the archive named `cnot7_ldrs_example.zip`.

Note: Sidechain clashes may appear if you use the FASPR method for packing on sidechains above. To guarantee no sidechain clashes, we recommended either lowering the steric-clash tolerance using the `-tol` flag above or generating backbone-only conformers first then packing sidechains later with MC-SCE as described below.

To generate backbone-only IDR tails on CNOT7 then pack sidechains on the IDRs with MC-SCE. We will be using agnostic secondary structure sampling here with `--dany`.:

```
idpconfgen ldrs \
    -db <PATH TO DATABASE.JSON> \
    -seq cnot7.fasta \
    --dloop-off \
    --dany \
    -etbb 100 \
    -dsd \
    -nc 1000 \
    -fld 4GMJ_CNOT7.pdb \
    -of ./cnot7_ldrs_ANY_bb \
    -n

idpconfgen resre \
    ./cnot7_ldrs_ANY_bb/ \
    -of ./cnot7_ldrs_ANY_bb_resre \
    -pt 126:HIP,157:HIP,225:HIP,249:HIP,258:HIP, \
    -n

mkdir cnot7_ldrs_ANY_mcsce

mcsce \
    ./cnot7_ldrs_ANY_bb_resre \
    64 \
    -w \
    -o ./cnot7_ldrs_ANY_mcsce \
    -l ./mcsce_log \
    -s \
    -m simple \
    -f 12-262
```

Note: You can access the MC-SCE software here to ignore folded regions and add post-translational modifications during the sidechain packing process.

If you run into an error with `mcsce` and your input PDB has histines labled as `HIS`, please change the three-letter code in the PDB file to `HIP` to account for all protonation states.

Using the `resre` subclient like so above can help you with this.

Any other parameters will only impact the disordered regions generated. Additional settings include `-tol` and `-kt`, where the former sets a tolerance for clashes between the disordered tail(s) and folded domain while the latter acts as a switch to retain the disordered tail(s) generated in the building process. By default, disordered tail only conformers are deleted after full length conformers are generated.

1.2.6 Modeling Disordered Regions Within Folded Domains

The following example will walk you through building an 86-residue-long IDR connecting two folded domains. By now, we expect you to be familiar with the Local Disordered Region Sampling (LDRS `ldrs` subclient). If not, please visit the `cnot7_example`.

For this exercise, we will construct the intrinsically disordered region from residues 568-652 on the STAS domain of SLC26A9 (PDB ID 7CH1, UniProt A0A7K9KBT8). In the `example/slc26a9_example` folder you will find the complete FASTA sequence: `SLC26A9_STAS.fasta`, and a PDB of the folded region from PDB ID 7CH1: `7CH1_SLC26A9.pdb`.

Note: Modeling an IDR between folded regions may take a while depending on various factors such as the length of the IDR to model, the distance between the chain breaks, the location of the chain break, and the presence of folded parts that restrict the growth of the chain.

To continue the tutorial, navigate to the `example/slc26a9_example` directory. Ensure you have already created your preferred reusable IDPConformerGenerator database. For instructions on the database, please visit the previous exercise, "A Real Case Scenario".

We will be using the `ldrs` subclient to model 50 conformations of the intrinsically disordered region. Sidechain clashes may exist if you decide to use the FASPR method for generating sidechains like so below.:

```
idpconfgen ldrs \
    -db <PATH TO DATABASE.JSON> \
    -seq SLC26A9_STAS.fasta \
    -etbb 100 \
    -etss 250 \
    -nc 50 \
    --dloop-off \
    --dany \
    -fld 7CH1_SLC26A9.pdb \
    -of ./slc26a9_ldrs_ANY_faspr \
    -n
```

From the `.fasta` file, the `ldrs` subclient will automatically identify the N-IDR, the C-IDR, and any IDRs missing between folded domains; and construct those. The speed of Linker-IDR generation varies with sequence length as well as its relative position to the folded domain.

To guarantee no sidechain clashes, we recommend either lowering the steric-clash tolerance using the `-tol` flag above or generating backbone-only conformers first then packing sidechains later with MC-SCE as described below in the

Advanced LDRS Usage section.

To check your outputs against what is to be expected for this tutorial section. Please click here and download the archive named `slc26a9_ldrs_example.zip`.

1.2.7 Advanced LDRS Usage

For those more proficient in Python, the modularity of LDRS allows users access to **advanced features** that we describe here. Such features include profiting from better parallelization that allows modeling longer IDRs in a shorter time (e.g., 221 residues in one or two days). Here, we explain how to use the modularity of IDPConformerGenerator to exploit its total capacity for modeling IDRs by writing two new Python scripts that import IDPConfGen machinery.

For template PDB structures, please ensure they have the element name at the second last column in the PDB file. If you're unsure about the formatting, you can use the `Export Molecule` feature in PyMOL. The element name column will be automatically added.

The logic behind the LDRS subclient for modeling and IDR connecting two folded domains assumes that we have an N-IDR-like case at the C-terminal region of the first folded domain and a C-IDR-like case at the N-terminal region of the second domain. Thus, when defining the IDR sequence in the fast file given to the *-seq* parameter, we need to provide two overlapping residues at each. Those will be "QK" and "LA" in this example.

We have already prepared the IDR sequence for this example; see the `SLC26A9_IDR.fasta` file. You can perform sequence alignment between the `SLC26A9_IDR.fasta` file and the `slc26a9_example.fasta` file to visualize where the IDR fits within the whole protein sequence.

Here is a brief overview of what we will do to speed up the process of closing the chain break (L-IDR) with an all-atom IDR model. We show two scripts you can use as templates for using the LDRS features of IDPConformgerGenerator.

1. Generate 10,000 backbone-only structures (more = better sampling) of the IDR:

The idea is to have IDPConformerGenerator generate a large library of IDPs that may represent the IDR to model. We are exploiting IDPConformerGenerator's speed and diversity for generating conformers. The time rate-limiting step here is the `next-seeker` protocol (see scripts), where we have to compare all of the structures in `slc26a9_cterm` to all structures in `slc26a9_nterm` to find our candidates for sidechain addition.

2. Create the necessary folders for the script to run:

3. Change the paths in the script `slc26a9_shortcut.py` and run it; always use the `idpconfgen` Python environment.

4. Use `psurgeon()` in `slc26a9_stitching.py` script to attach the all-atom IDR models to the folded domain. The output for this will be in `slc26a9_results`.

5. Use the `resre` module to rename any `HIS` to `HIP` that exist after stitching:

6. Model the sidechains onto the backbone-only L-IDRs stitched onto the folded region generated previously in the `results` folder using the MC-SCE software:

To further save time, especially on a computing cluster, we can split the conformers in the `nterm` folder and run jobs in parallel or request more workers. Furthermore, the conformers in `slc26a9_results` can be split to run `mcsce` in parallel as well. Please note that this shortcut is not a memory-intensive task, so 8 GB of RAM is sufficient to run the `next-seeker` protocol.

1.2.8 Processing Low-Confidence Predicted Residues

Although the `ldrs` subclient accepts any PDB or mmCIF file to be used as a template. A script has been prepared in this folder `/remove_lowconfidence_residues.py` to automate the removal of low-confidence predicted residues.

Sample thresholds have been given within the Python script that uses the `idpconfgen` environment based on the source of each structure prediction algorithm. For example, a threshold of 70 needs to be applied to AlphaFold structures and 0.7 for ESMFold structures.

Please edit the `input_file`, `output_file` and `threshold` variables in the script before running with `python remove_lowconfidence_residues.py` in the `idpconfgen` Python environment.

Note: It is sometimes preferable to remove residues manually using a molecular viewer such as PyMOL to avoid ending up with short segments of confident, yet unwanted residues.

For example, 2 "confident" residues in between 10 unconfident residues should be removed as well for optimal performance.

1.2.9 Modeling Disordered Regions in a Multi-Chain Protein Complex

The following example will walk you through building intrinsically disordered regions on multiple chains of a protein complex using the Local Disordered Region Sampling (LDRS `ldrs` subclient).

Note: Please ensure all sequences are within the .FASTA file even for chains you are not interested in LDRS processing. This will help LDRS determine which chain to process.

Please refer to `D1D2.fasta` for a formating example.

For this exercise, we will be constructing a combination of the three cases of N-IDR, L-IDR, and C-IDR on both chain A and chain B of the crystal structure of the D1D2 sub-complex from the human SNRNP core domain. Please enter the example `example/complex_example` folder where you will find the complete set of sequences: `D1D2.fasta`, and a PDB of the complex from the RCSB PDB ID 1B34: `1B34.pdb`.

Steps from now will assume you're in the `example/complex_example` directory and have already created your preferred reusable IDPConformerGenerator *database*. For instructions on the database, please visit the previous exercise "A Real Case Scenario".

Due to the automated process of multi-chain detection and building we can generate a set of 10 structures with a single command:

```
idpconfgen ldrs \
    -db <PATH TO DATABASE.JSON> \
    -seq D1D2.fasta \
    -nc 10 \
    -fld 1B34.pdb \
    --dloop-off \
    --dany \
    -of ./D1D2_ldrs_ANY_faspr \
    -n
```

The `ldrs` subclient with automatically detect the all IDRs and their corresponding chains based on sequence similarity and mismatches in the primary sequence of the `.fasta` file and the PDB file of the folded domain. This command took approximately an hour on a single workstation with 64 GB DDR4 RAM and 10 CPU threads (`-n 10`) clocked at 3.0 GHz.

To check your outputs against what is to be expected for this tutorial section. Please click here and download the archive named `d1d2_complex_ldrs_example.zip`.

1.2.10 Exploring IDPConfGen Analysis Functions

Our vision for IDPConformerGenerator as a platform includes the analysis of your database and the PDBs generated by IDPConfGen. To get started, the `stats` subclient is a quick way to check how many hits for different sequence fragment matches you will find in the database for your protein system of choice. It is also possible to include different secondary structure filters as well as amino-acid substitutions to get a more accurate representation the number of hits in the database for your system:

```
idpconfgen stats \
    -db idpconfgen_database.json \
    -seq drksh3.fasta \
    --dloop-off \
    --dany \
    -op drk_any \
    -of ./drk_any_dbStats
```

Another tool to investiagte the database is the `search` subclient. To use this, you will need a tarball or folder of raw PDBs required from the `fetch` subclient. The `search` function goes through the PDB headers to find keywords of your choice and returns the number of hits and their associated PDBIDs in .JSON format:

```
idpconfgen fetch \
    ../cull100 \
    -d ./cull100pdbs/ \
    -u \
    -n

idpconfgen search \
    -fpdb ./cull100pdbs/ \
    -kw 'thermococcus,pro,beta' \
    -n
```

After generating conformer ensembles with IDPConfGen, it is possible to do some basic plotting with the integrated plotting flags in the `torsions` and `sscalc` subclients. For `torsions`, you can choose to plot either omega, phi, or psi dihedral angle distributions in a scatter plot format. For `sscalc`, fractional secondary structure will be plotted in terms of DSSP codes as well as fractions from the alpha, beta, or other regions of the Ramachandran space for your conformers of choice. The following example plots the psi angle distributions and the fractional secondary structure of the `drk_CSSSd2D_nosub_mcsce` ensemble generated in the previous module:

```
idpconfgen torsions \
    ./drk_CSSSd2D_nosub_mcsce \
    -deg \
    -n \
    --plot angtype=psi xlabel=drk_residues
```

To plot the fractional Ramachandran space information:

```
idpconfgen torsions \
    ./drk_CSSSd2D_nosub_mcsce \
    -deg \
    -n \
    --ramaplot filename=fracDrkRama.png colors=['o', 'b', 'k']
```

To plot the fractional secondary structure information:

```
idpconfgen sscalc \
    ./drk_CSSSd2D_nosub_mcsce \
    -u \
    -rd \
    -n \
    --plot filename=dssp_reduced_drk_.png
```

To see which plotting parameters can be modified, please refer to `src/idpconfgen/plotfuncs.py`. We have given a short list of modifyable parameters here:

```
--plot title=<TITLE> title_fs=<TITLE FONT SIZE> xlabel=<X-AXIS LABEL> xlabel_fs=<X-AXIS␣
↪LABEL FONT SIZE> colors=<LIST_OF_COLORS>
```

1.2.11 Exploring MC-SCE and Int2Cart Integrations

Integrating the functions from our collaborators at the Head-Gordon Lab, IDPConformerGenerator has the ability to build with bond geometries derived from a recurrent neural network machine learning model Int2Cart. Furthermore, as we introduced the MC-SCE method for building sidechains in the previous modules, we would like to provide some examples on changing the default sidechain settings.

To use the Int2Cart method for bond geometries, the `--bgeo-strategy` flag needs to be defined with `int2cart` duringthe building stage:

```
idpconfgen build \
    -db idpconfgen_database.json \
    -seq drksh3.fasta \
    -etbb 100 \
    -etss 250 \
    -nc 100 \
    -csss csss_drk_d2D.json \
    --dloop-off \
    -et 'pairs' \
    -scm mcsce \
    --bgeo-strategy int2cart \
    -of ./drk_CSSSd2D_nosub_int2cart_mcsce \
    -n
```

To change the number of trials for MC-SCE to optimize success rate and overall speed:

```
idpconfgen build \
    -db idpconfgen_database.json \
    -seq drksh3.fasta \
    -etbb 100 \
    -etss 250 \
    -nc 100 \
    -csss csss_drk_d2D.json \
    --dloop-off \
    -et 'pairs' \
    -scm mcsce \
    --mcsce-n_trials 64 \
    -of ./drk_CSSSd2D_nosub_32_trials_mcsce \
    -n
```

1.2.12 How to Efficiently Set Jobs up for HPC Clusters

Using the `sethpc` subclient, users can generate bash scripts for SLURM managed systems. Due to architecture of Python's multiprocessing module, IDPConformerGenerator is unable to utilize the resources of multiple nodes on HPC clusters. However, with `sethpc`, users are able to request multiple nodes per job and `sethpc` will automatically generate the SBATCH scripts needed, along with an `all*.sh` and `cancel*.sh` script to run/cancel all of the jobs generated with ease.

Please note that on many HPC resources (such as Graham) your queuing priority will not change requesting 5 nodes per job or 1 node per 5 jobs, but this should be confirmed.

If multiple nodes are requested, at the end of all jobs, the `merge` subclient can be run to merge all of the conformers generated into one folder with the option of modify the naming-pattern for each structure. Please see below for an example of running `sethpc` and `merge`.

To request 3 nodes to generate 512,000 structures of the unfolded state of the drkN SH3 domain with 10 hours per node:

```
idpconfgen sethpc \
    -des ./drk_hpc_jobs/ \
    --account def-username \
    --job-name drk_hpc \
    --nodes 3 \
    --ntasks-per-node 32 \
    --mem 16g \
    --time-per-node 0-10:00:00 \
    --mail-user your@email.com \
    -db idpconfgen_database.json \
    -seq drksh3.fasta \
    -etbb 100 \
    -etss 250 \
    -nc 512000 \
    -csss csss_drk_d2D.json \
    --dloop-off \
    -et 'pairs' \
    -scm mcsce \
    --bgeo-strategy int2cart \
    -of /scratch/user/drk/ \
    -n 32 \
    -rs 12
```

To merge all of the folders created by the multi-node jobs:

```
idpconfgen merge \
    -tgt /scratch/user/drk/ \
    -des /scratch/user/drk/drk_CSSSd2D_nosub_multiple_mcsce \
    -pre drk_confs \
    -del
```

1.2.13 Using IDPConfgen as Python library

To use IDPConformerGenerator in your project, import it as a library:

```
import idpconfgen
```

From within the Python prompt you can get information on each module, class, and function with `help(idpconfgen)`. You can also access the whole API documentation here at *the reference page*.

1.3 F.A.Q.s

1.3.1 What are the recommended hardware specifications?

IDPConformerGenerator was built with computational efficiency in mind. However, having more CPU cores and more RAM would benefit larger protein systems (e.g. for systems greater than 350 residues, having more than 16 cores and 64GB of RAM is suggested). However, additional resources might be required to use certain third-party integrations. For example, Int2Cart, required high performance CUDA compatible GPUs.

IDPConformerGenerator can also run on HPC clusters easily. Please refer to the *installation* instructions to set-up IDPConformerGenerator using `conda` or `virtualenv`.

1.3.2 How long does the database generation take?

Although it is possible to generate the torsion angle and secondary structure database for IDPConformerGenerator on any system, including HPC clusters, we recommend local systems with the largest number of workers and the fastest read-write times. Due to the quantity of PDB files downloaded, many read-write requests will slow down the login node for other users on HPC resources as well as create errors in file I/O processing in IDPConfGen. For reference, generating a database using 24,002 PDB files takes 1 hour 13 minutes (using 31 workers) as a job on the Graham cluster, an HPC resource of the Digital Research Alliance of Canada (DRAC), an HPC resource. Generating this same database locally (using 63 workers of an AMD Threadripper 2990wx) takes only 37 minutes (approximately half the time compared to 31 workers on the Graham cluster (DRAC)) since the bulk of the time is the PDB downloading process. Importantly, this database can then be transferred onto HPC clusters and can be reused for all future calculations, as well as shared between collaborators.

1.3.3 What are the best options for combination of conformers and cores to use?

It is best to have the number of conformers/trials equal to an integer multiple of the number of cores/workers. For example, if you choose to use 32 cores via `-n 32`, it's ideal to have `-nc` as 32, 64, 128, etc. However, this is not mandatory and IDPConfGen will accommodate your *-nc* and *-n* request accordingly.

For hyperthreadded systems, `-n` will use the maximum number of THREADS - 1. For the best experience however, we recommend running on the maximum number of physical cores for hyperthreadded CPUs. For example, a Threadripper 2990wx has 32 cores and 64 threads, it's best to use `-n 32` to avoid overloading the system.

1.3.4 What's the difference between MC-SCE and FASPR?

IDPConfGen incorporates FASPR to natively build sidechains on the conformer's backbone.

A part from the algorithmic differences between FASPR and MC-SCE (please refer to their publications for details), FASPR does not add hydrogens to the sidechains, so other tools are required to add hydrogens if they are desired. MC-SCE does add hydrogens to the sidechains it generates.

Although MC-SCE is slower, it produces structures with no steric clashes, as it uses the same clash quality controls as IDPConfGen. Furthermore, it also has many settings that can be tweaked for flexible in silico experiments as opposed to the default settings of FASPR.

1.3.5 How can I optimize the MC-SCE settings for large proteins?

For larger proteins such as the Tau fragment (441 residues), we recommend generating backbone-only conformers and use MC-SCE as a stand-alone program afterwards.

During the backbone generation stage (i.e. `idpconfgen build ... -dsd`), speed could be improved by setting a higher backbone energy threshold (`-etbb`). We recommend 250 as a minimum.

For the MC-SCE sidechain step, we recommend doing a small benchmark with 128 trials to see what the median number of trials MC-SCE requires for a successful sidechain addition.

1.3.6 What does it mean that IDPConformerGenerator is deterministic?

Reproducibility is one of the most important pillars in scientific research. Thus, we've ensured reproducibility by implementing a `--random-seed` flag while building. Therefore, generating ensembles with the same database file and building parameters on the same processing system will generate the same set of conformers.

The random seed parameter is also helpful for appending to incomplete conformer pools. For example, if your target was 3000 conformers and the requested job-time was not enough and only 2000 were generated, 1000 more unique conformers can be generated simply by using a different integer for the `-rs` flag.

1.3.7 What forcefield does IDPConformerGenerator use by default to generate PDB files?

IDPConfGen uses the Amberff14SB forcefield by default from OpenMM. The forcefield can be changed via the `--forcefield` flag, but no other is currently implemented. For more advanced information see https://github.com/julie-forman-kay-lab/IDPConformerGenerator/blob/master/src/idpconfgen/core/data/README.md.

1.3.8 Out-of-memory (OOM) error when running the `count_clashes` function

If you are scripting the `count_clashes` function or using an exceptionally large template structure (e.g. greater than 450,000 atoms), you could run into a `numpy.core._exceptions._ArrayMemoryError`.

To avoid this, try reducing the number of cores/multiple processes with the `--ncores` flag. Or Decrease the size of your template structure, or request more RAM when submitting a job to a cluster.

1.4 Reference

1.4.1 Components

Residue tolerance during building

Implement residue replacement possibilities.

class idpconfgen.components.residue_tolerance.**EDSS50_indexes**(*option_strings*, *dest*, *nargs=None*, *const=None*, *default=None*, *type=None*, *choices=None*, *required=False*, *help=None*, *metavar=None*)

Convert the number indexes to EDSS50 table.

idpconfgen.components.residue_tolerance.**EDSSMat50_subs = {'A': ['', '', '', 'TV', 'SG'], 'C': ['', 'Y', 'W', 'F', 'SVH'], 'D': ['', '', 'E', '', ''], 'E': ['', '', 'D', '', ''], 'F': ['Y', '', 'WL', 'MIC', 'VH'], 'G': ['', '', '', '', 'A'], 'H': ['', '', 'YQ', 'N', 'FRC'], 'I': ['', 'VM', 'L', 'F', 'T'], 'K': ['', '', 'R', '', 'Q'], 'M': ['', 'I', 'VL', 'F', 'T'], 'N': ['', '', '', 'H', 'STD'], 'P': ['', '', '', '', ''], 'Q': ['', '', 'H', '', 'KR'], 'R': ['', '', 'K', '', 'HQW'], 'S': ['', '', '', '', 'TCNA'], 'T': ['', '', '', 'A', 'VSMIN'], 'W': ['', '', 'YFC', '', 'R'], 'Y': ['F', '', 'C', '', 'WH']}**

EDSSMat50 tolerance matrix.

idpconfgen.components.residue_tolerance.**add_res_tolerance_groups**(*ap*)

Add parameters related to residue tolerance.

idpconfgen.components.residue_tolerance.**make_EDSSMat50_subs**(*idx=(5, 3, 2, 1, 0)*)

Make EDSSMat50 table column indexes.

Sidechain packing methods

Modules to compute sidechains.

This package contains all methods integrated in idpconfgen to compute side chains. It implements a strategy pattern where depending on the sidechain option (str) the building workflow will use one function or another.

For this reason, all functions computing sidechains should output the same types and have a compatible input api.

All functions should return an array mask to be used in *all_atom_coords* and return the coordinate compatible with such a mask.

For that, in general, the `init_` functions receive the template and all_atom masks named tuples (see libbuild).

idpconfgen.components.sidechain_packing.**add_sidechain_method**(*parser*)

Add option to choose from sidechain packing algorithms.

idpconfgen.components.sidechain_packing.**get_sidechain_packing_parameters**(*parameters*, *mode*)

Extract sidechain packaging related parameters.

idpconfgen.components.sidechain_packing.**sidechain_packing_methods = {'faspr': <function init_faspr_sidechains>, 'mcsce': <function init_mcsce_sidechains>}**

Sidechain packing algorithms.

FASPR

Use FASPR to build sidechains.

idpconfgen.components.sidechain_packing.faspr.**init_faspr_sidechains**(*input_seq*, *template_masks*, *all_atom_masks*, ***ignore*)

Instantiate dedicated function environment for FASPR sidechain calculation.

Examples

```
>>> calc_faspr = init_fastpr_sidechains('MASFRTPKKLCVAGG', ...)
>>> # a (N, 3) array with the N,CA,C,O coordinates
>>> coords = np.array( ... )
>>> calc_faspr(coords)
```

Parameters

- **input_seq** (*str*) – The FASTA sequence of the protein for which this function will be used.
- **template_masks** (*libbuil.ConfMasks* object) – Related to the building template in *cli_build*.
- **all_atom_masks** (*libbuil.ConfMasks* object) – Related to the all atoom coordinate system in *cli_build*.

Returns

- *np.ndarray, dtype=bool, (M, 3)* – Array mask for the builder.
- *np.ndarray (M, 3)* – Heavy atom coordinates of the protein sequence.

MC-SCE

Implement MC-SCE sidechain packing algorithm logic.

MC-SCE repository at: https://github.com/THGLab/MCSCE

idpconfgen.components.sidechain_packing.mcsce.**add_mcsce_subparser**(*ap*)

Add MC-SCE related parameters to client.

idpconfgen.components.sidechain_packing.mcsce.**init_mcsce_sidechains**(*input_seq*, *template_masks*, *all_atom_masks*, *user_parameters=None*, ***ignore*)

Instantiate dedicated function environment for MC-SCE sidechain building.

Examples

```
>>> calc_mcsce = init_mcsce_sidechains('MASFRTPKKLCVAGG', ...)
>>> # a (N, 3) array with the N,CA,C,O coordinates
>>> coords = np.array( ... )
>>> calc_mcsce(coords)
```

Parameters

- **input_seq** (*str*) – The FASTA sequence of the protein for which this function will be used.
- **template_masks** (*libbuil.ConfMasks* object) – Related to the building template in *cli_build*.
- **all_atom_masks** (*libbuil.ConfMasks* object) – Related to the all atoom coordinate system in *cli_build*.
- **user_parameters** (*dict*) – Dictionary with additional parameters for the MC-SCE package.

Returns

- *np.ndarray, dtype=bool, (M, 3)* – Array mask for the builder.
- *np.ndarray (M, 3)* – Heavy atom coordinates of the protein sequence.

1.4.2 Core libraries

Lib build

Tools for conformer building operations.

class idpconfgen.libs.libbuild.**ConfLabels**(*atom_labels*, *res_nums*, *res_labels*)

Contain label information for a protein/conformer.

Variables

- **atom_labels** (*np.array*) –
- **res_nums** (*np.array*) –
- **res_labels** (*np.array*) –

atom_labels

Alias for field number 0

res_labels

Alias for field number 2

res_nums

Alias for field number 1

idpconfgen.libs.libbuild.**ConfMasks**

alias of ConfMaks

idpconfgen.libs.libbuild.**are_connected**(*n1*, *n2*, *rn1*, *a1*, *a2*, *bonds_intra*, *bonds_inter*)

Detect if a certain atom pair is bonded accordind to criteria.

Considers only to the self residue and next residue

idpconfgen.libs.libbuild.**build_regex_substitutions**(*s*, *options*, *pre_treatment=<class 'list'>*, *post_treatment=<built-in method join of str object>*)

Build character replacements in regex string.

Example

```
>>> build_regex_substitutions('ASD', {'S': 'SE'})
'A[SE]D'
```

```
>>> build_regex_substitutions('ASDS', {'S': 'SE'})
'A[SE]D[SE]'
```

```
>>> build_regex_substitutions('ASDS', {})
'ASDS'
```

Parameters

- **s** (*regex string*)
- **options** (*dict*) – Dictionary of char to multichar substitutions
- **pre_treatment** (*callable, optional*) – A treatment to apply in *s* before substitution. *pre_treatment* must return a list-like object. Default: list because it expects s to be a string.
- **post_treatment** (*callable, optional*) – A function to apply on the resulting list-like object before returning. Default: ''.join, to return a string.

idpconfgen.libs.libbuild.**create_Coulomb_params_raw**(*atom_labels*, *residue_numbers*, *residue_labels*, *force_field*)

.

idpconfgen.libs.libbuild.**create_LJ_params_raw**(*atom_labels*, *residue_numbers*, *residue_labels*, *force_field*)

Create ACOEFF and BCOEFF parameters.

idpconfgen.libs.libbuild.**create_bonds_apart_mask_for_ij_pairs**(*atom_labels*, *residue_numbers*, *residue_labels*, *bonds_intra*, *bonds_inter*, *base_bool=False*)

Create bool mask array identifying the pairs X bonds apart in ij pairs.

Given *bonds_intra* and *bonds_inter* criteria, idenfities those ij atom pairs in N*(N-1)/2 condition (upper all vs all diagonal) that agree with the described bonds.

Inter residue bonds are only considered for consecutive residues.

Parameters

- **atom_labels** (*iterable, list or np.ndarray*) – The protein atom labels. Ex: ['N', 'CA, 'C', 'O', 'CB', ...]
- **residue_numbers** (*iterable, list or np.ndarray*) – The protein residue numbers per atom in *atom_labels*. Ex: [1, 1, 1, 1, 1, 2, 2, 2, 2, ...]
- **residue_labels** (*iterable, list or np.ndarray*) – The protein residue labels per atom in *atom_labels*. Ex: ['Met', 'Met', 'Met', ...]

See also:

gen_ij_pairs_upper_diagonal(), *gen_atom_pair_connectivity_masks()*

idpconfgen.libs.libbuild.**create_conformer_labels**(*input_seq*, *atom_names_definition*, *transfunc=<function translate_seq_to_3l>*)

Create all atom/residue labels model based on an input sequence.

The labels are those expected for a all atom model PDB file. Hence, residue labels are repeated as needed in order to exist one residue label/number per atom.

Parameters

- **input_seq** (*str*) – The protein input sequence in 1-letter code format.
- **atom_names_definition** (*dict*) – Keys are residue identity and values are list/tuple of strings identifying atoms. Atom names should be sorted by the desired order.
- **transfunc** (*func*) – Function used to translate 1-letter input sequence to 3-letter sequence code.

Returns

tuple (atom labels, residue numbers, residue labels) – Each is a np.ndarray of types: '<U4', int, and '<U3' and shape (N,) where N is the number of atoms. The three arrays have the same length.

idpconfgen.libs.libbuild.**create_sidechains_masks_per_residue**(*residue_numbers*, *atom_labels*, *backbone_atoms*)

Create a map of numeric indexing masks pointing to side chains atoms.

Create separate masks per residue.

Parameters

- **residue_numbers** (*np.ndarray, shape (N,)*) – The atom residue numbers of the protein.
- **atom_labels** (*np.ndarray, shape (N,)*) – The atom labels of the protein.
- **backbone_atoms** (*list or tuple*) – The labels of all possible backbone atoms.

Returns

list of tuples of length 2 – List indexes refer to protein residues, index 0 is residue 1. Per residue, a tuple of length 2 is given. Tuple index 0 are the indexes of that residue sidechain atoms mapped to an array of the *atom_labels* and *residue_numbers* characteristics. The tuple index 1 is an array of length M, where M is the number of sidechain atoms for that residue, defaults to np.nan.

idpconfgen.libs.libbuild.**extract_ff_params_for_seq**(*atom_labels*, *residue_numbers*, *residue_labels*, *force_field*, *param*)

Extract a parameter from forcefield dictionary for a given sequence.

Parameters

- **atom_labels, residue_numbers, residue_labels** – As returned by *:func:create_conformer_labels*.
- **forcefield** (*dict*)
- **param** (*str*) – The param to extract from forcefield dictionary.

See also:

create_conformer_labels()

idpconfgen.libs.libbuild.**gen_3l_residue_labels_per_atom**(*input_seq_3letter*, *atom_labels*)

Generate residue 3-letter labels per atom.

Parameters

- **input_seq_3letter** (*list of 3letter residue codes*) – Most not be a generator.
- **atom_labels** (*list or tuple of atom labels*) – Most not be a generator.

Yields

String of length 3 – The 3-letter residue code per atom.

idpconfgen.libs.libbuild.**gen_atom_pair_connectivity_masks**(*res_names_ij*, *res_num_ij*, *atom_names_ij*, *connectivity_intra*, *connectivity_inter*)

Generate atom pair connectivity indexes.

Given atom information for the ij pairs and connectivity criteria, yields the index of the ij pair if the pair is connected according to the connectivity criteria.

For example, if the ij pair is covalently bonded, or 3 bonds apart, etc.

Parameters

- **res_names_ij**
- **res_num_ij,**
- **atom_names_ij, iterables of the same length and synchronized information.**
- **connectivity_intra,**
- **connectivity_inter, dictionaries mapping atom labels connectivity**
- **Depends**
- **——-**
- **`are_connected`**

idpconfgen.libs.libbuild.**gen_ij_pairs_upper_diagonal**(*data*)

Generate upper diagonal ij pairs in tuples.

The diagonal is not considered.

Yields

tuple of length 2 – IJ pairs in the form of N*(N-1) / 2.

idpconfgen.libs.libbuild.**gen_residue_number_per_atom**(*atom_labels*, *start=1*)

Create a list of residue numbers based on atom labels.

This is a contextualized function, not an abstracted one. Considers *N* to be the first atom of the residue.

Yields

ints – The integer residue number per atom label.

idpconfgen.libs.libbuild.**get_cycle_bond_type**()

Return an infinite interator of the bond types.

Labels returns are synced with bgeo library. See *core.definitions.bgeo_**.

idpconfgen.libs.libbuild.**get_cycle_distances_backbone**()

Return an inifinite iterator of backbone atom distances.

Sampling, in order, distances between atom pairs:

- N - Ca, used for OMEGA
- Ca - C, used for PHI
- C - N(+1), used for PSI

idpconfgen.libs.libbuild.**get_indexes_from_primer_length**(*sequence*, *plen*, *current_residue*)

Get sequence fragment based on position and length.

idpconfgen.libs.libbuild.**init_conflabels**(**args*, ***kwargs*)

Create atom and residue labels from sequence.

Parameters
***args, **kwargs** – Whichever *:func:create_conformer_labels* accepts.

Returns
namedtuple – ConfLabels named tuple populated according to input sequence.

See also:

create_conformer_labels, *ConfLabels*

idpconfgen.libs.libbuild.**init_confmasks**(*atom_labels*)

Create a ConfMask object (namedtuple).

ConfMask is a named tuple which attributes are integer masks for the respective groups.

Parameters
atom_labels (*array-like*) – The atom names of the protein.

Returns
namedtuple – ConfMasks object.

Notes

ConfMask attributes map to the following atom groups:

```
bb3 : N, CA, C
bb4 : N, CA, C, O
NHs : amide protons
Hterm : N-terminal protons
OXT1 : O atom of C-terminal carboxyl group
OXT2 : OXT atom of the C-terminal carboxyl group
cterm : (OXT2, OXT1)
non_Hs : all but hydrogens
non_Hs_non_OXT : all but hydrogens and the only OXT atom
non_NHs_non_OXT : all but NHs and OXT atom
H2_N_CA_CB : these four atoms from the first residue
             if Gly, uses HA3.
non_sidechains : all atoms except sidechains beyond CB
all_sidechain : all sidechain atoms including CB and HA
```

idpconfgen.libs.libbuild.**make_combined_regex**(*regexes*)

Make a combined regex with ORs.

To be used with re.fullmatch.

idpconfgen.libs.libbuild.**make_list_atom_labels**(*input_seq*, *atom_labels_dictionary*)

Make a list of the atom labels for an *input_seq*.

Considers the N-terminal to be protonated H1 to H3, or H1 only for the case of Proline. Adds also 'OXT' terminal label.

Parameters

- **input_seq** (*str*) – 1-letter amino-acid sequence.
- **atom_labels_dictionary** (*dict*) – The ORDERED atom labels per residue.

Returns

list – List of consecutive atom labels for the protein.

idpconfgen.libs.libbuild.**populate_dict_with_database**(*xmers*, *res_tolerance*, *primary*, *secondary*, *combined_dssps*)

Identify sampling positions.

Identifies slices in *primary* and *secondary* where the *combined_dssps* regexes apply. Considers also residue tolerance.

This function is used internally for *prepare_slice_dict* with multiprocessing.

Parameters

- **xmers** (*list*) – The list of all protein fragments we want to search in the *primary* and *secondary* "database" strings. This list should contain only sequence of the same length.
- **combined_dssps** (*string*) – A string regex prepared with all the combined DSSPs that need to be searched: see code in *prepare_slice_dict* and *make_combined_regex*.
- **others** – Other parameters are like described in *prepare_slice_dict*.

Returns

int, dict – The length of the xmers in *xmers* list. The dictionary with the identified slice positions.

idpconfgen.libs.libbuild.**prepare_energy_function**(*atom_labels*, *residue_numbers*, *residue_labels*, *forcefield*, *lj_term=True*, *coulomb_term=False*, *energy_type_ij='pairs'*, ***kwnull*)

Prepare energy function.

Parameters

- **lj_term** (*bool*) – Whether to compute the Lennard-Jones term during building and validation. If false, expect a physically meaningless result.
- **coulomb_term** (*bool*) – Whether to compute the Coulomb term during building and validation. If false, expect a physically meaningless result.
- **energy_type_ij** (*str*) – How to calculate the energy for *ij* pairs. See *libs.libenergyij.post_calc_options*.

idpconfgen.libs.libbuild.**prepare_slice_dict**(*primary*, *input_seq*, *csss=False*, *dssp_regexes=None*, *secondary=None*, *mers_size=(1, 2, 3, 4, 5)*, *res_tolerance=None*, *ncores=1*)

Prepare a dictionary mapping fragments to slices in *primary*.

Protocol:

1) The input sequence is split into all different possible smaller peptides according to the *mers_size* tuple. Let's call these smaller peptides, XMERS.

2) We search in the *primary* string where are all possible sequence matches for each of the XMERS. And, we consider possible residue substitution in the XMERS according to the *res_tolerance* dictionary, if given.

2.1) the found positions are saved in a list of slice objects that populates the final dictionary (see Return section).

2.2) We repeat the process but considering the XMER can be followed by a proline.

2.3) We save in the dictionary only those entries for which we find matches in the *primary*.

3) optional if *csss* is given. Here, we rearrange the output dictionary so it becomes compatible with the build process. The process goes as follows:

3.1) For each slice found in 2) we inspect the *secondary* string if it matches any of the *dssp_regex*. If it matches we consider that slice to the fragment-size, the XMER identify, the DSSP key. This allow us in the build process to specify which SS to sample for specific regions of the conformer.

Parameters

- **primary** (*str*) – A concatenated version of all primary sequences in the database. In the form of "QWERY|IPASDF", etc.
- **input_seq** (*str*) – The 1-letter code amino-acid sequence of the conformer to construct.
- **csss** (*bool*) – Whether to update the output according ot the CSSS probabilities of secondary structures per amino acid residue position. Will only be used when CSSS is activated.
- **dssp_regexes** (*list-like*) – List of all DSSP codes to look for in the sequence. Will only be used when *csss* is True.
- **secondary** (*str*) – A concatenated version of secondary structure codes that correspond to primary. In the form of "LLLL|HHHH", etc. Only needed if *csss* True.
- **mers_size** (*iterable*) – A iterable of integers denoting the size of the fragments to search for. Defaults from 1 to 5.
- **res_tolerance** (*dict*) – A dictionary mapping residue tolerances, for example: {"A": "AIL"}, noting Ala can be replaced by Ile and Leu in the search (this is a dummy example).
- **ncores** (*int*) – The number of processors to use.

Returns

dict –

A dict with the given mapping:

1) First key-level of the dict is the length of the fragments, hence, integers.

2) The second key level are the residue fragments found in the *primary*. A fragment in input_seq but not in *primary* is removed from the dict.

3) only if *csss* is True. Adds a new layer organizing the slice objects with the SS keys.

Lib Calculations and math

Mathemical calculations.

`idpconfgen.libs.libcalc.`**`calc_MSMV`**(*data*)

Calculate Mean, STD, Median, and Variance.

`idpconfgen.libs.libcalc.`**`calc_all_vs_all_dists`**(*coords*)

Calculate the upper half of all vs. all distances.

Reproduces the operations of scipy.spatial.distance.pdist.

Parameters
coords (*np.ndarray, shape (N, 3), dtype=np.float64*)

Returns
*np.ndarray, shape ((N * N - N) // 2,), dytpe=np.float64*

idpconfgen.libs.libcalc.**calc_angle**(*v1*, *v2*, *ARCCOS=<ufunc 'arccos'>*, *CLIP=<function clip>*, *DOT=<function dot>*, *NORM=<function norm>*)

Calculate the angle between two vectors.

idpconfgen.libs.libcalc.**calc_torsion_angles**(*coords*, *ARCTAN2=<ufunc 'arctan2'>*, *CROSS=<function cross>*, *DIAGONAL=<function diagonal>*, *MATMUL=<ufunc 'matmul'>*, *NORM=<function norm>*)

Calculate torsion angles from sequential coordinates.

Uses `NumPy` to compute angles in a vectorized fashion. Sign of the torsion angle is also calculated.

Uses Prof. Azevedo implementation: https://azevedolab.net/resources/dihedral_angle.pdf

Example

Given the sequential coords that represent a dummy molecule of four atoms:

```
>>> xyz = numpy.array([
>>>     [0.06360, -0.79573, 1.21644],
>>>     [-0.47370, -0.10913, 0.77737],
>>>     [-1.75288, -0.51877, 1.33236],
>>>     [-2.29018, 0.16783, 0.89329],
>>>     ])
```

A1—A2
A3—A4

Calculates the torsion angle in A2-A3 that would place A4 in respect to the plane (A1, A2, A3).

Likewise, for a chain of N atoms A1, …, An, calculates the torsion angles in (A2, A3) to (An-2, An-1). (A1, A2) and (An-1, An) do not have torsion angles.

If coords represent a protein backbone consisting of N, CA, and C atoms and starting at the N-terminal, the torsion angles are given by the following slices to the resulting array:

- phi (N-CA), [2::3]
- psi (CA-C), [::3]
- omega (C-N), [1::3]

Parameters
coords (*numpy.ndarray of shape (N>=4, 3)*) – Where *N* is the number of atoms, must be equal or above 4.

Returns
numpy.ndarray of shape (N - 3,) – The torsion angles in radians. If you want to convert those to degrees just apply `np.degrees` to the returned result.

idpconfgen.libs.libcalc.**make_axis_vectors**(*A*, *B*, *C*)

Make axis vectors.

For np.array([x,y,z]) of connected atoms A->B->C.

Example

```
>>> av = np.array([0.000, 0.000, 0.000])
>>> bv = np.array([1.458, 0.000, 0.000])
>>> cv = np.array([2.009, 1.420, 0.000])
>>> make_axis_vectors(av, bv, cv)
>>> (array([ 0.,  0., -1.]),
>>>  array([-1.,  0.,  0.]),
>>>  array([-0., -1., -0.]))
```

Parameters

A, B, C (*np.array of shape (3,).*) – 3D coordinate points in space.

Returns

tuple – Three vectors defining the of ABC plane.

idpconfgen.libs.libcalc.**make_coord**(*theta*, *phi*, *distance*, *parent*, *xaxis*, *yaxis*)

Make a new coordinate in space.

See also:

make_coord_from_angles(), *rotation_to_plane()*.

Parameters

- **theta** (*float*) – The angle in radians between *parent* and *yaxis*.
- **phi** (*float*) – The torsion angles in radians between *parent-xaxis-yaxis* plane and new coordinate.
- **distance** (*float*) – The distance between *parent* and the new coordinate.
- **parent** (*np.array of shape (3,)*) – The coordinate in space of the parent atom (point). The parent atom is the one that preceeds the newly added coordinate.
- **xaxis** (*np.array of shape (3,)*) – The coordinate in space that defines the xaxis of the parent-xaxis-yaxis coordinates plane.
- **yaxis** (*np.array of shape (3,)*) – The coordinate in space that defines the yaxis of the parent-xaxis-yaxis coordinates plane.

Returns

np.array of shape (3,) – The new coordinate in space.

idpconfgen.libs.libcalc.**make_coord_from_angles**(*theta*, *phi*, *distance*)

Make axis components from angles.

Performs:

np.array([
distance * math.cos(phi), distance * math.sin(phi) * math.cos(theta), distance * math.sin(phi) * math.sin(theta),])

Returns
np.array of shape (3,)

idpconfgen.libs.libcalc.**make_seq_probabilities**(*seq*, *reverse=False*)

Make probabilites from a sequence of numbers.

Sum of probabilities is 1.

idpconfgen.libs.libcalc.**multiply_upper_diagonal_raw**(*data*, *result*)

Calculate the upper diagonal multiplication with for loops.

The use of for-loop based calculation avoids the creation of very large arrays using numpy outer derivatives. This function is thought to be njit compiled.

Does not create new data structure. It requires the output structure to be provided. Hence, modifies in place. This was decided so because this function is thought to be jit compiled and errors with the creation of very large arrays were rising. By passing the output array as a function argument, errors related to memory freeing are avoided.

Parameters

- **data** (*an interable of Numbers, of length N*)
- **result** (*a mutable sequence, either list of np.ndarray,*) – of length N*(N-1)//2

idpconfgen.libs.libcalc.**rotate_coordinates_Q**(*coords*, *rot_vec*, *angle_rad*, *ARRAY=<built-in function array>*, *HMQ=<Mock name='mock.njit()' id='139777571626128'>*, *VSTACK=<function vstack>*)

Rotate coordinates by radians along an axis.

Rotates using quaternion operations.

Parameters

- **coords** (*nd.array (N, 3), dtype=np.float64*) – The coordinates to rotate.
- **rot_vec** (*(,3)*) – A 3D space vector around which to rotate coords. Rotation vector **must** be a unitary vector.
- **angle_rad** (*float*) – The angle in radians to rotate the coords.

Returns
nd.array shape (N, 3), dtype=np.float64 – The rotated coordinates

idpconfgen.libs.libcalc.**rotation_to_plane**(*A*, *B*, *C*)

Define rotations matricex for planar orientation.

Where, A->B determine the X axis, Y is the normal vector to the ABC
plane, and A is defined at 0,0,0.

See also:

make_axis_vectors()

Parameters
A, B, C (*np.array of shape (3,)*) – The 3D coordinates, where A is the parent coordinate, B is the Xaxis and C the Yaxis.

Returns
np.array of shape (3,3) – Rotational matrix

idpconfgen.libs.libcalc.**round_radian_to_degree_bin_10**(*x0*)

Round RADIAN to the nearest 10 degree bin.

23 degrees round to 20 degrees. 27 degrees round to 30 degrees.

Parameters
x0 (*float*) – The angle in radians.

Returns
int – The nearest bind of 10 degrees according to rounding rules.

idpconfgen.libs.libcalc.**sum_upper_diagonal_raw**(*data*, *result*)

Calculate outer sum for upper diagonal with for loops.

The use of for-loop based calculation avoids the creation of very large arrays using numpy outer derivates. This function is thought to be jut compiled.

Does not create new data structure. It requires the output structure to be provided. Hence, modifies in place. This was decided so because this function is thought to be jit compiled and errors with the creation of very large arrays were rising. By passing the output array as a function argument, errors related to memory freeing are avoided.

Parameters

- **data** (*an interable of Numbers, of length N*)
- **result** (*a mutable sequence, either list of np.ndarray,*) – of length N*(N-1)//2

idpconfgen.libs.libcalc.**unit_vector**(*vector*)

Calculate the unitary vector.

Lib Check

Gather check functions and decorators.

idpconfgen.libs.libcheck.**argstype**(**types*)

Decorate a function to check for args types.

@argstype(type1, type2, (type3, type4))

For each function argument provide a type or a tuple of types.

idpconfgen.libs.libcheck.**kwargstype**(**types*)

Decorate a function to check for kwargs types.

@kwargstype(type1, type2, (type3, type4))

For each function named argument provide a type or a tuple of types.

Lib mmCIF

Handle CIF data.

class idpconfgen.libs.libcif.**CIFParser**(*datastr*, *label_priority='auth'*)

mmCIFParser for structural data ONLY.

That is, fields from the m_site.` group.

Parameters
datastr (*str*) – The content of the mmCIF file in string format.

property altloc

The altloc field of the current *line*.

property atname

The atom name field of the current *line*.

property chainid

The chainID field of the current *line*.

property charge

The charge field of the current *line*.

property element

The element field of the current *line*.

get_line_elements_for_PDB(*line=None*)

Retrieve *line* elements.

Parameters

line (*int, optional*) – If given retrieve values for that line and sets that as current line. Else, retrieves values for current *line*.

Returns

list – The values of the line.

get_value(*label*)

Retrieve the label value for the current *line*.

property icode

The icode field of the current *line*.

property line

The current line number.

property occ

The occupancy field of the current *line*.

read_cif(*datastr*)

Read 'atom_site.' entries to dictionary.

property record

The record field of the current *line*.

property resname

The resname field of the current *line*.

property resseq

The resSeq field of the current *line*.

property serial

The serial field of the current *line*.

property tempfactor

The tempfactor field of the current *line*.

property xcoord

The Xcoord field of the current *line*.

property ycoord

The Ycoord field of the current *line*.

property zcoord

The Zcoord field of the current *line*.

idpconfgen.libs.libcif.**find_cif_atom_site_headers**(*lines*, *cif_dict*)

Find *_atom_site.* headers.

Given a list of mmCIF lines, finds headers of *_atom_site.*

Parameters

- **lines** (*list*) – The lines of the mmCIF files are read by *file.readlines()*.
- **cif_dict** (*dict*) – A dictionary where to initiate the *atom_site.* keys. Keys are assigned empty list as value.

Returns

int – The index of `lines` where the *_atom_site.* structural information starts.

Raises

CIFFileError – If any *_atom_site.* keys are found in `lines`.

See also:

populate_cif_dictionary

idpconfgen.libs.libcif.**is_cif**(*datastr*)

Detect if *datastr* is a CIF file.

idpconfgen.libs.libcif.**parse_cif_line**(*line*)

Parse CIF line according to *CIF_LINE_REGEX*.

Parameters

line (*str*) – The line to parse.

Returns

list – A list with the parsed line elements.

idpconfgen.libs.libcif.**populate_cif_dictionary**(*lines*, *start_index*, *cif_dict*)

Populate mmCIF dictionary.

Expects atom coordinate information to be consecutive, otherwise reading will be incomplete or errors will occur.

Parameters

- **lines** (*list*) – The mmCIF lines. Normally the whole mmCIF file content.
- **start_index** (*int*) – The line index, that is, line number 0-indexed, where the actual *atom_site.* structural information starts, that is, the first ATOM/HETATM line.
- **cif_dict** (*dict*) – The mmCIF dictionary to populate. The dict keys should be previously prepared and should match the data in the *lines*.

Returns

None – Edits dictionary in place.

Raises

InvalidCIFLineError – If the number of fields in parsed line do not match expected *_atom_site.* fields in the dictionary.

See also:

find_cif_atom_site_headers

Lib CLI

Operations shared by client interfaces.

class idpconfgen.libs.libcli.**AllParam**(*option_strings*, *dest*, *nargs=None*, *const=None*, *default=None*, *type=None*, *choices=None*, *required=False*, *help=None*, *metavar=None*)

Convert list of arguments in tuple.

class idpconfgen.libs.libcli.**ArgsToTuple**(*option_strings*, *dest*, *nargs=None*, *const=None*, *default=None*, *type=None*, *choices=None*, *required=False*, *help=None*, *metavar=None*)

Convert list of arguments in tuple.

class idpconfgen.libs.libcli.**CSV2Tuple**(*option_strings*, *dest*, *nargs=None*, *const=None*, *default=None*, *type=None*, *choices=None*, *required=False*, *help=None*, *metavar=None*)

Convert list of arguments in tuple.

idpconfgen.libs.libcli.**CheckExt**(*extensions*)

Check extension for arguments in argument parser.

class idpconfgen.libs.libcli.**CustomParser**(*prog=None*, *usage=None*, *description=None*, *epilog=None*, *parents=[]*, *formatter_class=<class 'argparse.HelpFormatter'>*, *prefix_chars='-'*, *fromfile_prefix_chars=None*, *argument_default=None*, *conflict_handler='error'*, *add_help=True*, *allow_abbrev=True*, *exit_on_error=True*)

Custom Parser class.

error(*message*)

Present error message.

class idpconfgen.libs.libcli.**FolderOrTar**(*option_strings*, *dest*, *nargs=None*, *const=None*, *default=None*, *type=None*, *choices=None*, *required=False*, *help=None*, *metavar=None*)

Controls if input is folder, files or tar.

class idpconfgen.libs.libcli.**ListOfIntsPositiveSum**(*option_strings*, *dest*, *nargs=None*, *const=None*, *default=None*, *type=None*, *choices=None*, *required=False*, *help=None*, *metavar=None*)

Convert list of str to list of ints.

class idpconfgen.libs.libcli.**ListOfPositiveInts**(*option_strings*, *dest*, *nargs=None*, *const=None*, *default=None*, *type=None*, *choices=None*, *required=False*, *help=None*, *metavar=None*)

Create list of positive integers from input string.

Raises error if non-positive integers are given.

class idpconfgen.libs.libcli.**ParamsToDict**(*option_strings*, *dest*, *nargs=None*, *const=None*, *default=None*, *type=None*, *choices=None*, *required=False*, *help=None*, *metavar=None*)

Convert command-line parameters in an argument to a dictionary.

Adapted from https://github.com/joaomcteixeira/taurenmd

Example

Where `-x` is an optional argument of the command-line client interface.

```
>>> par1=1 par2='my name' par3=[1,2,3]
>>> {'par1': 1, 'par2': 'my name', 'par3': [1, 2, 3]}
```

class idpconfgen.libs.libcli.**ReadDictionary**(*option_strings*, *dest*, *nargs=None*, *const=None*, *default=None*, *type=None*, *choices=None*, *required=False*, *help=None*, *metavar=None*)

Read to a dictionary.

class idpconfgen.libs.libcli.**SeqOrFasta**(*option_strings*, *dest*, *nargs=None*, *const=None*, *default=None*, *type=None*, *choices=None*, *required=False*, *help=None*, *metavar=None*)

Read sequence of FASTA file.

idpconfgen.libs.libcli.**add_argument_chunks**(*parser*)

Add fragments argument.

For those routines that are split into operative fragments.

idpconfgen.libs.libcli.**add_argument_cif**(*parser*)

Add command to prioritize downloading CIF parser.

idpconfgen.libs.libcli.**add_argument_cmd**(*parser*)

Add the command for the external executable.

idpconfgen.libs.libcli.**add_argument_dany**(*parser*)

Add argument *–dany*.

idpconfgen.libs.libcli.**add_argument_db**(*parser*)

Add argument to store path from database.

idpconfgen.libs.libcli.**add_argument_decimals**(*parser*)

Add decimals to parser.

idpconfgen.libs.libcli.**add_argument_degrees**(*parser*)

Add *degree* argument to parser.

idpconfgen.libs.libcli.**add_argument_destination_folder**(*parser*)

Add destination folder argument.

Accepts also a TAR file path.

Parameters

parser (*argparse.ArgumentParser* object)

idpconfgen.libs.libcli.**add_argument_dhelix**(*parser*)

Add argument *–dhelix*.

idpconfgen.libs.libcli.**add_argument_dloopoff**(*parser*)

Add argument *–dloop-off*.

idpconfgen.libs.libcli.**add_argument_dstrand**(*parser*)

Add argument *–dstrand*.

idpconfgen.libs.libcli.**add_argument_duser**(*parser*)

Add argument *–duser*.

idpconfgen.libs.libcli.**add_argument_idb**(*parser*)

Add argument for input database.

idpconfgen.libs.libcli.**add_argument_minimum**(*parser*)

Add argument for minimum size of fragments.

idpconfgen.libs.libcli.**add_argument_ncores**(*parser*)

Add argument for number of cores to use.

idpconfgen.libs.libcli.**add_argument_nohnterm**(*parser*)

Add boolean flag to H in N-terminal.

idpconfgen.libs.libcli.**add_argument_output**(*parser*)

Add argument for general output string.

idpconfgen.libs.libcli.**add_argument_output_folder**(*parser*)

Add argument for general output string.

idpconfgen.libs.libcli.**add_argument_pdb_files**(*parser*)

Add PDBs Files entry to argument parser.

Parameters

parser (*argparse.ArgumentParser* object)

idpconfgen.libs.libcli.**add_argument_pdbids**(*parser*)

Add arguments for PDBIDs.

This differs from `add_parser_pdbs()` that expects paths to files.

idpconfgen.libs.libcli.**add_argument_plot**(*parser*)

Add argument for plotting parameters.

Plot kwargs that will be passed to the plotting function. If given, plot results. Additional arguments can be given to specify the plot parameters.

Adapted from: https://github.com/joaomcteixeira/taurenmd/blob/6bf4cf5f01df206e9663bd2552343fe397ae8b8f/src/taurenmd/libs/libcli.py#L539-L570

Defined by `--plot`.

idpconfgen.libs.libcli.**add_argument_random_seed**(*parser*)

Add argument to select a random seed number.

idpconfgen.libs.libcli.**add_argument_record**(*parser*)

Add argument to select PDB RECORD identifier.

idpconfgen.libs.libcli.**add_argument_reduced**(*parser*)

Add *reduced* argument.

idpconfgen.libs.libcli.**add_argument_replace**(*parser*)

Add argument *replace*.

idpconfgen.libs.libcli.**add_argument_seq**(*parser*)

Add argument for input sequence.

idpconfgen.libs.libcli.**add_argument_source**(*parser*)

Add *source* argument to parser.

idpconfgen.libs.libcli.**add_argument_update**(*parser*)

Add update argument to argument parser.

idpconfgen.libs.libcli.**add_argument_vdWb**(*parser*)

Add argument for vdW bonds apart criteria.

Further read:

https://www.cgl.ucsf.edu/chimera/docs/ContributedSoftware/findclash/findclash.html

idpconfgen.libs.libcli.**add_argument_vdWr**(*parser*)

Add argument for vdW radii set selection.

idpconfgen.libs.libcli.**add_argument_vdWt**(*parser*)

Add argument for vdW tolerance.

idpconfgen.libs.libcli.**add_general_arg**(*parser*, **args*, ***kwargs*)

Add a general argument with args and kwargs.

idpconfgen.libs.libcli.**add_subparser**(*parser*, *module*)

Add a subcommand to a parser.

Parameters

- **parser** (argparse.add_suparsers object) – The parser to add the subcommand to.
- **module** – A python module containing the characteristics of a taurenmd client interface. Client interface modules require the following attributes: `__doc__` which feeds the description argument of add_parser, `_help` which feeds help, `ap` which is an ArgumentParser, and a `main` function, which executes the main logic of the interface.

idpconfgen.libs.libcli.**add_version**(*parser*)

Add version `-v` option to parser.

Displays a message informing the current version. Also accessible via `--version`.

Parameters

parser (argparse.ArgumentParser) – The argument parser to add the version argument.

idpconfgen.libs.libcli.**load_args**(*ap*)

Load argparse commands.

idpconfgen.libs.libcli.**maincli**(*ap*, *main*)

Command-line interface entry point.

idpconfgen.libs.libcli.**minimum_value**(*minimum*)

Define a minimum value for action.

idpconfgen.libs.libcli.**parse_doc_params**(*docstring*)

Parse client docstrings.

Separates PROG, DESCRIPTION and USAGE from client main docstring.

Parameters

docstring (*str*) – The module docstring.

Returns

tuple – (prog, description, usage)

Lib Download

Functions and variables to download files and data.

idpconfgen.libs.libdownload.**download_structure**(*pdbid*, *mmcif=False*, ***kwargs*)

Download a PDB/CIF structure chains.

Parameters

- **pdbid** (*tuple of 2-elements*) – 0-indexed, the structure ID at RCSB.org; 1-indexed, a list of the chains to download.
- ****kwargs** (as for `save_structure_by_chains()`.)

idpconfgen.libs.libdownload.**fetch_pdb_id_from_RCSB**(*pdbid*, *mmcif=False*)

Fetch PDBID from RCSB.

Returns

tuple (str, str) – urllib.request.urlopen.response.read() PDB file extension (.pdb, .cif, ...)

idpconfgen.libs.libdownload.**fetch_raw_CIFs**(*pdbid*, ***, *ext='cif'*, ***kwargs*)

Download raw structure from RCSB without any filtering.

idpconfgen.libs.libdownload.**fetch_raw_PDBs**(*pdbid*, ***, *ext='pdb'*, ***kwargs*)

Download raw structure from RCSB without any filtering.

idpconfgen.libs.libdownload.**fetch_raw_structure**(*pdbid*, *ext*, ***kwargs*)

Download raw structure from RCSB without any filtering.

Lib Energy

Functions to calculate energy from i-j pairs evaluation.

Contains only energy terms that evaluate energy for each atom-atom pair separately.

idpconfgen.libs.libenergyij.**energycalculator_ij**(*distf*, *efuncs*)

Calculate the sum of energy terms.

This function works as a closure.

Accepts only energy terms that compute for non-redundant ij-pairs.

Energy terms must have distances ij as unique positional parameter, and should return an integer.

Example

```
>>> ecalc = energycalculator_ij(calc_ij_pair_distances, [...])
>>> total_energy = ecalc(coords)
```

Where *[...]* is a list containing energy term functions.

See also:

init_lennard_jones_calculator, *init_coulomb_calculator*

Parameters

- **distf** (*func*) – The function that will be used to calculate ij-pair distances on each call. If performance is a must, this function should be fast. *distf* function should receive *coords* as unique argument where *coords* is a np.ndarray of shape (N, 3), where N is the number of atoms, and 3 represents the XYZ coordinates. This function should return a np.ndarray of shape (N * (N - 1)) / 2,), dtype=np.float.
- **efuncs** (*list*) – A list containing the energy terms functions. Energy term functions are prepared closures that accept the output of *distf* function.

Returns

func – A function that accepts coords in the form of (N, 3). The coordinates sent to the resulting function MUST be aligned with the labels used to prepare the *efuncs* closures.

idpconfgen.libs.libenergyij.**init_coulomb_calculator**(*charges_ij*, *postf='pairs'*)

Calculate Coulomb portential.

Parameters

- **charges_ij** (*np.ndarray, shape (N, 3), dtype=np.float*) – The *charges_ij* prepared already for the ij-pairs upon which the resulting function is expected to operate. IMPORTANT: it is up to the user to define the charge such that resulting energy is np.nan for non-relevant ij-pairs, for example, covalently bonded pairs, or pairs 2 bonds apart.
- **postf** (*str*) – There are different implementations of the energy calculation. Current options are: "whole": applies np.nansum after calculating the energy per pair. "pairs": returns the energies per pair.

Returns

numba.njitted func – Function closure with registered *charges_ij* that expects an np.ndarray of distances with same shape as *acoeff* and *bcoeff*: (N,). The *func* return value depends on the *postf* options.

idpconfgen.libs.libenergyij.**init_lennard_jones_calculator**(*acoeff*, *bcoeff*, *postf='pairs'*)

Calculate Lennard-Jones full pontential.

The LJ potential is calculated fully and no approximations to proximity of infinite distance are considered.

Parameters

- **acoeff, bcoeff** (*np.ndarray, shape (N, 3), dtype=np.float*) – The LJ coefficients prepared already for the ij-pairs upon which the resulting function is expected to operate. IMPORTANT: it is up to the user to define the coefficients such that resulting energy is np.nan for non-relevant ij-pairs, for example, covalently bonded pairs, or pairs 2 bonds apart.
- **postf** (*str*) – There are different implementations of the energy calculation. Current options are: "whole": applies np.nansum after calculating the energy per pair. "pairs": returns the energies per pair.

Returns

numba.njitted func – Function closure with registered *acoeff`s and `bcoeff`s that expects an np.ndarray of distances with same shape as `acoeff* and *bcoeff*: (N,). The *func* return value depends on the *postf* options.

idpconfgen.libs.libenergyij.**post_calc_options = ['pairs', 'whole']**

Option keys to apply after calculating energy value for each pair.

Lib Filter

Contain functions to filter information from the DB.

idpconfgen.libs.libfilter.**aligndb**(*db*, *exact=False*)

Aligns IDPConfGen DB.

idpconfgen.libs.libfilter.**make_any_overlap_regex**(*range_*)

Make an overlap regex.

idpconfgen.libs.libfilter.**make_helix_overlap_regex**(*range_*)

Make an overlap regex.

idpconfgen.libs.libfilter.**make_loop_overlap_regex**(*range_*)

Make an overlap regex.

idpconfgen.libs.libfilter.**make_overlap_regex**(*s*, *range_*)

Make an overlap regex.

idpconfgen.libs.libfilter.**make_ranges**(*regex_string*, *rang_rex=re.compile('(\\{\\d+\\,\\d+\\}|\\{\\d+\\}|\\{\\d+\\,\\}|\\{\\,\\d+\\})')*, *char_rex=re.compile('((\\w)\\{|\\[([\\w]+)\\]\\{)')*, *max_range=30*)

Define a set of ranges and characters from *regex_string*.

Examples

```
>>> make_range('L{1}H{1,2}')
[range(1, 2), range(1, 3)], ['L', 'H']
```

```
>>> make_range('L{1,}H{,2}')
[range(1, None), range(1, 3)], ['L', 'H']
```

idpconfgen.libs.libfilter.**make_regex_combinations**(*ranges*, *chars*, *pre=None*, *suf=None*)

Make combinations of regexes from *ranges* and *chars*.

This function is not a general abstraction. Is instead a partial abstraction within the problem of IDPConfGen.

Parameters

- **ranges** (*list of range objects*) – Ranges where to start and stop searching in the regex. Ranges should follow regex conventions, usually 1-indexed and stop inclusive.
- **chars** (*list of 1 letter chars*) – The chars that will be searched in ranges.
- **pre, suf** (*str*) – Strings to add as prefix and suffix of the generates ranges.

idpconfgen.libs.libfilter.**make_regex_combinations_from_ranges**(*regex_string*, ***kwargs*)

.

idpconfgen.libs.libfilter.**make_strand_overlap_regex**(*range_*)

Make an overlap regex.

idpconfgen.libs.libfilter.**regex_forward_no_overlap**(*sequence*, *regex*)

Search for regex forward without overlap.

Examples

r'L' r'L{2}' r'L{1,3}'

In the first example, returns all indexes of single char L without overlap.

On the second example, returns all indexes of entire 'LL' sequences without overlap. So, 'LLL' returns only slice(0, 2).

3) **returns all sequences with 'LLL' without overlap, if a terminal 'LL'** is found, returns that. Same for 'L' if found at the end.

Using expressions such as r'(?=(L))' give not correct results. Use *regex_forward_with_overlap* instead.

idpconfgen.libs.libfilter.**regex_forward_with_overlap**(*sequence*, *regex*)

Find matches for regex in sequence of chars.

Considers regex defines overlap.

Accepts only regex expressions with overlap, for example:

r'(?=(L{3}))'

Returns
list of slices – Where slice.start and slice.stop follow Python conventions.

idpconfgen.libs.libfilter.**regex_has_overlap**(*regex_string*, *overlap_rex=re.compile('\\(\\?\\=\\(.+\\)')*)

Find if a *regex_string* defines overlap.

Parameters

- **regex_string** (*str*) – The regex string.
- **overlap_fmt** (*str*) – The regex to find overlap in regex_string.

Returns
bool

idpconfgen.libs.libfilter.**regex_range**(*sequence*, *regex_string*, *ncores=1*)

Find slices of sequence where regex_string applies.

Searches forward and backwards by means of exploring ranges as individual regexes.

idpconfgen.libs.libfilter.**regex_search**(*sequence*, *regex_string*, *rex_range=re.compile('(\\{\\d+\\,\\d+\\}|\\{\\d+\\}|\\{\\d+\\,\\}|\\{\\,\\d+\\})')*, ***kwargs*)

Search for regex in sequence.

Parameters

- **sequence** (*str*) – The sequence where to apply the regex.
- **regex_string** (*str*) – The regex to search for.
- **regex_range** (*compiled regex*) – A regex to apply in *regex_string* to identify if *regex_string* refers to a range.

Returns
list of slices

Lib Functional

Contain functions to enhance functional programming in Python.

idpconfgen.libs.libfunc.**ITE**(*iflogic*, *assertion*, *elselogic*)

Apply ternary operator logic executing functions.

Functions should be preconfigured and accept no arguments.

Better if you see the code:

return iflogic() if assertion() else elselogic()

idpconfgen.libs.libfunc.**ITEX**(*x*, *iflogic*, *assertion*, *elselogic*)

Apply ternary operator logic executing functions.

Functions should receive a single value: *x*.

Better if you see the code:

return iflogic(x) if assertion(x) else elselogic(x)

Parameters

x – The value to pass to each function.

idpconfgen.libs.libfunc.**chainf**(*init*, **funcs*)

Run functions in sequence starting from an initial value.

Example

```
>>> chainf(2, [str, int, float])
2.0
```

idpconfgen.libs.libfunc.**chainfs**(**funcs*)

Store functions be executed on a value.

Example

```
>>> do = chainfs(str, int, float)
>>> do(2)
2.0
```

idpconfgen.libs.libfunc.**consume**(*gen*)

Consume generator in a single statement.

Example

```
>>> consume(generator)
```

idpconfgen.libs.libfunc.**context_engine**(*func*, *exceptions*, *doerror*, *doelse*, *dofinally*, **args*, ***kwargs*)

Make a context engine.

idpconfgen.libs.libfunc.**f1f2**(*f1*, *f2*, **a*, ***k*)

Apply one function after the other.

Call *f1* on the return value of *f2*.

Args and kwargs apply to *f2*.

Example

```
>>> f1f2(str, int, 2)
"2"
```

idpconfgen.libs.libfunc.**f2f1**(*f1*, *f2*, **a*, ***k*)

Apply the second function after the first.

Call *f2* on the return value of *f1*.

Args and kwargs apply to *f1*.

Example

```
>>> f2f1(str, int, 2)
2
```

idpconfgen.libs.libfunc.**flatlist**(*list_*)

Flat a list recursively.

This is a generator.

idpconfgen.libs.libfunc.**give**(*value*)

Preare a function to return a value when called.

Ignore **args* and ***kwargs*.

Example

```
>>> true = give(True)
>>> true()
True
```

```
>>> five = give(5)
>>> five(4, 6, 7, 8, some_args='some string')
5
```

idpconfgen.libs.libfunc.**if_elif_else**(*value*, *condition_function_pair*)

Apply logic if condition is True.

Parameters

- **value** (*anything*) – The initial value
- **condition_function_pair** (*tuple*) – First element is the assertion function, second element is the logic function to execute if assertion is true.

Returns
The result of the first function for which assertion is true.

idpconfgen.libs.libfunc.**ite**(*iflogic*, *assertion*, *elselogic*)

Apply ternary operator logic executing functions.

Functions should be preconfigured and accept no arguments.

Better if you see the code:

return iflogic() if assertion() else elselogic()

idpconfgen.libs.libfunc.**itev**(*x*, *iflogic*, *assertion*, *elselogic*)

Apply ternary operator logic executing functions.

Functions should receive a single value: *x*.

Better if you see the code:

return iflogic(x) if assertion(x) else elselogic(x)

Parameters
x – The value to pass to each function.

idpconfgen.libs.libfunc.**make_iterable**(*value*)

Transform into an iterable.

Transforms a given *value* into an iterable if it is not. Else, return the value itself.

Example

```
>>> make_iterable(1)
[1]
```

```
>>> make_iterable([1])
[1]
```

idpconfgen.libs.libfunc.**mapc**(*f*, **iterables*)

Consume map function.

Like *map()* but it is not a generator; *map* is consumed immediately.

idpconfgen.libs.libfunc.**pass_**(*value*)

Do nothing, just pass the value.

Example

```
>>> pass_(1)
1
```

idpconfgen.libs.libfunc.**raise_**(*exception*, **ignore*, ***everything*)

Raise exception.

idpconfgen.libs.libfunc.**reduce_helper**(*value*, *f*, **a*, ***k*)

Help in *reduce*.

Helper function when applying *reduce* to a list of functions.

Parameters

- **value** (*anything*)
- **f** (*callable*) – The function to call. This function receives *value* as first positional argument.
- ***a, ****k** – Args and kwargs passed to *f*.

idpconfgen.libs.libfunc.**ternary_operator**(*iflogic*, *assertion*, *elselogic*)

Apply ternary operator logic executing functions.

Functions should be preconfigured and accept no arguments.

Better if you see the code:

return iflogic() if assertion() else elselogic()

idpconfgen.libs.libfunc.**ternary_operator_v**(*x*, *iflogic*, *assertion*, *elselogic*)

Apply ternary operator logic executing functions.

Functions should receive a single value: *x*.

Better if you see the code:

return iflogic(x) if assertion(x) else elselogic(x)

Parameters

x – The value to pass to each function.

idpconfgen.libs.libfunc.**ternary_operator_x**(*x*, *iflogic*, *assertion*, *elselogic*)

Apply ternary operator logic executing functions.

Functions should receive a single value: *x*.

Better if you see the code:

return iflogic(x) if assertion(x) else elselogic(x)

Parameters

x – The value to pass to each function.

idpconfgen.libs.libfunc.**vartial**(*func*, **args*, ***keywords*)

Prepare a function with args and kwargs except for the first arg.

Functions like *functools.partial* except that the resulting preprepared function expects the first positional argument.

Example

```
>>> pow2 = vartial(math.pow, 2)
>>> pow2(3)
9
>>> pow2(4)
16
```

This is different from: >>> pow_base_3 = partial(math.pow, 3) >>> pow_base_3(2) 9 >>> pow_base_3(4) 81

idpconfgen.libs.libfunc.**whileloop**(*cond*, *func*, *do_stopiteration=<function give.<locals>.newfunc>*, *do_exhaust=<function give.<locals>.newfunc>*)

Execute while loop.

All function accept no arguments. If state needs to be evaluated, *cond* and *func* need to be synchronized.

Parameters

- **cond** (*callable*) – The *while* loop condition.
- **func** (*callable*) – The function to call on each while loop iteration.
- **do_stopiteration** (*callable*) – The function to execute when *func* raises StopIteration error.
- **do_exhaust** (*callable*) – The function to execute when while loop exhausts.

Returns
None

Lib higher level

Higher level functions.

Function which operate with several libraries and are defined here to avoid circular imports.

`idpconfgen.libs.libhigherlevel.`**`bgeo_reduce`**(*bgeo*)

Reduce BGEO DB to trimer and residue scopes.

`idpconfgen.libs.libhigherlevel.`**`cli_helper_calc_bgeo`**(*fname*, *fdata*, ***kwargs*)

Help *cli_bgeodb* to operate.

Returns
dict – key: *fname* value -> dict, *Ca_C_Np1*, *Ca_C_O*, *Cm1_N_Ca*, *N_Ca_C*

`idpconfgen.libs.libhigherlevel.`**`cli_helper_calc_torsions`**(*fname*, *fdata*, ***kwargs*)

Help *cli_torsion* to operate.

Returns
dict – key: *fname* value: -> dict, *phi*, *phi*, *omega* -> list of floats

`idpconfgen.libs.libhigherlevel.`**`cli_helper_calc_torsionsJ`**(*fdata_tuple*, ***kwargs*)

Help cli_torsionsJ.py.

`idpconfgen.libs.libhigherlevel.`**`convert_bond_geo_lib`**(*bond_geo_db*)

Convert bond geometry library to bond type first hierarchy.

Restructure the output of *read_trimer_torsion_planar_angles* such that the main keys are the bond types, followed by the main residue, the pairs in the trimer, and, finally, the torsion angles.

Parameters
bond_geo_db (*dict*) – The output of *read_PDBID_from_source*.

Returns
dict

`idpconfgen.libs.libhigherlevel.`**`download_pipeline`**(*func*, *logfilename='.download'*)

Context pipeline to download PDB/mmCIF files.

Exists because fetching and download filtered PDBs shared the same operational contextm, only differing on the function which orchestrates the download process.

Parameters

- **func** (*function*) – The actual function that orchestrates the download operation.
- **logfilename** (*str, optional*) – The common stem name of the log files.

idpconfgen.libs.libhigherlevel.**extract_secondary_structure**(*pdbid*, *ssdata*, *atoms='all'*, *minimum=0*, *structure='all'*)

Extract secondary structure elements from PDB data.

Parameters

- **pdbid** (*tuple*) – Where index 0 is the PDB id code, for example *12AS.pdb*, or *12AS_A*, or *12AS_A_seg1.pdb*. And, index 1 is the PDB data itself in bytes.
- **ssdata** (*dict*) – Dictionary containing the DSSP information. Must contain a key equal to *Path(pdbid).stem, where `dssp* key contains the DSSP information for that PDB.
- **atoms** (*str or list of str or bytes, optional*) – The atom names to keep. Defaults to *all*.
- **minimum** (*int, optional*) – The minimum size a segment must have in order to be considered. Defaults to 0.
- **structure** (*str or list of chars*) – The secondary structure character to separate. Multiple can be given in the form of a list.

idpconfgen.libs.libhigherlevel.**get_bond_geos**(*fdata*)

Calculate bond angles from structure.

Parameters

- **fdata** (data to **:pyclass:`idpconfgen.libstructure.Structure`**)
- **degrees** (*bool, optional*) – Defaults to False.
- **decimals** (*int, optional*) – Defaults to 3.

idpconfgen.libs.libhigherlevel.**get_separate_torsions**(*torsions_array*)

Separate torsion angles according to the protein backbone concept.

Considers torsion angles for bonds in between atom pairs:

- CA - C
- C - N
- N - CA

Backbone obeys the order: N-CA-C-N-CA-C(...)

And the first value corresponds to a CA-C pair, because the first N-CA pair of the protein backbone has no torsion angle.

idpconfgen.libs.libhigherlevel.**get_torsions**(*fdata*, *degrees=False*, *decimals=3*, *validate=True*)

Calculate torsion angles from structure.

Corrects for labels not sorted according to (N, CA, C). But does not correct for non-sorted residues (do these exist?).

Calculates torsion angles with *:func:libcalc.calc_torsion_angles*.

Parameters

- **fdata** (*str, bytes or Path*) – A path to the structure file, or the string representing the file. In fact, accepts any type *:class:libstructure.Structure* would accept.
- **degrees** (*bool, optional*) – Whether to return torsion angles in degrees or radians.
- **decimals** (*int, optional*) – The number of decimals to return. Defaults to 3.

- **validate** (*bool*) – Validates coordinates and labels according to general expectations. Expectations are as described by functions: - :func:libhigherlevel.validate_backbone_labels_for_torsion - :func:libhigherlevel.validate_coords_for_backbone_torsions

  If *False*, does not perform validation. Be sure to provide PDBs what will output meaningful results.

Returns

np.ndarray – As described in *:func:libcalc.calc_torsion_angles()* but with *decimals* and *degrees* applied.

See also:

`libhigherlevel.validate_backbone_labels_for_torsion`, `libhigherlevel.validate_coords_for_backbone_torsions`, `libcalc.calc_torsion_angles`

`idpconfgen.libs.libhigherlevel.`**`get_torsionsJ`**`(`*fdata*, *decimals=5*, *degrees=False*, *hn_terminal=True*, *hn_labels=('H', 'H1')*, *proline_value=nan*`)`

Calculate HN-CaHA torsion angles from a PDB/mmCIF file path.

Needs atom labels: H or H1, N, CA, HA or HA2 (Glycine).

Parameters

- **decimals** (*int*) – The decimal number to round the result.
- **degrees** (*bool*) – Whether or not to return values as degrees. If *False* returns radians.
- **hn_terminal** (*bool*) – If the N-terminal has no hydrogens, flag *hn_terminal* should be
- **provided as `False`, and the first residue will be discarded.**
- **If `True` expects N-terminal to have `H` or `H1`.**

Returns

np.ndarray – The NH-CaHA torsion angles for the whole protein. Array has the same length of the protein if N-terminal has H, otherwise has length of protein minus 1.

Notes

Not optimized for speed. Not slow either.

`idpconfgen.libs.libhigherlevel.`**`read_trimer_torsion_planar_angles`**`(`*pdb*, *bond_geometry*`)`

Create a trimer/torsion library of bend/planar angles.

Given a PDB file:

1) reads each of its trimers, and for the middle residue:
2) Calculates phi/psi and rounds them to the closest 10 degree bin
3) assign planar angles found for that residue to the trimer/torsion key.
4) the planar angles are converted to the format needed by cli_build, which is that of (pi - angle) / 2.
5) updates that information in *bond_gemetry*.

Created key:values have the following form in *bond_geometry* dict:

```
{
    'AAA:10,-30': {
        'Cm1_N_Ca': [],
        'N_Ca_C': [],
        'Ca_C_Np1': [],
        'Ca_C_O': [],
        }
    }
```

Parameters

- **pdb** (any input of *libstructure.Structure*) – The PDB/mmCIF file data.
- **bond_geometry** (*dict*) – The library dictionary to update.

Returns

None

idpconfgen.libs.libhigherlevel.**validate_backbone_labels_for_torsion**(*labels*, *minimum=2*)

Validate labels for torsion angle calculation.

Assumes labels are aligned with their corresponding coordinates. Yet, coordinates have no scope in this function.

Excepts only the mininal backbone labels, these are: N, CA, and C.

Parameters

- **labels** (*np.array of shape (N,) or alike*) – Where N % 3 equals 0.
- **minimum** (*int*) – The minimum number of residues to consider valid.

idpconfgen.libs.libhigherlevel.**validate_coords_for_backbone_torsions**(*coords*, *minimum=2*)

Validate coords for torsions.

Does NOT validate with values have physical sense. Validates only valid input for *:func:libcalc.calc_torsion_angles*.

Validations performed:

- coords are two-dimensional arrays
- coords have 3 values in the first dimension (XYZ) (shape[1])
- number of coords is multiple of 3.

Returns

str – A string explaining the error if an error is found. An empty string if no error is found.

Lib I/O

Functions and Classes regarding Input/Output.

class idpconfgen.libs.libio.**FileIterator**(*origin*, *ext='.pdb'*)

Iterate over files.

class idpconfgen.libs.libio.**FileIteratorBase**

File iterator base class.

class idpconfgen.libs.libio.**FileReaderIterator**(*origin*, ***kwargs*)

Dispaches file read iteractor.

Here I created a class interface instead of a function for convinience.

class idpconfgen.libs.libio.**TarFileIterator**(*origin*, *ext='.pdb'*)

Iterate over files in tarfiule.

idpconfgen.libs.libio.**add_existent_files**(*storage*, *source*)

Add files that exist to a list.

Given a list of *source* Paths, if Path exist adds it to *storage*.

Adds Path instances.

idpconfgen.libs.libio.**concatenate_entries**(*entry_list*)

Concatente entries.

Entries can be given in a list of entries or file paths with entry lists. Single entries in the input list are used directly while files are read and their lines added one by one to the concatenated list.

Notice:

Does not descriminate between single entries and mispelled file paths. Every string that cannot be openned as path is considered an individual entry.

Parameters

entry_list (*lits*) – List containing strings or file paths

Returns

list – Concatenated strings plus lines in files.

idpconfgen.libs.libio.**extract_from_tar**(*tar_file*, *output=None*, *ext='.pdb'*)

Extract files from tarfile.

Parameters

- **tar_file** (*str*) – The tarfile to extract.
- **output** (*str, optional*) – The folder to where extract tar files. Defaults to current working directory.

Returns

list – A list of Path-objects pointing to the extracted files.

idpconfgen.libs.libio.**file_exists**(*path*, *ifdir=<function get_false>*, *doelse=<function get_false>*)

Confirm file exists.

Parameters

path (*path-object or str*)

Returns

bool

idpconfgen.libs.libio.**glob_folder**(*folder*, *ext*)

List files with extention *ext* in *folder*.

Does NOT perform recursive search.

Parameters

- **folder** (*str*) – The path to the folder to investigate.
- **ext** (*str*) – The file extention. Can be with or without the dot [.] preffix.

Returns
list of Path objects – SORTED list of matching results.

idpconfgen.libs.libio.**has_suffix**(*path*, *ext=None*)

Evaluate file suffix according to *ext* condition.

Parameters

- **path** (*str of Path*) – The file path.
- **ext** (*str*) – The file extension. Can be dotted '.' (.csv) or plain (csv).

Returns

bool –

True
Always if *ext* is None. If path suffix equals *ext*.

False
Otherwise

idpconfgen.libs.libio.**has_suffix_fasta**(*path*, *, *ext='.fasta'*)

Evaluate file suffix according to *ext* condition.

Parameters

- **path** (*str of Path*) – The file path.
- **ext** (*str*) – The file extension. Can be dotted '.' (.csv) or plain (csv).

Returns

bool –

True
Always if *ext* is None. If path suffix equals *ext*.

False
Otherwise

idpconfgen.libs.libio.**is_valid_fasta_file**(*path*)

Return True if FASTA file is valid.

idpconfgen.libs.libio.**list_files_recursively**(*folder*, *ext=None*)

List files recursively from source folder.

Parameters

- **folder** (*string or Path*) – The folder from where to start searching.
- **ext** (*string*) – The file extension to consider. Files without the defined *ext* will be ignored. Defaults to `None`, all files are considered.

Returns
unsorted list – Of the file paths relative to the source *folder*.

idpconfgen.libs.libio.**log_nonexistent**(*files*)

Log to ERROR files that do not exist.

Parameters
files (*iterable of Paths*)

idpconfgen.libs.libio.**make_destination_folder**(*dest*)

Make a destination folder.

Returns
Path-object – A path pointing to the folder created. If `dest` is `None` returns a Path poiting to the CWD.

idpconfgen.libs.libio.**make_folder_or_cwd**(*folder*)

Make a folder or five CWD.

Parameters
folder (*str or Path*) – Make the folder. If folder is None return the CWD.

Returns
Path

idpconfgen.libs.libio.**parse_suffix**(*ext*)

Represent a suffix of a file.

Example

parse_suffix('.pdf') '.pdf'

parse_suffix('pdf') '.pdf'

Parameters
ext (*str*) – String to extract the suffix from.

Returns
str – File extension with leading period.

idpconfgen.libs.libio.**paths_from_flist**(*path*)

Read Paths from file listing paths.

Returns
Map generator – Path representation of the entries in the file.

idpconfgen.libs.libio.**read_FASTAS_from_file**(*fpath*)

Read FASTA sequence from file.

idpconfgen.libs.libio.**read_FASTAS_from_file_to_strings**(*fpath*)

Read FASTA sequences from file.

FASTA is output as string.

Note that there should be no blank spaces between different sequences. The final return will be a dictionary where the key value will be ">XYZ" header and the value will be a list of individual residue letters ['X', 'Y', 'Z'].

idpconfgen.libs.libio.**read_PDBID_from_folder**(*folder*)

Read PDBIDs from folder.

Parameters
folder (*str*) – The folder to read.

Returns
idpconfgen.libs.libpdb.PDBList

idpconfgen.libs.libio.**read_PDBID_from_source**(*source*)

Read PDBIDs from destination.

Accepted destinations:

- folder
- tarfile

Returns
idpconfgen.libs.libpdb.PDBList.

idpconfgen.libs.libio.**read_PDBID_from_tar**(*tar_file*)

Read PDBIDs from tarfile.

Note: Case-specific function, not an abstraction.

Parameters
tar_file (idpconfgen.Path) – The tarfile to read.

Returns
idpconfgen.libs.libpdb.PDBList – If file is not found returns an empty PDBList.

idpconfgen.libs.libio.**read_dict_from_json**(*path*)

Read dict from json.

idpconfgen.libs.libio.**read_dict_from_pickle**(*path*)

Read dictionary from pickle.

idpconfgen.libs.libio.**read_dict_from_tar**(*path*)

Read dictionary from .tar file.

idpconfgen.libs.libio.**read_dictionary_from_disk**(*path*)

Read a dictionary from disk.

Accepted formats:

- pickle
- json

Returns
dict

idpconfgen.libs.libio.**read_lines**(*fpath*)

Read lines from path.

idpconfgen.libs.libio.**read_path_bundle**(*path_bundle*, *ext=None*, *listext='.list'*)

Read path bundle.

Read paths encoded in strings, folders or files that are list of files.

If a string or Path object points to an existing file,
register that path.

If a string points to a folder, registers all files in that folder
that have extension *ext*, recursively.

If a string points to a file with extension *listext*, registers
all files referred in the *listext* file and which exist in disk.

Non-existing files are log as error messages.

Parameters

- **path_bundle** (*list-like*) – A list containing strings or paths that point to files or folders.
- **ext** (*string*) – The file extension to consider. If `None` considers all files. Defaults to `None`.
- **listext** (*string*) – The file extension to consider as a file listing other files. Defaults to `.flist`.

Returns
generator – A generator that complies with the specifications described

`idpconfgen.libs.libio.read_text`(*fpath*)
Read text from path.

`idpconfgen.libs.libio.save_dict_to_json`(*mydict*, *output='mydict.json'*, *indent=True*, *sort_keys=True*)
Save dictionary to JSON.

`idpconfgen.libs.libio.save_dict_to_pickle`(*mydict*, *output='mydict.pickle'*, ***kwargs*)
Save dictionary to pickle file.

`idpconfgen.libs.libio.save_dictionary`(*mydict*, *output='mydict.pickle'*)
Save dictionary to disk.

Accepted formats:

- pickle
- json

Parameters

- **mydict** (*dict*) – The dict to be saved.
- **output** (*str or Path*) – The output file. Format is deduced from file extension.

Raises
`KeyError` – If extension is not a compatible format.

`idpconfgen.libs.libio.save_file_to_tar`(*tar*, *fout*, *data*)
Save content to a file inside a tar.

Parameters

- **tar** (*openned tar-file instance*)
- **fout** (*str*) – The name under which to save the `data`.
- **data** (*str or bytes*) – Data to save to *fout* named file inside *tar*.

`idpconfgen.libs.libio.save_pairs_to_disk`(*pairs*, *destination*)
Save pairs to files.

Parameters

- **pairs** (*list or tuple*) – First indexes are used as file names, second indexes as info to write to files.
- **destination** (*str or Path-like*) – Destination in the disk where to save the files. Current options are: a folder, a TAR file.

idpconfgen.libs.libio.**save_pairs_to_files**(*pairs*, *destination=None*)

Save pairs to files.

Where each pair (tuple or list of length 2) is composed of:
(file name, data content)

Parameters

- **pairs** (*list or tuple*) – The pairs of information to save to the disk. Index 0 is the file name, and index 1 is the data content.
- **destination** (*str or Path, optional*) – The folder where to save the files. It is NOT the file name. Defauls to the CWD.

idpconfgen.libs.libio.**save_pairs_to_tar**(*pairs*, *destination*)

Save pairs to files inside a TAR file.

Where each pair (tuple or list of length 2) is composed of:
(file name, data content)

Parameters

- **pairs** (*list or tuple*) – The pairs of information to save to the disk. Index 0 is the file name, and index 1 is the data content.
- **destination** (*str or Path*) – The TAR file where to save the files. It is NOT the file name. If exists appends, if not creates.

Lib Multicore

Multi-core related objects.

idpconfgen.libs.libmulticore.**consume_iterable_in_list**(*func*, **args*, ***kwargs*)

Consumes a generator to a list.

Used as a wrapper to send generators to Python multiprocessing lib.

Returns
list of length N – Where N is the number of times *func* yields.

idpconfgen.libs.libmulticore.**flat_results_from_chunk**(*execute*, *func*, **args*, ***kwargs*)

Flattens a result coming from consume_iterable_in_list execution.

Parameters

- **execute** (*callable*) – The prepared multiprocessing function. Each item of its iteration should be iterable of iterable.
- **func** (*callable*) – The function to process each yielded result from the results in the multiprocessing fragment.

idpconfgen.libs.libmulticore.**pool_function**(*func*, *items*, *method='imap_unordered'*, *ncores=1*)

Multiprocess Pools a function.

Parameters

- **func** (*callable*) – The function to execute along the iterable *items*. If this function expects additional *args and `kwargs*, prepare it previously using *functools.partial*.
- **items** (*interable*) – Elements to pass to the *func*.

- **method** (*str*) – The `Pool` method to execute. Defaults to *imap_unordered*.
- **ncores** (*int*) – The number of cores to use. Defaults to *1*.

`idpconfgen.libs.libmulticore.`**`pool_function_in_chunks`**(*func*, *items*, *chunks=5000*, ***kwargs*)

Execute `func` in `chunks` of `items` using *Pool*.

Yields the results after each fragment.

Parameters

- **func** (*callable*) – The function to execute over `items`. If this function expects additional *args and `kwargs*, prepare it previously using *functools.partial*.
- **items** (*iterable*) – The items to process by the `funct`.
- ***args** (*any*) – Additional positional arguments to send to `func`.
- **chunks** (*int*) – The size of each fragment processed multiprocessing before yielding.
- ****kwargs** (*any*) – Additional keyword arguments to send to `func`.

Yields

list – Containing the results after each fragment.

`idpconfgen.libs.libmulticore.`**`starunpack`**(*func*, **args*, ***kwargs*)

Unpack first argument of *args*.

Lib Parse

Parsing routines for different data structure.

All functions in this module receive a certain Python native datastructure, parse the information inside and return/yield the parsed information.

`idpconfgen.libs.libparse.`**`convert_int_float_lines_to_dict`**(*lines*)

Convert string lines composed of putative int and float to dict.

Example

```
>>> convert_int_float_lines_to_dict(['1 2'])
{1: 2.0}
```

```
>>> convert_int_float_lines_to_dict(['1 2\n', '3 45.5\n'])
{1: 2.0, 3: 45.5}
```

`idpconfgen.libs.libparse.`**`convert_tuples_to_lists`**(*data*)

Recursively processes input data and converts it all to list of lists.

Parameter

data : list of tuple

returns
result (*list of list*)

idpconfgen.libs.libparse.**fill_list**(*seq*, *fill*, *size*)

Fill list with *fill* to *size*.

If seq is not a list, converts it to a list.

Returns
list – The original with fill values.

idpconfgen.libs.libparse.**get_diff_between_aa1l**(*group1*, *, *group2={'A', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'K', 'L', 'M', 'N', 'P', 'Q', 'R', 'S', 'T', 'V', 'W', 'Y', 'd', 'e', 'p', 's', 't'}*)

Get difference between groups as a set.

idpconfgen.libs.libparse.**get_diff_between_groups**(*group1*, *group2*)

Get difference between groups as a set.

idpconfgen.libs.libparse.**get_mers**(*seq*, *size*)

Get X-mers from seq.

Example

```
>>> get_mers('MEAIKHD', 3)
{'MEA', 'EAI', 'AIK', 'KHD'}
```

idpconfgen.libs.libparse.**get_seq_chunk**(*seq*, *idx*, *size*)

Get a fragment from sequence at start at *idx* with *size*.

idpconfgen.libs.libparse.**get_seq_next_residue**(*seq*, *idx*, *size*)

Get the next residue after the fragment.

idpconfgen.libs.libparse.**get_trimer_seq**(*seq*, *idx*)

Get sequence of trimer.

idpconfgen.libs.libparse.**group_by**(*data*)

Group data by indexes.

Parameters
data (*iterable*) – The data to group by.

Returns
list (*[[type, slice],]*)

Examples

```
>>> group_by('LLLLLSSSSSSEEEEE')
[['L', slice(0, 5)], ['S', slice(5, 11)], ['E', slice(11,16)]]
```

idpconfgen.libs.libparse.**group_runs**(*li*, *tolerance=1*)

Group consecutive numbers given a tolerance.

idpconfgen.libs.libparse.**is_valid_fasta**(*fasta*)

Confirm string is a valid FASTA primary sequence.

Does not accept headers, just the protein sequence in a single string.

idpconfgen.libs.libparse.**make_list_if_not**(*item*)

Make a list from item.

idpconfgen.libs.libparse.**mkdssp_w_split**(*pdb*, *cmd*, ***kwargs*)

Execute *mkdssp* from DSSP.

Saves the data splitted accoring to backbone continuity as identified by *mkdssp*. Splits the input PDB into bb continuity segments.

https://github.com/cmbi/dssp

Parameters

- **pdb** (*Path*) – The path to the pdb file.
- **cmd** (*str*) – The command to execute the external DSSP program.

Yields

from *split_pdb_by_dssp*

idpconfgen.libs.libparse.**parse_dssp**(*data*, *reduced=False*)

Parse DSSP file data.

JSON doesn't accept bytes That is why *data* is expected as str.

idpconfgen.libs.libparse.**pop_difference_with_log**(*dict1*, *dict2*, *logmsg='Removing {} from the dictionary.\n'*)

Pop keys in *dict1* that are not present in *dict2*.

Reports pop'ed keys to log INFO.

Operates *dict1* in place.

Parameters

dict1, dict2 (*dict*)

Returns

None

idpconfgen.libs.libparse.**remap_sequence**(*seq*, *target='A'*, *group=('P', 'G')*)

Remap sequence.

Parameters

- **seq** (*Protein primary sequence in FASTA format.*)
- **target** (*str (1-char)*) – The residue to which all other residues will be converted to.
- **group** (*tuple*) – The list of residues that excape map/conversion.

Returns
str – The remaped string.

Examples

```
>>> remap_sequence('AGTKLPHNG')
'AGAAAPAAG'
```

idpconfgen.libs.libparse.**remove_empty_keys**(*ddict*)

Remove empty keys from dictionary.

idpconfgen.libs.libparse.**sample_case**(*input_string*)

Sample all possible cases combinations from *string*.

Examples

```
>>> sample_case('A')
{'A', 'a'}
```

```
>>> sample_case('Aa')
{'AA', 'Aa', 'aA', 'aa'}
```

idpconfgen.libs.libparse.**split_by_ranges**(*seq*, *ranges*)

Split a string into substrings based on a list of custom ranges.

Parameters

- **seq** (*str*) – String or sequence of desire to be split.
- **ranges** (*list of int*) – Integers represent the index of which the split will occur. Each value is not inclusive.

Returns
chunks (*list*) – List of split strings at their desired locations.

idpconfgen.libs.libparse.**split_into_chunks**(*string*, *size=150*)

Split a string into chunks of characters.

The last chunk may be longer or shoter.

Parameters

- **string** (*str*) – String of characters to split
- **size** (*int*) – Integer value of chunk sizes. Defaults to 200.

Returns
chunks (*list*) – List of strings split into chunks of pre-determined sizes.

idpconfgen.libs.libparse.**split_pdb_by_dssp**(*pdbfile*, *dssp_text*, *minimum=2*, *reduced=False*)

Split PDB file based on DSSP raw data.

Parameters

- **minimum** (*int*) – The minimum length allowed for a segment.
- **reduce** (*bool*) – Whether to reduce the DSSP nomemclature to H/E/L.

idpconfgen.libs.libparse.**translate_seq_to_3l**(*input_seq*)

Translate 1-letter sequence to 3-letter sequence.

Currently translates 'H' to 'HIP', to accommodate double protonation. Editing to 'HIS' causes issues with libbuild.

idpconfgen.libs.libparse.**values_to_dict**(*values*)

Generalization of converting parameters to dict.

Adapted from: https://github.com/joaomcteixeira/taurenmd/blob/6bf4cf5f01df206e9663bd2552343fe397ae8b8f/src/taurenmd/libs/libcli.py#L94-L138

Parameters
values (*string*) – List of values with the format "par1=1 par2='string' par3=[1,2,3]

Returns
param_dict (*dictionary*) – Converted string above to dictionary with = denoting linkage E.g. {'par1': 1, 'par2':'string', 'par3': [1,2,3]}

Lib PDB

Contain handlers of PDB information.

class idpconfgen.libs.libpdb.**PDBID**(*name*, *chain=None*, *segment=None*)

PDB object identifier.

Identifies unique downloadable/stored units.

In the current implmentation each unit is one PDB chain, which is identified by the PDBID and its chain identifier.

Parameters

- **name** (obj:*str*) – The PDBID, for example: 1ABC
- **chain** (obj:*str*) – The chain identifier. Defaults to None.

Variables

- **name** – The four character PDB identifier.
- **chain** – The chain identifier.

class idpconfgen.libs.libpdb.**PDBIDFactory**(*name*)

Parse input for PDBID instatiation.

Parameters
name (*str or Path*) – The code name or ID that identified the PDB. Possible formats:

- XXXX
- XXXXC*
- XXXX_C*
- *.pdb

where XXXX is the PDBID code, C is the chain ID and * means any number of characters. PDB and chaind ID codes are any digits, lower and upper case letters.

Returns
PDBID object.

class idpconfgen.libs.libpdb.**PDBList**(*pdb_names*)

List of PDBID objects.

Parameters
pdb_names (*obj:iterator*) – An iterator containing the PDB names. PDB names can be in the form accepted by PDBIDFactory or PDBID objects.

difference(*other*)

Difference between self and other.

Returns
PDBList

property name_chains_dict

Chains dictionary map.

Type
Export PDBIDs

property pdbids

Generate the PDBID names.

to_tuple()

Convert PDBList to sorted tuple.

write(*filename='PDBIDs.list'*)

Write to a file the PDBIDs in the PDBList.

Parameters
filename (*str, optional*) – The output file name.

idpconfgen.libs.libpdb.**delete_insertions**(*lines*)

Delete insertions.

Adapted from pdbtools and optimized for this context.

Visit pdb-tools at: https://github.com/haddocking/pdb-tools/blob/master/pdbtools/pdb_delinsertion.py

idpconfgen.libs.libpdb.**format_atom_name**(*atom*, *element*, *AFD={1: {1: ' {:<3s}', 2: ' {:<3s}', 3: ' {:<3s}', 4: '{:<4s}'}, 2: {1: '{:<4s}', 2: '{:<4s}', 3: '{:<4s}', 4: '{:<4s}'}}*)

Format PDB Record line Atom name.

Further Reading:

- https://www.cgl.ucsf.edu/chimera/docs/UsersGuide/tutorials/pdbintro.html

Parameters

- **atom** (*str*) – The atom name.
- **element** (*str*) – The atom element code.

Returns
str – Formatted atom name.

idpconfgen.libs.libpdb.**format_chainid**(*chain*)

Format chain identifier to one letter.

This is required to receive chain IDs from mmCIF files, which may have more than one letter.

idpconfgen.libs.libpdb.**get_fasta_from_PDB**(*pdbid*)

Extract FASTA from PDB.

idpconfgen.libs.libpdb.**is_pdb**(*datastr*)

Detect if *datastr* if a PDB format v3 file.

Lib Structure

Store internal protein structure representation.

Classes

Structure

The main API that represents a protein structure in IDPConfGen.

class idpconfgen.libs.libstructure.**Structure**(*data*, ***kwargs*)

Hold structural data from PDB/mmCIF files.

Run the `.build()` method to read the structure.

Cases for PDB Files: * If there are several MODELS only the first model is considered.

Parameters

data (*str, bytes, Path*) – Raw structural data from PDB/mmCIF formatted files. If *data* is a path to a file it *must* be a *pathlib.Path* object. If string or bytes, it must be the raw content of the input file.

Examples

Opens a PDB file, selects only chain 'A' and saves selection to a file. >>> s = Structure(Path('1ABC.pdb')) >>> s.build() >>> s.add_filter_chain('A') >>> s.write_PDB('out.pdb')

Opens a mmCIF file, selects only residues above 50 and saves selection to a file. >>> s = Structure(Path('1ABC.cif')) >>> s.build() >>> s.add_filter(lambda x: int(x[col_resSeq]) > 50) >>> s.write_PDB('out.pdb')

```
>>> with open('1ABC.pdb', 'r') as fin:
>>>     lines = fin.read()
>>> s = Structure(lines)
>>> s.build()
```

add_filter(*function*)

Add a function as filter.

add_filter_backbone(*minimal=False*)

Add filter to consider only backbone atoms.

add_filter_chain(*chain*)

Add filters for chain.

add_filter_record_name(*record_name*)

Add filter for record names.

build()

Read structure raw data in `rawdata`.

After *.build()*, filters and data can be accessed.

property chain_set

All chain IDs present in the raw dataset.

clear_filters()

Clear/Deletes registered filters.

property consecutive_residues

Consecutive residue groups from filtered atoms.

property coords

Coordinates of the filtered atoms.

As float.

property data_array

Contain structure data in the form of a Numpy array.

property fasta

FASTA sequence of the *filtered_atoms* lines.

HETATM residues with non-canonical codes are represented as X.

property filtered_atoms

Filter data array by the selected filters.

Returns
list – The data in PDB format after filtering.

property filtered_residues

Filter residues according to *filters*.

property filters

Filter functions registered ordered by registry record.

get_PDB(*pdb_filters=None*, *renumber=True*)

Convert Structure to PDB format.

Considers only filtered lines.

Returns
generator

get_sorted_minimal_backbone_coords(*filtered=False*)

Generate a copy of the backbone coords sorted.

Sorting according N, CA, C.

This method was created because some PDBs may not have the backbone atoms sorted properly.

Parameters
filtered (*bool, optional*) – Whether consider current filters or raw data.

pop_last_filter()

Pop last filter.

property residues

Residues of the structure.

Without filtering, without chain separation.

write_PDB(*filename*, ***kwargs*)

Write Structure to PDB file.

idpconfgen.libs.libstructure.**concatenate_residue_labels**(*labels*)

Concatenate residue labels.

This function is a generator.

Parameters

labels (*numpy array of shape (N, M)*) – Where N is the number of rows, and M the number of columns with the labels to be concatenated.

idpconfgen.libs.libstructure.**detect_structure_type**(*datastr*)

Detect structure data parser.

Uses `structure_parsers`.

Returns

parser – That which can parse *datastr* to a :py::class:`Structure'.

idpconfgen.libs.libstructure.**filter_record_lines**(*lines*, *which='both'*)

Filter lines to get record lines only.

idpconfgen.libs.libstructure.**gen_empty_structure_data_array**(*number_of_atoms*)

Generate an array data structure to contain structure data.

Parameters

number_of_atoms (*int*) – The number of atoms in the structure. Determines the size of the axis 0 of the structure array.

Returns

np.ndarray of (N, (attr:*libpdb.atom_slicers), dtype = '<U8') – Where N is the ``number_of_atoms`.*

idpconfgen.libs.libstructure.**generate_backbone_pairs_labels**(*da*)

Generate backbone atom pairs labels.

Used to create columns in report summaries.

Parameters

da (*Structure.data_array - like*)

Returns

Numpy Array of dtype str, shape (N,) – Where N is the number of minimal backbone atoms.

idpconfgen.libs.libstructure.**generate_residue_labels**(**residue_labels*, *fmt=None*, *delimiter=' - '*)

Generate residue labels column.

Concatenate labels in *residue_labels* using
concatenate_residue_labels.

Parameters

fmt (*str, optional*) – The string formatter by default we consider backbone atoms of a protein with less than 1000 residues. Defaults to *None*, uses '{:<8}', 8 or multiple of 8 according to length of residue_labels.

idpconfgen.libs.libstructure.**get_datastr**(*data*)

Get data in string format.

Can parse data from several formats:

- Path, reads file content
- bytes, converst to str
- str, returns the input

Returns
str – That represents the data

idpconfgen.libs.libstructure.**is_backbone**(*atom*, *element*, *minimal=False*)

Whether *atom* is a protein backbone atom or not.

Parameters

- **atom** (*str*) – The atom name.
- **element** (*str*) – The element name.
- **minimal** (*bool*) – If *True* considers only *C* and *N* elements. *False*, considers also *O*.

idpconfgen.libs.libstructure.**parse_cif_to_array**(*datastr*, ***kwargs*)

Parse mmCIF protein data to array.

Array is as given by *gen_empty_structure_data_array()*.

idpconfgen.libs.libstructure.**parse_pdb_to_array**(*datastr*, *which='both'*)

Transform PDB data into an array.

Parameters

- **datastr** (*str*) – String representing the PDB format v3 file.
- **which** (*str*) – Which lines to consider ['ATOM', 'HETATM', 'both']. Defaults to *'both'*, considers both 'ATOM' and 'HETATM'.

Returns
numpy.ndarray of (N, len(*libpdb.atom_slicers*)) – Where N are the number of ATOM and/or HETATM lines, and axis=1 the number of fields in ATOM/HETATM lines according to the PDB format v3.

idpconfgen.libs.libstructure.**populate_structure_array_from_pdb**(*record_lines*, *data_array*)

Populate structure array from PDB lines.

Parameters

- **record_lines** (*list-like*) – The PDB record lines (ATOM or HETATM) to parse.
- **data_array** (*np.ndarray*) – The array to populate.

Returns
None – Populates array in place.

idpconfgen.libs.libstructure.**save_structure_by_chains**(*pdb_data*, *pdbname*, *altlocs=('A', '', ' ', '1')*, *chains=None*, *record_name=('ATOM', 'HETATM')*, *renumber=True*, ***kwargs*)

Save PDBs/mmCIF in separated chains (PDB format).

Logic to parse PDBs from RCSB.

idpconfgen.libs.libstructure.**structure_to_pdb**(*atoms*)

Convert table to PDB formatted lines.

Parameters

atoms (*np.ndarray, shape (N, 16) or similar data structure*) – Where N is the number of atoms and 16 the number of cols.

Yields

Formatted PDB line according to *libpdb.atom_line_formatter*.

idpconfgen.libs.libstructure.**write_PDB**(*lines*, *filename*)

Write Structure data format to PDB.

Parameters

- **lines** (*list or np.ndarray*) – Lines contains PDB data as according to *parse_pdb_to_array*.
- **filename** (*str or Path*) – The name of the output PDB file.

Lib Timer

Manages time.

class idpconfgen.libs.libtimer.**ProgressBar**(*total*, *prefix=''*, *suffix=''*, *decimals=1*, *bar_length=20*)

Contextualize a Progress Bar.

Parameters

- **total** (*int convertable*) – The total number o iteractions expected.
- **prefix** (*str*) – Some prefix to enhance readability.
- **suffix** (*str*) – Some suffix to enhance readability.
- **decimals** (*int-convertable*) – The demicals to show in percentage. Defaults to *1*.
- **bar_length** (*int, float, -convertable*) – The length of the bar. If not provided (`None`), uses 20.
- **Thanks to for the initial template function**
- **https** (*//dev.to/natamacm/progressbar-in-python-a3n*)

Examples

```
>>> with ProgressBar(5000, suffix='frames') as PB:
>>>     for i in trajectory:
>>>         # do something
>>>         PB.increment()
```

increment()

Print next progress bar increment.

class idpconfgen.libs.libtimer.**ProgressCounter**(*suffix='Running operations'*, ***kwargs*)

Represent progression via a counter.

increment()

Increment counter by one.

Represent progression in terminal.

class idpconfgen.libs.libtimer.**ProgressFake**(**args*, ***kwargs*)

Simulate the interface of ProgressBar but does nothing.

Used in servers where ProgressBar makes no sense.

increment()

Simulate ProgressBar interface.

Does nothing.

class idpconfgen.libs.libtimer.**ProgressWatcher**(*items*, **args*, ***kwargs*)

Construct a Progress Watcher context.

idpconfgen.libs.libtimer.**record_time**(*process_name=''*, **args*, ***kwargs*)

Record time of function execution.

Use as decorator.

idpconfgen.libs.libtimer.**timeme**(*func*, **args*, ***kwargs*)

Log the time taken to run a function.

Lib Validate

Tools to validate conformers.

Recognition of this module should be grated to:

- @AlaaShamandy
- @joaomcteixeira

Please see: https://github.com/julie-forman-kay-lab/IDPConformerGenerator/pull/23

idpconfgen.libs.libvalidate.**eval_bb_bond_length_distribution**(*name*, *pdb_data*)

.

idpconfgen.libs.libvalidate.**evaluate_vdw_clash_by_threshold_from_disk**(*name*, *pdb_data*, *atoms_to_consider*, *elements_to_consider*, ***kwargs*)

Evaluate clashes in a structure.

Created to evaluate conformers from disk.

idpconfgen.libs.libvalidate.**report_sequential_bon_len**(*bond_distances*, *expected_bond_length*, *invalid_bool*, *labels*)

Generate a report from bond distances of sequential bonds.

Returns
string

idpconfgen.libs.libvalidate.**report_vdw_clash**(*data_array*, *pair1*, *pair2*, *distances*, *radii_sum*, *overlap*)

Prepare a report of the identified clashes.

Parameters

- **data_array** (*np.ndarray, shape (N, M)*) – A numpy data_array as given by *idpconfgen.libs.libstructure.Structure.data_array*.

- **pair1, pair2** (*ordered iterable of integers*) – The row indexes where to retrieve atom information from `data_array´. pair1 and pair2 must be aligned in order for the report to make sense, that it, the first item of pair1 clashes with the first item of pair2.
- **distances, threshold** (*indexable of length equal to pair1/2 length*) – Contain distances and clash thresholds for the different identified clashes.

Returns
str – The report.

idpconfgen.libs.libvalidate.**validate_bb_bond_len**(*coords*, *tolerance=0.01*)

Validate backbone bond lengths of *coords*.

Considers *coords* are already sorted to (N, CA, C) per residue. Considers only (N, CA, C) atoms are present in *coords*. Evalutes against N-CA, CA-C and C-Np1 distances used in IDPConfGen.

Parameters
tolerance (*float*) – A tolerance in the same units as *coords*. Conflicts under the tolerance are consider valid.

Returns
np.array, dtype=bool, shape (N-1,) – True if bond length is invalid. False if bond length is valid.

np.array, dtype=np.float, shape (N-1,)
The computed bond distances.

np.array, dtype=np.float, shape (N-1,)
The expected bond lengths

idpconfgen.libs.libvalidate.**validate_bb_bonds_len_from_disk**(*name=None*, *pdb_data=None*, *tolerance=0.1*)

Validate backbone bond lengths of a structure stored in disk.

idpconfgen.libs.libvalidate.**validate_conformer_for_builder**(*coords*, *atom_labels*, *residue_numbers*, *bb_mask*, *carbonyl_mask*, *LOGICAL_NOT=<ufunc 'logical_not'>*, *ISNAN=<ufunc 'isnan'>*)

.

idpconfgen.libs.libvalidate.**vdw_clash_by_threshold**(*coords*, *protein_atoms*, *protein_elements*, *atoms_to_consider*, *elements_to_consider*, *residue_numbers*, *residues_apart=2*, *vdW_radii='tsai1999'*, *vdW_overlap=0.0*)

Calculate vdW clashes from XYZ coordinates and identity masks.

Parameters

- **coordinates** (*numpy array, dtype=float, shape (N, 3)*) – The atom XYZ coordinates.
- **protein_atoms** (*numpy array, dtype=str, shape (N,)*) – The protein atom names.
- **protein_elements** (*numpy array, dtype=str, shape(N,)*) – The protein atom elements.
- **atoms_to_consider** (*list-like*) – The atoms in *protein_atoms* to consider in the vdW clash analysis.
- **elements_to_consider** (*list-like*) – The elements in *protein_elements* to consider in the vdW clash analysis.

- **residue_number** (*numpy array, dtype=int, shape (N,)*) – The residue number corresponding to each atom.
- **residues_apart** (*int, optional*) – The minimum number of residues apart to consider for a clash. Defaults to 2.
- **vdW_radii** (*str, optional*) – The VDW radii set to consider. Defaults to 'tsai1999'.

vdW_overlap
[float, optional] An overlap allowance in Angstroms. Defaults to 0.0, any distance less than vdW+vdW is considered a clash.

Returns
Same *vdw_clash_by_threshold_calc()* returns.

idpconfgen.libs.libvalidate.**vdw_clash_by_threshold_calc**(*coords*, *atc_mask*, *pure_radii_sum*, *distances_apart*, *vdW_overlap=0.0*)

Calculate van der Waals clashes from a pure sphere overlap.

Other masks used as parameters will be applied to the result of:

scipy.distance.cdist(coords, coords, 'euclidean')

Parameters

- **coords** (*np.array, dtype=np.float, shape (N, 3)*) – The protein XYZ coordinates.
- **atc_mask** (*np.array, dtype=np.bool, shape (N, N)*) – A boolean masks to filter only the atoms relevant to report. Usually this mask is prepared beforehand and can contain different considerations, such as residues apart and especific atom types. If *coords* contain only the coordinates desired to compute, then *atc_mask* should contain only TRUE entries.
- **pure_radii_sum** (*np.array, dtype=float, shape (N, N)*) – An all-to-all sum of the vdW radii. In other words, the threeshold after which a clash is considered to exist for each atom pair, before applying *vdW_overlap* allowance.
- **distances_apart** (*np.array, dtype=int, shape (N, N)*) – An all-to-all atom-to-atom residue to residue distance matrix.
- **vdW_overlap** (*float*) – The vdW overlap tolerance to apply.

Returns
tuple – rows, cols : of cdist applied to *coords* where clashes where found. distances found for those clashes computed distance threshold overlap, computed overlap distance between threshold and distance

idpconfgen.libs.libvalidate.**vdw_clash_by_threshold_common_preparation**(*protein_atoms*, *protein_elements*, *residue_numbers*, *atoms_to_consider=False*, *elements_to_consider=False*, *residues_apart=2*, *vdW_radii='tsai1999'*)

Prepare masks for vdW clash calculation.

Masks are prepared considering all-to-all distances will be computed using scipy.distance.cdist, so a (N, 3) array originates a (N, N) distance result.

atoms_to_consider and *elements_to_consider* are evaluated with logical AND, that is, only entries that satisfy both are considered.

Parameters

- **protein_atoms** (*np.array, of shape (N,), dtype string*) – A sequence of the protein atom names.
- **protein_elements** (*np.array, of shape (N,), dytpe compatible with '<U2'*) – A sequence of the protein atom elements aligned with *protein_atoms*.
- **residue_numbers** (*np.array of shape (N,), dtype=int*)
- **atoms_to_consider** (*sequence, optional*) – A tuple of the atom names to consider in the calculation. Defaults to FALSE, considers all atoms.
- **elements_to_consider** (*sequence, optional*) – A tuple of the element types to consider in the calculation. Defaults to FALSE, considers all elements.

1.5 Contributing

How to contribute to this project.

1.5.1 Fork this repository

Fork this repository before contributing.

Clone your fork

Next, clone your fork to your local machine, keep it up to date with the upstream, and update the online fork with those updates.

```
git clone https://github.com/YOUR-USERNAME/IDPConformerGenerator.git
cd IDPConformerGenerator
git remote add upstream git://github.com/julie-forman-kay-lab/IDPConformerGenerator.git
git fetch upstream
git merge upstream/master
git pull origin master
```

1.5.2 Install for developers

Create a dedicated Python environment where to develop the project. If you are using Anaconda go for:

```
conda env create -f requirements.yml
```

Install the package in the repository:

```
python setup.py develop --no-deps
```

This configuration, together with the use of the `src` folder layer, guarantee that you will always run the code after installation. Also, thanks to the `develop` flag, any changes in the code will be automatically reflected in the installed version.

1.5.3 Make a new branch

From the `master` branch create a new branch where to develop the new code:

```
git checkout master
git checkout -b new_branch
```

Develop the feature and keep regular pushes to your fork with comprehensible commit messages.

```
git status
git add (the files you want)
git commit (add a nice commit message)
git push origin new_branch
```

While you are developing, you can execute `tox` as needed to run your unittests or inspect lint, etc. `tox` usage is described in the last section of this page.

Update CHANGELOG

Update the changelog file under `docs/CHANGELOG.rst` with an explanatory bullet list of your contribution. Add that list right after the main title and before the last version subtitle:

```
Changelog
=========

* here goes my new additions explain them shortly and well

vX.X.X (1900-01-01)
-------------------
```

Also add your name to the authors list at `docs/AUTHORS.rst`.

Pull Request

Once you are finished, you can Pull Request you additions to the main repository, and engage with the community. Please read the docs/PULLREQUEST.rst guidelines first, you will see them when you open a PR.

Before submitting a Pull Request, verify your development branch passes all tests as *described bellow* . If you are developing new code you should also implement new test cases.

1.5.4 Uniformed Tests with tox

Thanks to Tox we can have a unified testing platform where all developers are forced to follow the same rules and, above all, all tests occur in a controlled Python environment.

Before creating a Pull Request from your branch, certify that all the tests pass correctly by running:

```
tox
```

These are exactly the same tests that will be performed online in the Github Actions.

Also, you can run individual environments if you wish to test only specific functionalities, for example:

```
tox -e lint  # code style
tox -e build  # packaging
tox -e docs  # only builds the documentation
tox -e tests  # runs code unit tests for your python version
tox -e tests -- -vv  # for full verbosity
```

1.6 How to cite

If you use IDPConformerGenerator, please always cite its original publication:

```
IDPConformerGenerator: A Flexible Software Suite for Sampling the Conformational Space␣
↪of Disordered Protein States
João M. C. Teixeira, Zi Hao Liu, Ashley Namini, Jie Li, Robert M. Vernon, Mickaël␣
↪Krzeminski, Alaa A. Shamandy, Oufan Zhang, Mojtaba Haghighatlari, Lei Yu, Teresa Head-
↪Gordon, and Julie D. Forman-Kay
The Journal of Physical Chemistry A 2022 126 (35), 5985-6003
DOI: 10.1021/acs.jpca.2c03726
```

If you use the Local Disordered Region Sampling (LDRS) module, please also cite:

```
Zi Hao Liu, João M C Teixeira, Oufan Zhang, Thomas E Tsangaris, Jie Li, Claudiu C␣
↪Gradinaru, Teresa Head-Gordon,
Julie D Forman-Kay, Local Disordered Region Sampling (LDRS) for ensemble modeling of␣
↪proteins with experimentally undetermined
or low confidence prediction segments, Bioinformatics, Volume 39, Issue 12, December␣
↪2023, btad739,
https://doi.org/10.1093/bioinformatics/btad739
```

1.7 Authors

1.7.1 Main Developers/Maintainers

- João M. C. Teixeira
- Zi Hao (Nemo) Liu

1.7.2 Contributors

- Alaa A. Shamandy (Co-op student University of Toronto, Canada)
- Jie (Jerry) Li (PhD student University of California Berkeley, USA)

1.7.3 All Authors

- João M. C. Teixeira
- Zi Hao (Nemo) Liu
- *Ashley Namini*
- Jie (Jerry) Li
- Robert M. Vernon
- *Mickaël Krzeminski*
- Alaa A. Shamandy
- *Oufan Zhang*
- Mojtaba Haghighatlari
- Teresa Head-Gordon
- Julie D. Forman-Kay

1.8 Versioning

This project follows strictly Semantic Versioning 2.0 for version control.

All additions to the `master` branch are done by PR followed by its respective version increment. While in version `0`, minor and patch upgrades converge in the `patch` number while major upgrades are reflected in the `minor` number.

1.9 Changelog

1.9.1 v0.7.23 (2024-03-27)

- Fixes bug in `inter_chain_cc()` for `ldrs` so conformers generated with `-scm mcsce` also work

1.9.2 v0.7.22 (2024-03-22)

- Fixes issue where clash-checking IDR combinations was not being performed between different chains

1.9.3 v0.7.21 (2024-03-13)

- When a given sequence overlaps with sequence from folded template in `ldrs`, ignore building on that chain
- Fixes issue #269

1.9.4 v0.7.20 (2024-02-01)

- Add a note for users in the `README.rst` regarding conformers generated with `ldrs` prior to version 0.7.17

1.9.5 v0.7.19 (2024-01-29)

- Update Sphinx requirement to v5 to fix `docs` workflow and `tox` error

1.9.6 v0.7.18 (2024-01-25)

- Correct case where multiple chains are given in a template and only some need `ldrs`
- Update documentation to clarify sequence formatting example for complexes

1.9.7 v0.7.17 (2024-01-20)

- Fix bug where torsion angles from `ldrs` would be swapped sometimes due to mirroring in rotation matrix

1.9.8 v0.7.16 (2024-01-11)

- Update citation for `ldrs` module to Bioinformatics paper

1.9.9 v0.7.15 (2024-01-05)

- Bug-fix for `ldrs` file processing during `psurgeon` stage for single IDR cases

1.9.10 v0.7.14 (2024-01-04)

- Correct reference hyperlink in usage documentation

1.9.11 v0.7.13 (2024-01-03)

- Update documentation for clarity
- Update example folder with processing AlphaFold structures

1.9.12 v0.7.12 (2023-11-17)

- Update documentation for DSSP due to clarity with version requirements
- Set default of `-cmd` in `sscalc` to `mkdssp`

1.9.13 v0.7.11 (2023-11-02)

- Update citing information.

1.9.14 v0.7.10 (2023-10-27)

- Correct reading CIF files without # end block.
- Fix minor bugs for reading CIF templates in `ldrs`

1.9.15 v0.7.9 (2023-10-27)

- Update *libstructure.Structure* documentation.

1.9.16 v0.7.8 (2023-10-02)

- Bug-fix in `ldrs` for input sequence recognition

1.9.17 v0.7.7 (2023-09-26)

- Bug-fix MC-SCE integration with force-field topology generation

1.9.18 v0.7.6 (2023-09-25)

- Update documentation for installation and usage

1.9.19 v0.7.5 (2023-09-11)

- Fix tests for `cli_build.py` that arose from #245

1.9.20 v0.7.4 (2023-09-11)

- Add `--long` and `--long-ranges` to the `build` subclient to build long (300+ AA) IDPs much quicker
- Change default backbone energy `-etbb` in `build` to 100.0
- Change default sidechain energy `-etss` in `build` to 250.0

1.9.21 v0.7.3 (2023-09-11)

- Add select ionic radii from CRC Handbook of Chemistry and Physics, 82nd Ed
- Update `merge` module to use multiprocessing
- Increase efficiency of clash-checking algorithm in `ldrs_helper.py`
- Acceptance and automated processing of proteins with a membrane/bilayer
- Automated clash-checking between all combinations of IDRs built
- Added ability for multi-chain protein detection and processing

- Added ability to build IDRs on multi-chain complexes
- Change default backbone energy `-etbb` in `ldrs` to 100.0
- Change default sidechain energy `-etss` in `ldrs` to 250.0

1.9.22 v0.7.2 (2023-07-27)

- Correct documentation formatting reference to MC-SCE

1.9.23 v0.7.1 (2023-07-27)

- Update documentation formatting for LDRS
- Update README citations
- Temporarily removed MacOS GitHub actions tests

1.9.24 v0.7.0 (2023-07-25)

- Added new module `ldrs` to build IDRs in the context of a folded region
- Added new module `resre` to rename certain residue names of multiple PDB files
- Minor bug-fixes to numpy references (#231)

1.9.25 v0.6.17 (2023-06-12)

- Bug-fix for installation and tox build/pr

1.9.26 v0.6.16 (2022-11-08)

- Update citation to J Phys Chem A reference

1.9.27 v0.6.15 (2022-09-26)

- Added new subclient `bgeodb` to append to the database exact bgeos
- Implement `exact` bgeo-strategy
- Add documentation regarding new features with `exact` and `bgeodb`
- PR #220

1.9.28 v0.6.14 (2022-07-13)

- Add tests to `cli_build.main` (#225)

1.9.29 v0.6.13 (2022-07-13)

- Correct how some loops are written due to redundancies (#226)

1.9.30 v0.6.12 (2022-07-07)

- lint files

1.9.31 v0.6.11 (2022-06-14)

- Add option to build backbone conformers with fixed bond angle and distances. (#217)

1.9.32 v0.6.10 (2022-06-14)

- Implement boxplot plotting feature for bond angle distributions
- Append documentation for `bgeo` subclient
- Ability to use `degrees` for `bgeo` subclient
- Ability to change the name of the ouput file for `bgeo` subclient
- PR #219

1.9.33 v0.6.9 (2022-06-13)

- Correct CLI help for the `--mem` flag in the `sethpc` subclient (#218)

1.9.34 v0.6.8 (2022-06-13)

- Add bioRxiv citation

1.9.35 v0.6.7 (2022-05-31)

- update actions from `master` to `main`
- update install instructions for MC-SCE in GRAHAM (#215)

1.9.36 v0.6.6 (2022-05-27)

- Add `pr` env to `tox`
- Update CI workflows
- Update ReadTheDocs python version to 3.8
- Dropped python 3.7 after Numpy

1.9.37 v0.6.5 (2022-05-25)

- Correct typo bugs in `fastext` and `bgeo`
- General lints

1.9.38 v0.6.4 (2022-05-25)

- Re-licensed to Apache-2.0

1.9.39 v0.6.3 (2022-05-25)

1.9.40 v0.6.2 (2022-05-25)

- Update usage instructions for *bgeo*

1.9.41 v0.6.1 (2022-05-25)

- updated GRAHAM install instruction (#207)

1.9.42 v0.6.0 (2022-05-24)

- Add bond geometry option to build with *Int2Cart* software
- PR #203

1.9.43 v0.5.1 (2022-05-24)

- add plot functions to `sscalc` and `torsions`
- PR #198

1.9.44 v0.5.0 (2022-05-24)

- Add residue tolerance matrices: EDSS50
- Update/improve parameters to residue tolerance options
- PR #183

1.9.45 v0.4.10 (2022-05-23)

- Add documentation RTD format
- Add documentation for several features and examples
- PR #171

1.9.46 v0.4.9 (2022-05-23)

- Add `sethpc` client.
- Add `merge` client.
- PR #202

1.9.47 v0.4.8 (2022-05-23)

- Add `stats` client
- Add `search` client
- PR #200

1.9.48 v0.4.7 (2022-05-23)

- update CI methods
- PR #205

1.9.49 v0.4.6 (2022-04-22)

1.9.50 v0.4.5 (2022-04-21)

1.9.51 v0.4.4 (2022-03-29)

- Fixes MC-SCE integration when sidechain packing fails
- Corrects MC-SCE installation
- #190

1.9.52 v0.4.3 (2022-03-26)

1.9.53 v0.4.2 (2022-03-20)

1.9.54 v0.4.1 (2022-03-17)

- Adds support for single residues when not specified. Addresses #184

1.9.55 v0.4.0 (2022-03-15)

- Integrates the MC-SCE protocol in the building process as part of the sidechain packing method options.

1.9.56 v0.3.3 (2022-03-14)

- removes assert in 0.3.2

1.9.57 v0.3.2 (2022-03-14)

- improves regex creation to avoid silent bugs in possible parallel futures

1.9.58 v0.3.1 (2022-03-13)

- incorporates *G* in *H* when treating DSSP with reduced labels

1.9.59 v0.3.0 (2022-03-13)

- see #168
- Revisited the whole regex sampling machinery during conformer building
- A initial major part for preparing the regex database was dropped
- applied multiprocessing to the regex database preparation steps
- updated the *cli_build* API with 4 new command options
- dropped using regex in the *cli_build* command line

1.9.60 v0.2.6 (2022-03-13)

- corrected *sscalc* from * input in command-line #175

1.9.61 v0.2.5 (2022-03-11)

- Implemented capacity to read PDBs with names different from cull #167

1.9.62 v0.2.4 (2022-03-11)

- implemented support for N-terminal Proline residues #166

1.9.63 v0.2.3 (2022-03-08)

- corrected energy.log #162

1.9.64 v0.2.2 (2022-03-07)

- incorporated *libfuncpy* internally

1.9.65 v0.2.1 (2022-03-03)

1.9.66 v0.2.0 (2022-02-10)

1.9.67 v0.1.0 (2021-07-24)

- Implements energy calculation to individual pairs. Energy threshold can now be compared to *pairs* or *whole*.

1.9.68 v0.0.24 (2021-07-01)

- Corrects *make_folder* function in *cli_build*.

1.9.69 v0.0.23 (2021-07-01)

- Added libfuncpy to requirements.yml

1.9.70 v0.0.22 (2021-06-30)

- **Users are now able to fully configure the size of fragments and probabilities,**
  via the flag *-xp* that expects a two column file.

1.9.71 v0.0.21 (2021-06-28)

- Now build prints log to terminal.
- improved other minor logging issues

1.9.72 v0.0.20 (2021-06-21)

- **Decoupled `energy-threshold` parameters. Now Backbone and sidechains,**
  can be configured separately.

1.9.73 v0.0.19 (2021-06-14)

- Saves a table with energy values per conformer.
- Crash reports now saved in execution folder (CLI build).

1.9.74 v0.0.18 (2021-06-10)

- **Improves sampling of multiple secondary structure regexes.**
  Now, when given multiple regex, angle sampling will be biased towards the number of occurrences in each regex.

1.9.75 v0.0.17 (2021-06-10)

- Corrects bug in Coulomb formula

1.9.76 v0.0.16 (2021-06-09)

- Add output-folder option for the `build` interface

1.9.77 v0.0.15 (2021-06-09)

- corrected typo in example/ commands

1.9.78 v0.0.14 (2021-06-05)

- Users can now select single residue fragment size
- `-xp` parameter was updated with checks and completion

1.9.79 v0.0.13 (2021-05-28)

- Added usage example and documentation.

1.9.80 v0.0.12 (2021-05-28)

- Corrects path suffix evaluation in `cli_torsions.py`

1.9.81 v0.0.11 (2021-05-28)

- corrects var name bug in ProgressBar

1.9.82 v0.0.10 (2021-05-27)

- Implements residue substitution/tolerance during conformer build

1.9.83 v0.0.9 (2021-05-27)

- user can now define the fragment size selection probabilities

1.9.84 v0.0.8 (2021-05-09)

- Expands try:catch to avoid index error when restarting conformer

1.9.85 v0.0.7 (2021-05-09)

- saves version number to file before running a client

1.9.86 v0.0.6 (2021-04-20)

- additional functions for logging
- add logging to build and other parts

1.9.87 v0.0.5 (2021-04-19)

- added `--energy-threshold` flag to control energy threshold after sidechain addition

1.9.88 v0.0.4 (2021-04-19)

- `builder` CLI now accepts `.fasta` files.

1.9.89 v0.0.3 (2021-04-19)

- added matplotlib in requirements.yml as dependency

1.9.90 v0.0.2 (2021-04-03)

- **corrects variable name in `libbuild` that was breaking sidechain**
  construction.

1.9.91 v0.0.1 (2021-04-02)

- added CI integration files

1.9.92 v0.0.0

- Any development previous to version 0.0.1 is registered in PRs up to #102.

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

i

INDEX

A

B

C

D

E

F

G

H

I

K

L

M

U

V

W

X

Y

Z