GaudiMM, for Genetic Algorithms with Unrestricted Descriptors for Intuitive Molecular Modeling, helps to sketch new molecular designs that require complex interactions.
How to install

Quick steps:

1 - Download the latest stable copy of UCSF Chimera and install it with:

```
chmod +x chimera-*.bin && sudo ./chimera-*.bin
```

2 - Install Miniconda Python 2.7 Distribution for your platform and install it with:

```
bash Miniconda2*.sh
```

3 - Install gaudi with conda in a new environment called insilichem (or whatever name you prefer after the -n flag), using these custom channels (-c flags):

```
conda create -n insilichem -c omnia -c salilab -c insilichem -c conda-forge -c bioconda -c tpeulen gaudi
```

4 - Activate the new environment as proposed:

```
conda activate insilichem
```

or

```
source activate insilichem
```

5 - Run it!

```
gaudi
```

### 1.1 Check everything is OK

If everything went OK, you will get the usage screen:
Usage: gaudi [OPTIONS] COMMAND [ARGS]...

GaudiMM: Genetic Algorithms with Unrestricted Descriptors for Intuitive Molecular Modeling

(C) 2017, InsiliChem
https://github.com/insilichem/gaudi

Options:
--version    Show the version and exit.
-h, --help    Show this message and exit.

Commands:
prepare    Create or edit a GAUDI input file.
run         Launch a GAUDI input file.
view        Analyze the results in a GAUDI output file.
Running GAUDI jobs is quite easy with `gaudi.cli.gaudi_run`:

```
 gaudi run /path/to/some_file.gaudi-input
```

To learn how to create input files, go to Input files, and make sure to check the tutorials!

- GaudiMM basics (start here!)
- Tutorial: Your first GaudiMM calculation

After the job is completed, you can check the results with `gaudi.cli.gaudi_view`:

```
 gaudi view /path/to/some_file.gaudi-output
```

You can choose between two molecular visualization tools: Chimera itself (using GaudiView extension), or our in-house GUI, GAUDInspect.
Both tools support lazy-load, filtering and sorting, so choose whichever you prefer. A quick tutorial on GaudiView is available here:

- **Tutorial: Visualizing your first results**
GaudiMM uses YAML-formatted files for both input and output files. YAML is a human-readable serialization format, already implemented in a broad range of languages. Input files must contain these five sections:

- **output**: Project options. Configure it to your liking
- **ga**: Genetic algorithm configuration. Normally, you don’t have to touch this, except maybe the number of generations and population size.
- **similarity**: The similarity function to compare potentially redundant solutions.
- **genes**: List of descriptors used to define an individual
- **objectives**: The list of functions that will evaluate your individuals.

You can check some sample input files in the examples directory.

### 3.1 How to create GaudiMM input files

If you don’t mind installing GAUDInspect, it provides a full GUI to create GAUDI input files, step by step. Add genes and objectives, configure the paths, number of generations and population size, and run it. Simple and easy.

**Note:** The development of GAUDInspect is currently stalled.

However, you can also edit them manually, since they are just plain text files. Create a copy of one of the examples and edit them to your convenience. While you can check the API documentation of `gaudi.genes` and `gaudi.objectives` to find a list of available components, we really recommend checking the beginners tutorial:

- **Tutorial: Your first GaudiMM calculation.**
GAUDI output files are also YAML-formatted, so you can just open the file and read it in your favourite editor. However, that’s not probably what you were expecting to do.

For the time being, your best bet to understand how to analyze the results is to read the visualization tutorial:

- *Tutorial: Visualizing your first results*

**Todo:**

- Multi-objective solutions: Pareto front vs lexicographically sorted elite
CHAPTER 5

FAQ & Known issues

5.1 Known issues

Most problems we have faced have to do with the installation and Chimera-Conda incompatibilities. Those are covered in the pychimera documentation, so please check that list before raising an issue in this repository!

5.2 How can I cite GaudiMM? What licenses apply?

GaudiMM is scientific software, funded by public research grants. If you make use of GaudiMM in scientific publications, please cite it. It will help measure the impact of our research and future funding!

\begin{verbatim}
@preamble{ " \newcommand{noop}[1]{} " }
\@article{GaudiMM2017,
  title = {GaudiMM: A Modular Multi-Objective Platform for Molecular Modeling},
  author = {Rodríguez-Guerra Pedregal, Jaime and Sciortino, Giuseppe and Guasp, Jordi and Municoy, Martí and Marchal, Jean-Didier},
  year = {\noop{2017}submitted},
}
\end{verbatim}

GaudiMM itself is licensed under Apache License 2.0, but includes work from other developers, whose licenses apply. Please check the LICENSE file in the root directory for further details.

5.3 How many generations / Which population size should I pick?

This depends on the complexity of your search space (related to the number and type of genes in use), and the evaluation power of the chosen objectives. A simple job would work OK with 50 generations and populations of 100 individuals, while more complex ones would require 500 generations for populations of 1000 individuals. This also depends on the values of mu and lambda parameters.
5.4 The output produces a lot of similar results and I want to focus on diversity

You should play with the cutoff value of the RMSD similarity section. If the RMSD of two potentially similar solutions is under the cutoff, one of them is discarded. However, this is only applied if the score of two solutions are the same; ie, if there is a draw.

To force draws, one can reduce the number of decimal positions returned by every objective, which is controlled by the precision parameter. It is setup globally in the output section, but can also be overridden by any objective. By default, precision is 3. Also, if you are using the `gaudi.genes.search.Search` gene, make sure its precision parameter is small enough so you don’t lose exploration efforts in too similar positions.

5.5 The output produces very different results and I am interested in clustered solutions

In this case, one should increase the precision value (both globally and in the `gaudi.genes.search.Search` gene, if it applies) and reduce the RMSD similarity cutoff.

5.6 What is the difference between atom_names and atom_types in some objectives or genes?

`atom_names` refers to the name attribute of `chimera.Atom` objects, while `atom_types` is applied with idatmType attributes. Normally, the name attribute is picked directly from the molecule file (PDB, mol2), while the idatmType is assigned algorithmically by UCSF Chimera.

Tip: Any further questions? Feel free to submit your inquiries to our issues page!
The main strength of GaudiMM is, possibly, its flexible approach to solve molecular modeling problems. However, flexibility does not come without a price: the learning curve can be somewhat steep for beginners. In this tutorial we will try to smooth that curve out.

GaudiMM makes a big effort to clearly separate exploration from evaluation. As such, both stages are configured in different sections of the input file: genes and objectives.

### 6.1 Exploration and genes

Most molecular modeling problems can be explained using search space analogies. This is, given a problem, one can identify the dimensions it must explore to find adequate solutions for that problem. That n-dimensional space can be comprised of one or more (bio)chemical structures and their respective spatial variability. Modifying the topology and/or composition of such structures would be equivalent to moving along the (bio)chemical axes of such search space, while modifying their atomic positions would be equivalent to moving along the cartesian and/or internal coordinates axes. Thus, visiting different regions of the whole search space means creating new candidate solutions to the problem... We will call this stage exploration, and it is detailed in the genes section of the input file. If you want to know why they are called genes, check the Genetic Algorithms Primer in the Developers guide.

The more important gene is, almost always, the gaudi.genes.molecule.Molecule gene. This is used to load (bio)chemical structures, such as proteins, peptides and small cofactors. Normally they are straight PDB or Mol2 files, but can also be more complex constructs. As of now, it is the only one capable of providing topologies. The other genes are responsible of applying changes to the coordinates of that initial structure. Some examples include torsion angles of rotatable bonds (gaudi.genes.torsion.Torsion gene), translating and rotating a Molecule in the 3D space (gaudi.genes.search.Search gene), or importing the atomic positions of a molecular dynamics trajectory frame (gaudi.genes.trajectory.Trajectory gene).

### 6.2 Evaluation and objectives

However, there is no guarantee that, given a new position on such search space, that hypothetical solution is actually good enough to solve the problem. To assess the quality of a random solution, it must be evaluated with some criteria
or objectives. All objectives have something in common: they take a candidate solution (ie, a point of the search space) and measure some property to return a numeric score. Some examples include the distance between two given atoms (gaudi.objectives.distance.Distance objective), the steric clashes of the molecules (gaudi.objectives.contacts.Contacts objective) or the forcefield energy of the system (gaudi.objectives.energy.Energy objective).

6.3 Examples of application

Now we got the basics covered, we can start to think of how we could solve some standard (and not so standard) molecular modeling problems with GaudiMM.

6.3.1 Docking

Docking problems devote to finding the correct orientation and position of a small molecule (the ligand) within the cavity of a bigger one (normally, a protein). For the exploration stage, at least three genes must be defined:

- A gaudi.genes.molecule.Molecule gene to load the file corresponding to the protein structure.
- Another gaudi.genes.molecule.Molecule gene to load the ligand itself.
- A gaudi.genes.search.Search gene to translate and rotate the ligand around the protein surroundings.

This is all you need to move a rigid ligand around a protein. However, most of the time you want to implement some kind of internal flexibility, such as torsions in the ligand or some residues sidechains. To do that, add these to your genes section:

- A gaudi.genes.torsion.Torsion gene with the ligand as target to implement dihedral angle torsions along the rotatable bonds of the ligand.
- A gaudi.genes.rotamers.Rotamers gene with some residues of the protein as target. This will randomly modify the dihedral torsions of the sidechains of the specified residues according to the angles provided by the rotamer library (defaults to Dunbrack’s).

There is also a gaudi.genes.mutamers.Mutamers gene that allows you to mutate residues in addition to rotameric exploration. This will provide some traversal along the biochemical axis of your protein, but careful, because the current implementation is RAM hungry!

The evaluation stage can be comprised of several objectives, but normally you’d want to:

- Minimize steric clashes with gaudi.objectives.contacts.Contacts.
- Maximize hydrophobic patches with another gaudi.objectives.contacts.Contacts entry.
- Use a proper docking scoring function, such as gaudi.objectives.ligscore.LigScore or gaudi.objectives.dsx.DSX. Depending on the one you choose, it should be minimized (the usual case) or maximized.

6.3.2 Covalent docking

Not all docking studies consider a free ligand. Sometimes the ligand is anchored to some part of the protein. While there’s no specific gene to implement a covalent bond (for now, at least), you can mimic it with a Search gene with radius=0 and rotate=True. The origin of this null search sphere should be set to the atom that is part of the covalent bond, so you would have add an extra atom in one of the molecules. Also, we recommend setting an gaudi.objectives.angle.Angle objective between the involved atoms in the covalent bond so that the resulting rotation matches the expected geometry of the new bond (109.5° for sp3, 120° for sp2, 180° for sp1).
6.3.3 Naked metal ions docking

If instead a small organic molecule you choose to load a naked metal ion... would it work? Well, in most docking programs probably not, but with GaudiMM the answer is... it depends!

It depends on the objectives you choose. Since we ship a `gaudi.objectives.coordination.Coordination` objective that can deal with coordination geometries, you can dock naked metal ions in any protein or peptide.

6.3.4 Competitive docking

Since you can instantiate as many `gaudi.genes.molecule.Molecule` genes as you want, nothing prevents you from adding more ligand molecules at the same time. They will compete to find its place in the protein(s). Just remember to replicate any dependent genes (`gaudi.genes.search.Search`, `gaudi.genes.torsion.Torsion` and so on) accordingly. Also, why not two proteins which the ligand should choose from (but it’s true there are finer ways to assess that)?

6.3.5 Hacking Molecule genes for complex studies

In addition to loading molecular structure files, the `gaudi.genes.molecule.Molecule` does a couple of extra things. The path parameter can be set to two different values:

- The path to a PDB or Mol2 file (or any format that Chimera can open). This is the standard behaviour. It will load the structure and that’s it.
- The path to a directory, whose contents determine the final behaviour:
  - A) If the directory contains molecule files, GaudiMM will choose one of them randomly for each case.
  - B) If the directory contains subdirectories which, in turn, contain molecules files, GaudiMM will sort those subdirectories by name and then pick one molecule from each, in that order. The chosen molecules will be chained linearly as specified in the accompanying *.attr files. Perfect for drug design, since you can fill a couple of directories with different alkanes to link a cofactor

Case A allows GaudiMM to test a library of compounds against certain criteria: ie, virtual screening!

Case B makes drug design studies possible. If you want to test different linker molecules to anchor a cofactor to a protein, you can use a two-directories scheme: one directory would contain the aforementioned linkers, and the other one just the cofactor (it’s fine if there’s only one molecule in one directory).

6.4 Peptide folding & Conformational analysis

Loading an unfolded peptide with `gaudi.genes.molecule.Molecule` is pretty easy. Just specify the path. Then, if you apply a `gaudi.genes.torsion.Torsion` gene on such peptide, but considering only the bonds involving an alpha-carbon, you can effectively explore the conformational space of its folding. Use `gaudi.genes.rotamers.Rotamers` to implement sidechain flexibility.

For the evaluation, you can use the `gaudi.objectives.energy.Energy` objective to assess the forcefield energy as provided by OpenMM GPU calculations. This type of study could be considered some sort of highly explosive metadynamics.
6.4.1 Homology modeling

In the same fashion, given an unfolded peptide or protein segment, you could apply the same Torsion scheme to explore different conformations. A hypothetical objective could be devised to calculate the RMS deviation of the new fragment and a reference one, which then the GA would minimize to obtain a similar structure.

6.4.2 Conformational analysis

Another possible variation of this scheme is to impose geometric guides as additional objectives, like `gaudi.objectives.distance.Distance`, `gaudi.objectives.angle.Angle` or `gaudi.objectives.coordination.Coordination`. The resulting calculation could be regarded as a restrained conformational analysis, very useful for finding initial structures of unparametrized small molecules you want to study with higher levels of theory, such as QM.

6.4.3 Trajectory analysis

GaudiMM features a `gaudi.genes.trajectory.Trajectory` gene capable of importing frames of MD movies and apply them to the corresponding `gaudi.genes.molecule.Molecule` instance. This way, all the objectives are available as trajectory analysis tools, maybe not present in other specific software. For example, finding coordination geometries of a given metal ion along a molecular mechanics simulation.
CHAPTER 7

Tutorial: Your first GaudiMM calculation

Running a GaudiMM calculation is easy. All you need is a simple command:

```
gaudi run input_file.yaml
```

The question is... how do I create that `input_file.yaml`.

### 7.1 How to create an input file from scratch

Input files in GaudiMM are formatted with YAML, which follows readable conventions that are still parseable by computers. To easily edit YAML files, we recommend using a text editor that supports syntax highlighting, such as Sublime Text 3, Atom or Visual Studio Code. If you use `.yaml` as the extension of the input filename, the editor will colorize the text automatically. Else, you can always configure it manually. Check the docs of your editor to do so. If you don’t want syntax highlighting it’s fine, GaudiMM will work the same.

Input files must contain the sections mentioned above: output, ga, similarity, genes, and objectives. Each of this sections it’s defined with a colon and a newline, and its contents will be inside an indented block. For readability, I usually insert a blank line between sections.

```yaml
output:
    contents of output

ga:
    contents of ga

similarity:
    contents of similarity

genes:
    contents of genes

objectives:
    contents of objectives
```
7.1.1 The output section

This section governs how the results and reports will be created. Everything is optional, since each key has a default value, but it is preferable to at least specify the name of the job (otherwise, it will be set to five random characters), and the path where the result files will be written. Like this:

```yaml
output:
  name: some_example
  path: results
```

Note: All the relative paths in GaudiMM are relative to the location of the input file, not the working directory. To ease this difference, we recommend running the jobs from the same folder where the input file is located. Of course, you can always use absolute paths.

7.1.2 The ga section

This section hosts the parameters of the Genetic Algorithm GaudiMM uses. Unless you know what you are doing, the only values you should modify are population and generations. To see the appropriate values, refer to FAQ & Known issues. For example:

```yaml
ga:
  population: 200
  generations: 100
```

7.1.3 The similarity section

This section contains the parameters to the similarity operator, which, given two individuals with the same fitness, whether they can be considered the same solution or not. This section is deliberatedly loose: you define the Python function to call, together with its positional and keyword arguments.

For the time being, the only similarity function we ship is based on the RMSD of the two structures: `gaudi.similarity.rmsd()`. The arguments are which Molecule genes should be compared and the RMSD threshold to consider whether they are equivalent or not.

```yaml
similarity:
  module: gaudi.similarity.rmsd
  args: [[Ligand], 1.0]
  kwargs: {}
```
7.1.4 The genes section

This section describes the components of the exploration stage of the algorithm; ie, the features of each Individual in the population. While the previous sections were dictionaries (this is, a collection key-value pairs), the genes and objectives section is actually a list of dictionaries. As a result, you need to specify them like this:

```
genes:
  - name: Protein
    module: gaudi.genes.molecule
    path: /path/to/protein.mol2
  - name: Torsion
    module: gaudi.genes.torsion
    target: Ligand
    flexibility: 360
```

Notice the dash – next to name. This, and the extra indentation, define a list. Each element of this list is a new gene. Each gene must include two compulsory values:

- name. A unique identifier for this gene. If you add two genes with the same name, GaudiMM will complain.
- module. The Python import path to the module that contains the gene. All GaudiMM builtin genes are located at `gaudi.genes`.

All other parameters are determined by the chosen gene. Check the corresponding documentation for each one!

Note:

**How do I know which genes to use?**

Unless you code a gene of your own to replace it, you will always need one or more `gaudi.genes.molecule.Molecule` genes. Then, choose the flexibility models you want to implement on top of such molecule. Several examples are provided in *GaudiMM basics (start here!)*.

7.1.5 The objectives section

Like the genes section, the objectives section is also a list of dictionaries, so they follow the same syntax:

```
- name: Clashes
  module: gaudi.objectives.contacts
  which: clashes
  weight: -1.0
  probes: [Ligand]
  radius: 5.0
- name: LigScore
  module: gaudi.objectives.ligscore
  weight: -1.0
  proteins: [Protein]
  ligands: [Ligand]
  method: pose
```

In addition to the required name and module parameters, each objective needs a weight parameter. If set to 1.0, the algorithm will maximize the score returned by the objective; if set to -1.0, it will be minimized. Theoretically, any other positive or negative float will work, but stick to the convention of using 1.0 or -1.0.
Any other parameters present in an objective are responsibility of that objective, and are specified in its corresponding documentation.

That’s it! Now save it with a memorable filename and run it!

## 7.2 How to run your input file

Let’s get back to the beginning of the tutorial: all you need to do is typing:

```bash
gaudi run input_file.yaml
```

If everything is fine, you’ll see the following output in the console:

```bash
$> gaudi run input_file.yaml

Loaded input input_file.yaml
Launching job with...
  Genes: Protein, Ligand, Rotamers, Torsion, Search
  Objectives: Clashes, Contacts, HBonds, LigScore

After the first iteration is complete, the realtime report data will kick in:

<table>
<thead>
<tr>
<th>gen</th>
<th>progress</th>
<th>nevals</th>
<th>speed</th>
<th>eta</th>
<th>avg</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>4.76%</td>
<td>20</td>
<td>1.25</td>
<td>0:16:34</td>
<td>[ 3.080e+03 -2.690e+02 7.500e-01 1.179e+03]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>9.52%</td>
<td>60</td>
<td>1.28</td>
<td>0:16:32</td>
<td>[ 2.787e+03 -2.659e+02 1.400e+02 4.758e+02]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The first time you see this it might result too confusing, especially if the terminal wraps long lines. Let’s describe each tab-separated column:

- **gen.** The current generation.
- **progress.** Percentage of completion of the job. This is estimated with the expected number of operations: 
  \[(generations + 1) \times \lambda \times (cxpb + mutpb)\]
• **nevals.** Number of evaluations performed in current generation.

• **speed.** Estimated number of evaluations per second. This does not take into account the time spent in the variation stage.

• **eta.** Estimated time left.

• **avg.** Average of all the fitness values reported by each objective in the current generation. They are listed in the order given in the input file, and also reflected above, after the *Launching job with...* line.

• **std.** Same as avg, but for the standard deviation.

• **max.** The maximum fitness value reported by each objective in the current generation.

• **min.** Same as above, but for the minimum value.

If the setting *check_every* in the output section is greater than zero, GaudiMM will dump the current population every *check_every* generations. That way, you can assess the progress visually along the simulation.

Also, if you feel that the algorithm has progressed enough to satisfy your needs, you can cancel it prematurely with Ctrl+C. GaudiMM will detect the interruption and offer to dump the current state of the simulation:

```
^C[]
```

Interruption detected. Write results so far? (y/N):

Answer *y* and wait a couple of seconds while GaudiMM writes the results. To analyze them, check the following tutorial:

• *Tutorial: Visualizing your first results*
CHAPTER 8

Tutorial: Visualizing your first results

When a GaudiMM job finishes correctly, you will find the results in the directory specified at output.path. There you will find:

- *.gaudi-input. An (updated) copy of the input file used in this job, for reproducibility purposes.
- *.gaudi-log. A detailed log of the simulation, for debugging purposes.
- *.gaudi-output. A YAML file with the scores and paths of the solutions.
- (Potentially lots of) *.zip. As many as the elite population size. Each zip contains an Individual, which means that you will find molecule files (normally, mol2 files), as well as the allele metadata of the rest of the genes.

To analyze the results, currently the best tool is our own extension for UCSF Chimera, GaudiView. After installing it, you can type the following:

```
 gaudi view *.gaudi-output
```

, and hopefully it will open Chimera and GaudiView to load the output file. If it doesn’t work (please, if that’s the case, report it in the issues page!), you can always UCSF Chimera, fire up GaudiView from Tools> InsiliChem> GAUDIView and browse the file manually.
One way or another, you will end up seeing the following dialog:
8.1 Actions

The following actions are available:

- **Depiction**
  - Click on a row to show that solution as is. Double-click may trigger additional post-processing as details in the genes metadata, depending on the chosen genes.
  - Ctrl-click or Shift-click to select more than one solution at the same time.
  - Use up and down arrows to move along the list.

- **Sorting**
  - Click on the column headers to sort by that key.

- **Filtering**
  - Press +Add to load a new filter. Select the filter key, specify the cutoff and press Filter. Repeat to add more filters with the selected boolean.
  - Remove filters by pressing on -.
  - Clear all the filters with Reset.

---

8.1. Actions
• **Commands**
  
  – Every time you change the selected row, the command specified in the *command input* panel will be run. You can force it by pressing **Enter** key or click on **Run**.
  
  – Combine the command line with the dynamic selection panel to run commands like `show sel zr < 5` to depict the surrounding residues of docking poses.
  
  – Multiple commands can be chained with `;`.

• **Clustering**
  
  – First, choose the column to use as *energy* criteria.
  
  – Select the RMSD cutoff. If two solutions have RMSD bigger than this value, they won’t be considered similar and thus not clustered together.
  
  – Press **Cluster**!. Since this method requires to load all solutions at once, it can take some time if you have a lot of solutions.
  
  – When it’s done, a new column **Cluster** will be available at the end of the table. Sort by that column to group them together and **Shift-click** to depict the full cluster.

When you are done with the analysis, you can exit GaudiView in different ways:

• **Click** **OK**. The depicted solutions at that moment won’t be closed, and thus available for further post-processing in Chimera. Every other solution previously loaded will be removed from the canvas and closed.

• **Click** **Close** or upper-corner **X**. All solutions, visible or not, will be removed from canvas and closed.

**Error:** There’s a known bug in GaudiView that may trigger a non-critical exception if you filter the solutions and then try to sort them again without selecting a new row beforehand. You can dismiss the dialog and simply click in any row to avoid this bug. It will be fixed soon, promise!
9.1 Introduction to Genetic Algorithms

The GA in GaudiMM stands for Genetic Algorithm, a search heuristic inspired by natural selection that is used for optimization processes.

Genetic Algorithms use a biologist's terminology. Each candidate solution to the problem is considered an **individual**, which is part of the so-called **population** (the set of all candidate solutions).

The initial population is generated from scratch, almost always randomly. These individuals also comprise the first generation of the evolution process. As in nature, only the **fittest** survive. The survival process is simulated with an evaluation function, that tests them against the optimization variable(s). This is called **selection**.

- How do we create an individual? With one or more **genes**, naturally. Genes describe how an individual should look like, of course!
- How do we evaluate that individual? With one or more **objectives**.

Also, as in nature, the fittest individuals are allowed to **mate** (exchange their defining values), and **mutate** (spontaneously modify their own defining values). This adds some more variability to the process, and that is key to survival.

After a number of generations repeating the process of selection, choosing the fittest over the weakest, we will obtain better and better solutions to our problem. Since it’s all heuristics, it’s up to us to stop at some point. We won’t probably get the solution, but we can live with pretty good ones, right?

9.1.1 The One Max Problem

Ok, that was a lot of biology and we are trying to code, I get it. Let’s explain a classical GA example, the trivial **One Max Problem**, adapted from the original deap documentation.

In this problem, we have a list of integers that can be either 0 or 1, and we want to obtain a list full of 1s. So, in this example, we have individuals defined by a single **gene** and evaluated with a single **objective**.

We build individuals with the gene **onemax**:
import random.randint

def onemax(size=5):
    return [random.randint(0, 1) for i in range(size)]

adam = onemax(5)  # returns [0, 0, 0, 1, 0]
eve = onemax(5)  # returns [0, 1, 1, 0, 0]

The objective is also trivial. We have to maximize the sum of the numbers inside a given individual:

def evaluate(individual):
    return sum(individual)

So... which is one is fittest, adam or eve? Obviously, eve:

evaluate(adam)  # returns 1
evaluate(eve)  # returns 2

Of course, an initial population is usually larger! At least, a hundred individuals. With such a trivial case, given a big enough population, we may obtain the solution in the first generation by pure change. However, we must not rely on the initial population as the only diversity source.

Additional diversity is achieved with the mutation and mating operations, implemented as additional functions:

def mate(a, b):
    """ Let a and b mate, in hope of fitter children """
    i = crossover_point = random.random() * min(len(a), len(b))
    c, d = a[:], b[:]
    c[i:], d[i:] = d[i:], c[i:]
    return c, d

def mutate(individual, probability):
    """ Spontaneous mutation at random places can result in a fitter individual """
    return [random.randint(0, 1) for i in individual if random.random() < probability]

Let's see how this is useful:

cain, abel = mate(adam, eve)
# cain = [ 0, 1, 1, 1, 0 ]
# abel = [ 0, 0, 0, 0, 0 ]
evaluate(cain)  # returns 3
evaluate(abel)  # returns 0

See? adam and eve gave birth to cain and abel. cain had luck and inherited the good parts, while abel... Well, he was not that lucky. In the next selection process, cain will be selected over abel, and probably over its own father adam. Now, the population (cain and eve) as a whole is fitter, with an average fitness of 2.5. That's higher than the average in the previous generation (1.5). Evolution!

Mutation works similarly:

enoch = mutate(cain)
# enoch = [ 1, 1, 1, 1, 0]
seth = mutate(eve)
# seth = [ 0, 0, 1, 0, 0]

Take into account that mutations can be beneficial, like in the case of enoch, but also detrimental, as in the case of seth. Some of them will contribute to evolution, and some of them not. Lucky ones will be selected, the others, discarded.
By the way, deap already defines some mutation and mating operators for you that will work in most cases. So, hopefully, this part will be trivial.

And that’s it! Deap does the rest! So, to sum up, you only need to worry about:

- How to define your individuals.
- How to evaluate them.
- How to implement mutation and mating (normally, with deap built-in operators).

If you want to know more about Deap and Genetic Algorithms, go check their documentation. It’s great!

### 9.2 Our implementation

GaudiMM is built as an extensible and highly modular Python platform. Although the main focus is Chemistry and molecular design, you can use your own genes and objectives. You can think of GaudiMM as a new API for deap that provides an object-oriented interface to easily create new individuals and objectives.

In deap an individual can be any Python object (check their overview and GA examples), which is a very versatile approach, but it tends to be very limited when your individual gets complex. For example, if an individual needs to be defined by several genes with different mutation strategies.

In GaudiMM, each individual is a `gaudi.base.Individual`, which is a very (bio)fancy name for a list of genes. To create a gene, you just subclass `gaudi.genes.GeneProvider` and define the needed methods: `express`, `unexpress`, `mutate`, and `mate`. The `gaudi.base.Individual` class then provides some wrapper methods that call the respective counterparts in each gene.

To evaluate the fitness of an individual, you must first define the set of evaluation functions. Each function is called `objective`, and you keep them inside a `gaudi.base.Environment`.

To create a new objective, you have to subclass `gaudi.objectives.ObjectiveProvider`, which provides a very simple interface: `evaluate`. Define your function there, and that’s it!

---

**Todo:**

- Tutorial: How to create your own gene
- Tutorial: How to create your own objective
The GaudiMM package is comprised of several core modules that establish the base architecture to build an extensible platform of molecular design.

The main module is `gaudi.base`, which defines the `gaudi.base.Individual`, whose instances represent the potential solutions to the proposed problem. Two plugin packages allow easy customization of how individuals are defined (`gaudi.genes`) and how they are evaluated (`gaudi.objectives`). Additionally:

- `gaudi.algorithms` is the place to look for the actual GA implementation
- `gaudi.box` is a placeholder for several small functions that are used across GaudiMM.
- `gaudi.exceptions` defines custom exceptions.
- `gaudi.parse` contains parsing utilities to retrieve the configuration files.
- `gaudi.parallel` contains helpers to deal with parallel GaudiMM jobs.
- `gaudi.plugin` holds some magic to make the plugin system work.
- `gaudi.similarity` defines the diversity enhancers.

### 10.1 gaudi.cli

This package holds all the CLI scripts and wrappers provided by GAUDI.

#### 10.1.1 gaudi.cli.gaudi_cli

`gaudi.cli.gaudi_cli` is the CLI entry point for all GaudiMM scripts.

Available commands:

- run
- view
- prepare
gaudi.cli.gaudi_cli.echo_banner()

```python
gaudi.cli.gaudi_cli.load_chimera(nogui=True)
```

```python
gaudi.cli.gaudi_cli.test_import(command, module)
```

```python
gaudi.cli.gaudi_cli.timeit(func, *args, **kwargs)
```

### 10.1.2 gaudi.cli.gaudi_run

`gaudi.cli.gaudi_run` is the main hub for launching GAUDI jobs.

It sets up the configuration environment needed by DEAP (responsible for the GA) and ties it up to the GAUDI custom classes that shape up the invididuals and objectives. All in a loosely-coupled approach based on Python modules called on-demand.

**Usage.** Simply, type:

```bash
gaudi run /path/to/job.gaudi-input
```

```python
gaudi.cli.gaudi_run.enable_logging(path=None, name=None, debug=False)
```

Register loggers and handlers for both stdout and file

```python
gaudi.cli.gaudi_run.launch(cfg)
```

Runs a GAUDI job

**Parameters**

- `cfg` ([`gaudi.parse.Settings`]) – Parsed YAML dict with attribute-like access

```python
gaudi.cli.gaudi_run.main(cfg, debug=False)
```

Starts a GAUDI job

**Parameters**

- `cfg` (`str` or `gaudi.parse.Settings`) – Path to YAML input file or an already parsed YAML file via `gaudi.parse.Settings` class
- `debug` (`bool`, optional, default=`False`) – Whether to enable verbose logging or not.

```python
gaudi.cli.gaudi_run.unbuffer_stdout()
```

### 10.2 gaudi.genes

These are the built-in genes in GAUDI. You can also build your own, but these are ready to use.

#### 10.2.1 Molecule gene

This gene implements a wrapper around Chimera.molecule objects to expand its original features, such as appending new molecules.

This allows to build new structures with a couple of building blocks as a starting point, as well as keeping several ligands as different potential solutions to the essay (think about multi-molecule alternative docking). The user can also request more `gaudi.genes.molecule` instances for the genome of the individual, resulting in a competitive multi-docking essay.

To handle all this diversity, each construction is cached the first time it is built.

This class is a dependency of most of the other genes (and even objectives), so it will be requested almost always.
class gaudi.genes.molecule.Compound(molecule=None, origin=None, seed=0.0, **kwargs)

Bases: object

Wraps chimera.Molecule instances and allows to perform copies, appending new fragments, free placement and extended attributes.

Parameters

- **molecule** (chimera.Molecule or str, optional) – The chimera.Molecule object to wrap, or
  - 'dummy' (for an empty molecule), or
  - The path to a mol2 file that will be parsed by Chimera to return a valid chimera.Molecule object.
- **origin** (3-tuple of float, optional) – Coordinates to the place where the molecule should be placed
- **seed** (float, optional) – Random seed, used by the vertex chooser on appending.
- **optional** (kwargs,) – Additional parameters that should be injected as attributes

mol

Type chimera.Molecule
donor

If self were to be bonded to another Compound, this atom would be the one involved in the bond. By default, first element in chimera.Molecule.atoms

Type chimera.Atom
acceptor

If another Compound wanted to be bonded to self, this atom would be the one involved in the bond. By default, last element in chimera.Molecule.atoms

Type chimera.Atom
origin

Default location of mol in the 3D canvas.

Type 3-tuple of float
seed

Randomness seed for vertex computation.

Type float
rotatable_bonds

Bonds that can be torsioned

Type list of chimera.Bond
nonrotatable

Bonds that, although possible, should not be torsioned. Ie, fixed bonds.

Type list of chimera.Bond
built_atoms

Memo of already built_atoms

Type list of chimera.Atom
**Notes**

**Todo:** Instead of a `molecule` parameter overload, think of using equivalent classmethods.

```python
add_dummy_atom(where, name='dum', element=None, residue=None, bonded_to=None, serial=None)
```

Adds a placeholder atom at the coordinates specified by `where`

**Parameters**

- `where` (*chimera.Atom or 3-tuple of float*) – Coordinates of target location. A chimera.Atom can be supplied, in which case its coordinates will be used (via `.coord()`)
- `name` (*str, optional*) – Name for the new atom
- `element` (*chimera.Element, optional*) – Element of the new atom
- `residue` (*chimera.Residue, optional*) – Residue that will incorporate the new atom
- `bonded_to` (*chimera.Atom, optional*) – Atom that will form a bond with new atom
- `serial` (*int*) – Serial number that will be assigned to atom

```python
add_hydrogens()
```

Add missing hydrogens to current molecule

```python
append(molecule)
```

Wrapper around `attach` to add a new molecule to `self`, using `self.acceptor` as bonding atom.

**Parameters**

- `molecule` (*Compound*) –

```python
apply_pdbfix(pH=7.0)
```

Run PDBFixer and replace original molecule with new one

```python
attach(molecule, acceptor, donor)
```

Call `join` to bond `molecule` to `self.mol` and updates attributes.

**Parameters**

- `molecule` (*Compound*) – The molecule that will be attached to `self`
- `acceptor` (*chimera.Atom*) – Atom of `self` that will participate in the new bond
- `donor` (*chimera.Atom*) – Atom of `molecule` that will participate in the new bond

**Notes**

**Note:** After joining the two molecules together, we have to update the attributes of `self` to match the new molecular reality. For example, the atom participating in the bond will not be available for new bonds (we are forcing linear joining for now), so the new `donor` will be inherited from `molecule`, before deleting the object.

```python
destroy()
```

Removes `mol` from canvas, deletes it and then, deletes itself
**join** (*molecule, acceptor, donor, newres=False*)

Take a molecule and bond it to **self.mol**, copying the atoms in the process so they’re contained in the same chimera.Molecule object.

**Parameters**

- **molecule** (*Compound*) – The molecule that will be attached to **self**
- **acceptor** (*chimera.Atom*) – Atom of **self** that will participate in the new bond
- **donor** (*chimera.Atom*) – Atom of **molecule** that will participate in the new bond
- **newres** (*bool*) – If True, don’t reuse acceptor.residue for the new molecule, and create a new blank one instead.

**Returns** **builtAtoms** – Maps original atoms in **molecules** to their new counterparts in **self.mol**.

**Return type** dict

**Notes**

**Note:** Chimera does not allow bonds between different chimera.Molecule objects, so firstly, we have to copy the atoms of **molecule** to **self.mol** and, only then, make the joining bond.

It traverses the atoms of **molecule** and adds a copy of each of them to **self.mol** using chimera.molEdit.addAtom in the same spot of space. All the bonds are preserved and, finally, bond the two molecules.

The algorithm starts by adding the bonding atom of **molecule** (**donor**), to the **sprouts** list. Then, the loop starts:

```python
while sprouts contains atoms:
    sprout = sprouts.pop(0)
    copy sprout to **self.mol**
    for each neighbor of sprout
        copy neighbor to **self.mol**
        if neighbor itself has more than one neighbor (ie, sprout)
            add neighbor to sprouts
```

Along the way, we have to take care of already processed sprouts, so we don’t repeat ourselves. That’s what the **builtAtoms** dict is for.

Also, instead of letting addAtom guess new serial numbers, we calculate them beforehand by computing the highest serial number in **self.mol** prior to the additions and then incrementing one by one on a per-element basis.

**Todo:** This code is UGLY. Find a better way!

**parse_attr()**

**place** (*where, anchor=None*)

Convenience wrapper around **translate**, supplying default

**Parameters**

- **where** (*3-tuple of float, or chimera.Atom*) – Coordinates of destination. If chimera.Atom, use supplied **coord()**
• anchor (chimera.Atom) – The atom that will guide the translation.

**Notes**

**Note:** Anchor atoms are called that way in the API because I picture them as the one we pick with our hands to drag the molecule to the desired place.

place_for_bonding (target, anchor=None)

Translate self.mol to a covalent distance of target atom, with an adequate orientation.

**Parameters**

• target (chimera.Atom) – The atom we are willing to bond later on.

• anchor (chimera.Atom, optional) –

prepend (molecule)

Wrapper around attach to add a new molecule to self, using self.donor as bonding atom.

**Parameters**

molecule (Compound) –

set_vdw_radii (vdw_radii)

update_attr (d)

class gaudi.genes.molecule.Molecule (path=None, symmetry=None, hydrogens=False, pdbfix=False, vdw_radii=None, **kwargs)

Bases: gaudi.genes.GeneProvider

Interface around the gaudi.genes.molecule.Compound to handle the GAUDI protocol and caching features.

**Parameters**

• path (str, optional) – Path to a molecule file or a directory containing dirs of molecule files.

• symmetry (str, optional) – If path is a directory, list of pairs of directories whose chosen molecule must be the same, thus enabling symmetry.

• hydrogens (bool, optional) – Add hydrogens to Molecule (True) or not (False).

• pdbfix (bool, optional) – Only for testing and debugging. Better run pdbfixer prior to GAUDI. Fix potential issues that may cause troubles with OpenMM forcefields.

• vdw_radii (dict {str: float} , optional) – Set a specific vdw_radius for a particular element (instead of standard Chimera VdW table). It can be useful in particular cases together with a contacts objective. Example of use in the .yaml file: vdw_radii: {
  Fe: 2.00, Cu: 2.16}

Defaults to None

allele

Paths to every fragment that composes a given Compound. It will consist of a single value tuple if there’s no dynamic building involved; ie, a mol2 or pdb file as is.

**Type** tuple of str

__CATALOG

Class attribute (shared among all Molecule instances) that holds all the possible molecules GAUDI can build given current path.
If path is a single molecule file, that’s the only possibility, but if it’s set to a directory, the engine can potentially build all the combinations of molecule blocks found in those subdirectories.

Normally, it is accessed via the catalog property.

Type dict

Notes

Use of ‘_cache’

Molecule class uses _cache to store already built molecules. Its entry in the _cache directory is a boltons.cacheutils.LRU cache (least recently used) set to a maximum size of 300 entries.

Todo: The LRU cache size should be proportional to job size, depending on the population size and number of generations, but also taking available memory into account (?).

SUPPORTED_FILETYPES = ('mol2', 'pdb')

build(key, where=None)

Builds a Compound following the recipe contained in key through Compound.append methods.

Parameters key (tuple of str) – Paths to the molecule blocks that comprise the final molecule. A single molecule is just a one-block recipe.

Returns The final molecule result of the sequential appending.

Return type Compound

classmethod clear_cache()
Todo: Allow mating while preserving symmetry

**mutate**(indpb)

VERY primitive. It only gets another compound.

**unexpress**()

Removes the Chimera molecule from the viewer canvas (without deleting the object).

**write**(paths=None, names=None, absolute=None, combined_with=None, filetype='mol2')

Writes full mol2 to disk.

Todo: It'd be preferable to get a string instead of a file

**xyz**(transformed=True)

gaudi.genes.molecule.enable(**kwargs)

### 10.2.2 Mutamers gene

This module allows to explore side chains flexibility in proteins, as well as mutation.

It needs that at least a `gaudi.genes.rotamers.molecule.Molecule` has been requested in the input file. Residues of those are referenced in the `residues` argument.

It also allows mutations in the selected residues. However, the resulting structure keeps the same backbone, which may not be representative of the in-vivo behaviour. Use with caution.

```python
class gaudi.genes.mutamers.Mutamers(residues=None, library='Dunbrack', avoid_replacement=False, mutations=[], ligation=False, hydrogens=False, **kwargs)
```

Bases: `gaudi.genes.GeneProvider`

Mutamers class

**Parameters**

- **residues**(list of str) – Residues that can mutate. This has to be in the form:

  ```
  [ Protein/233, Protein/109 ]
  ```

  where the first element (before slash) is the gaudi.genes.molecule name and the second element (after slash) is the residue position number in that molecule.

  This list of str is later parsed to the proper chimera.Residue objects

- **library**({'Dunbrack', 'Dynameomics'}) – The rotamer library to use.

- **mutations**(list of str, required) – Aminoacids (in 3-letter codes) residues can mutate to.

- **ligation**(bool, optional) – If True, all residues will mutate to the same type of aminoacid.

- **hydrogens**(bool, optional) – If True, add hydrogens to replacing residues (buggy)

**allele**

For i residues, it contains i tuples with two values each: residue type and a float within [0, 1), which will be used to pick one of the rotamers for that residue type.

**Type** list of 2-tuple (str, float)
static add_hydrogens_to_isolated_rotamer(rotamers)

choice(l)
Overrides random.choice with custom one so we can reuse a previously obtained random number. This helps dealing with the ligation parameter, which forces all the requested residues to mutate to the same type

express()
Compile the gene to an evaluable object.

get_rotamers(mol, pos, restype)
Gets the requested rotamers out of cache and if not found, creates the library and stores it in the cache.

Parameters

• mol (str) – gaudi.genes.molecule name that contains the residue
• pos – Residue position in mol
• restype – Get rotamers of selected position with this type of residue. It does not need to be the original type, so this allows mutations

Returns

Return type List of rotamers returned by Rotamers.getRotamers.

mate (mate)
Perform a crossover with another gene of the same kind.

mutate(indpb)
Perform a mutation on the gene.

unexpress()
Revert expression.

static update_rotamer_coords(residue, rotamer)

10.2.3 NormalModes gene

This module allows to explore molecular folding through normal modes analysis. It works by calculating normal modes for the input molecule and moving along a combination of normal modes. It needs at least a gaudi.genes.rotamers.molecule.Molecule.

class gaudi.genes.normalmodes.NormalModes (method='prody', target=None, modes=None, n_samples=10000, rmsd=1.0, group_by=None, group_lambda=None, path=None, write_modes=False, **kwargs)

Bases: gaudi.genes.GeneProvider

NormalModes class

Parameters

• method (str, optional, default=prody) – Either: - prody : calculate normal modes using prody algorithms - gaussian : read normal modes from a gaussian output file
• target (str) – Name of the Gene containing the actual molecule
• modes (list, optional, default=range(12)) – Modes to be used to move the molecule
\textbf{group\_by} (str or callable, optional, default=None) – Available str names: residues, mass. This is, consider only pseudoatoms that group a whole residue, groups of contiguous atoms that amount for certain mass, or only alpha carbons. If residues or mass, the \textbf{group\_lambda} parameter can customize the behaviour.

\textbf{group\_lambda} (int, optional) – Either: number of residues per group (default=7), or number of groups with same mass (default=100).

\textbf{path} (str) – Gaussian or prody modes output path. Required if \textbf{method} is \textit{gaussian}.

\textbf{write\_modes} (bool, optional) – write a molecule\_modes.nmd file with the ProDy modes

\textbf{n\_samples} (int, optional, default=10000) – number of conformations to generate

\textbf{rmsd} (float, optional, default=1.0) – RMSD (in angstrom) that the conformations will have with respect to the initial conformation. In some cases, it can be slightly higher than the requested threshold (e.g. 1.07Å instead of 1Å)

\textbf{allele} Randomly picked coordinates from NORMAL\_MODE\_SAMPLES

\textbf{NORMAL\_MODES} normal modes calculated for the molecule or readed from the gaussian frequencies output file stored in a prody modes class (ANM or RTB)

\textbf{NORMAL\_MODE\_SAMPLES} configurations applying modes to molecule

\textbf{_original\_coords} Parent coordinates

\textbf{_chimera2prody} \_chimera2prody[chimera\_index] = prody\_index

\textbf{Todo:} Combine coords between two samples in NORMAL\_MODE\_SAMPLES? Or two samples between different NORMAL\_MODE\_SAMPLES? Or combine samples between two NORMAL\_MODE\_SAMPLES?
molecule

mutate(indpb)
(mutate to get) another SAMPLE with probability = indpb

read_gaussian_normal_modes()
read normal modes, creates a dictionary between chimera and prody indices and calculate n_confs number of configurations using this modes

read_prody_normal_modes()

unexpress()
Undo coordinates change

gaudi.genes.normalmodes.alg3(moldy, max_bonds=3, **kwargs)
TESTS PENDING!

Coarse Grain Algorithm 3: Graph algorithm.

New group when a vertice: have more than n, have 0 edges new chain

Parameters

- moldy(prody.AtomGroup) -
- n(int, optional, default=2) - maximum bonds number

Returns moldy – New Betas added

Return type prody.AtomGroup

gaudi.genes.normalmodes.chimeracoords2numpy(molecule)

Parameters molecule(chimera.molecule) -

Returns

Return type numpy.array with molecule.atoms coordinates

gaudi.genes.normalmodes.chunker(end, n)
divide end integers in closed groups of n

gaudi.genes.normalmodes.convert_chimera_molecule_to_prody(molecule)
Function that transforms a chimera molecule into a prody atom group

Parameters molecule(chimera.Molecule) -

Returns

- prody_molecule (prody.AtomGroup())
- chimera2prody (dict) – dictionary: chimera2prody[chimera_atom.coordIndex] = i-thm element prody getCoords() array

10.2. gaudi.genes
gaudi.genes.normalmodes.group_by_mass(molecule, n=100)
Coarse Grain Algorithm 2: groups per mass percentage

Parameters
- molecule (prody.AtomGroup)
- n (int, optional, default=100) – Intended number of groups. The mass of the system will be divided by this number, and each group will have the corresponding proportional mass. However, the final number of groups can be slightly different.

Returns molecule – New Betas added
Return type prody.AtomGroup

gaudi.genes.normalmodes.group_by_residues(molecule, n=7)
Coarse Grain Algorithm 1: groups per residues

Parameters
- molecule (prody.AtomGroup)
- n (int, optional, default=7) – number of residues per group

Returns molecule – New betas added
Return type prody.AtomGroup

gaudi.genes.normalmodes.prody_modes(molecule, max_modes, algorithm=None, **options)

Parameters
- molecule (prody.AtomGroup)
- max_modes (int) – number of modes to calculate
- algorithm (callable, optional, default=None) – coarseGrain(prm) wich make molecule.select().setBetas(i) where i is the index Coarse Grain group Where prm is prody AtomGroup
- options (dict, optional) – Parameters for algorithm callable

Returns modes
Return type ProDy modes ANM or RTB

10.2.4 Rotamers gene

This modules allows to explore side chains flexibility in proteins, as well as mutation.

It needs that at least a gaudi.genes.rotamers.molecule.Molecule has been requested in the input file. Residues of those are referenced in the residues argument.

It also allows mutations in the selected residues. However, the resulting structure keeps the same backbone, which may not be representative of the in-vivo behaviour. Use with caution.

class gaudi.genes.rotamers.Rotamers(residues=None, library='Dunbrack', with_original=True, **kwargs)
Bases: gaudi.genes.GeneProvider
Rotamers class

Parameters
- residues (list of str) – Residues that should be analyzed. This has to be in the form:
where the first element (before slash) is the gaudi.genes.molecule name and the second element (after slash) is the residue position number in that molecule.

This list of str is later parsed to the proper chimera.Residue objects

- **library** ("Dunbrack", "Dynameomics") – The rotamer library to use.
- **with_original** (bool, defaults to True) – Whether to include the original set of chi angles as part of the rotamer library.

**allele**

For i residues, it contains i floats within [0, 1), that will point to the selected rotamer torsions for each residue.

- **Type** list of float

**static all_chis**(residue)

**express**()

Compile the gene to an evaluable object.

**mate**(mate)

Perform a crossover with another gene of the same kind.

**mutate**(indpb)

Perform a mutation on the gene.

**static patch_residue**(residue)

**retrieve_rotamers**(molecule, position, residue, library='Dunbrack')

**unexpress**()

Revert expression.

**static update_rotamer**(residue, chis)

gaudi.genes.rotamers.enable(**kwargs)

### 10.2.5 Search gene

This module provides spatial exploration of the environment.

It works by creating a sphere with radius `self.radius` and origin at `self.origin`. The movement is achieved with three matrices that contain a translation, a rotation, and a reference position.

It depends on `gaudi.genes.molecule.Molecule`, since these are the ones that will be moved around. Combined with the adequate objectives, this module can be used to implement docking experiments.

**class gaudi.genes.search.Search**(target=None, center=None, radius=None, rotate=True, precision=0, interpolation=0.5, **kwargs)

**Bases:** `gaudi.genes.GeneProvider`

**Parameters**

- **target**(namedtuple or Name of gaudi.genes.molecule) – Can be either:
  - The anchor atom of the molecule we want to move, with syntax `<molecule_name>/index`. For example, if we want to move Ligand using atom with serial number = 1 as pivot, we would specify Ligand/1. It’s parsed to the actual chimera.Atom later on. - A name of gaudi.genes.molecule instance. In this case, the anchor atom for the movement of the molecule will be set to its nearest atom to the geometric center of the molecule.
• **center** *(3-item list or tuple of float, optional)* – Coordinates to the center of the desired search sphere

• **radius** *(float)* – Maximum distance from center that the molecule can move

• **rotate** *(bool, bool)* – If False, don’t rotate the molecule - only translation

• **precision** *(int, bool)* – Rounds the decimal part of the 3D search matrix to get a coarser model of space. I.e, less points can be accessed, the search is less exhaustive, more variability in less runs.

**allele**  
A 4x3 matrix of float, as explained in Notes.  
**Type** 3-tuple of 4-tuple of floats

**origin**  
The initial position of the requested target molecule. If we don’t take this into account, we can’t move the molecule around was not originally in the center of the sphere.  
**Type** 3-tuple of float

**Notes**

**How matricial translation and rotation takes place**

A single movement is summed up in a 4x3 matrix:

\[  
\begin{pmatrix}  
R1 & R2 & R3 & T1 \\
R4 & R5 & R6 & T2 \\
R7 & R8 & R9 & T3  
\end{pmatrix}  
\]

R-elements contain the rotation information, while T elements account for the translation movement. That matrix can be obtained from multiplying three different matrices with this expression:

\[
multiply_matrix(translation, rotation, to_zero)  
\]

To understand the operation, it must be read from the right:

1. First, translate the molecule the origin of coordinates 0,0,0
2. In that position, the rotation can take place.
3. Then, translate to the final coordinates from zero. There’s no need to get back to the original position.

How do we get the needed matrices?

• **to_zero**. Record the original position *(origin)* of the molecule and multiply it by -1. Done with method `to_zero()`.

• **rotation**. Obtained directly from `FitMap.search.random_rotation`

• **translation**. Check docstring of `random_translation()` in this module.

**center**

express()  
Multiply all the matrices, convert the result to a chimera.CoordFrame and set that as the xform for the target molecule. If precision is set, round them.

**mate** *(mate)*  
Interpolate the matrices and assign them to each individual. Ind1 gets the rotated interpolation, while Ind2 gets the translation.

**molecule**
**mutate** *(indpb)*
Perform a mutation on the gene.

**origin**

**random_transform()**
Wrapper function to provide translation and rotation in a single call

**to_zero**
Return a translation matrix that takes the molecule from its original position to the origin of coordinates (0,0,0).

Needed for rotations.

**unexpress()**
Reset xform to unity matrix.

**gaudi.genes.search.center** *(mol)*

**gaudi.genes.search.distance** *(a, b)*

**gaudi.genes.search.enable** *(**kwargs)*

**gaudi.genes.search.nearest_atom** *(mol, position)*

**gaudi.genes.search.parse_origin** *(origin, individual=None)*

The center of the sphere can be given as an Atom, or directly as a list of three floats (x,y,z). If it’s an Atom, find it and return the xyz coords. If not, just turn the list into a tuple.

**Parameters**

- **origin** *(3-item list of coordinates, or chimera.Atom)* –
- **genes** *(gaudi.parse.Settings.genes)* – List of gene-dicts to look for molecules that may contain the referred atom in origin

**Returns**

The x,y,z coordinates

**Return type**

Tuple of float

**gaudi.genes.search.rand_xform** *(origin, destination, r, rotate=True)*

**gaudi.genes.search.random_translation** *(center, r)*

Get a random point from the cube built with l=r and test if it’s within the sphere. Most of the points will be, but not all of them, so get another one until that criteria is met.

**Parameters**

- **center** *(3-tuple of float)* – Coordinates of the center of the search sphere
- **r** *(float)* – Radius of the search sphere

**Returns**

**Return type**

A translation matrix to a random point in the search sphere, with no rotation.

**gaudi.genes.search.rotate** *(molecule, at, alpha)*

**gaudi.genes.search.translate** *(molecule, anchor, target)*

### 10.2.6 Torsions gene

This module helps explore small molecules flexibility.

It does so by performing bond rotations in the selected `gaudi.genes.molecule.Molecule` objects, if they exhibit free bond rotations.
class gaudi.genes.torsion.Torsion(target=None, flexibility=360.0, max_bonds=None, anchor=None, rotatable_atom_types=('C3', 'N3', 'C2', 'N2', 'P'), rotatable_atom_names=(), rotatable_elements=(), non_rotatable_bonds=(), rotatable_bonds=(), precision=1, **kwargs)

Bases: gaudi.genes.GeneProvider

Parameters

• **target** *(str)* – Name of gaudi.genes.molecule instance to perform rotation on

• **flexibility** *(int or float)* – Maximum number of degrees a bond can rotate

• **max_bonds** – Expected number of free rotations in molecule. Needed to store arbitrary rotations.

• **anchor** *(str)* – Molecule/atom_serial_number of reference atom for torsions

• **rotatable_atom_types** *(list of str)* – Which type of atom types (as in chimera.Atom.idatmType) should rotate. Defaults to (`'C3', 'N3', 'C2', 'N2', 'P'`).

• **rotatable_atom_names** *(list of str)* – Which type of atom names (as in chimera.Atom.name) should rotate. Defaults to ()

• **rotatable_bonds** *(list of [SerialNumberAtom1, SerialNumberAtom2, SerialNumberAnchor])* – Concrete bonds that are allowed to rotate. Atoms have to be designated using their chimera serial number. IMPORTANT: if set, these will be the ONLY bonds allowed to rotate, ignoring other possible conditions (e.g. rotatable_atom_types, rotatable_atom_names...).

allele
For i rotatable bonds in molecule, it contains i floats which correspond to each torsion angle. As such, each falls within [-180.0, 180.0).

Type tuple of float

Notes

Todo: max_bonds is now automatically computed, but probably won’t deal nicely with block-built ligands.

BONDS_ROTS = {}

anchor
Get the molecular anchor. Ie, the root of the rotations, the fixed atom of the molecule.

Usually, this is the target atom in the Search gene, but if we can’t find it, get the nearest atom to the geometric center of the molecule, and if it’s not possible, the donor atom of the molecule.

clear_cache()

express()
Apply rotations to rotatable bonds

mate *(mate)*
Perform a crossover with another gene of the same kind.

molecule

mutate *(indpb)*
Perform a mutation on the gene.
random_angle()
   Returns a random angle within flexibility limits

rotatable_bonds
   Gets potentially rotatable bonds in molecule

   First, it retrieves all the atoms. Then, the bonds are filtered, discarding coordination (pseudo)bonds and sort them by atom serial.

   For each bond, try to retrieve it from the cache. If unavailable, request a bond rotation object to chimera.BondRot.

   In this step, we have to discard non rotatable atoms (as requested by the user), and make sure the involved atoms are of compatible type. Namely, one of them must be either C3, N3, C2 or N2, and both of them, non-terminal (more than one neighbor).

   If the bond is valid, get the BondRot object. Chimera will complain if we already have requested that bond previously, or if the bond is in a cycle. Handle those exceptions silently, and get the next bond in that case.

   If no exceptions are raised, then store the rotation anchor in the BondRot object (that’s the nearest atom in the bond to the molecular anchor), and store the BondRot object in the rotations cache.

unexpress()
   Undo the rotations

   gaudi.genes.torsion.center(mol)
   gaudi.genes.torsion.distance(a, b)
   gaudi.genes.torsion.enable(**kwargs)
   gaudi.genes.torsion.nearest_atom(mol, position)

10.2.7 Trajectory gene

This module provides spatial exploration of the environment through a MD trajectory file.

class gaudi.genes.trajectory.Trajectory(target=None, path=None, max_frame=None, stride=1, preload=False, **kwargs)
   Bases: gaudi.genes.GeneProvider

   Parameters

   • target (str) – The Molecule that contains the topology of the trajectory.
   • path (str) – Path to a MD trajectory file, as supported by mdtraj.
   • max_frame (int) – Last frame of the trajectory that can be loaded.
   • stride (int, optional) – Only load one in every stride frames
   • preload (bool, optional) – Load the full trajectory in memory to accelerate expres-
      sion. Not recommended for large files!

   allele
   The index of a frame in the MD trajectory.

   Type  int

   _traj
   Alias to the frames cache

   Type  dict
express()
Load the frame requested by the current allele into a new CoordSet object (always at index 1) and set that as the active one.

load_frame(n)

mate(mate)
Simply exchange alleles. Can’t try to interpolate an intermediate structure because the result wouldn’t probably belong to the original trajectory!

molecule
The target Molecule gene

mutate(indpb)
Perform a mutation on the gene.

random_frame_number()

topology
Returns the equivalent mdtraj Topology object of the currently expressed Chimera molecule

unexpress()
Set the original coordinates (stored at mol.coordSets[0]) as the active ones.

gaudi.genes.trajectory.enable(**kwargs)

10.2.8 Base class for all genes

Genes are modules that reside in the gaudi.genes package, and have a certain class structure.

class gaudi.genes.GeneProvider(parent=None, name=None, cx_eta=5.0, mut_eta=5.0, mut_indpb=0.75, **kwargs)
Bases: object
Base class that every genes plugin MUST inherit.

The methods listed here are compulsory for all subclasses, since the individual will be using them anyway. If it’s not relevant in your plugin, just define them with a single pass statement. Also, don’t forget to call GeneProvider.__init__ in your overridden __init__ function; it registers compulsory

— From (M.A. itself)[http://martyalchin.com/2008/jan/10/simple-plugin-framework/]: Now that we have a mount point, we can start stacking plugins onto it. As mentioned above, individual plugins will subclass the mount point. Because that also means inheriting the metaclass, the act of subclassing alone will suffice as plugin registration. Of course, the goal is to have plugins actually do something, so there would be more to it than just defining a base class, but the point is that the entire contents of the class declaration can be specific to the plugin being written. The plugin framework itself has absolutely no expectation for how you build the class, allowing maximum flexibility. Duck typing at its finest.

classmethod clear_cache()

express()
Compile the gene to an evaluable object.

mate(gene)
Perform a crossover with another gene of the same kind.

mutate()
Perform a mutation on the gene.

plugins = [<class 'gaudi.genes.search.Search'>, <class 'gaudi.genes.molecule.Molecule'>,
unexpress()
    Revert expression.

classmethod validate(data, schema=None)
classmethod with_validation(**kwargs)
write(path, name, *args, **kwargs)
    Write results of expression to a file representation.

10.3 gaudi.objectives

These are the built-in objectives in GaudiMM. You can also build your own, but these are ready to use.

10.3.1 Angle objective

This objective calculates the angle formed by three given atoms (or the dihedral, if four atoms are given) and returns the absolute difference of that angle and the target value.

class gaudi.objectives.angle.Angle(threshold=None, probes=None, *args, **kwargs)
    Bases: gaudi.objectives.ObjectiveProvider

    Angle class

    Parameters

    • threshold (float) – Optimum angle
    • probes (list of str) – Atoms that make the angle, expressed as a series of <molecule_name>/<serial_number> strings

    Returns  Deviation from threshold angle, in degrees

    Return type  float

evaluate(ind)
    Return the score of the individual under the current conditions.

probes(ind)
gaudi.objectives.angle.enable(**kwargs)

10.3.2 Contacts objective

This objective provides a wrapper around Chimera’s DetectClash that detects clashes and contacts. Clashes are understood as steric conflicts that increases the energy of the system. They are evaluated as the sum of volumetric overlapping of the Van der Waals’ spheres of the implied atoms. Contacts are considered as stabilizing, and they are evaluated with a Lennard-Jones 12-6 like function.

class gaudi.objectives.contacts.Contacts(probes=None, radius=5.0, which='hydrophobic', clash_threshold=0.6, hydrophobic_threshold=-0.4, cutoff=0.0, hydrophobic_elements=('C', 'S'), bond_separation=4, same_residue=True, only_internal=False, *args, **kwargs)
    Bases: gaudi.objectives.ObjectiveProvider

    Contacts class

    :param probes: Name of molecule gene that is object of contacts analysis
    :type probes: str
    :param radius: Maximum distance from any point of probes that is searched
for possible interactions

Parameters

- **which(**'hydrophobic', 'clashes')** – Type of interactions to measure
- **clash_threshold** (float, optional) – Maximum overlap of van-der-Waals spheres that is considered as a contact (attractive). If the overlap is greater, it’s considered a clash (repulsive)
- **hydrophobic_threshold** (float, optional) – Maximum overlap for hydrophobic patches.
- **hydrophobic_elements** (list of str, optional, defaults to [C, S]) – Which elements are allowed to interact in hydrophobic patches
- **cutoff** (float, optional) – If the overlap volume is greater than this, a penalty is applied. Useful to filter bad solutions.
- **bond_separation** (int, optional) – Ignore clashes or contacts between atoms within n bonds.
- **only_internal** (bool, optional) – If set to True, take into account only intramolecular interactions, defaults to False

Returns Lennard-Jones-like energy when which='hydrophobic', and volumetric overlap of VdW spheres in A^3 if which='clashes.

Return type float

evaluate_clashes (ind)
evaluate_hydrophobic (ind)
find_interactions (ind)
molecules (ind)
probes (ind)
gaudi.objectives.contacts.enable(**kwargs)

10.3.3 Coordination objective

This objective performs rough estimations of good orientations of ligating residues in a protein to coordinate a given metal or small molecule. The geometry is approximated by computing average distances from ligating atoms the metal centre (self.probe) as well as the angles formed by the probe, the ligating atom and its immediate neighbor. Good planarity is assured by a dihedral check.

class gaudi.objectives.coordination.Coordination (probe=None, radius=3.0, atom_types=(), atom_elements=(), atom_names=(), residues=(), geometry='tetrahedral', distance=0, min_atoms=1, prevent_intruders=True, enforce_all_residues=False, only_one_ligand_per_residue=False, center_of_mass_correction=False, distance_correction=False, **kwargs)
Coordination class

**Parameters**

- **probe** *(tuple)* − The atom that acts as the metal center, expressed as `<molecule_name>/<atom serial>`. This will be parsed later on.
- **residues** *(list of str)* − Residues that must coordinate to `probe`, expressed as `<molecule_name>/<residue position>`. Position can be *.
- **radius** *(float, optional, default=3.0)* − Distance from `probe` where ligating atoms must be found
- **atom_types** *(list of str, optional)* − Types of atoms that are considered ligands to `probe`
- **atom_names** *(list of str, optional)* − Names of atoms that are considered ligands to `probe`
- **atom_elements** *(list of str, optional)* − Elements of atoms that are considered ligands to `probe`
- **distance** *(float, optional)* − Perfect distance a ligand atom should be from target.
- **geometry** *(str or list of 3-tuple floats, optional)* − Which geometry should be fitted. Choose from `GEOMETRIES` dict or specify a set of vectors.
- **enforce_all_residues** *(bool, optional)* − Whether to force or not if all specified residues should coordinate.
- **only_one_ligand_per_residue** *(bool, optional)* − Enforce that only one ligand for each residue should coordinate.
- **prevent_intruders** *(bool, optional)* − Don’t let non-ligand atoms to be closer to the target than the selected ligand atoms.
- **center_of_mass_correction** *(bool, optional)* − If True, calculate the distance between the metal center and the center of mass of the ligand atoms, and sum that to the final score.
- **distance_correction** *(bool, optional)* − If True, report the deviation of the experimental coordination bond length and the ideal one, as tabulated by chimera. Element, and sum that to the final score.

**Returns** Sum of RMSD of vertices from ideal RMSD and average cosine of angle deviation from ideal orientation of ligand neighbors. A perfect match should report 0.0.

**Return type** float

**coordination_sphere** *(ind)*

1. Get atoms and residues found within `self.radius` angstroms from `self.probe`. Found residues MUST include `self.residues`. Otherwise, apply penalty.

2. Sort atoms by absolute difference of `self.distance` and distance to `self.probe`. That way, nearest atoms are computed first. If found atoms do not include some of the requested types, apply penalty.

**evaluate** *(ind)*

1. Get requested atoms sorted by distance

2. If they meet the minimum quantity, return the rmsd for that geometry

3. If that’s not possible of they are not enough, return penalty

**molecules** *(ind)*
probe (ind)
residues (ind)
gaudi.objectives.coordination.enable(**kwargs)
gaudi.objectives.coordination.ideal_bond_deviation(metal, ligand, other_ligands=())
    Assess if the current bond vector is well oriented with respect to the ideal bond vector.

Parameters

• metal (chimera.Atom) – The ion ligands are coordinating to
• ligand (chimera.Atom) – Potential ligand atoms to metal

Returns Absolute sine of the angle between the ideal vector and the ligand-metal one.

Return type float

gaudi.objectives.coordination.ideal_bonded_positions(atom, element, geometry=None)

10.3.4 Distance objective

This objective calculates the distance between two given atoms. It returns the absolute difference between the calculated distance and the target value.

class gaudi.objectives.distance.Distance(threshold=None, tolerance=None, target=None, probes=None, center_of_mass=False, *args, **kwargs)

    Bases: gaudi.objectives.ObjectiveProvider

    Distance class

    Parameters

• threshold (float) – Optimum distance to meet
• tolerance (float) – Maximum deviation from threshold that is not penalized
• target (str) – The atom to measure the distance to, expressed as <molecule name>/atom serial>
• probes (list of str) – The atoms whose distance to target is being measured, expressed as <molecule name>/atom serial>. If more than one is provided, the average of all of them is returned
• center_of_mass (bool) –

    Returns (Mean of) absolute deviation from threshold distance, in A.

    Return type float

    atoms (ind, *targets)

    evaluate_center_of_mass (ind)

    evaluate_distances (ind)
        Measure the distance

gaudi.objectives.distance.enable(**kwargs)
10.3.5 DrugScoreX objective

This objective is a wrapper around the binaries provided by Neudert and Klebe and calculates the score of the current pose. The lower, the better, so usually you will use a -1.0 weight.

```python
class gaudi.objectives.dsx.DSX(binary=None, potentials=None, proteins=('Protein', ), ligands=('Ligand', ), terms=None, sorting=1, cofactor_mode=0, with_covalent=False, with_metals=True, *args, **kwargs)
```

Bases: `gaudi.objectives.ObjectiveProvider`

**DSX class**

**Parameters**

- **protein** *(str)* – The molecule name that is acting as a protein
- **ligand** *(str)* – The molecule name that is acting as a ligand
- **binary** *(str, optional)* – Path to the DSX binary. Only needed if drugscorex is not in PATH.
- **potentials** *(str, optional)* – Path to DSX potentials. Only needed if DSX_POTENTIALS env var has not been set by the installation process (conda install -c insilichem drugscorex normally takes care of that).
- **terms** *(list of bool, optional)* – Enable (True) or disable (False) certain terms in the score function in this order: distance-dependent pair potentials, torsion potentials, intramolecular clashes, sas potentials, hbond potentials
- **sorting** *(int, defaults to 1)* – Sorting mode. An int between 0-6, read binary help for -S:

  `-S int : Here you can specify the mode that affects how the_ 
  ←results
  will be sorted. The default mode is '-S 1', which sorts the 
  ligands in the same order as they are found in the lig_ 
  ←file. 
  The following modes are possible::

  0: Same order as in the ligand file
  1: Ordered by increasing total score
  2: Ordered by increasing per-atom-score
  3: Ordered by increasing per-contact-score
  4: Ordered by increasing rmsd
  5: Ordered by increasing torsion score
  6: Ordered by increasing per-torsion-score

  **cofactor_mode** *(int, defaults to 0)* – Cofactor handling mode. An int between
  0-7, read binary help for -I:

  `-I int : Here you can specify the mode that affects how cofactors, 
  →waters and metals will be handeled. 
  The default mode is '-I 1', which means, that all molecules 
  →are treated as part of the protein. If a structure should 
  not be treated as part of the protein you have supply a 
  separate file with separate MOLECULE entries corresponding 
  to each MOLECULE entry in the ligand file (It is assumed 
  that the structure, e.g. a cofactor, was kept flexible in 
  docking, so that there should be a different geometry
```

(continues on next page)
corresponding to each solution. Otherwise it won't make
sense not to treat it as part of the protein.).
The following modes are possible:
0: cofactors, waters and metals interact with protein,
    ligand and each other
1: cofactors, waters and metals are treated as part of
    the protein
2: cofactors and metals are treated as part of the
    protein
3: cofactors and waters are treated as part of the
    protein
4: cofactors are treated as part of the protein
5: metals and waters are treated as part of the protein
6: metals are treated as part of the protein
7: waters are treated as part of the protein
Please note: Only those structures can be treated
individually, which are supplied in separate files.

• with_covalent (bool, defaults to False) – Whether to deal with covalently
  bonded atoms as normal atoms (False) or not (True)

• with_metals (bool, defaults to True) – Whether to deal with metal atoms as
  normal atoms (False) or not (True)

Returns Interaction energy as reported by DSX output logs.

Return type float

clean ()
evaluate (ind)
    Run a subprocess calling DSX binary with provided options, and parse the results. Clean tmp files at exit.

get_molecule_by_name (ind, *names)
    Get a molecule gene instance of individual by its name

parse_output (stream)

prepare_command ()

prepare_ligands (ligands)

prepare_proteins (proteins)

gaudi.objectives.dsx.enable (**kwargs)

### 10.3.6 Energy (OpenMM) objective

This objective is a wrapper around OpenMM, providing a GPU-accelerated energy calculation of the system with a
simple forcefield evaluation.

class gaudi.objectives.energy.Energy (targets=None, forcefields=('amber99sbildn.xml', ),
    auto_parametrize=None, parameters=None, platform=None, *args, **kwargs)

Bases: gaudi.objectives.ObjectiveProvider

Calculate the energy of a system

Parameters
• **targets** *(list of str, default=None)* – If set, which molecules should be evaluated. Else, all will be evaluated.

• **forcefields** *(list of str, default=('amber99sbildn.xml',))* – Which forcefields to use

• **auto_parametrize** *(list of str, default=None)* – List of Molecule instances GAUDI should try to auto parametrize with antechamber.

• **parameters** *(list of 2-item list of str)* – List of (gaff.mol2, .frcmod) files to use as parametrization source.

• **platform** *(str)* – Which platform to use for calculations. Choose between CPU, CUDA, OpenCL.

**Returns**  The estimated potential energy, in kJ/mol

**Return type** float

**calculate_energy** *(coordinates)*

Set up an OpenMM simulation with default parameters and return the potential energy of the initial state

**Parameters** coordinates *(simtk.unit.Quantity)* – Positions of the atoms in the system

**Returns** potential_energy – Potential energy of the system, in kJ/mol

**Return type** float

**static chimera_molecule_to_openmm_positions** *(molecules)*

**static chimera_molecule_to_openmm_topology** *(molecules)*

Convert a Chimera Molecule object to OpenMM structure, providing topology and coordinates.

**Parameters** molecule *(chimera.Molecule)* –

**Returns**

• topology *(simtk.openmm.app.topology.Topology)*

• coordinates *(simtk.unit.Quantity)*

**evaluate** *(individual)*

Calculates the energy of current individual

**Notes**

For static calculations, where molecules are essentially always the same, but with different coordinates, we only need to generate topologies once. However, for dynamic jobs, with potentially different molecules involved each time, we cannot guarantee having the same topology. As a result, we generate it again for each evaluation.

**molecules** *(individual)*

**simulation**

Build a new OpenMM simulation if not yet defined and return it

**Notes**

self.topology must be defined previously! Use self.chimera_molecule_to_openmm_topology to set it.
gaudi.objectives.energy.calculate_energy(filename, forcefields=None)

Calculate energy from PDB file with desired forcefields. If not specified, amber99sbildn will be used. Returns potential energy in kJ/mol.

gaudi.objectives.energy.enable(**kwargs)

## 10.3.7 GOLD objective

This objective is a wrapper around the scoring functions provided by CCDC’s GOLD.

It will use the rescoring abilities in GOLD to extract the fitness corresponding to any of the available scoring functions:

- GoldScore (goldscore)
- ChemScore (chemscore)
- Astex Statistical Potential (asp)
- CHEMPLP (chemplp)

Since GOLD is commercial software, you will need to install it separately and provide a valid license! This is just a wrapper. Make sure to set all the needed environment variables, such as CCDC_LICENSE_FILE, and that ‘gold_auto’ is in $PATH. Check tests/test_objectives_gold.py for an example; make sure to have GOLDXX/bin before GOLDXX/GOLD/bin!

```python
class gaudi.objectives.gold.Gold(protein='Protein', ligand='Ligand', scoring='chemscore', score_component='Score', radius=10, *args, **kwargs)
```

Bases: gaudi.objectives.ObjectiveProvider

Gold class

**Parameters**

- **protein** (str) – The name of molecule acting as protein
- **ligand** (str) – The name of molecule acting as ligand
- **scoring** (str, optional, defaults to chemscore) – Fitness function to use. Choose between chemscore, chemplp, goldscore and asp.
- **score_component** (str, optional, defaults to 'Score') – Scoring fields to parse out of the rescore.log file, such as Score, DG, S(metal), etc.
- **radius** (float, optional, defaults to 10.0) – Radius (in A) of binding site sphere, the origin of which is automatically centered at the ligand’s center of mass.

**Returns** Interaction energy as reported by GOLD’s chosen scoring function

**Return type** float

clean()

evaluate(ind)

Run a subprocess calling LigScore binary with provided options, and parse the results. Clean tmp files at exit.

get_molecule_by_name(ind, *names)

Get a molecule gene instance of individual by its name

origin(molecule)

parse_output(filename)

Get last word of first line (and unique) and parse it into float

prepare_command(protein_path, ligand_path, origin)
prepare_ligands(ligands)
prepare_proteins(proteins)
gaudi.objectives.gold.enable(**kwargs)

10.3.8 Hydrogen bonds objective

This objective is a wrapper around Chimera’s *FindHBond*. It returns the number of hydrogen bonds that can be formed between the target molecule and its environment. .. todo:

```mermaid
evaluationbubble
Evaluate the possible HBonds with some kind of function that gives a rough idea of the strength (energy) of each of them.
```

class gaudi.objectives.hbonds.Hbonds(probes=None, radius=5.0, distance_tolerance=0.4,
angle_tolerance=20.0, only_intermolecular=True,
only_probes=False, *args, **kwargs)

Bases: gaudi.objectives.ObjectiveProvider

Hbonds class :

**param probes:** Names of molecules being object of analysis

**type probes:** list of str

**param radius:** Maximum distance from any point of probe that is searched for a possible interaction

**Parameters**

- **distance_tolerance** *(float, optional)* – Allowed deviation from ideal distance to consider a valid H bond.

- **angle_tolerance** *(float, optional)* – Allowed deviation from ideal angle to consider a valid H bond.

- **only_intermolecular** *(boolean, optional)* – Only intermolecular interactions are considered (defaults to True)

- **only_probes** *(boolean, optional)* – Only interactions between probe molecules are considered, excluding other molecule genes. (defaults to False)

**Returns** Number of detected Hydrogen bonds.

**Return type** int

display(bonds)
Mock method to show a graphical depiction of the found H Bonds.

evaluate(ind)
Find H bonds within self.radius angstroms from self.probes, and return only those that interact with probe. Ie, discard those hbonds in that search space whose none of their atoms involved are not part of self.probe.

molecules(ind)

probes(ind)
gaudi.objectives.hbonds.enable(**kwargs)

10.3.9 Inertia objective

This objective calculates the alignment between the axes of inertia of the given molecules.
class gaudi.objectives.inertia.AxesOfInertia

Calculates the axes of inertia of given molecules and returns their alignment deviation.

**Parameters**

- **reference** *(str)* – Molecule name targets should align to.
- **targets** *(list of str)* – Names of molecules to be aligned to reference.
- **threshold** *(float)* – Target average of cosine of angle of alignment between targets and reference.
- **only_primaries** *(bool)* – Consider only the largest inertia vectors.

**Returns**

Mean absolute difference of threshold alignment and mean of all the cosines involved for each axis.

**Return type**

float

evaluate

Return the score of the individual under the current conditions.

reference

The reference molecule. Usually, the biggest in size.

targets

gaudi.objectives.inertia.calculate_alignment

gaudi.objectives.inertia.calculate_axes_of_inertia

gaudi.objectives.inertia.calculate_inertial_matrix

gaudi.objectives.inertia.centroid

gaudi.objectives.inertia.enable

## 10.3.10 LigScore objective

This objective is a wrapper around the scoring function provided by IMP’s ligand_score.

The lower, the better, so usually you will use a -1.0 weight.

class gaudi.objectives.ligscore.LigScore

**Parameters**

- **proteins** *(list of str)* – The name of molecules that are acting as proteins.
- **ligands** *(list of str)* – The name of molecules that are acting as ligands.
- **binary** *(str, optional)* – Path to ligand_score executable.
- **library** *(str, optional)* – Path to LigScore lib file.

**Returns**

Interaction energy as reported by IMP’s ligand_score.
Return type  float

clean()  

evaluate(ind)  
Run a subprocess calling LigScore binary with provided options, and parse the results. Clean tmp files at exit.

get_molecule_by_name(ind, *names)  
Get a molecule gene instance of individual by its name

parse_output(stream)  
Get last word of first line (and unique) and parse it into float

prepare_command(protein_path, ligand_path)  
prepare_ligands(ligands)  
prepare_proteins(proteins)

10.3.11 NWChem objective

This objective is a wrapper around NWChem.

It expects an additional input template with the keyword $MOLECULE, which will be replaced by the currently expressed molecule(s). See TEMPLATE for an example, which works as a default template if none is provided.

A ~/.nwchemrc file should be present. If you installed NWChem with our conda recipe, you will find the file in $CONDA_PREFIX/etc/default.nwchemrc. Copy it to your $HOME.

class gaudi.objectives.nwchem.NWChem(template=None, targets=('Ligand', ), parser=None, title=None, executable=None, basis_library=None, processors=None, *args, **kwargs)

Bases: gaudi.objectives.ObjectiveProvider

NWChem class

Parameters

• targets (list of str) – Molecule name(s) to be processed with NWChem. Small ones!

• template (str, optional) – NWChem input template (or path to a file with such contents) containing a $MOLECULE placeholder to be replaced by the currently expressed molecule(s) requested in targets, and optionally, a $TITLE placeholder to be replaced by the job name. If not provided, it will default to the TEMPLATE example (single-point dft energy).

• parser (str, optional) – Path to a Python script containing a top-level function called parse_output which will parse the NWChem output and return a float. This replaces the default parser, which looks for the last ‘Total <whatever> energy’ value.

• processors (int, optional=None) – Number of physical processors to use with openmpi

Returns Any numeric value as reported by the parser routines. By default, last ‘Total <whatever> energy’ value.

Return type  float

clean()}
**evaluate** *(ind)*
Run a subprocess calling DSX binary with provided options, and parse the results. Clean tmp files at exit.

**get_molecule_by_name** *(ind, *names)*
Get a molecule gene instance of individual by its name

**get_xyz** *(*molecules)*

**parse_output** *(stream)*

**prepare_nwfile** *(*molecules)*

**gaudi.objectives.nwchem.enable** (**kwargs**)

### 10.3.12 Solvation objective

This objective calculates SASA for the given system (or region).

**class** `gaudi.objectives.solvation.Solvation` *(targets=None, threshold=0.0, radius=5.0, method='area', *args, **kwargs)*

**Bases:** `gaudi.objectives.ObjectiveProvider`

Solvation class

**Parameters**

- **targets** *(str)* – Names of the molecule genes being analyzed
- **threshold** *(float, optional, default=0.0)* – Optimize the difference to this value
- **radius** *(float, optional, default=5.0)* – Max distance to search for neighbor atoms from targets.
- **method** *(str, optional, default=area)* – Which method should be used. Both methods compute the surface of the solvated molecule. *area* returns the surface area of such surface, while *volume* returns the volume occupied by the model.

**Returns** Surface area of solvated shell, in Å² (if method=area), or volume of solvated shell, in Å³ (if method=volume).

**Return type** float

**evaluate_area** *(ind)*

**evaluate_volume** *(ind)*

**molecules** *(ind)*

**surface** *(ind)*

**targets** *(ind)*

**zone_atoms** *(probes, molecules)*

**gaudi.objectives.solvation.enable** (**kwargs**)

**gaudi.objectives.solvation.grid_sas_surface** *(atoms, probe_radius=1.4, grid_spacing=0.5)*

Stripped from Chimera’s Surface.gridsurf
10.3.13 Vina objective

This objective is a wrapper around the scoring functions provided by AutoDock Vina.

class gaudi.objectives.vina.Vina(receptor='Protein', ligand='Ligand', prepare_each=False, *args, **kwargs)

Bases: gaudi.objectives.ObjectiveProvider

Vina class

Parameters

• **receptor** (str) – Key of the gene containing the molecule acting as receptor (protein)
• **ligand** (str) – Key of the gene containing the molecule acting as ligand
• **prepare_each** (bool) – Whether to prepare receptors and ligands in every evaluation or try to cache the results for faster performance.

Returns Interaction energy in kcal/mol, as reported by AutoDock Vina --score-only.

Return type float

Notes

• AutoDock scripts prepare_ligand4.py and prepare_receptor4.py are used to prepare the corresponding .pdqt files that will be used as input for AutoDock Vina scorer. - No repairs nor cleanups will be performed on ligand/receptor molecules, so the user has to take into account that provided .mol2 or .pdb files have correct atom types and correct structure (including Hydrogen atoms that will be taken into account in the docking evaluation). Otherwise, AutoDock errors/warnings could appear (e.g. ValueError: Could not find atomic number for Lp Lp) - Gasteiger charges will be added during the preparation of the .pdbqt files. - All torsions of the ligand will be marked as inactive for AutoDock, because torsion changes are part of GaudiMM genes.

clean()

evaluate(ind)

Run a subprocess calling Vina binary with provided options, and parse the results. Clean tmp files at exit.

parse_output(stream)

prepare_ligand(molecule)

prepare_receptor(molecule)

tmpfile
gauid.objectives.vina.enable(**kwargs)

10.3.14 Volume objective

This objective calculates the volume occupied by the requested Molecule gene instance.

Note: Volume is calculated from the surfacePiece created by a new experimental method found in Chimera's Surface.gridsurf. This could be used as an objective for SES, instead of Solvation.
class gaudi.objectives.volume.Volume(threshold=0.0, target=None, cavities=False, *args, **kwargs)

Bases: gaudi.objectives.ObjectiveProvider

Volume class

Parameters

- **threshold** (*float or 'auto'*) – Final volume to target. If 'auto', it will calculate the sum of VdW volumes of all requested atoms in probes. (Unimplemented!)
- **target** (*list of str*) – Molecule gene name to calculate volume over
- **cavities** (*boolean, optional, default=False*) – If True, evaluate cavities volume creating a convex hull and calculating the difference between convex hull volume and molecule volume

Returns  Calculated volume in \(\text{A}^3\)

Return type  float

evaluate_convexhull(ind)

evaluate_volume(ind)

target(ind)

gaudi.objectives.volume.convexhull_volume(surface)

This function gets a surface, creates the convex hull and calculates its volume

Parameters  **surface** (*Surface.gridsurf.ses_surface(molecule.atoms)*) –

Returns  **volume** – Convex hull volume

Return type  float

Notes

Some systems may produce small volume blobs, resulting in a number of different surface pieces. This should be discussed in the future. 

```
# points = surface.surfacePieces[0].geometry[0]  
# convexhull = scipy.spatial.ConvexHull(points)
```

gaudi.objectives.volume.enable(**kwargs)

10.3.15 Base class for all objectives

Objectives are modules that reside in the `gaudi.objectives` package, and have a certain class structure

class gaudi.objectives.ObjectiveProvider(environment=None, name=None, weight=None, zone=None, precision=3, **kwargs)

Bases: object

Base class that every objectives plugin MUST inherit.

Mount point for plugins implementing new objectives to be evaluated by DEAP. The objective resides within the Fitness attribute of the individual. Do whatever you want, but use an evaluate() function to return the results. Apart from that, there’s no requirements.

The base class includes some useful attributes, so don’t forget to call **ObjectiveProvider.__init__** in your over-ridden **init**. For example, **self.zone** is a **Chimera.selection.ItemizedSelection** object which is shared among all objectives. Use that to get atoms in the surrounding of the target gene, and remember to **self.zone.clear()** it before use.
Now that we have a mount point, we can start stacking plugins onto it. As mentioned above, individual plugins will subclass the mount point. Because that also means inheriting the metaclass, the act of subclassing alone will suffice as plugin registration. Of course, the goal is to have plugins actually do something, so there would be more to it than just defining a base class, but the point is that the entire contents of the class declaration can be specific to the plugin being written. The plugin framework itself has absolutely no expectation for how you build the class, allowing maximum flexibility. Duck typing at its finest.

classmethod clear_cache()

evaluate(individual)
     Return the score of the individual under the current conditions.

plugins = [<class 'gaudi.objectives.angle.Angle'>, <class 'gaudi.objectives.contacts.Contacts'>, <class 'gaudi.objectives.solvation.Solvation'>, <class 'gaudi.objectives.vina.Vina'>, <class 'gaudi.objectives.volume.Volume'>]

classmethod validate(data, schema=None)

classmethod with_validation(**kwargs)

10.4 gaudi.algorithms

This module implements evolutionary algorithms as seen in DEAP, and extends their functionality to make use of GAUDI goodies.

Todo:
   • Genealogy

gaudi.algorithms.dump_population(population, cfg, subdir=None)

gaudi.algorithms.ea_mu_plus_lambda(population, toolbox, mu, lambda_, cxpb, mutpb, ngen, cfg, stats=None, halloffame=None, verbose=True, prompt_on_exception=True)

This is the ($\mu + \lambda$) evolutionary algorithm.

Parameters
   • population – A list of individuals.
   • toolbox – A Toolbox that contains the evolution operators.
   • mu – The number of individuals to select for the next generation.
   • lambda_ – The number of children to produce at each generation.
   • cxpb – The probability that an offspring is produced by crossover.
   • mutpb – The probability that an offspring is produced by mutation.
   • ngen – The number of generation.
   • stats – A Statistics object that is updated inplace, optional.
   • halloffame – A HallOfFame object that will contain the best individuals, optional.
   • verbose – Whether or not to log the statistics.

Returns The final population.
First, the individuals having an invalid fitness are evaluated. Then, the evolutionary loop begins by producing \( \lambda \) offspring from the population, the offspring are generated by a crossover, a mutation or a reproduction proportionally to the probabilities \( cxpb \), \( mutpb \) and \( 1 - (cxpb + mutpb) \). The offspring are then evaluated and the next generation population is selected from both the offspring and the population. Briefly, the operators are applied as following:

```python
evaluate(population)
for i in range(ngen):
    offspring = varOr(population, toolbox, lambda_, cxpb, mutpb)
    evaluate(offspring)
    population = select(population + offspring, mu)
```

This function expects `toolbox.mate()`, `toolbox.mutate()`, `toolbox.select()` and `toolbox.evaluate()` aliases to be registered in the toolbox. This algorithm uses the `varOr()` variation.

10.5 gaudi.base

Contains the core classes we use to build individuals (potential solutions of the optimization process).

```python
class gaudi.base.BaseIndividual(cfg=None, cache=None, **kwargs):
    Bases: object
    Base class for individual objects that are evaluated by DEAP.

    Each individual is a potential solution. It contains all that is needed for an evaluation. With multiprocessing in mind, individuals should be self-contained so it can be passed between threads.

    The defined methods are only wrapper calls to the respective methods of each gene.

    Parameters

    - ** cfg (gaudi.parse.Settings) – The full parsed object from the configuration YAML file. **
    - ** cache (dict or dict-like) – A mutable object that can be used to store values across instances. **
    - ** dummy (bool) – If True, create an uninitialized Individual, only containing the cfg attribute. If false, call __ready__ and complete initialization. **

    __CACHE
    Class attribute that caches gene data across instances
    Type dict

    __CACHE_OBJ
    Class attribute that caches objectives data across instances
    Type dict
```

**Todo:** `write()` should use Pickle and just save the whole object, but Chimera’s immutable objects (Atoms, Residues, etc) get in the way. A workaround may be found if we take a look a the session saving code.

```python
clear_cache()
```

evaluate(environment)

Express individual, evaluate it and unexpress it.

**Parameters**

- ** environment (Environment) – Objectives that will evaluate the individual**
express()
Express genes in this environment. Very much like ‘compiling’ the individual to a chimera.Molecule.

mate(other)
Recombine genes of self with other. It simply calls mate on each gene instance

Parameters other(Individual) – Another individual to mate with.

mutate(indpb)
Trigger a round of possible mutations across all genes

Parameters indpb(float) – Probability of suffering a mutation

post_express()
post_unexpress()
pre_express()
pre_unexpress()
similar(other)
Compare self and other with a similarity function.

Returns
Return type bool

unexpress()
Undo express()

write(i, path=None)
Export the individual to a mol2 file

Parameters i(int) – Individual identifier in current generation or hall of fame

:param .. note :::: Maybe someday we can pickle it all :/

```python
>>> filename = os.path.join(path, '{}/{}.pickle.gz'.format(name, i))
>>> with gzip.GzipFile(filename, 'wb') as f:
...     cPickle.dump(self, f, 0)
>>> return filename
```

class gaudi.base.Environment(cfg=None, *args, **kwargs)
Bases: object
Objective container and helper to evaluate an individual. It must be instantiated with a gaudi.parse.Settings object.

Parameters cfg(gaudi.parse.Settings) – The parsed configuration YAML file that contains objectives information

clear_cache()

evaluate(individual)
individual : Individual

class gaudi.base.Fitness(weights)
Bases: deap.base.Fitness

wvalues = ()

class gaudi.base.MolecularIndividual(*args, **kwargs)
Bases: gaudi.base.BaseIndividual
find_molecule(name)
post_express()
xyz(gene=None)
gaudi.base.expressed(*args,**kwds)

10.6 gaudi.box

This module is a messy collection of useful functions used all along GAUDI.

Todo: Some of these functions are hardly used, so maybe we should clean it a little in the future...

gaudi.box.atoms_between(atom1, atom2)
    Finds all connected atoms between two given atoms

gaudi.box.atoms_by_serial(*serials, **kw)
    Find atoms in kw['atoms'] with serialNumber = serials.

    Parameters
    • serials(int) – List of serial numbers to match
    • atoms(list of chimera.Atom, optional) – List of atoms to be traversed while looking for serial numbers

    Returns
    Return type list of chimera.Atom

gaudi.box.create_single_individual(path)
    Create an individual within Chimera. Convenience method for Chimera IDLE.

gaudi.box.do_cprofile(func)
    Decorator to cProfile a certain function and output the results to cprofile.out

gaudi.box.draw_interactions(interactions, startCol='FF0000', endCol='FFFF00', key=None, name='Custom pseudobonds')
    Draw pseudobonds depicting atoms relationships.

    Parameters
    • interactions(list of tuples) – Each tuple contains an interaction, defined, at least, by the two atoms involved.
    • startCol(str, optional) – Hex code for the initial color of the pseudobond (closer to the first atom of the pair).
    • endCol(str, optional) – Hex code for the final color of the pseudobond. (closer to the second atom of the pair)
    • key(int, optional) – The index of an interaction tuple that represent the alpha channel in the color used to depict the interaction.
    • name(str, optional) – Name of the pseudobond group created.

    Returns
    Return type chimera.pseudoBondGroup
gaudi.box.files_in(path, ext=None)
    Returns all the files in a given directory, filtered by extension if desired.

    Parameters
    ----------
    path : str
        Path to the directory.
    ext : list of str, optional
        File extension(s) to filter on.

    Returns
    -------
    List of absolute paths

gaudi.box.find_nearest(anchor, atoms)
    Find the atom of atoms that is closer to anchor, in terms of number of atoms in between.

gaudi.box.highest_atom_indices(r)
    Returns a dictionary with highest atom indices in given residue. Key: value -> element.name: highest index in residue

gaudi.box.incremental_existing_path(path, separator='__')

gaudi.box.open_models_and_close(*args, **kwds)

gaudi.box.pseudobond_to_bond(molecule, remove=False)
    Transforms every pseudobond in molecule to a covalent bond

    Parameters
    ----------
    molecule : chimera.Molecule
        The molecule to transform.
    remove : bool
        If True, remove original pseudobonds after actual bonds have been created.

gaudi.box.rmsd(a, b)

gaudi.box.sequential_bonds(atoms, s)
    Returns bonds in atoms in sequential order, beginning at atom s

gaudi.box.silent_stdout(*args, **kwds)

gaudi.box.stdout_to_file(workspace, stderr=True)

gaudi.boxsuppress_ksdssp(trig_name, my_data, molecules)
    Monkey-patch Chimera triggers to disable KSDSSP computation

gaudi.box.write_individuals(inds, outpath, name, evalfn, remove=True)
    Write an individual to disk.

Note: Deprecated since an Individual object is able to write itself to disk.

10.7 gaudi.exceptions

This module collects more meaningful exceptions than builtins.

exception gaudi.exceptionsAtomsNotFound
    Bases: exceptions.Exception

exception gaudi.exceptions.MoleculesNotFound
    Bases: exceptions.Exception
exception gaudi.exceptions.ResiduesNotFound
    Bases: exceptions.Exception

exception gaudi.exceptions.TooManyAtoms
    Bases: exceptions.Exception

exception gaudi.exceptions.TooManyResidues
    Bases: exceptions.Exception

10.8 gaudi.parallel

Helper functions to deal with parallel execution of GaudiMM jobs.
Useful for benchmarks.

    gaudi.parallel.run_parallel(fn, args=(), processes=None, initializer=None, initargs=(), max_tasksperchild=1, map_timeout=9999999, map_chunksize=1, map_callback=None)

Create a pool instance with built-in exception handling.

10.9 gaudi.parse

This module parses and validates YAML input files into convenient objects that allow per-attribute access to configuration parameters.

    gaudi.parse.AssertList(*validators, **kwargs)
    Make sure the value is contained in a list

    gaudi.parse.Coordinates(v)

    gaudi.parse.Degrees(v)

    gaudi.parse.ExpandUserPathExists(v)

    gaudi.parse.Importable(v)

    gaudi.parse.MakeDir(validator)

class gaudi.parse.MoleculeAtom(molecule, atom)
    Bases: tuple
        atom
            Alias for field number 1

        molecule
            Alias for field number 0

class gaudi.parse.MoleculeResidue(molecule, residue)
    Bases: tuple
        molecule
            Alias for field number 0

        residue
            Alias for field number 1

gaudi.parse.Molecule_name(v)
    Ideal implementation:
def fn(v):
    valid = [i['name'] for i in items if i['module'] == 'gaudi.genes.molecule']
    if v not in valid:
        raise Invalid("{} is not a valid Molecule name").format(v)
    return v

return fn

However, I must figure a way to get the gene list beforehand

gaudi.parse.Named_spec(*names)
    Assert that str is formatted like ”Molecule/123”, with Molecule being a valid name of a Molecule gene and 123 a positive int or *

gaudi.parse.RelPathToInputFile(inputpath=None)

gaudi.parse.ResidueThreeLetterCode(v)

class gaudi.parse.Settings(path=None, validation=True)
    Bases: munch.Munch
    Parses a YAML input file with PyYAML, validates it with voluptuous and builds a attribute-accessible dict with Munch.

    Hence, all the attributes in this class are generated automatically from the default values, and the updated with the contents of the YAML file.

    Parameters path (str) – Path to YAML file

output
    Contains the parameters to determine how to write and report results.

        Type dict

output.path
    Directory that will contain the result files. If it does not exist, it will be created. If it does, the contents could be overwritten. Better change this between different attempts.

        Type str, optional, defaults to . (current dir)

output.name
    A small identifier for your calculation. If not set, it will use five random characters.

        Type str, optional

output.precision
    How many decimals should be used along the simulation. This won’t only affect reports, but also the reported scores by the objectives during the selection process.

        Type int, optional, defaults to 3

output.compress
    Whether to apply compression to the individual ZIP files or not.

        Type bool, optional, defaults to True

output.history
    Whether to save all the genealogy of the individuals created along the simulation or not. Only for advanced users.

        Type bool, optional, defaults to False

output.pareto
    If True, the elite population will be the Pareto front of the population. If False, the elite population will be the best solutions, according to the lexicographic sorting of the fitness values.
output.verbose

Whether to realtime report the progress of the simulation or not.

Type bool, optional, defaults to True

output.check_every

Dump the elite population every n generations. Switched off if set to 0.

Type int, optional, defaults to 10

output.prompt_on_exception

When an exception is raised, GaudiMM tries to dump the current population to disk as an emergency rescue plan. This includes pressing Ctrl+C. If this happens, it prompts the user whether to dump it or not. For interactive sessions this is desirable, but no so much for unsupervised cluster jobs. If set to False, this behaviour will be disabled.

Type bool, optional, defaults to True

Contains the genetic algorithm parameters.

Type dict

ga

Size of the starting population, in number of individuals.

Type int

ga.generations

Number of generations to simulate.

Type float

ga.mu

The number of children to select at each generation, expressed as a multiplier of ga.population.

Type float, optional, defaults to 1.0

ga.lambda_

The number of children to produce at each generation, expressed as a multiplier of ga.population.

Type float, optional, defaults to 3.0

ga.mut_eta

Crowding degree of the mutation. A high eta will produce a mutant resembling its parent, while a small eta will produce a solution much more different.

Type float, optional, defaults to 5

ga.mut_pb

The probability that an offspring is produced by mutation.

Type float, optional, defaults to 0.5

ga.mut_indpb

Independent probability for each gene to be mutated.

Type float, optional, defaults to 0.75

ga.cx_eta

Crowding degree of the crossover. A high eta will produce children resembling to their parents, while a small eta will produce solutions much more different.

Type float, optional, defaults to 5
The probability that an offspring is produced by crossover.

**Type** float, optional, defaults to 0.5

**similarity**
Contains the parameters to the similarity operator, which, given two individuals with the same fitness, whether they can be considered the same solution or not.

**Type** dict

**similarity.module**
The function to call when a fitness draw happens. It should be expressed as Python importable path; ie, separated by dots: gaudi.similarity.rmsd.

**Type** str

**similarity.args**
Positional arguments to the similarity function.

**Type** list

**similarity.kwargs**
Optional arguments to the similarity function.

**Type** dict

**genes**
Contains the list of genes that each Individual will have.

**Type** list of dict

**objectives**
Contains the list of objectives that will make the Environment object to evaluate the Individuals.

**Type** list of dict

```python
default_values = {'ga': {'cx_eta': 5, 'cx_pb': 0.5, 'generations': 3, 'lambda_': 3, 'mu': 1, 'mut_eta': 5, 'mut_indpb': 1}, 'similarity': {'args': [['Ligand'], 2.5], 'kwargs': {}, 'module': 'gaudi.similarity.rmsd'}}
```

**name_objectives**

```python
schema = {'genes': All(Length(min=1, max=None), [<type 'dict'>], msg=None), 'objectives': All(Length(min=1, max=None), [<type 'dict'>], msg=None)}
```

**validate**(data=None)

**weights**

```python
gaudi.parse.deep_update(source, overrides)
```

Update a nested dictionary or similar mapping.

Modify `source` in place.

```python
gaudi.parse.parse_rawstring(s)
```

It parses reference strings contained in some fields.

These strings contain references to `genes.molecule` instances, and one of its atoms

```python
gaudi.parse.validate(schema, data)
```

## 10.10 gaudi.plugin

This module provides the basic functionality for the plugin system of `genes` and `objectives`. 
class gaudi.plugin.PluginMount(name, bases, attrs)

Bases: type

Base class for plugin mount points.

Metaclass trickery obtained from Marty Alchin’s blog Each mount point (ie, genes and objectives), MUST inherit this one.

gaudi.plugin.import_plugins(*pluginlist)

Import requested modules, only once, when launch.py is called and the configuration is parsed successfully.

Parameters

pluginlist (list of gaudi.parse.Param) – Usually, the genes or objectives list resulting from the configuration parsing.

gaudi.plugin.load_plugins(plugins, container=None, **kwargs)

Requests an instance of the class that resides in each plugin. For genes, each individual has its own instance, but objectives are treated like a singleton. So, they are only instantiated once. That’s the reason behind usen a mutable container.

Parameters

• plugins (list of gaudi.parse.Param) – Modules to load. Each Param must have a module attr with a full import path.

• container (dict or dict-like) – If provided, use this container to retain instances across individuals.

• kwargs – Everything else will be passed to the requested plugin instances.

10.11 gaudi.similarity

This module contains the similarity functions that are used to discard individuals that are not different enough.

gaudi.similarity.rmsd(ind1, ind2, subjects, threshold, *args, **kwargs)

Returns the RMSD between two individuals

Parameters

• ind1 (gaudi.base.Individual) –

• ind2 (gaudi.base.Individual) –

• subjects (list of str) – Name of gaudi.genes.molecule instances to measure

• threshold (float) – Maximum RMSD value to consider two individuals as similar. If rmsd > threshold, they are considered different.

Returns

True if rmsd is within threshold, False otherwise. It will always return False if number of atoms is not equal in the two Individuals.

Return type bool

10.12 gaudi._cpdrift

Coherent Point Drift (affine and rigid) Python2/3 implementation, adapted from kwohlfahrt’s.

Only 3D points are supported in this version.

Depends on:

• Python 2.7, 3.4+
- Numpy
- Matplotlib (plotting only)

```python
class gaudi._cpdrift.Quaternion(s, i, j, k)
    Bases: object
    axis_angle
    conjugate(other=None)
    classmethod fromAxisAngle(v, theta)
    matrix()
    norm()
    reciprocal()
    unit()
    vector

gaudi._cpdrift.RMSD(X, Y)
gaudi._cpdrift.affine_cpd(X, Y, w=0.0, B=None)
gaudi._cpdrift.affine_xform(X, B=<Mock name='mock.array()' id='140298192794000'>, t=0)
gaudi._cpdrift.coherent_point_drift(X, Y, w=0.0, B=None, guess_steps=5, max_iterations=20, method='affine')
gaudi._cpdrift.common_steps(X, Y, Y_, w, sigma_sq)

class gaudi._cpdrift.frange(start, stop, step)
    Bases: object

gaudi._cpdrift.last(sequence)
gaudi._cpdrift.pairwise_sqdist(X, Y)
gaudi._cpdrift.plot(x, y, t)
    Plot the initial datasets and registration results.

Parameters
- x (ndarray) – The static shape that y will be registered to. Expected array shape is [n_points_x, n_dims]
- y (ndarray) – The moving shape. Expected array shape is [n_points_y, n_dims]. Note that n_dims should be equal for x and y, but n_points does not need to match.
- t (ndarray) – The transformed version of y. Output shape is [n_points_y, n_dims].

gaudi._cpdrift.rigid_cpd(X, Y, w=0.0, R=None)
gaudi._cpdrift.rigid_xform(X, R=<Mock name='mock.array()' id='140298192794000'>, t=0.0, s=1.0)
gaudi._cpdrift.rotation_matrix(*angles)
gaudi._cpdrift.spaced_rotations(N)
gaudi._cpdrift.std(x)
```
CHAPTER 11

Indices and tables

- genindex
- modindex
- search
Python Module Index

g
  gaudi, 31
  gaudi._cpdrift, 72
  gaudi.algorithms, 63
  gaudi.base, 64
  gaudi.box, 66
  gaudi.cli, 31
  gaudi.cli.gaudi_cli, 31
  gaudi.cli.gaudi_run, 32
  gaudi.exceptions, 67
  gaudi.genes, 48
  gaudi.genes.molecule, 32
  gaudi.genes.mutamers, 38
  gaudi.genes.normalmodes, 39
  gaudi.genes.rotamers, 42
  gaudi.genes.search, 43
  gaudi.genes.torsion, 45
  gaudi.genes.trajectory, 47
  gaudi.objectives, 62
  gaudi.objectives.angle, 49
  gaudi.objectives.contacts, 49
  gaudi.objectives.coordination, 50
  gaudi.objectives.distance, 52
  gaudi.objectives.dsx, 53
  gaudi.objectives.energy, 54
  gaudi.objectives.gold, 56
  gaudi.objectives.hbonds, 57
  gaudi.objectives.inertia, 57
  gaudi.objectives.ligscore, 58
  gaudi.objectives.nwchem, 59
  gaudi.objectives.solvation, 60
  gaudi.objectives.vina, 61
  gaudi.objectives.volume, 61
  gaudi.parallel, 68
  gaudi.parse, 68
  gaudi.plugin, 71
  gaudi.similarity, 72
Symbols

CATALOG (gaudi.genes.molecule.Molecule attribute), 36
_CACHE (gaudi.base.BaseIndividual attribute), 64
_CACHE_OBJ (gaudi.base.BaseIndividual attribute), 64
_chimera2prody (gaudi.genes.normalmodes.NormalModes attribute), 40
_original_coords (gaudi.genes.normalmodes.NormalModes attribute), 40
_traj (gaudi.genes.trajectory.Trajectory attribute), 47

A

acceptor (gaudi.genes.molecule.Compound attribute), 33
add_dummy_atom() (gaudi.genes.molecule.Compound method), 34
add_hydrogens() (gaudi.genes.molecule.Compound method), 34
add_hydrogens_to_isolated_rotamer() (gaudi.genes.mutamers.Mutamers static method), 38
affine_cpd() (in module gaudi._cpdrift), 73
affine_xform() (in module gaudi._cpdrift), 73
alg3() (in module gaudi.genes.normalmodes), 41
all_chis() (gaudi.genes.rotamers.Rotamers static method), 43
allele (gaudi.genes.molecule.Molecule attribute), 36
allele (gaudi.genes.mutamers.Mutamers attribute), 38
allele (gaudi.genes.normalmodes.NormalModes attribute), 40
allele (gaudi.genes.rotamers.Rotamers attribute), 43
allele (gaudi.genes.search.Search attribute), 44
allele (gaudi.genes.torsion.Torsion attribute), 46
anchor (gaudi.genes.torsion.Torsion attribute), 46
Angle (class in gaudi.objectives.angle), 49
append() (gaudi.genes.molecule.Compound method), 34
apply_pdbfix() (gaudi.genes.molecule.Compound method), 34
args (gaudi.parse.Settings.similarity attribute), 71
AssertList() (in module gaudi.parse), 68
atom (gaudi.parse.MoleculeAtom attribute), 68
atoms() (gaudi.objectives.distance.Distance method), 52
atoms_between() (in module gaudi.box), 66
atoms_by_serial() (in module gaudi.box), 66
AtomsNotFound, 67
attach() (gaudi.genes.molecule.Compound method), 34
AxesOfInertia (class in gaudi.objectives.inertia), 57
axis_angle (gaudi._cpdrift.Quaternion attribute), 73

B

BaseIndividual (class in gaudi.base), 64
BONDS_ROT (gaudi.genes.torsion.Torsion attribute), 46
build() (gaudi.genes.molecule.Molecule method), 37
built_atoms (gaudi.genes.molecule.Compound attribute), 33

calculate_alignment() (in module gaudi.objectives.inertia), 58
calculate_axes_of_inertia() (in module gaudi.objectives.inertia), 58
calculate_energy() (gaudi.objectives.energy.Energy method), 55
calculate_energy() (in module gaudi.objectives.energy), 55
calculate_inertial_matrix() (in module gaudi.objectives.inertia), 58
calculate_prody_normal_modes() (gaudi.genes.normalmodes.NormalModes method), 40
center (gaudi.genes.search.Search attribute), 44
evaluate() (gaudi.objectives.angle.Angle method), 49
evaluate() (gaudi.objectives.coordination.Coordination method), 51
evaluate() (gaudi.objectives.dsx.DSX method), 54
evaluate() (gaudi.objectives.energy.Energy method), 55
evaluate() (gaudi.objectives.gold.Gold method), 56
evaluate() (gaudi.objectives.hbonds.Hbonds method), 57
evaluate() (gaudio.objectives.inertia.AxesOfInertia method), 58
evaluate() (gaudio.objectives.ligscore.LigScore method), 59
evaluate() (gaudio.objectives.nwchem.NWChem method), 59
evaluate() (gaudio.objectives.ObjectiveProvider method), 63
evaluate_area() (gaudio.objectives.solvation.Solvation method), 60
evaluate_center_of_mass() (gaudio.objectives.distance.Distance method), 52
evaluate_clashes() (gaudio.objectives.contacts.Contacts method), 50
evaluate_convexhull() (gaudio.objectives.volume.Volume method), 62
evaluate_distances() (gaudio.objectives.distance.Distance method), 52
evaluate_hydrophobic() (gaudio.objectives.contacts.Contacts method), 50
evaluate_volume() (gaudio.objectives.solvation.Solvation method), 60
evaluate_volume() (gaudio.objectives.volume.Volume method), 62
ExpandUserPathExists() (in module gaudi.parse), 68
express() (gaudio.genes.search.Search method), 44
express() (gaudio.genes.torsion.Torsion method), 46
express() (gaudio.genes.trajectory.Trajectory method), 47
expressed() (in module gaudi.base), 66
express() (gaudio.genes.molecule.Molecule method), 37
express() (gaudio.genes.mutamers.Mutamers method), 39
express() (gaudio.genes.normalmodes.NormalModes method), 40
express() (gaudio.genes.rotamers.Rotamers method), 43
express() (in module gaudi.box), 66
express() (gaudio.genes.molecule.Molecule method), 37
express() (gaudio.genes.normalmodes.NormalModes method), 40
express() (gaudio.genes.rotamers.Rotamers method), 43
Fitness (class in gaudi.base), 65
frange (class in gaudi._cpdrift), 73
find_atom() (gaudio.genes.molecule.Molecule method), 37
find_atoms() (gaudio.genes.molecule.Molecule method), 37
find_interactions() (gaudio.objectives.contacts.Contacts method), 50
find_molecule() (gaudio.base.MolecularIndividual method), 65
find_nearest() (in module gaudi.box), 67
find_residue() (gaudio.genes.molecule.Molecule method), 37
find_residues() (gaudio.genes.molecule.Molecule method), 37
files_in() (in module gaudi.box), 66
fromAxisAngle() (gaudio._cpdrift.Quaternion class method), 73
G

ga (gaudio.parse.Settings attribute), 70
gaudi (module), 31
gaudi._cpdrift (module), 72
gaudi.algorithms (module), 63
gaudi.base (module), 64
gaudi.box (module), 66
gaudi.cli (module), 31
gaudi.cli.gaudi_cli (module), 31
gaudi.cli.gaudi_run (module), 32
gaudi.exceptions (module), 67
gaudi.genes (module), 48
gaudi.genes.molecule (module), 32
gaudi.genes.mutamers (module), 38
gaudi.genes.normalmodes (module), 39
gaudi.genes.rotamers (module), 42
gaudi.genes.search (module), 43
gaudi.genes.torsion (module), 45
gaudi.genes.trajectory (module), 47
gaudio.objectives (module), 62
gaudio.objectives.angle (module), 49
gaudio.objectives.contacts (module), 49
gaudio.objectives.coordination (module), 50

Index
molecules() (gaudi.objectives.solvation.Solvation method), 60
MoleculesNotFound, 67
mu (gaudi.parse.Settings.ga attribute), 70
mut_eta (gaudi.parse.Settings.ga attribute), 70
mut_indpb (gaudi.parse.Settings.ga attribute), 70
mut_pb (gaudi.parse.Settings.ga attribute), 70
Mutamers (class in gaudi.genes.mutamers), 38
mutate() (gaudi.base.BaseIndividual method), 65
mutate() (gaudi.genes.GeneProvider method), 48
mutate() (gaudi.genes.molecule.Molecule method), 38
mutate() (gaudi.genes.mutamers.Mutamers method), 39
mutate() (gaudi.genes.normalmodes.NormalModes method), 41
mutate() (gaudi.genes.rotamers.Rotamers method), 43
mutate() (gaudi.genes.search.Search method), 44
mutate() (gaudi.genes.torsion.Torsion method), 46
mutate() (gaudi.genes.trajectory.Trajectory method), 48

N
name (gaudi.parse.Settings.output attribute), 69
name_objectives (gaudi.parse.Settings attribute), 71
Named_spec() (in module gaudi.parse), 69
nearest_atom() (in module gaudi.genes.search), 45
nearest_atom() (in module gaudi.genes.torsion), 47
nonrotatable (gaudi.genes.molecule.Compound attribute), 33
norm() (gaudi._cpdrift.Quaternion method), 73
NORMAL_MODE_SAMPLES
(gaudi.genes.normalmodes.NormalModes attribute), 40
NORMAL_MODES (gaudi.genes.normalmodes.NormalModes attribute), 40
NORMAL_MODES_SAMPLES
(gaudi.genes.normalmodes.NormalModes attribute), 40
NormalModes (class in gaudi.genes.normalmodes), 39
NWChem (class in gaudi.objectives.nwchem), 59

O
ObjectiveProvider (class in gaudi.objectives), 62
objectives (gaudi.parse.Settings attribute), 71
open_models_and_close() (in module gaudi.box), 67
origin (gaudi.genes.molecule.Compound attribute), 33
origin (gaudi.genes.search.Search attribute), 44, 45
origin() (gaudi.objectives.gold.Gold method), 56
output (gaudi.parse.Settings attribute), 69

P
pairwise_sqdist() (in module gaudi._cpdrift), 73
pareto (gaudi.parse.Settings.output attribute), 69
parse_attr() (gaudi.genes.molecule.Compound method), 35
parse_origin() (in module gaudi.genes.search), 45
parse_output() (gaudi.objectives.dsx.DSX method), 54
parse_output() (gaudi.objectives.gold.Gold method), 56
parse_output() (gaudi.objectives.ligscore.LigScore method), 59
parse_output() (gaudi.objectives.nwchem.NWChem method), 60
parse_output() (gaudi.objectives.vina.Vina method), 61
parse_rawstring() (in module gaudi.parse), 71
patch_residue() (gaudi.genes.rotamers.Rotamers static method), 43
path (gaudi.parse.Settings.output attribute), 69
place() (gaudi.genes.molecule.Compound method), 35
place_for_bonding()
(gaudi.genes.molecule.Compound method), 36
plot() (in module gaudi._cpdrift), 73
PluginMount (class in gaudi.plugin), 71
plugins (gaudi.genes.GeneProvider attribute), 48
plugins (gaudi.objectives.ObjectiveProvider attribute), 63
population (gaudi.parse.Settings.ga attribute), 70
post_express() (gaudi.base.BaseIndividual method), 65
post_express() (gaudi.base.MolecularIndividual method), 66
post_unexpress() (gaudi.base.BaseIndividual method), 65
pre_express() (gaudi.base.BaseIndividual method), 65
pre_unexpress() (gaudi.base.BaseIndividual method), 65
precision (gaudi.parse.Settings.output attribute), 69
prepare_command() (gaudi.objectives.dsx.DSX method), 54
prepare_command() (gaudi.objectives.gold.Gold method), 56
prepare_command() (gaudi.objectives.ligscore.LigScore method), 59
prepare_ligand() (gaudi.objectives.vina.Vina method), 61
prepare_ligands() (gaudi.objectives.dsx.DSX method), 54
prepare_ligands() (gaudi.objectives.gold.Gold method), 56
prepare_ligands()
(gaudi.objectives.ligscore.LigScore method), 59
Index 85