structopt Documentation

Release 0.5

UW Madison, Materials Science

Sep 05, 2017

Contents

1 What is reverse structure determination? 3

2 Overview of StructOpt 5

3 Contents 11

4 Contributing 57

5 License Agreement 59

6 Index and Search 61

StructOpt is a reverse structure determination toolkit.

CHAPTER 1

What is reverse structure determination?

Reverse structure determination is a type of structural refinement that iteratively modifies and optimizes a structural model. In atomistic simulations, the positions of the atoms are moved within the model at every step. After the atoms have moved, the structure is evaluated to see how "good" it is. If the structure is "better" than the previous step, the moved atoms are more likely to persist into the next generation. This process is repeated many times until acceptable structure(s) have been generated.

Many different metrics can be used to determine how "good" a structure is, and this is often material-dependent. The average energy per atom is one commonly used metric, and others include fits to experimental data (e.g. S(q) or g(r) data), medium-range order information available via FEM measurements, average coordination number, and bond angle constraints.

The specific algorithms that can be used in reverse structure determination are numerous and include Monte Carlo, Genetic Algorithm, and Particle Swarm.

CHAPTER 2

Overview of StructOpt

2.1 User Documentation

StructOpt is a structure optimization framework that incorportates multiple forward simulation techniques into its optimization scheme with the goal of identifying stable and realistic atomic structures. It is designed with modularity in mind, and encourages simplicity in both its codebase and usage without sacrificing powerful functionality. Nearly any forward simulation technique that takes an atomic model as input and outputs a fitness value can be integrated into this framework.

This documentation serves as both a user and developer guide for StructOpt.

StructOpt serves the purpose of structure refinment for multiple different materials including nanoparticles, defects, and metallic glasses. As such, it is highly customizable and extendable. There are many different types of simulations that can be set up, which requires getting to know the relevent parameters. Multiple examples are included in this documentation and comments (via issues on our github page) are welcome.

The *Examples* section provides multiple examples of basic StructOpt configurations. Details on the inputs and outputs can be found in the parameters and *Outputs* sections, respectively. Details on the many options currently available for StructOpt are provided in the parameters section of this document. An explanation of commonly generated errors and troubleshooting advice is provided in the section entitled *Troubleshooting*.

2.2 Developer Documentation

2.2.1 Architecture Details

Core Concepts

Overview and General Workflow

StructOpt uses a Genetic Algorithm to optimize a set of atomic structures according to a customizable objective function (aka cost function).

Genetic Algorithm

A genetic algorithm utilizes a population of structures rather than a single individual. A genetic algorithm, or evolutionary algorithm, is conceptually similar to genetic Darwinism where animals are replaced by "individuals" (in the case of StructOpt a "individual" is an atomic model). A population of atomic models is first generated. Given a population, pairs of individuals are mated (aka crossed over) by selecting different aspects of each model and pasting them into each other. Crossovers always produce two children, one for each section of the models combined together. The offspring are added to the population. After the mating scheme has finished, single individuals can "mutate" (i.e. moving atoms in a unique way) to add new genes to the population's gene pool. After the atoms have been moved via crossovers and mutations, the structures are relaxed. Finally, each structure is run though a series of "fitness" evaluations to determine how "fit to survive" it is, and the population is then reduced to its original size based on a number of optional selection criteria. This process is repeated many times.

In summary:

1. Generate initial structures
2. Locally relax structures
3. Calculate fitness values (e.g. energies) of each structure
4. Remove some individuals from the population based on their fitness value
5. Perform crossovers and selected individuals to generate offspring for the next generation
6. Perform mutations on the selected individuals in the current population and offspring for the next generation
7. Repeat steps 2-6 until the convergence criteria are met

The relaxation and fitness evaluations are only performed on individuals that have been modified via crossovers and mutations. This avoids recomputing these expensive calculations for individuals that were unchanged during the generation's crossover/mutation scheme.

During crossovers, the offspring are collected into a list. After all crossovers have been completed, these offspring are added to the entire population. Each individual in the entire population then has a chance to be mutated. There will therefore be a variable number of modified individuals that will need to be relaxed and fit during each generation. The number of modified individuals can only be predicted by using the probability of mutation and crossover.

Cost Function

Individual

Structure Types

Crystal

Implemented by default.

Has periodic boundary conditions along all dimensions. The entire model is relaxed.

Cluster

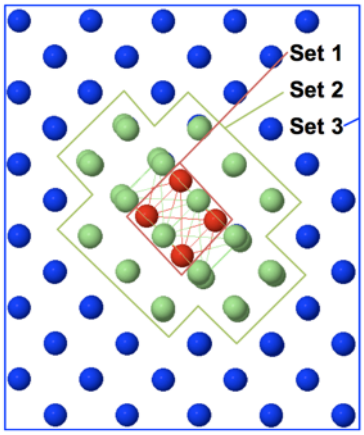
Not implemented.

Does not have periodic boundary conditions. The entire model is relaxed.

Defect

Not implemented.

There are three layers to this structure type. The outer layer contains the fixed atoms that are never moved but are used as constrains in the relaxations and fitnesses. The middle layer contains atoms that are part of the crystal structure but that will be mutated, crossed-over, and relaxed. The inner layer contains the defect and it will also be mutated cross-over, and relaxed.



Surface

Not implemented.

Population

Crossovers

Crossover Selection Schemes

Mutations

Predators and Predator Selection Schemes

Roulette Wheel Selection

Fingerprinters

Relaxations

Fitnesses

Relevant References

- Crystals: Artem Oganov, Alex Zunger, Scott Woodley, Richard Catlow

- Clusters: Roy L. Johnston, Bernd Hartke, David Deaven
- Surfaces: Cristian V. Ciobanu, Kai-Ming Ho

Parallelism

In general, the only parallelized parts of StructOpt are the fitness and relaxation modules that can be plugged in.

StructOpt has two parallelization mechanisms. The first is the simplest case where each structure is assigned to a single core. The core does the significant processing for one structure by processing the module's code. This is optimal when the module does not implement MPI, or the code is relatively fast.

The second parallelization method, called MPMD (via `MPI_Comm_spawn_multiple`), is a type of advanced dynamic process management but remains relatively easy to use within StructOpt. It allows MPI code to be used within modules and for those modules to be processes on an arbitrary number of cores.

For functions that are only run on the root core (e.g. crossovers and mutations), the root decorator is used on the main `fitness` or `relaxation` function to broadcast the return value of the function to all cores.

StructOpt acts as a master process ("master program" may be a better word) that runs in Python and uses MPI (via mpi4py) to communicate between cores. This master process/program makes `MPI_Comm_spawn_multiple` calls to C and Fortran programs (which also use MPI). While the C and Fortran processes run, the master python program waits until they are finished. As an example in this section, we will assume StructOpt is using 16 cores to do calculations on 4 structures.

In terms of MPMD parallelization, StructOpt does two primary things:

1. Uses MPI to do preprocessing for the spawning in step (2). `MPI_Barrier` is called after this preprocessing to ensure that all ranks have finished their preprocessing before step (2) begins. Note that the preprocessing is distributed across all 16 cores (via the one-core-per structure parallelism using `mpi4py`), and at the end of the preprocessing the resulting information is passed back to the root rank (e.g. rank == 0).
2. After the preprocessing, the root rank spawns 4 workers, each of which use 4 cores (i.e. all 16 cores are needed to run all 4 processes at the same time). These workers are spawned through either a relaxation or fitness evaluation module, which is done via `MPI_Comm_spawn_multiple`. These workers can use MPI to communicate within their 4 cores. In the master StructOpt program, only the root rank spawns the C or Fortran subprocesses, and the modules wait until the spawned processes finish before they continue execution.
3. Step (1) and (2) are repeated until the convergence criteria are satisfied.

Cores per Structure Use Cases

- `ncores == len(population)`: One core per structure
- `ncores < len(population)`: One core per structure, but all the structure cannot be run at once
- `ncores > len(population)`: Multiple cores per structures

Unfortunately, it is impossible to predict the number of structures that will be need to be relaxed and fitted after crossovers and mutations have been performed on the population. As a result, all of the above cases are possible (and probable) for any given simulation.

mpi4py: One structure per core

Main idea: One structure per core, or multiple structures per core that execute serially in a for-loop. The module must be written in python (or callable from python like LAMMPS through ASE) and implemented directly into StructOpt.

mpi4py allows MPI commands to be run within python.

Installation

TODO: Change OpenMPI 1.10.2 to the correct version after the bugfixes have been made. In the meantime, use `-mca btl tcp,sm,self` to use TCP rather than infiniband.

Note: mpi4py needs to be installed from source against OpenMPI 1.10.2. Follow the instructions here under "3.3: Using distutils". In short:

```
# Setup modules so that `mpi/intel/openmpi` is loaded and `mpiexec` finds that␣
↪executable
wget https://bitbucket.org/mpi4py/mpi4py/downloads/mpi4py-X.Y.tar.gz
tar -zxf mpi4py-X.Y.tar.gz
cd mpi4py-X.Y
python setup.py build
python setup.py install --user
```

MPMD: Multiple cores per structure

Multiple program, multiple data (MPMD) is a form of MPI parallelization where multiple MPI communicators are used synchonously to run multiple MPI processes at the same time. MPMD can be used within `mpiexec` by separating each command by colons. Each command is preceded by the `-n` option whcih specifies the number of cores to be used for that executable. MPMD can also be used from another MPI master process which calls `MPI_Comm_spawn_multiple`. This is how StructOpt implements its advanced parallelization techniques to integrate MPI relaxation and fitness programs into its framework. The executable needs to implement MPMD by disconnecting a parent process if it exists (see here and here for an example parent/child implementation).

CHAPTER 3

Contents

3.1 Installation and Setup

StructOpt is written in Python 3 and as such requires a working Python 3 installation. We recommend setting up an Anaconda virtual environment exclusively for StructOpt.

3.1.1 Python Libraries

```
conda install numpy
conda install scipy
pip install ase
pip install natsorted
# Install mpi4py from source (below)
```

mpi4py

On Madison's ACI cluster:

```
module load compile/intel
module load mpi/intel/openmpi-1.10.2
```

Follow these instructions:

```
wget https://bitbucket.org/mpi4py/mpi4py/downloads/mpi4py-X.Y.tar.gz
tar -zxf mpi4py-X.Y.tar.gz
cd mpi4py-X.Y
python setup.py build
python setup.py install --user
```

You can test your installation by following these instructions.

3.1.2 Installing StructOpt

To get the code, fork and clone the StructOpt repository or download the zip here. Add the location of the StructOpt folder (e.g. `$HOME/repos/StructOpt`) to your `PATH` environment variable.

Create an environment variable called `STRUCTOPT_HOME` with the same folder location as you added to your path.

3.1.3 Additional Modules

Depending on the type of calculation you wish to run, you will need to install specific relaxation and fitness modules. VASP and LAMMPS are two examples that function as both a relaxation and fitness. They can both move atoms and relax the structure, and they output an energetic term that quantifies how "good" or fit the input structure is. These modules should be installed using their standard installation procedures unless specified otherwise in the **'Modules <>'_** section.

3.2 Input Parameters

The input parameters is a JSON file, which holds a single dictionary. Due to the modular nature of StructOpt, the input file is a dictionary of embedded dictionary, where dictionary key and values often determine the function name and kwargs, respectively. An example Au nanoparticle input file is given below. Each part will be discussed in the following sections.

Example:

```
{
  "seed": 0,
  "structure_type": "cluster",
  "generators": {
      "fcc": {"number_of_individuals": 5,
              "kwargs": {"atomlist": [["Au", 55]],
                         "orientation": "100",
                         "cell": [20, 20, 20],
                         "a": 4.08}}
  },
  "fitnesses": {
      "LAMMPS": {"weight": 1.0,
                 "kwargs": {"use_mpi4py": false,
                            "MPMD": 0,
                            "keep_files": true,
                            "min_style": "cg",
                            "min_modify": "line quadratic",
                            "minimize": "1e-8 1e-8 5000 10000",
                            "pair_style": "eam",
                            "potential_file": "$STRUCTOPT_HOME/potentials/Au_u3.eam",
                            "thermo_steps": 0,
                            "reference": {"Au": -3.930}}}
  },
  "relaxations": {
      "LAMMPS": {"order": 0,
                 "kwargs": {"use_mpi4py": false,
                            "MPMD": 0,
                            "keep_files": true,
                            "min_style": "cg",
                            "min_modify": "line quadratic",
                            "minimize": "1e-8 1e-8 5000 10000",
```

```
                            "pair_style": "eam",
                            "potential_file": "$STRUCTOPT_HOME/potentials/Au_u3.eam",
                            "thermo_steps": 0}}
  },
  "convergence": {
      "max_generations": 10
  },
  "mutations": {
      "move_surface_atoms": {"probability": 0.0},
      "move_atoms": {"probability": 0.0},
      "move_atoms_group": {"probability": 0.0},
      "rotate_atoms": {"probability": 0.0},
      "rotate_cluster": {"probability": 0.0},
      "rotate_all": {"probability": 0.0},
      "move_surface_defects": {"probability": 1.0}
  },
  "crossovers": {
      "rotate": {"probability": 0.7,
                 "kwargs": {"repair_composition": true}}
  },
  "predators": {
      "fuss": {"probability": 0.9},
      "diversify_module": {"probability": 0.1,
                           "kwargs": {"module": "LAMMPS",
                                      "min_diff": 0.01}}
  },
  "selections": {
      "rank": {"probability": 1.0}
  }
}
```

3.2.1 Global Parameters

Global parameters are those that determine how the optimizer should run.

structure_type

`structure_type` (str) is a key parameter for determining operations will be run. Currently, only cluster is supported, but StructOpt is written in a way that *how* the operations are carried out is seperated from the operations themselves. Hence, one can easily write new crossover, mutation, selection, and predator operations that are unique to their structure, test them, and incorporate them inside StructOpt seamlessly.

seed

`seed` (int): seed for the psuedo-random number generator. Two runs with the exact same input *and* seed should run exactly the same way. Almost all operations use random number generators. See should be an int.

convergence

`convergence` (dict): Convergence is a dictionary that determines when to stop the calculation. Currently, the only convergence criteria is `max_generations`, which is set to an int. For example, the setting below runs the optimizer for 200 generations.

Example:

```
"convergence": {
    "max_generations": 200
}
```

In the future, more `convergence` options will be added.

post_processing

`post_processing` (dict): Determines what is output as the optimizer is run. Currently, the only option is `XYZs`, which determines how frequentely the xyz files of each generation should be printed. The rules for this are as follows.

- `XYZs` = 0: all generations are kept
- `XYZs` > 0: every `XYZs` generation is kept
- `XYZs` < 0: the last `XYZs` generations are kept

The below example is a run where only the last generation is kept. This behavior is by default and encouraged for saving space.

Example:

```
"post_processing": {
    "XYZs": -1
}
```

3.2.2 Generators

Generators are functions for initializing the population. These are pseudo-random generators that depend on the `seed` global parameter.

Generators are given as a dictionary entry defined by the `generators` key in the input file. The structure of the generators dictionary with *N* desired generators is given below.

Example:

```
"generators": {
    generator_1: {"number_of_individuals": n_1,
                  "kwargs": kwargs_1}
    generator_2: {"number_of_individuals": n_2,
                  "kwargs": kwargs_2}
    generator_3: {"number_of_individuals": n_3,
                  "kwargs": kwargs_3}
    ...
    generator_N: {"number_of_individuals": n_N,
                  "kwargs": kwargs_N}
}
```

The string for *generator_i*, is the name of the generator one wants to use. The number of individuals that generator should generate is determined by the integer *n_i*. The sum of all *n_i* values determines the total size of the population, which is fixed throughout the run. *kwargs_i* are dictionaries that input the kwargs to the generator function one is using. These will be specific to the function and can be found in their help function, show below.

structopt.cluster.individual.generators.**ellipsoid**(*atomlist, fill_factor=0.74, radii=None, ratio=[1, 1, 1], cell=None*)

Generates a random ellipsoid by rejection sampling.

Parameters

- **atomlist** (*list*) – A list of [sym, n] pairs where sym is the chemical symbol and n is the number of of sym's to include in the individual
- **fill_factor** (*float*) – Determines how "close" the atoms are packed. See structopt.tools.get_particle_radius for description
- **radii** (*list*) – The size, in angstroms, of the ellipsoid in the x, y and z direction. If None, with ratio parameters and the average atomic radii, radii is automatically calculated.
- **ratio** (*list*) – The ratio of the dimensions of the ellipsoid in the x, y and z direction.
- **cell** (*list*) – The size, in angstroms, of the dimensions that holds the atoms object. Must be an orthogonal box.

structopt.cluster.individual.generators.**sphere**(*atomlist, cell, fill_factor=0.74, radius=None*)

Generates a random sphere of particles given an atomlist and radius.

Parameters

- **atomlist** (*list*) – A list of [sym, n] pairs where sym is the chemical symbol and n is the number of of sym's to include in the individual
- **cell** (*list*) – The x, y, and z dimensions of the cell which holds the particle.
- **fill_factor** (*float*) – How densely packed the sphere should be. Ranges from 0 to 1.
- **radius** (*float*) – The radius of the sphere. If None, estimated from the atomic radii

structopt.cluster.individual.generators.**fcc**(*atomlist, cell, a, shape=[1, 1, 1], orientation=None, size=21, roundness=0.5, alpha=10, v=None, angle=None*)

Generates a fcc nanoparticle of varying shape and orientation. For multi-component particles, elements are randomly distributed.

Parameters

- **atomlist** (*list*) – A list of [sym, n] pairs where sym is the chemical symbol and n is the number of of sym's to include in the individual
- **cell** (*list*) – A 3 element list that defines the x, y, and z dimensions of the simulation box
- **a** (*float*) – fcc lattice constant
- **shape** (*list*) – The ratio of the x, y, and z dimensions of the particle.
- **orientation** (*str*) – The facet that is parallel to the xy plane. This is useful for LAMMPS+STEM calculations where one already knows the orientation. If None, a random orientation is chosen.
- **size** (*int*) – Size of the fcc grid for building the nanoparticle. For example, a size of 21 means 21 x 21 x 21 supercell of fcc primitive cells will be used. Note, if the size is too small, the function will automatically expand the cell to fit the particle.
- **roundness** (*float*) – Determines the "roundness" of the particle. A more round particle will have a smaller surface area to volume ratio, but more undercoordinated surface sites. A less round particle will take more and more the form of an octahedron.

- **alpha** (*float*) – Parameter for determining how defective the particle will be. Higher alpha means less defective particle.
- **v** (*list*) – Used for a custom orientation. V is the vector in which to rotate the particle. Requires angle parameters to be entered. All rotations are done with respect to the 100 plane.
- **angle** (*float*) – Angle, in radians, to rotate atoms around vector v. All rotations are done with respect to the 100 plane.

3.2.3 Crossovers

Crossovers are operations for mating two individuals chosen by a selection algorithm. The purpose of the crossover is to intelligently combine (mate) different individuals (parents) in a way to create new individuals (children) that have the features of the current best individuals in the population.

Crossovers are given as a dictionary entry defined by the `crossovers` key in the input file. The structure of the crossovers dictionary with *N* desired selections is given below.

Example:

```
"crossovers": {
    crossover_1: {"probability": p_1,
                  "kwargs": kwargs_1}
    crossover_2: {"probability": p_2,
                  "kwargs": kwargs_2}
    crossover_3: {"probability": p_3,
                  "kwargs": kwargs_3}
    ...
    crossover_N: {"probability": p_N,
                  "kwargs": kwargs_N}
}
```

The string for *crossover_i*, is the name of the crossover one wants to use. The probability *p_i* is the probability of the crossover occuring if a mate is determined to happen in the population. *p_i* values should sum to 1. *kwargs_i* are dictionaries that input the kwargs to the crossover function one is using. These will be specific to the function and can be found in their help function.

Currently the only crossover in use in the algorithm is the cut-and-splice operator introduced by Deaven and Ho. The description is shown below.

`structopt.cluster.population.crossovers.`**rotate**(*individual1*, *individual2*, *center_at_atom=True*, *repair_composition=True*)

Rotates the two individuals around their centers of mass, splits them in half at the xy-plane, then splices them together. Maintains number of atoms. Note, both individuals are rotated in the same way.

Parameters

- **individual1** (Individual) – The first parent
- **individual2** (Individual) – The second parent
- **center_at_atom** (*bool*) – This centers the cut at an atom. This is particularly useful when one desires a crystalline solution. If both parents are crystalline, the children will likely not have grain boundaries.
- **repair_composition** (*bool*) – If True, conserves composition. For crossovers that create children with more (less) atoms, atoms are taken from (added to) the surface of the

particle. For incorrect atomic ratios, atomic symbols are randomly interchanged throughout the particle

Returns

- **Individual** (*The first child*)
- **Individual** (*The second child*)

3.2.4 Selections

Selections are operations for choosing which individuals to "mate" in producing new individuals. Individuals are chosen based on their fitness, and different selection functions determine how diverse the children should be.

Selections are given as a dictionary entry defined by the `selections` key in the input file. The structure of the selections dictionary with *N* desired selections is given below.

Example:

```
"selections": {
    selection_1: {"probability": p_1,
                  "kwargs": kwargs_1}
    selection_2: {"probability": p_2,
                  "kwargs": kwargs_2}
    selection_3: {"probability": p_3,
                  "kwargs": kwargs_3}
    ...
    selection_N: {"probability": p_N,
                  "kwargs": kwargs_N}
}
```

The string for *selection_i*, is the name of the selection one wants to use. The probability *p_i* is the probability of the selection occuring if a mate is determined to happen in the population. *p_i* values should sum to 1. *kwargs_i* are dictionaries that input the kwargs to the selection function one is using. These will be specific to the function and can be found in their help function.

`structopt.common.population.selections.`**`tournament`**(*population*, *fits*, *tournament_size=5*, *unique_pairs=False*, *unique_parents=False*, *keep_best=False*)

Selects pairs in seperate "tournaments", where a subset of the population are randomly selected and the highest fitness allowed to pass. In addition to a population, their fits, and end population size, takes in a tournament size parameter.

Parameters

- **`population`** (`Population`) – The population of individuals needed to be trimmed
- **`fits`** (*`list`*) – List of fitnesses that correspond to the population.
- **`tournament_size`** (*`int`*) – The number of individuals in each tournament. If 1, tournament is the same as random selection. If len(population), corresponds to the "best" selection process
- **`unique_pairs`** (*`bool`*) – If True, all combinations of parents are unique, though parents can show up in different pairs. True increases the diversity of the population.
- **`unique_parents`** (*`bool`*) – If True, all parents can only mate with on other individual. True increases the diversity of the population.

structopt.common.population.selections.**random_selection**(*population*, *fits*)
Randomly selects parents

Parameters

- **population** (Population) – An population of individuals
- **fits** (*list*) – Fitnesses that corresponds to population

structopt.common.population.selections.**best**(*population*, *fits*)
Deterministic selection function that chooses adjacently ranked individuals as pairs.

Parameters

- **population** (Population) – An population of individuals
- **fits** (*list*) – Fitnesses that corresponds to population

structopt.common.population.selections.**rank**(*population*, *fits*, *p_min=None*, *unique_pairs=False*, *unique_parents=False*)
Selection function that chooses pairs of structures based on linear ranking.

See "Grefenstette and Baker 1989 Whitley 1989".

Parameters

- **population** (Population) – An object inherited from list that contains StructOpt individual objects.
- **fits** (*list*) – A list of fitnesses of the population
- **p_min** (*float*) – The probability of choosing the lowest ranked individual. Given population of size N, this should be below 1/nindiv. The probability of selecting rank N (worst) to rank 1 (best) increases from p_min to (2/N - p_min) in even, (1/N - p_min) increments. Defaults to (1/N)^2.
- **unique_pairs** (*bool*) – If True, all combinations of parents are unique. True increases the diveristy of the population.
- **unique_parents** (*bool*) – If True, all parents can only mate with on other individual. True increases the diversity of the population.

structopt.common.population.selections.**roulette**(*population*, *fits*, *unique_pairs=False*, *unique_parents=False*)
Selection function that chooses pairs of structures based on their fitness. Fitnesses are normalized from 0 to 1.

See "Grefenstette and Baker 1989 Whitley 1989".

Parameters

- **population** (*StructOpt population object*) – An object inherited from list that contains StructOpt individual objects.
- **fits** (*list*) – A list of fitnesses of the population
- **unique_pairs** (*bool*) – If True, all combinations of parents are unique. True increases the diveristy of the population.
- **unique_parents** (*bool*) – If True, all parents can only mate with on other individual. True increases the diversity of the population.

3.2.5 Predators

Similar to selections, predators are selection processes that selects individuals based on their fitness. The distinction is that while selections select individuals with positive features to duplicate in children, predators select which individuals to keep in the next generation. Note, this must be done because crossovers and (sometimes) mutations increase the population every generation, and hence each generation requires a predator step.

Predators are given as a dictionary entry defined by the `predators` key in the input file. The structure of the predators dictionary with *N* desired predators is given below

Example:

```
"predators": {
    predator_1: {"probability": p_1,
                 "kwargs": kwargs_1}
    predator_2: {"probability": p_2,
                 "kwargs": kwargs_2}
    predator_3: {"probability": p_3,
                 "kwargs": kwargs_3}
    ...
    predator_N: {"probability": p_N,
                 "kwargs": kwargs_N}
}
```

The string for *predator_i*, is the name of the predator one wants to use. The probability *p_i* is the probability of the predator occuring on every individual in the population. *p_i* values should sum to 1. *kwargs_i* are dictionaries that input the kwargs to the predator function one is using. These will be specific to the function and can be found in their help function.

`structopt.common.population.predators.`**`best`**(*fits*, *nkeep*)

Sorts individuals by fitness and keeps the top nkeep fitnesses.

Parameters

- **`fits`** (*dict<int, float>*) – Dictionary of <individual.id, fitness> pairs.
- **`nkeep`** (*int*) – The number of individuals to keep. In a GA run, corresponds to the sum of each generators number_of_individuals

`structopt.common.population.predators.`**`roulette`**(*fits*, *nkeep*, *T=None*)

Select individuals with a probability proportional to their fitness. Fitnesses are renormalized from 0 - 1, which means minimum fitness individual is never included in in the new population.

Parameters

- **`fits`** (*dict<int, float>*) – Dictionary of <individual.id, fitness> pairs.
- **`nkeep`** (*int*) – The number of individuals to keep. In a GA run, corresponds to the sum of each generators number_of_individuals
- **`T`** (*float*) – If T is not None, a boltzman-like transformation is applied to all fitness values with T.

`structopt.common.population.predators.`**`rank`**(*fits*, *nkeep*, *p_min=None*)

Selection function that chooses pairs of structures based on linear ranking.

See “Grefenstette and Baker 1989 Whitley 1989”.

Parameters

- **`fits`** (*dict<int, float>*) – Dictionary of <individual.id, fitness> pairs.

- **nkeep** (*int*) – The number of individuals to keep. In a GA run, corresponds to the sum of each generators number_of_individuals
- **p_min** (*float*) – The probability of choosing the lowest ranked individual. Given population of size N, this should be below 1/nindiv. The probability of selecting rank N (worst) to rank 1 (best) increases from p_min to (2/N - p_min) in even, (1/N - p_min) increments. Defaults to (1/N)^2.

structopt.common.population.predators.**fuss** (*fits*, *nkeep*, *nbest=0*, *fusslimit=10*)

Fixed uniform selection scheme. Aimed at maintaining diversity in the population. In the case where low fit is the highest fitness, selects a fitness between min(fits) and min(fits) + fusslimit, if the difference between the min(fit) and max(fit) is larger than fusslimit.

Parameters

- **fits** (*dict<int, float>*) – Dictionary of <individual.id, fitness> pairs.
- **nkeep** (*int*) – The number of individuals to keep. In a GA run, corresponds to the sum of each generators number_of_individuals
- **nbest** (*int*) – The top n individuals to always keep (default 0)
- **fusslimit** (*float*) – Individuals that have fitness fusslimit worse than the max fitness will not be considered

structopt.common.population.predators.**tournament** (*fits*, *nkeep*, *tournament_size=5*)

Selects individuals in seperate "tournaments", where a subset of the population are randomly selected and the highest fitness allowed to pass. In addition to a population, their fits, and end population size, takes in a tournament size parameter.

Parameters

- **fits** (*dict<int, float>*) – Dictionary of <individual.id, fitness> pairs.
- **nkeep** (*int*) – The number of individuals to keep. In a GA run, corresponds to the sum of each generators number_of_individuals
- **tournament_size** (*int*) – The number of individuals in each tournament. If 1, tournament is the same as random selection. If len(population), corresponds to the "best" selection process

3.2.6 Mutations

Mutations are operations applied to individuals that change its structure and composition. It is a local search operation, though the mutation itself can be written to perform small or larger changes.

Mutations are given as a dictionary entry defined by the `mutations` key in the input file. The structure of the mutations dictionary with *N* desired mutations is given below

Example:

```
"mutations": {
    "preserve_best": "true" or "false",
    "keep_original": "true" or "false",
    "keep_original_best": "true" or "false,
    mutation_1: {"probability": p_1,
                 "kwargs": kwargs_1}
    mutation_2: {"probability": p_2,
                 "kwargs": kwargs_2}
    mutation_3: {"probability": p_3,
                 "kwargs": kwargs_3}
```

```
    ...
    mutation_N: {"probability": p_N,
                 "kwargs": kwargs_N}
}
```

The string for *mutation_i*, is the name of the mutation one wants to use. The probability *p_i* is the probability of the mutation occuring on every individual in the population. *p_i* values should sum to any value between 0 and 1. *kwargs_i* are dictionaries that input the kwargs to the mutation function one is using. These will be specific to the function and can be found in their help function.

In addition to specifying the mutations you want to use, the `mutations` dictionary takes three special kwargs: `preserve_best`, `keep_original`, and `keep_original_best`. Setting `preserve_best` to `true`, means the highest fitness individual will **never** be mutated. Setting `keep_original` to `true` means mutations will be applied to copies of individuals, not the individual itself. This means, the original individual is not changed through a mutation. `keep_original_best` applies `keep_original` to only the best individual.

The currently implemented mutations are shown below. Note in all functions, the first argument is the atomic structure, which inserted by the optimizer. The user defines all of the other kwargs *after* the first input.

`structopt.common.individual.mutations.`**`swap_positions`**(*individual*, *max_natoms=0.2*)

Randomly swaps the positions atoms within the individual (in place).

Parameters

- **`individual`** (`Individual`) – an individual
- **`max_natoms`** (*`float or int`*) – if float, the maximum number of atoms whose positions will be swapped is max_natoms*len(individual) if int, the maximum number of atoms whose positions will be swapped is max_natoms if the number of atoms to be swapped is (or evaluates to) an odd integer, it is rounded down to an even integer max_natoms corresponds to the maximum number of atoms whose positions will change default: 0.20

`structopt.common.individual.mutations.`**`swap_species`**(*individual*, *max_natoms=0.2*)

Randomly swaps the species of atoms within the individual (in place).

Parameters

- **`individual`** (`Individual`) – an individual
- **`max_natoms`** (*`float or int`*) – if float, the maximum number of atoms that will be swapped is max_natoms*len(individual) if int, the maximum number of atoms that will be swapped is max_natoms if the number of atoms to be swapped is (or evaluates to) an odd integer, it is rounded down to an even integer max_natoms corresponds to the maximum number of atoms whose species will change default: 0.20

`structopt.common.individual.mutations.`**`rotate_atoms`**(*individual*, *max_natoms=0.2*)

Randomly rotates a number of random atoms within the individual (in place).

Parameters

- **`individual`** (`Individual`) – an individual
- **`max_natoms`** (*`float or int`*) – if float, the maximum number of atoms that will be rotated is max_natoms*len(individual) if int, the maximum number of atoms that will be rotated is max_natoms default: 0.20

`structopt.common.individual.mutations.`**`rotate_all`**(*atoms*, *vector=None*, *angle=None*, *center=None*)

Rotate all atoms around a single point. Most suitable for cluster calculations.

Parameters

- **individual** (Individual) – An individual.
- **vector** (*string or list*) – The list of axes in which to rotate the atoms around. If None, is a randomly chosen direction. If 'random' in list, a random vector can be chosen.
- **angle** (*string or list*) – A list of angles that will be chosen to rotate. If None, is randomly generated. Angle must be given in radians. If 'random' in list, a random angle is included.
- **center** (*string or xyz iterable*) – The center in which to rotate the atoms around. If None, defaults to center of mass. Acceptable strings are COM = center of mass COP = center of positions COU = center of cell

structopt.common.individual.mutations.**permutation**(*individual*)

Swaps the chemical symbol between two elements

Parameters **individual** (Individual) – An individual or atoms object.

structopt.common.individual.mutations.**rattle**(*individual*, *stdev=0.5*, *x_avg_bond=True*)

Randomly displace all atoms in a random direction with a magnitude drawn from a gaussian distribution.

Parameters

- **individual** (Individual) – An individual
- **stdev** (*float*) – The standard deviation of the gaussian distribution to rattle all the atoms. If x_avg_bond is set to True, given as the fraction of the average bond length of the material.
- **x_avg_bond** (*bool*) – If True, the gaussian distributions standard deviation is stdev * avg_bond_length. Note, this only applies to fcc, hcp, or bcc materials.

structopt.cluster.individual.mutations.**move_atoms**(*individual*, *max_natoms=0.2*)

Randomly moves atoms within a cluster.

Parameters

- **individual** (Individual) – An individual
- **max_natoms** (*float or int*) – if float, the maximum number of atoms that will be moved is max_natoms*len(individual). if int, the maximum number of atoms that will be moved is max_natoms default: 0.20

structopt.cluster.individual.mutations.**move_surface_atoms**(*individual*, *max_natoms=0.2*, *move_CN=11*, *surf_CN=11*)

Randomly moves atoms at the surface to other surface sites

Parameters

- **individual** (Individual) – The individual object to be modified in place
- **max_natoms** (*float or int*) – if float, the maximum number of atoms that will be moved is max_natoms*len(individual) if int, the maximum number of atoms that will be moved is max_natoms default: 0.20
- **move_CN** (*int*) – The coordination number to determine which atoms can move moved. Any atom with coordination number above move_CN will not be moved
- **surf_CN** (*int*) – The coordination number to determine which atoms are considered surface atoms. Surface atoms are used to estimating new surface sites

structopt.cluster.individual.mutations.**rotate_cluster**(*individual*, *max_natoms=0.2*)
Chooses a random number of atoms nearest to a random point in the cluster. These atoms are then rotated randomly around this point

Parameters

- **individual** (Individual) – An individual object
- **max_natoms** (*float*) – The fraction of the total atoms to rotate

structopt.cluster.individual.mutations.**twist**(*individual*, *max_radius=0.9*)
Splits the particle randomly in half and rotates one half.

Parameters

- **individual** (*structopt.Individual object*) – Individual to be mutated
- **max_natoms** (*float*) – That maximum relative distance from the center of the particle the twist is initiated

structopt.cluster.individual.mutations.**swap_core_shell**(*individual*, *surf_CN=11*)
Swaps atoms on the surface with an atom in the core. Only does it for different element types.

Parameters

- **individual** (Individual) – An individual
- **surf_CN** (*int*) – The maximum coordination number of an atom to be considered surface

structopt.cluster.individual.mutations.**rich2poor**(*individual*)
Used for multi-component systems. Swaps atoms A and B so that atom A moves from a region with a high number of A-A bonds to a low number of A-A bonds.

Parameters **individual** (Individual) – An individual

structopt.cluster.individual.mutations.**poor2rich**(*individual*)
Used for multi-component systems. Swaps atoms A and B so that atom A moves from a region with a low number of A-A bonds to a high number of A-A bonds.

Parameters **individual** (Individual) – An individual

structopt.cluster.individual.mutations.**move_surface_defects**(*individual*, *surf_CN=11*)
Moves atoms around on the surface based on coordination number Moves a surface atom with a low CN to an atom with a high CN

Parameters

- **individual** (Individual) – The individual object to be modified in place
- **surf_CN** (*int*) – The maximum coordination number to considered a surface atom

structopt.cluster.individual.mutations.**enrich_surface**(*individual*, *surf_CN=11*, *species=None*)
Mutation that selectively enriches the surface with a species.

Parameters

- **individual** (Individual) – An individual
- **surf_CN** (*int*) – The maximum coordination number of an atom to be considered surface
- **species** (*str*) – The surface to enrich with. If None, takes the lowest concentration species

structopt.cluster.individual.mutations.**enrich_bulk**(*individual*, *surf_CN=11*, *species=None*)

Mutation that selectively enriches the bulk with a species

Parameters

- **individual** (Individual) – An individual
- **surf_CN** (*int*) – The maximum coordination number of an atom to be considered surface
- **species** (*str*) – The surface to enrich with. If None, takes the lowest concentration species

structopt.cluster.individual.mutations.**enrich_surface_defects**(*individual*, *surf_CN=11*, *species=None*)

Mutation that selectively enriches defects with a species. Defects are defined as atoms atoms with lower coordination numbers

Parameters

- **individual** (Individual) – An individual
- **surf_CN** (*int*) – The maximum coordination number of an atom to be considered surface
- **species** (*str*) – The surface to enrich with. If None, takes the lowest concentration

structopt.cluster.individual.mutations.**enrich_surface_facets**(*individual*, *surf_CN=11*, *species=None*)

Mutation that selectively enriches facets with a species. Facets are defined as atoms atoms with higher coordination numbers.

Parameters

- **surf_CN** (*int*) – The maximum coordination number of an atom to be considered surface
- **species** (*str*) – The surface to enrich with. If None, takes the lowest concentration

3.2.7 Relaxations

Relaxations performs a local relaxation to the atomic structure before evaluating their fitness. This is typically done after crossover and mutation operators are applied.

Relaxations differ than the previous operations in that they require varying amounts of resources. Hence, a subsequent section, Parallelization, will introduce ways to run your job with varying levels of parallel performance.

Relaxations are given as a dictionary entry defined by the `relaxations` key in the input file. The structure of these dictionaries is shown below.

Example:

```
"relaxations": {
    relaxation_1: {"order": o_1,
                   "kwargs": kwargs_1}
    relaxation_2: {"order": o_2,
                   "kwargs": kwargs_2}
    relaxation_3: {"order": o_3,
                   "kwargs": kwargs_3}
    ...
    relaxation_N: {"order": o_N,
                   "kwargs": kwargs_N}
}
```

The string for *relaxation_i*, is the name of the relaxation one wants to use. The order *o_i* is the order of the relaxation occuring on every individual in the population. *kwargs_i* are dictionaries that input the kwargs to the relaxation function one is using. These will be specific to the function. More details of each relaxation module will be given in the following subsections

LAMMPS

The LAMMPS relaxation module calls LAMMPS to relax according to some potential. Most of the kwargs can be found from the LAMMPS documentation.

class `structopt.common.individual.relaxations.`**LAMMPS**(*parameters*)

LAMMPS class for running LAMMPS on a single individual. Takes a dictionary, where the key: value are the parameters for running LAMMPs.

Parameters

- **min_style** (*str*) – The minimization scheme for running LAMMPS. See LAMMPS doc.
- **min_modify** (*str*) – Parameters for min_style energy minimization algorithm. See LAMMPS doc.
- **minimize** (*str*) – Convergence criteria for minimization algorithm. See LAMMPS doc.
- **pair_style** (*str*) – Type of potential used. See LAMMPS doc.
- **potential_file** (*str*) – The path to the potential_file. Should be absolute.
- **thermo_steps** (*int*) – How much output to print of thermodynamic information. If set to 0, only the last step is printed.See LAMMPS doc.
- **keep_file** (*bool*) – Will keep all of the LAMMPS input and output files for each individual. Use with caution.
- **repair** (*bool*) – Determines whether to run an algorithm to make sure no atoms are in "space". Atoms can be in space due to a mutation or crossover that results in a large force that shoots the atom outside of the particle.

The potential files available to use are listed below and are from the default potentials included from LAMMPS. Given a potential, enter in the `potential_file` kwarg as "$STRUCTOPT_HOME/potentials/<name>". Note also that different potentials will have different lines of the `pair_style` kwarg. If the user would like to use an unavailable potential file, please submit an email to zxu39@wisc.edu, and the potential will be added.

AlCu.eam.alloy: Aluminum and copper alloy EAM (Cai and Ye, Phys Rev B, 54, 8398-8410 (1996))

Au_u3.eam: Gold EAM (SM Foiles et al, PRB, 33, 7983 (1986))

ZrCuAl2011.eam.alloy: Zirconium, copper, and aluminum glass (Howard Sheng at GMU. (hsheng@gmu.edu))

3.2.8 Fitnesses

Fitness evaluates how closely the individual satisfies the minimization criteria. One typical minimization criteria is the stability of a structure, and the formation energy is the fitness. Note, all fitness modules operate so that the *lower* the fitness value the *more* fit it is.

Fitnesses differ than the previous operations in that they require varying amounts of resources. Hence, a subsequent section, Parallelization, will introduce ways to run your job with varying levels of parallel performance.

Fitnesses are given as a dictionary entry defined by `fitnesses` key in the input file. The structure of these dictionaries is shown below.

Example:

```
"fitnesses": {
    fitness_1: {"weight": w_1,
                "kwargs": kwargs_1}
    fitness_2: {"weight": w_2,
                "kwargs": kwargs_2}
    fitness_3: {"weight": w_3,
                "kwargs": kwargs_3}
    ...
    fitness_N: {"weight": w_N,
                "kwargs": kwargs_N}
}
```

The string for *fitness_i*, is the name of the fitness one wants to use. The weight *w_i* is the constant to multiply the fitness value returned by the *fitness_i* module. Not that all selections and predators operate on the **total** fitness, which is a sum of each fitness and their weight. *kwargs_i* are dictionaries that input the kwargs to the fitness function one is using. These will be specific to the function. More details of each fitness module will be given in the following subsections

LAMMPS

The LAMMPS fitness module calls LAMMPS to calculate the potential energy of the structure. Most of the kwargs can be found from the LAMMPS documentation. In addition, most of the *kwargs* are the same as relaxations, except the fitness module of LAMMPS has a number of normalization options for returning the potential energy. These are described below.

class `structopt.common.individual.fitnesses.`**LAMMPS**(*parameters*)

LAMMPS class for running LAMMPS on a single individual. Takes a dictionary, where the key: value are the parameters for running LAMMPs.

Parameters

- **min_style** (*str*) – The minimization scheme for running LAMMPS. See LAMMPS doc.
- **min_modify** (*str*) – Parameters for min_style energy minimization algorithm. See LAMMPS doc.
- **minimize** (*str*) – Convergence criteria for minimization algorithm. Note for fitness values, the last two values are set to 0, so no relaxation is done. See LAMMPS doc.
- **pair_style** (*str*) – Type of potential used. See LAMMPS doc.
- **potential_file** (*str*) – The path to the potential_file. Should be absolute.
- **thermo_steps** (*int*) – How much output to print of thermodynamic information. If set to 0, only the last step is printed.See LAMMPS doc.
- **keep_file** (*bool*) – Will keep all of the LAMMPS input and output files for each individual. Use with caution.
- **reference** (*dict*) – Reference energies of the particle. These are values to subtract from the values returned by LAMMPS. Given as a dictionary of {sym : E} pairs, where sym is a str denoating the the element, while E is the value to be subtracted per sym. This is typically the pure component formation energy calculated with LAMMPS. Note since this is merely a fixed subtraction, should not change the performance in constant composition runs.

The potential files available to use are listed below and are from the default potentials included from LAMMPS. Given a potential, enter in the `potential_file` kwarg as "$STRUCTOPT_HOME/potentials/<name>". Note also that different potentials will have different lines of the `pair_style` kwarg. If the user would like to use an unavailable potential file, please submit an email to zxu39@wisc.edu, and the potential will be added.

AlCu.eam.alloy: Aluminum and copper alloy EAM (Cai and Ye, Phys Rev B, 54, 8398-8410 (1996))

Au_u3.eam: Gold EAM (SM Foiles et al, PRB, 33, 7983 (1986))

ZrCuAl2011.eam.alloy: Zirconium, copper, and aluminum glass (Howard Sheng at GMU. (hsheng@gmu.edu))

3.2.9 Parallelization

In addition to the module-specific parameters, each module requires two parallelization entries: `use_mpi4py` and `MPMD_cores_per_structure`. These two entries are mutually exclusive, meaning that only one can be turned on at a time. `use_mpi4py` can take two values, `true` or `false` depending on whether the module should use the **'one-structure-per-core <>'_** parallelization.

`MPMD_cores_per_structure` can be disabled (if `use_mpi4py` is `true`) by setting it to `0`, but otherwise specifies the number of cores that each process/structure should be allocated within the `MPI_Comm_spawn_multiple` command. There are two types of valid values for this parameter: 1) an integer specifying the number of cores per structure, or 2) a string of two integers separated by a dash specifying the minimum and maximum number of cores allowed (e.g. `"4-16"`). `MPMD_cores_per_structure` can also take the value of `"any"`, and StructOpt will use as many cores as it can to run each individual.

Example:

```
"relaxations": {
    "LAMMPS": {"order": 0,
               "kwargs": {"use_mpi4py": true,
                          "MPMD_cores_per_structure": 0,
                          "keep_files": true,
                          "min_style": "cg\nmin_modify line quadratic",
                          "minimize": "1e-8 1e-8 5000 10000",
                          "pair_style": "eam/alloy",
                          "potential_file": "$STRUCTOPT_HOME/potentials/ZrCuAl2011.
↪eam.alloy",
                          "thermo_steps": 1000}}
}
```

3.3 Outputs

3.4 Examples

The following below are examples of runs you can use to test StructOpt. They exclusively use LAMMPS to relax the structures and calculate its fitness. All of the input files can be found in the StructOpt_modular/examples folder

3.4.1 Running StructOpt

StructOpt can be run on a single processor or in parallel. In a single score environment, the command is given below

```
python $STRUCTOPT_HOME/structopt/genetic.py structopt.in.json
```

In a parallel environment with N processors, StructOpt can be run with the following command

```
mpirun -n N python $STRUCTOPT_HOME/structopt/genetic.py structopt.in.json
```

The output will exist in the folder the command was run from

3.4.2 Example 1: cluster/Au55

3.4.3 Example 2: cluster/Au55-parallel

3.5 JobManager

3.5.1 Introduction

The purpose of the `JobManager` module is to provide a python wrapper for submitting and tracking jobs in a queue environment.

3.5.2 Configuration

The `JobManager` is initially built for a PBS queue environment, so many of the commands will have to be modified for usage in a different queue environment. These customizations will likely take place in the following files.

1. The `submit` and `write_submit` function in the `structopt/utilities/job_manager.py` file will likely need to be updated to reflect your specific queue environment.
2. The dictionaries held in `structopt.utilities/rc.py` is the first attempt to store some dictionaries specific to the queue environment. Many queue specific variables are drawn from here.

3.5.3 Submitting jobs

Single job

The script below is an example script of submitting a single job to a queue using the `JobManager`. The optimization run is a short run of a Au_{55}nanoparticle using only LAMMPS. A large part of the script is defining the input, which goes into the `JobManager` class. These inputs are given below.

1. `calcdir`: This is a string that tells where the calculation is run. Note that the calculation itself is run within the `calcdir/logs{time}` directory, which is created when the job starts to run on the queue. Unless an absolute path, the `calcdir` directory is always given with respect to directory that the job script is run from
2. `optimizer`: This is a string of the optimizer file used for the calculation. These files can be found in the `structopt/optimizers` folder. Upon run, a copy of this script is placed insde of the `calcdir` directory and accessed from there.
3. `StructOpt_parameters`: This is a dictionary object that should mirror the input file you are trying to submit
4. `submit_parameters`: This dictionary holds the submit parameters. These will be specific to the queue system in use. In this example, we specify the the submission system, queue, number of nodes, number of cores, and walltime.

```
from structopt.utilities.job_manager import JobManager
from structopt.utilities.exceptions import Running, Submitted, Queued

calcdir = 'job_manager_examples/Au55-example'

LAMMPS_parameters = {"use_mpi4py": True,
                     "MPMD": 0,
                     "keep_files": False,
                     "min_style": "cg",
                     "min_modify": "line quadratic",
                     "minimize": "1e-8 1e-8 5000 10000",
                     "pair_style": "eam",
                     "potential_file": "$STRUCTOPT_HOME/potentials/Au_u3.eam",
                     "thermo_steps": 0}

StructOpt_parameters = {
    "seed": 0,
    "structure_type": "cluster",
    "generators": {"sphere": {"number_of_individuals": 20,
                              "kwargs": {"atomlist": [["Au", 55]],
                                         "cell": [20, 20, 20]}}},
    "fitnesses": {"LAMMPS": {"weight": 1.0,
                  "kwargs": LAMMPS_parameters}},
    "relaxations": {"LAMMPS": {"order": 0,
                               "kwargs": LAMMPS_parameters}},
    "convergence": {"max_generations": 10},
    "mutations": {"move_atoms": {"probability": 0.1},
                  "rotate_cluster": {"probability": 0.1}},
    "crossovers": {"rotate": {"probability": 0.7}},
    "predators": {"best": {"probability": 1.0}},
    "selections": {"rank": {"probability": 1.0,
                            "kwargs": {"unique_pairs": False,
                                       "unique_parents": False}}},
    "fingerprinters": {"keep_best": True,
                       "diversify_module": {"probability": 1.0,
                                            "kwargs": {"module": "LAMMPS",
                                                       "min_diff": 0.001}}},
    "post_processing": {"XYZs": -1},
}

submit_parameters = {'system': 'PBS',
                     'queue': 'morgan2',
                     'nodes': 1,
                     'cores': 12,
                     'walltime': 12}

optimizer = 'genetic.py'

job = JobManager(calcdir, optimizer, StructOpt_parameters, submit_parameters)
job.optimize()
```

Upon running this script, the user should get back an exception called `structopt.utilities.exceptions.Submitted` with the jobid. This is normal behavior and communicates that the job has successfully been submitted.

Multiple jobs

One advantage of the job manager is that it allows one to submit multiple jobs to the queue. This is often useful for tuning the optimizer against different inputs. The script below is an example of submitting the same job at different seeds.

In the previous script, submitting a single job successfully with `JobManager.optimizer` method resulted in an exception. We can catch this exception with a `try` and `except` statement. This is shown below in the script where upon a successful submission, the script prints out the jobid to the user.

```
from structopt.utilities.job_manager import JobManager
from structopt.utilities.exceptions import Running, Submitted, Queued

LAMMPS_parameters = {"use_mpi4py": True,
                     "MPMD": 0,
                     "keep_files": False,
                     "min_style": "cg",
                     "min_modify": "line quadratic",
                     "minimize": "1e-8 1e-8 5000 10000",
                     "pair_style": "eam",
                     "potential_file": "$STRUCTOPT_HOME/potentials/Au_u3.eam",
                     "thermo_steps": 0}

StructOpt_parameters = {
    "seed": 0,
    "structure_type": "cluster",
    "generators": {"sphere": {"number_of_individuals": 20,
                              "kwargs": {"atomlist": [["Au", 55]],
                                         "cell": [20, 20, 20]}}},
    "fitnesses": {"LAMMPS": {"weight": 1.0,
                  "kwargs": LAMMPS_parameters}},
    "relaxations": {"LAMMPS": {"order": 0,
                               "kwargs": LAMMPS_parameters}},
    "convergence": {"max_generations": 10},
    "mutations": {"move_atoms": {"probability": 0.1},
                  "rotate_cluster": {"probability": 0.1}},
    "crossovers": {"rotate": {"probability": 0.7}},
    "predators": {"best": {"probability": 1.0}},
    "selections": {"rank": {"probability": 1.0,
                            "kwargs": {"unique_pairs": False,
                                       "unique_parents": False}}},
    "fingerprinters": {"keep_best": True,
                       "diversify_module": {"probability": 1.0,
                                            "kwargs": {"module": "LAMMPS",
                                                       "min_diff": 0.001}}},
    "post_processing": {"XYZs": -1},
}

submit_parameters = {'system': 'PBS',
                     'queue': 'morgan2',
                     'nodes': 1,
                     'cores': 12,
                     'walltime': 12}

optimizer = 'genetic.py'

seeds = [0, 1, 2, 3, 4]
for seed in seeds:
```

```
    StructOpt_parameters['seed'] = seed
    calcdir = 'job_manager_examples/Au55-seed-{}'.format(seed)

    job = JobManager(calcdir, optimizer, StructOpt_parameters, submit_parameters)

    try:
        job.optimize()
    except Submitted:
        print(calcdir, job.get_jobid(), 'submitted')
```

```
job_manager_examples/Au55-seed-0 936454.bardeen.msae.wisc.edu submitted
job_manager_examples/Au55-seed-1 936455.bardeen.msae.wisc.edu submitted
job_manager_examples/Au55-seed-2 936456.bardeen.msae.wisc.edu submitted
job_manager_examples/Au55-seed-3 936457.bardeen.msae.wisc.edu submitted
job_manager_examples/Au55-seed-4 936458.bardeen.msae.wisc.edu submitted
```

3.5.4 Tracking jobs

In the previous section, we covered how to submit a new job from an empty directory. This is done by first initializing an instance of the `StructOpt.utilities.job_manager.JobManager` class with a calculation directory along with some input files and then submitting the job with the `JobManager.optimize` method. The `JobManager.optimize` method knows what to do because upon initialization, it detected an empty directory. If the directory was not empty and contained a StructOpt job, the `JobManager` knows what to do with it if `optimize` was run again. This is all done with exceptions.

The four primary exceptions that are returned upon executing the `optimize` method are below along with their explanations.

1. `Submitted`: This exception is returned if a job is submitted from the directory. This is done when `JobManager.optimize` is called in an empty directory or `JobManager.optimize` is called with the kwarg `restart=True` in a directory where a job is not queued or running.
2. `Queued`: The job is queued and has not started running. There should be no output files to be analyzed.
3. `Running`: The job is running and output files should be continously be updated. These output files can be used for analysis before the job has finished running.
4. `UnknownState`: This is returned if the `calcdir` is not an empty directory doesn't detect it as a StructOpt run. A StructOpt run is detected when a `structopt.in.json` file is found in the `calcdir`.

Note that if no exception is returned, it means the job is done and is ready to be analyzed. `Job.optimize` does nothing in this case.

One way of using these three exceptions is below. If the job is submitted or Queued, we want the script to stop and not submit the job. If it is running, additional commands can be used to track the progress of the job.

```
from structopt.utilities.job_manager import JobManager
from structopt.utilities.exceptions import Running, Submitted, Queued

calcdir = 'job_manager_examples/Au55-example'

LAMMPS_parameters = {"use_mpi4py": True,
                     "MPMD": 0,
                     "keep_files": False,
                     "min_style": "cg",
                     "min_modify": "line quadratic",
                     "minimize": "1e-8 1e-8 5000 10000",
```

```
                    "pair_style": "eam",
                    "potential_file": "$STRUCTOPT_HOME/potentials/Au_u3.eam",
                    "thermo_steps": 0}

StructOpt_parameters = {
    "seed": 0,
    "structure_type": "cluster",
    "generators": {"sphere": {"number_of_individuals": 20,
                              "kwargs": {"atomlist": [["Au", 55]],
                                         "cell": [20, 20, 20]}}},
    "fitnesses": {"LAMMPS": {"weight": 1.0,
                  "kwargs": LAMMPS_parameters}},
    "relaxations": {"LAMMPS": {"order": 0,
                               "kwargs": LAMMPS_parameters}},
    "convergence": {"max_generations": 10},
    "mutations": {"move_atoms": {"probability": 0.1},
                  "rotate_cluster": {"probability": 0.1}},
    "crossovers": {"rotate": {"probability": 0.7}},
    "predators": {"best": {"probability": 1.0}},
    "selections": {"rank": {"probability": 1.0,
                            "kwargs": {"unique_pairs": False,
                                       "unique_parents": False}}},
    "fingerprinters": {"keep_best": True,
                       "diversify_module": {"probability": 1.0,
                                            "kwargs": {"module": "LAMMPS",
                                                       "min_diff": 0.001}}},
    "post_processing": {"XYZs": -1},
}

submit_parameters = {'system': 'PBS',
                     'queue': 'morgan2',
                     'nodes': 1,
                     'cores': 12,
                     'walltime': 12}

optimizer = 'genetic.py'

job = JobManager(calcdir, optimizer, StructOpt_parameters, submit_parameters)
try:
    job.optimize()
except (Submitted, Queued):
    print(calcdir, job.get_jobid(), 'submitted or queued')
except Running:
    pass
```

```
job_manager_examples/Au55-example 936453.bardeen.msae.wisc.edu submitted or queued
```

3.5.5 Restarting jobs

Sometimes jobs need to be restarted or continued from the last generation. The `JobManager` does this by submitting a new job from the same `calcdir` folder the previous job was run in. Because calculations take place in unique `log{time}` directories, the job will run in a new `log{time}` directory. Furthermore, the `JobManager` modifies the `structopt.in.json` file so the initial population of the new job are the XYZ files of the last generation of the previous run. The code below is an example of restarting the first run of this example. The only difference between this code and the one presented in the previous section is that a `restart=True` kwarg has been added to the `JobManager.optimize` command.

```
from structopt.utilities.job_manager import JobManager
from structopt.utilities.exceptions import Running, Submitted, Queued

calcdir = 'job_manager_examples/Au55-example'

LAMMPS_parameters = {"use_mpi4py": True,
                     "MPMD": 0,
                     "keep_files": False,
                     "min_style": "cg",
                     "min_modify": "line quadratic",
                     "minimize": "1e-8 1e-8 5000 10000",
                     "pair_style": "eam",
                     "potential_file": "$STRUCTOPT_HOME/potentials/Au_u3.eam",
                     "thermo_steps": 0}

StructOpt_parameters = {
    "seed": 0,
    "structure_type": "cluster",
    "generators": {"sphere": {"number_of_individuals": 20,
                              "kwargs": {"atomlist": [["Au", 55]],
                                         "cell": [20, 20, 20]}}},
    "fitnesses": {"LAMMPS": {"weight": 1.0,
                  "kwargs": LAMMPS_parameters}},
    "relaxations": {"LAMMPS": {"order": 0,
                               "kwargs": LAMMPS_parameters}},
    "convergence": {"max_generations": 10},
    "mutations": {"move_atoms": {"probability": 0.1},
                  "rotate_cluster": {"probability": 0.1}},
    "crossovers": {"rotate": {"probability": 0.7}},
    "predators": {"best": {"probability": 1.0}},
    "selections": {"rank": {"probability": 1.0,
                            "kwargs": {"unique_pairs": False,
                                       "unique_parents": False}}},
    "fingerprinters": {"keep_best": True,
                       "diversify_module": {"probability": 1.0,
                                            "kwargs": {"module": "LAMMPS",
                                                       "min_diff": 0.001}}},
    "post_processing": {"XYZs": -1},
}

submit_parameters = {'system': 'PBS',
                     'queue': 'morgan2',
                     'nodes': 1,
                     'cores': 12,
                     'walltime': 12}

optimizer = 'genetic.py'

job = JobManager(calcdir, optimizer, StructOpt_parameters, submit_parameters)
job.optimize(restart=True)
```

3.6 Relaxation and Fitness Modules

3.6.1 LAMMPS

Installation

Follow the standard installation instructions.

Create an environment variable called `LAMMPS_COMMAND` that points to the serial LAMMPS executable after installation.

Package Documentation

3.6.2 VASP

Installation

Follow the standard installation instructions.

Create an environment variable called `VASP_COMMAND` that points to the VASP executable after installation.

Package Documentation

3.6.3 FEMSIM

Installation

Fork and clone the repository from github.

Using OpenMPI 1.10.2 compilers, follow the instructions to compile femsim.

Create an environment variable called `FEMSIM_COMMAND` pointing to the newly created `femsim` executable.

Package Documentation

3.6.4 STEM

References: http://pubs.acs.org/doi/abs/10.1021/acsnano.5b05722

3.6.5 Creating Your Own Module

Any forward simulation that takes an atomic model as input and outputs a "fitness" value that can be interpreted as a measure of "goodness" of the structure can be integrated into StructOpt. Contact the developers by making an issue on github to get in touch with us.

3.7 Why Python?

Python has been widely accepted by scientific community. From the invaluable scientific software packages such as numpy, scipy, mpi4py, dask, and pandas to the thousands of specialized software packages, the scientific support through Python is enormous.

StructOpt is meant to solve new problems rather than be a better tool for solving well understood problems. As a result, many of the users of StructOpt will be exploring new scientific territory and will be in the development process of creation and iteration on their tools. Python is a forerunner for development applications due to its ability to scale from off-hand scripts to large packages and applications.

Via Jupyter notebooks, Python code is on its way to becoming readable for the general community. This, combined with the drive toward more accessible and better documented scientific code, may provide a powerful combination to encourage scientific reproducability and archival. To this end, StructOpt's data explorer is meant to ease the process of analyzing and displaying useful information.

3.8 Future Work

- Add additional optimzers (such as particle swarm)
- Implement uncertainty quantification
- Implement Baysian statistics to estimate and automate setting correct values of the weights between modules in the cost function

3.9 Troubleshooting

3.10 StructOpt Package

3.10.1 Submodules

structopt.io

`structopt.io.`**`parameters`**`()`
Contains functionality for reading, writing, and parsing StrcutOpt parameters.

`structopt.io.`**`logger_utils`**`()`
Contains functionality for creating and using loggers.

`structopt.io.`**`read_xyz`**`(`*filename*, *index=None*, *format=None*, ***kwargs*`)`
Reads an xyz file into an ASE Atoms object and returns it.

`structopt.io.`**`write_xyz`**`(`*fileobj*, *atoms*, *comment=''*, *append=False*`)`
Writes xyz files from an Individual object. Adapted from ase.io.xyz.

structopt.tools

`structopt.tools.parallel.`**`root`**`(`*method=None*, *broadcast=True*`)`
A decorator to make the function only run on the root node. The returned data from the root is then broadcast to all the other nodes and each node returns the root's data.

`structopt.tools.parallel.`**`single_core`**`(`*method*`)`
A place holder decorator that does nothing except document that the function is designed to be run on a single core.

`structopt.tools.parallel.`**`parallel`**`(`*method*`)`
A decorator that does nothing except document that the function is designed to run in parallel.

structopt.tools.parallel.**allgather**(*stuff*, *stuffs_per_core*)

Performs an MPI.allgather on a selection of data and uses stuffs_per_core to parse out the correct information and return it.

Parameters

- **stuff** (*any*) – any piece of data (e.g. fitnesses), some of which have been updated on their respective cores and some of which haven't. each piece of data should be of the same length
- **stuffs_per_core** (*dict<int, list<int>>*) – a dictionary containing a mapping of the cores that contain the correct information to the corresponding indices in the pieces of data

Returns the correct stuff is returned on each core

Return type type(stuff)

Example

This is going to take the values:

```
values = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

and convert each of them to strings.

In this example there are 5 cores, so `stuffs_per_core` looks like:

```
stuffs_per_core = {0: [0, 5], 1: [1, 6], 2: [2, 7], 3: [3, 8], 4: [4, 9]}
```

Now for the code that precedes `allgather()` and then calls `allgather()`:

```
# This for-loop modifies different parts of `values` on each core by
# converting some elements in `values` from an int to a str.
# We then want to collect the values that each core independently updated
# and allgather them so that every core has all of the updated values,
# even though each core only did part of the work.
for i in stuffs_per_core[rank]:
    values[i] = str(inds[i])
x = allgather(values, stuffs_per_core)
print(x)  # returns:  ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
```

structopt.postprocessing

3.10.2 The Optimizer

structopt.Optimizer

structopt.common.population.Population

class structopt.common.population.**Population**(*parameters*, *individuals=None*)

Bases: structopt.tools.sorted_dict.SortedDict

A list-like class that contains the Individuals and the operations to be run on them.

add(*individual*)

Adds an Individual to the population.

allgather(*individuals_per_core*)

Performs an MPI.allgather on self (the population) and updates the correct individuals that have been modified based on the inputs from individuals_per_core.

See stuctopt.tools.parallel.allgather for a similar function.

(@parallel) Designed to run code that runs differently on different cores. The MPI functionality should be implemented inside these functions.

apply_fingerprinters()
Apply fingerprinters on the entire population.

(@parallel) Designed to run code that runs differently on different cores. The MPI functionality should be implemented inside these functions.

bcast()
Performs and MPI.bcast on self.

(@parallel) Designed to run code that runs differently on different cores. The MPI functionality should be implemented inside these functions.

calculate_fitnesses()
Perform the fitness evaluations on the entire population.

(@parallel) Designed to run code that runs differently on different cores. The MPI functionality should be implemented inside these functions.

clear() → None. Remove all items from od.

copy() → a shallow copy of od

crossover(*pairs*)
Perform crossovers on the population.

(@parallel) Designed to run code that runs differently on different cores. The MPI functionality should be implemented inside these functions.

extend(*individuals*)
Overwrites and adds to the population using the *id* attribute of the individuals as a keyword. Assigns an id to an individual if it doesn't already have one.

fromkeys(*S*[, *v*]) → New ordered dictionary with keys from S.
If not specified, the value defaults to None.

get(*k*[, *d*]) → D[k] if k in D, else d. d defaults to None.

get_by_position(*position*)
Returns the individual at position *position*.

get_new_id()

items() → a set-like object providing a view on D's items

keys() → a set-like object providing a view on D's keys

kill()
Remove individuals from the population based on a predator scheme.

(@parallel) Designed to run code that runs differently on different cores. The MPI functionality should be implemented inside these functions.

load_modules()

mutate()
Perform mutations on the population.

(@root) Designed to run on the root node only.

pop(*k*[, *d*]) → v, remove specified key and return the corresponding
value. If key is not found, d is returned if given, otherwise KeyError is raised.

popitem() → (k, v), return and remove a (key, value) pair.
Pairs are returned in LIFO order if last is true or FIFO order if false.

position(*individual*)
Returns the position of the individual in the population.

relax()
Relax the entire population.

(@parallel) Designed to run code that runs differently on different cores. The MPI functionality should be implemented inside these functions.

remove(*individual*)
Removes *individual* form the population.

replace(*a_list*)
Deletes the current list of individuals and replaces them with the ones in a_list.

run_pso_moves(*best_swarm*, *best_particles*)
Perform PSO moves on the population.

(@root) Designed to run on the root node only.

select()
Select the individuals in the population to perform crossovers on.

(@root) Designed to run on the root node only.

setdefault(*k*[, *d*]) → od.get(k,d), also set od[k]=d if k not in od

update(*individuals*)
Overwrites and adds to the population using the *id* attribute of the individuals as a keyword. Assigns an id to an individual if it doesn't already have one.

values() → an object providing a view on D's values

structopt.common.population.crossovers

class structopt.common.population.crossovers.**Crossovers**(*parameters*)
Bases: object

crossover(*pairs*)
(@parallel) Designed to run code that runs differently on different cores. The MPI functionality should be implemented inside these functions.

post_processing(*parent_pair*, *child_pair*)

static rotate(*individual1*, *individual2*, *conserve_composition=True*)
Rotates the two individuals around their centers of mass, splits them in half at the xy-plane, then splices them together. Maintains number of atoms.

Parameters

- **individual1** (Individual) – The first parent

- **individual2** (Individual) – The second parent
- **conserve_composition** (*bool*) – default True. If True, conserves composition.

Returns The first child Individual: The second child

Return type *Individual*

The children are returned without indicies.

select_crossover()

structopt.common.population.fitnesses

class structopt.common.population.fitnesses.**Fitnesses**(*parameters*)
Bases: object

Holds the parameters for each fitness module and defines a utility function to compute the fitnesses for each fitness module.

calculate_fitnesses(*population*)
Perform the fitness calculations on an entire population.

Args: population (Population): the population to evaluate

(@parallel) Designed to run code that runs differently on different cores. The MPI functionality should be implemented inside these functions.

post_processing(*fitnesses*)

structopt.common.population.fitnesses.LAMMPS.**fitness**(*population*, *parameters*)
Perform the LAMMPS fitness calculation on an entire population.

Args: population (Population): the population to evaluate

(@parallel) Designed to run code that runs differently on different cores. The MPI functionality should be implemented inside these functions.

structopt.common.population.fitnesses.FEMSIM.**fitness**(*population*, *parameters*)
Perform the FEMSIM fitness calculation on an entire population.

Args: population (Population): the population to evaluate

(@root) Designed to run on the root node only.

structopt.common.population.relaxations

class structopt.common.population.relaxations.**Relaxations**(*parameters*)
Bases: object

Holds the parameters for each relaxation module and defines a utility function to run the relaxations for each relaxation module.

post_processing()

relax(*population*)
Relax the entire population using all the input relaxation methods.

Args: population (Population): the population to relax

(@parallel) Designed to run code that runs differently on different cores. The MPI functionality should be implemented inside these functions.

structopt.common.population.relaxations.LAMMPS.**relax**(*population*, *parameters*)
Relax the entire population using LAMMPS.

Args: population (Population): the population to relax

(@parallel) Designed to run code that runs differently on different cores. The MPI functionality should be implemented inside these functions.

structopt.common.population.relaxations.hard_sphere_cutoff.**relax**(*population*, *parameters*)
Relax the entire population using a hard-sphere cutoff method.

Args: population (Population): the population to relax

(@parallel) Designed to run code that runs differently on different cores. The MPI functionality should be implemented inside these functions.

structopt.common.population.mutations

class structopt.common.population.mutations.**Mutations**(*parameters*)
Bases: object

mutate(*population*)

post_processing()

structopt.common.population.predators

class structopt.common.population.predators.**Predators**(*parameters*)
Bases: object

static best(*fits*, *nkeep*)
Sorts individuals by fitness and keeps the top nkeep fitnesses.

Parameters

- **fits** (*dict<int, float>*) – Dictionary of <individual.id, fitness> pairs.
- **nkeep** (*int*) – The number of individuals to keep. In a GA run, corresponds to the sum of each generators number_of_individuals

static fuss(*fits*, *nkeep*, *nbest=0*, *fusslimit=10*)
Fixed uniform selection scheme. Aimed at maintaining diversity in the population. In the case where low fit is the highest fitness, selects a fitness between min(fits) and min(fits) + fusslimit, if the difference between the min(fit) and max(fit) is larger than fusslimit.

Parameters

- **fits** (*dict<int, float>*) – Dictionary of <individual.id, fitness> pairs.
- **nkeep** (*int*) – The number of individuals to keep. In a GA run, corresponds to the sum of each generators number_of_individuals
- **nbest** (*int*) – The top n individuals to always keep (default 0)
- **fusslimit** (*float*) – Individuals that have fitness fusslimit worse than the max fitness will not be considered

kill(*population*, *nkeep*, *keep_best=True*)
Removes some individuals from the population.

Parameters

- **nkeep** (*int*) – The number of individuals to keep.
- **keep_best** (*bool*) – If set to True, the best individual is always included in the following generation.

Returns

Return type The individuals that were removed from the population.

post_processing(*killed*)

static rank(*fits*, *nkeep*, *p_min=None*)

Selection function that chooses pairs of structures based on linear ranking.

See "Grefenstette and Baker 1989 Whitley 1989".

Parameters

- **fits** (*dict<int, float>*) – Dictionary of <individual.id, fitness> pairs.
- **nkeep** (*int*) – The number of individuals to keep. In a GA run, corresponds to the sum of each generators number_of_individuals
- **p_min** (*float*) – The probability of choosing the lowest ranked individual. Given population of size N, this should be below 1/nindiv. The probability of selecting rank N (worst) to rank 1 (best) increases from p_min to (2/N - p_min) in even, (1/N - p_min) increments. Defaults to (1/N)^2.

static roulette(*fits*, *nkeep*, *T=None*)

Select individuals with a probability proportional to their fitness. Fitnesses are renormalized from 0 - 1, which means minimum fitness individual is never included in in the new population.

Parameters

- **fits** (*dict<int, float>*) – Dictionary of <individual.id, fitness> pairs.
- **nkeep** (*int*) – The number of individuals to keep. In a GA run, corresponds to the sum of each generators number_of_individuals
- **T** (*float*) – If T is not None, a boltzman-like transformation is applied to all fitness values with T.

select_predator()

static tournament(*fits*, *nkeep*, *tournament_size=5*)

Selects individuals in seperate "tournaments", where a subset of the population are randomly selected and the highest fitness allowed to pass. In addition to a population, their fits, and end population size, takes in a tournament size parameter.

Parameters

- **fits** (*dict<int, float>*) – Dictionary of <individual.id, fitness> pairs.
- **nkeep** (*int*) – The number of individuals to keep. In a GA run, corresponds to the sum of each generators number_of_individuals
- **tournament_size** (*int*) – The number of individuals in each tournament. If 1, tournament is the same as random selection. If len(population), corresponds to the "best" selection process

structopt.common.population.selections

class structopt.common.population.selections.**Selections**(*parameters*)
Bases: object

static best(*population*, *fits*)
Deterministic selection function that chooses adjacently ranked individuals as pairs.

Parameters

- **population** (Population) – An population of individuals
- **fits** (*list*) – Fitnesses that corresponds to population

post_processing(*pairs*)

static random_selection(*population*, *fits*)
Randomly selects parents

Parameters

- **population** (Population) – An population of individuals
- **fits** (*list*) – Fitnesses that corresponds to population

static rank(*population*, *fits*, *p_min=None*, *unique_pairs=False*, *unique_parents=False*)
Selection function that chooses pairs of structures based on linear ranking.

See "Grefenstette and Baker 1989 Whitley 1989".

Parameters

- **population** (Population) – An object inherited from list that contains StructOpt individual objects.
- **fits** (*list*) – A list of fitnesses of the population
- **p_min** (*float*) – The probability of choosing the lowest ranked individual. Given population of size N, this should be below 1/nindiv. The probability of selecting rank N (worst) to rank 1 (best) increases from p_min to (2/N - p_min) in even, (1/N - p_min) increments. Defaults to (1/N)^2.
- **unique_pairs** (*bool*) – If True, all combinations of parents are unique. True increases the diveristy of the population.
- **unique_parents** (*bool*) – If True, all parents can only mate with on other individual. True increases the diversity of the population.

static roulette(*population*, *fits*, *unique_pairs=False*, *unique_parents=False*)
Selection function that chooses pairs of structures based on their fitness. Fitnesses are normalized from 0 to 1.

See "Grefenstette and Baker 1989 Whitley 1989".

Parameters

- **population** (*StructOpt population object*) – An object inherited from list that contains StructOpt individual objects.
- **fits** (*list*) – A list of fitnesses of the population
- **unique_pairs** (*bool*) – If True, all combinations of parents are unique. True increases the diveristy of the population.
- **unique_parents** (*bool*) – If True, all parents can only mate with on other individual. True increases the diversity of the population.

select(*population*)

select_selection()

static tournament(*population*, *fits*, *tournament_size=5*, *unique_pairs=False*, *unique_parents=False*, *keep_best=False*)

Selects pairs in seperate "tournaments", where a subset of the population are randomly selected and the highest fitness allowed to pass. In addition to a population, their fits, and end population size, takes in a tournament size parameter.

Parameters

- **population** (Population) – The population of individuals needed to be trimmed
- **fits** (*list*) – List of fitnesses that correspond to the population.
- **tournament_size** (*int*) – The number of individuals in each tournament. If 1, tournament is the same as random selection. If len(population), corresponds to the "best" selection process
- **unique_pairs** (*bool*) – If True, all combinations of parents are unique, though parents can show up in different pairs. True increases the diversity of the population.
- **unique_parents** (*bool*) – If True, all parents can only mate with on other individual. True increases the diversity of the population.

structopt.common.individual.Individual

class structopt.common.individual.**Individual**(*id=None*, *load_modules=True*, *relaxation_parameters=None*, *fitness_parameters=None*, *mutation_parameters=None*, *pso_moves_parameters=None*, *generator_parameters=None*, ***kwargs*)

Bases: ase.atoms.Atoms

An abstract base class for a structure.

adsorbate_info

Return the adsorbate information set by one of the surface builder functions. This function is only supplied in order to give a warning if this attribute (atoms.adsorbate_info) is asked for. The dictionary with adsorbate information has been moved to the info dictionary, i.e. atoms.info['adsorbate_info'].

append(*atom*)

Append atom to end.

calc

Calculator object.

calculate_fitness()

Perform the fitness calculations on an individual.

Args: individual (Individual): the individual to evaluate

(@parallel) Designed to run code that runs differently on different cores. The MPI functionality should be implemented inside these functions.

cell

Attribute for direct manipulation of the unit cell.

center(*vacuum=None*, *axis=(0*, *1*, *2)*, *about=None*)
Center atoms in unit cell.

Centers the atoms in the unit cell, so there is the same amount of vacuum on all sides.

vacuum: float (default: None) If specified adjust the amount of vacuum when centering. If vacuum=10.0 there will thus be 10 Angstrom of vacuum on each side.

axis: int or sequence of ints Axis or axes to act on. Default: Act on all axes.

about: float or array (default: None) If specified, center the atoms about <about>. I.e., about=(0., 0., 0.) (or just "about=0.", interpreted identically), to center about the origin.

clear()

constraints
Constraints of the atoms.

copy(*include_atoms=True*)
Return a copy.

edit()
Modify atoms interactively through ASE's GUI viewer.

Conflicts leading to undesirable behaviour might arise when matplotlib has been pre-imported with certain incompatible backends and while trying to use the plot feature inside the interactive GUI. To circumvent, please set matplotlib.use('gtk') before calling this method.

euler_rotate(*phi=0.0*, *theta=0.0*, *psi=0.0*, *center=(0*, *0*, *0)*)
Rotate atoms via Euler angles (in degrees).

See e.g http://mathworld.wolfram.com/EulerAngles.html for explanation.

Parameters:

center : The point to rotate about. A sequence of length 3 with the coordinates, or 'COM' to select the center of mass, 'COP' to select center of positions or 'COU' to select center of cell.

phi : The 1st rotation angle around the z axis.

theta : Rotation around the x axis.

psi : 2nd rotation around the z axis.

extend(*other*)
Extend atoms object by appending atoms from *other*.

fitness
The total fitness of the individual.

fits

generate()
Generate an individual using generator_kwargs parameter. By defualt it extends the current atoms object

get_all_distances(*mic=False*)
Return distances of all of the atoms with all of the atoms.

Use mic=True to use the Minimum Image Convention.

get_angle(*a1*, *a2=None*, *a3=None*, *mic=False*)
Get angle formed by three atoms.

calculate angle in degrees between the vectors a2->a1 and a2->a3.

Use mic=True to use the Minimum Image Convention and calculate the angle across periodic boundaries.

get_angular_momentum()
Get total angular momentum with respect to the center of mass.

get_array(*name*, *copy=True*)
Get an array.

Returns a copy unless the optional argument copy is false.

get_atom_indices_within_distance_of_atom(*atom_index*, *distance*)

get_atomic_numbers()
Get integer array of atomic numbers.

get_calculator()
Get currently attached calculator object.

get_cell(*complete=False*)
Get the three unit cell vectors as a 3x3 ndarray.

get_cell_lengths_and_angles()
Get unit cell parameters. Sequence of 6 numbers.

First three are unit cell vector lengths and second three are angles between them:

```
[len(a), len(b), len(c), angle(a,b), angle(a,c), angle(b,c)]
```

in degrees.

get_celldisp()
Get the unit cell displacement vectors.

get_center_of_mass(*scaled=False*)
Get the center of mass.

If scaled=True the center of mass in scaled coordinates is returned.

get_charges()
Get calculated charges.

get_chemical_formula(*mode='hill'*)
Get the chemial formula as a string based on the chemical symbols.

Parameters:

mode: str There are three different modes available:

'all': The list of chemical symbols are contracted to at string, e.g. ['C', 'H', 'H', 'H', 'O', 'H'] becomes 'CHHHOH'.

'reduce': The same as 'all' where repeated elements are contracted to a single symbol and a number, e.g. 'CHHHOCHHH' is reduced to 'CH3OCH3'.

'hill': The list of chemical symbols are contracted to a string following the Hill notation (alphabetical order with C and H first), e.g. 'CHHHOCHHH' is reduced to 'C2H6O' and 'SOOHOHO' to 'H2O4S'. This is default.

'metal': The list of checmical symbols (alphabetical metals, and alphabetical non-metals)

get_chemical_symbols()
Get list of chemical symbol strings.

get_dihedral(*a1*, *a2=None*, *a3=None*, *a4=None*, *mic=False*)
Calculate dihedral angle.

Calculate dihedral angle (in degrees) between the vectors a1->a2 and a3->a4.

Use mic=True to use the Minimum Image Convention and calculate the angle across periodic boundaries.

get_dipole_moment()
Calculate the electric dipole moment for the atoms object.

Only available for calculators which has a get_dipole_moment() method.

get_distance(*a0*, *a1*, *mic=False*, *vector=False*)
Return distance between two atoms.

Use mic=True to use the Minimum Image Convention. vector=True gives the distance vector (from a0 to a1).

get_distances(*a*, *indices*, *mic=False*, *vector=False*)
Return distances of atom No.i with a list of atoms.

Use mic=True to use the Minimum Image Convention. vector=True gives the distance vector (from a to self[indices]).

get_forces(*apply_constraint=True*, *md=False*)
Calculate atomic forces.

Ask the attached calculator to calculate the forces and apply constraints. Use *apply_constraint=False* to get the raw forces.

For molecular dynamics (md=True) we don't apply the constraint to the forces but to the momenta.

get_initial_charges()
Get array of initial charges.

get_initial_magnetic_moments()
Get array of initial magnetic moments.

get_kinetic_energy()
Get the kinetic energy.

get_magnetic_moment()
Get calculated total magnetic moment.

get_magnetic_moments()
Get calculated local magnetic moments.

get_masses()
Get array of masses.

get_momenta()
Get array of momenta.

get_moments_of_inertia(*vectors=False*)
Get the moments of inertia along the principal axes.

The three principal moments of inertia are computed from the eigenvalues of the symmetric inertial tensor. Periodic boundary conditions are ignored. Units of the moments of inertia are amu*angstrom**2.

get_nearest_atom_indices(*atom_index*, *count*)

get_number_of_atoms()
Returns the global number of atoms in a distributed-atoms parallel simulation.

DO NOT USE UNLESS YOU KNOW WHAT YOU ARE DOING!

Equivalent to len(atoms) in the standard ASE Atoms class. You should normally use len(atoms) instead. This function's only purpose is to make compatibility between ASE and Asap easier to maintain by having a few places in ASE use this function instead. It is typically only when counting the global number of degrees of freedom or in similar situations.

get_pbc()
Get periodic boundary condition flags.

get_positions(*wrap=False*)
Get array of positions. If wrap==True, wraps atoms back into unit cell.

get_potential_energies()
Calculate the potential energies of all the atoms.

Only available with calculators supporting per-atom energies (e.g. classical potentials).

get_potential_energy(*force_consistent=False*, *apply_constraint=True*)
Calculate potential energy.

Ask the attached calculator to calculate the potential energy and apply constraints. Use *apply_constraint=False* to get the raw forces.

When supported by the calculator, either the energy extrapolated to zero Kelvin or the energy consistent with the forces (the free energy) can be returned.

get_reciprocal_cell()
Get the three reciprocal lattice vectors as a 3x3 ndarray.

Note that the commonly used factor of 2 pi for Fourier transforms is not included here.

get_scaled_positions(*wrap=True*)
Get positions relative to unit cell.

If wrap is True, atoms outside the unit cell will be wrapped into the cell in those directions with periodic boundary conditions so that the scaled coordinates are between zero and one.

get_stress(*voigt=True*)
Calculate stress tensor.

Returns an array of the six independent components of the symmetric stress tensor, in the traditional Voigt order (xx, yy, zz, yz, xz, xy) or as a 3x3 matrix. Default is Voigt order.

get_stresses()
Calculate the stress-tensor of all the atoms.

Only available with calculators supporting per-atom energies and stresses (e.g. classical potentials). Even for such calculators there is a certain arbitrariness in defining per-atom stresses.

get_tags()
Get integer array of tags.

get_temperature()
Get the temperature in Kelvin.

get_total_energy()
Get the total energy - potential plus kinetic energy.

get_velocities()
Get array of velocities.

get_volume()
Get volume of unit cell.

has(*name*)
Check for existence of array.

name must be one of: ‘tags’, ‘momenta’, ‘masses’, ‘magmoms’, ‘charges’.

load_modules()
Loads the relevant modules.

(@parallel) Designed to run code that runs differently on different cores. The MPI functionality should be implemented inside these functions.

mutate(*select_new=True*)
Mutate an individual.

Args: individual (Individual): the individual to mutate

(@parallel) Designed to run code that runs differently on different cores. The MPI functionality should be implemented inside these functions.

new_array(*name*, *a*, *dtype=None*, *shape=None*)
Add new array.

If *shape* is not *None*, the shape of *a* will be checked.

number_of_lattice_vectors
Number of (non-zero) lattice vectors.

numbers
Attribute for direct manipulation of the atomic numbers.

pbc
Attribute for direct manipulation of the periodic boundary condition flags.

pop(*i=-1*)
Remove and return atom at index *i* (default last).

positions

rattle(*stdev=0.001*, *seed=42*)
Randomly displace atoms.

This method adds random displacements to the atomic positions, taking a possible constraint into account. The random numbers are drawn from a normal distribution of standard deviation stdev.

For a parallel calculation, it is important to use the same seed on all processors!

relax()
Relax an individual.

Args: individual (Individual): the individual to relax

(@parallel) Designed to run code that runs differently on different cores. The MPI functionality should be implemented inside these functions.

repeat(*rep*)
Create new repeated atoms object.

The *rep* argument should be a sequence of three positive integers like *(2,3,1)* or a single integer (*r*) equivalent to *(r,r,r)*.

rotate(*a*, *v=None*, *center=(0, 0, 0)*, *rotate_cell=False*)
Rotate atoms based on a vector and an angle, or two vectors.

Parameters:

a = None: Angle that the atoms is rotated around the vecor 'v'. 'a' can also be a vector and then 'a' is rotated into 'v'.

v: Vector to rotate the atoms around. Vectors can be given as strings: 'x', '-x', 'y', ...

center = (0, 0, 0): The center is kept fixed under the rotation. Use 'COM' to fix the center of mass, 'COP' to fix the center of positions or 'COU' to fix the center of cell.

rotate_cell = False: If true the cell is also rotated.

Examples:

Rotate 90 degrees around the z-axis, so that the x-axis is rotated into the y-axis:

```
>>> from math import pi
>>> atoms = Atoms()
>>> atoms.rotate(90, 'z')
>>> atoms.rotate(90, (0, 0, 1))
>>> atoms.rotate(-90, '-z')
>>> atoms.rotate('x', 'y')
```

rotate_dihedral(*a1*, *a2=None*, *a3=None*, *a4=None*, *angle=None*, *mask=None*)
Rotate dihedral angle.

Complementing the two routines above: rotate a group by a predefined dihedral angle, starting from its current configuration

rotate_euler(*center=(0, 0, 0)*, *phi=0.0*, *theta=0.0*, *psi=0.0*)

set_angle(*a1*, *a2=None*, *a3=None*, *angle=None*, *mask=None*)
Set angle (in degrees) formed by three atoms.

Sets the angle between vectors a2->a1 and a2->a3.

Same usage as in set_dihedral().

set_array(*name*, *a*, *dtype=None*, *shape=None*)
Update array.

If *shape* is not *None*, the shape of *a* will be checked. If *a* is *None*, then the array is deleted.

set_atomic_numbers(*numbers*)
Set atomic numbers.

set_calculator(*calc=None*)
Attach calculator object.

set_cell(*cell*, *scale_atoms=False*)
Set unit cell vectors.

Parameters:

cell: 3x3 matrix or length 3 or 6 vector Unit cell. A 3x3 matrix (the three unit cell vectors) or just three numbers for an orthorhombic cell. Another option is 6 numbers, which describes unit cell with lengths of unit cell vectors and with angles between them (in degrees), in following order: [len(a), len(b), len(c), angle(b,c), angle(a,c), angle(a,b)]. First vector will lie in x-direction, second in xy-plane, and the third one in z-positive subspace.

scale_atoms: bool Fix atomic positions or move atoms with the unit cell? Default behavior is to *not* move the atoms (scale_atoms=False).

Examples:

Two equivalent ways to define an orthorhombic cell:

```
>>> atoms = Atoms('He')
>>> a, b, c = 7, 7.5, 8
>>> atoms.set_cell([a, b, c])
>>> atoms.set_cell([(a, 0, 0), (0, b, 0), (0, 0, c)])
```

FCC unit cell:

```
>>> atoms.set_cell([(0, b, b), (b, 0, b), (b, b, 0)])
```

Hexagonal unit cell:

```
>>> atoms.set_cell([a, a, c, 90, 90, 120])
```

Rhombohedral unit cell:

```
>>> alpha = 77
>>> atoms.set_cell([a, a, a, alpha, alpha, alpha])
```

set_celldisp (*celldisp*)
Set the unit cell displacement vectors.

set_chemical_symbols (*symbols*)
Set chemical symbols.

set_constraint (*constraint=None*)
Apply one or more constrains.

The *constraint* argument must be one constraint object or a list of constraint objects.

set_dihedral (*a1*, *a2=None*, *a3=None*, *a4=None*, *angle=None*, *mask=None*, *indices=None*)
Set the dihedral angle (degrees) between vectors a1->a2 and a3->a4 by changing the atom indexed by a4 if mask is not None, all the atoms described in mask (read: the entire subgroup) are moved. Alternatively to the mask, the indices of the atoms to be rotated can be supplied.

example: the following defines a very crude ethane-like molecule and twists one half of it by 30 degrees.

```
>>> from math import pi
>>> atoms = Atoms('HHCCHH', [[-1, 1, 0], [-1, -1, 0], [0, 0, 0],
...                          [1, 0, 0], [2, 1, 0], [2, -1, 0]])
>>> atoms.set_dihedral(1, 2, 3, 4, 210, mask=[0, 0, 0, 1, 1, 1])
```

set_distance (*a0*, *a1*, *distance*, *fix=0.5*, *mic=False*)
Set the distance between two atoms.

Set the distance between atoms *a0* and *a1* to *distance*. By default, the center of the two atoms will be fixed. Use *fix=0* to fix the first atom, *fix=1* to fix the second atom and *fix=0.5* (default) to fix the center of the bond.

set_initial_charges (*charges=None*)
Set the initial charges.

set_initial_magnetic_moments (*magmoms=None*)
Set the initial magnetic moments.

Use either one or three numbers for every atom (collinear or non-collinear spins).

set_masses (*masses='defaults'*)
Set atomic masses.

The array masses should contain a list of masses. In case the masses argument is not given or for those elements of the masses list that are None, standard values are set.

set_momenta (*momenta*, *apply_constraint=True*)
Set momenta.

set_pbc (*pbc*)
Set periodic boundary condition flags.

set_positions(*newpositions*, *apply_constraint=True*)
Set positions, honoring any constraints. To ignore constraints, use *apply_constraint=False*.

set_scaled_positions(*scaled*)
Set positions relative to unit cell.

set_tags(*tags*)
Set tags for all atoms. If only one tag is supplied, it is applied to all atoms.

set_velocities(*velocities*)
Set the momenta by specifying the velocities.

translate(*displacement*)
Translate atomic positions.

The displacement argument can be a float an xyz vector or an nx3 array (where n is the number of atoms).

velocities

wrap(*center=(0.5, 0.5, 0.5)*, *pbc=None*, *eps=1e-07*)

write(*filename*, *format=None*, ***kwargs*)
Write atoms object to a file.

see ase.io.write for formats. kwargs are passed to ase.io.write.

structopt.common.individual.mutations

class structopt.common.individual.mutations.**Mutations**(*parameters*)
Bases: object

static move_atoms(*individual*, *max_natoms=0.2*)
Randomly moves atoms within the individual (in place).

Parameters

- **individual** (Individual) – an individual
- **max_natoms** (*float or int*) – if float, the maximum number of atoms that will be moved is max_natoms*len(individual) if int, the maximum number of atoms that will be moved is max_natoms default: 0.20

mutate(*individual*)

static permutation(*individual*)
Swaps the chemical symbol between two elements

Parameters **individual** (Individual) – An individual or atoms object.

post_processing(*individual*)

static rattle(*individual*, *stdev=0.5*, *x_avg_bond=True*)
Randomly displace all atoms in a random direction with a magnitude drawn from a gaussian distribution.

Parameters

- **individual** (Individual) – An individual
- **stdev** (*float*) – The standard deviation of the gaussian distribution to rattle all the atoms. If x_avg_bond is set to True, given as the fraction of the average bond length of the material.
- **x_avg_bond** (*bool*) – If True, the gaussian distributions standard deviation is stdev * avg_bond_length. Note, this only applies to fcc, hcp, or bcc materials.

static rotate_all(*atoms*, *vector=None*, *angle=None*, *center=None*)
Rotate all atoms around a single point. Most suitable for cluster calculations.

Parameters

- **individual** (Individual) – An individual.
- **vector** (*string or list*) – The list of axes in which to rotate the atoms around. If None, is a randomly chosen direction. If 'random' in list, a random vector can be chosen.
- **angle** (*string or list*) – A list of angles that will be chosen to rotate. If None, is randomly generated. Angle must be given in radians. If 'random' in list, a random angle is included.
- **center** (*string or xyz iterable*) – The center in which to rotate the atoms around. If None, defaults to center of mass. Acceptable strings are COM = center of mass COP = center of positions COU = center of cell

static rotate_atoms(*individual*, *max_natoms=0.2*)
Randomly rotates a number of random atoms within the individual (in place).

Parameters

- **individual** (Individual) – an individual
- **max_natoms** (*float or int*) – if float, the maximum number of atoms that will be rotated is max_natoms*len(individual) if int, the maximum number of atoms that will be rotated is max_natoms default: 0.20

static rotate_cluster(*individual*, *max_natoms=0.2*)
Randomly rotates a random cluster of atoms within the individual (in place).

Parameters

- **individual** (Individual) – an individual
- **max_natoms** (*float or int*) – if float, the maximum number of atoms that will be rotated is max_natoms*len(individual) if int, the maximum number of atoms that will be rotated is max_natoms default: 0.20

select_mutation()

static swap_positions(*individual*, *max_natoms=0.2*)
Randomly swaps the positions atoms within the individual (in place).

Parameters

- **individual** (Individual) – an individual
- **max_natoms** (*float or int*) – if float, the maximum number of atoms whose positions will be swapped is max_natoms*len(individual) if int, the maximum number of atoms whose positions will be swapped is max_natoms if the number of atoms to be swapped is (or evaluates to) an odd integer, it is rounded down to an even integer max_natoms corresponds to the maximum number of atoms whose positions will change default: 0.20

static swap_species(*individual*, *max_natoms=0.2*)
Randomly swaps the species of atoms within the individual (in place).

Parameters

- **individual** (Individual) – an individual
- **max_natoms** (*float or int*) – if float, the maximum number of atoms that will be swapped is max_natoms*len(individual) if int, the maximum number of atoms that will be

swapped is max_natoms if the number of atoms to be swapped is (or evaluates to) an odd integer, it is rounded down to an even integer max_natoms corresponds to the maximum number of atoms whose species will change default: 0.20

structopt.common.individual.fitnesses

class structopt.common.individual.fitnesses.**Fitnesses**(*parameters*)
Bases: object

calculate_fitness(*individual*)
Perform the fitness calculations on an individual.

Args: individual (Individual): the individual to evaluate

(@parallel) Designed to run code that runs differently on different cores. The MPI functionality should be implemented inside these functions.

post_processing()

class structopt.common.individual.fitnesses.**LAMMPS**(*parameters*)
Bases: object

LAMMPS class for running LAMMPS on a single individual. Takes a dictionary, where the key: value are the parameters for running LAMMPs.

Parameters

- **min_style** (*str*) – The minimization scheme for running LAMMPS. See LAMMPS doc.
- **min_modify** (*str*) – Parameters for min_style energy minimization algorithm. See LAMMPS doc.
- **minimize** (*str*) – Convergence criteria for minimization algorithm. Note for fitness values, the last two values are set to 0, so no relaxation is done. See LAMMPS doc.
- **pair_style** (*str*) – Type of potential used. See LAMMPS doc.
- **potential_file** (*str*) – The path to the potential_file. Should be absolute.
- **thermo_steps** (*int*) – How much output to print of thermodynamic information. If set to 0, only the last step is printed.See LAMMPS doc.
- **keep_file** (*bool*) – Will keep all of the LAMMPS input and output files for each individual. Use with caution.
- **reference** (*dict*) – Reference energies of the particle. These are values to subtract from the values returned by LAMMPS. Given as a dictionary of {sym : E} pairs, where sym is a str denoating the the element, while E is the value to be subtracted per sym. This is typically the pure component formation energy calculated with LAMMPS. Note since this is merely a fixed subtraction, should not change the performance in constant composition runs.

calculate_fitness(*individual*)

get_command(*individual*)

normalize(*E*, *individual*)

reference(*E*, *individual*)
References the energy of the cluster to a reference energy

class structopt.common.individual.fitnesses.**FEMSIM**(*parameters*)
Bases: object

Contains parameters and functions for running FEMSIM through Python.

chi2(*vk*)

get_spawn_args(*individual*)
Returns a dictionary of arguments to be passed to MPI.COMM_SELF.Spawn which will be collected for all structures and concatenated into MPI.COMM_SELF.Spawn_multiple: https://github.com/mpi4py/mpi4py/blob/2acfc552c42846628304e54a3b87e2bf3a59af07/src/mpi4py/MPI/Comm.pyx#L1555

get_vk_data()

read_inputs(*parameters*)

setup_individual_evaluation(*individual*)

update_parameters(***kwargs*)

write_paramfile(*individual*)

structopt.common.individual.relaxations

class structopt.common.individual.relaxations.**Relaxations**(*parameters*)
Bases: object

post_processing()

relax(*individual*, *generation=None*)
Relax an individual.

Args: individual (Individual): the individual to relax

(@parallel) Designed to run code that runs differently on different cores. The MPI functionality should be implemented inside these functions.

class structopt.common.individual.relaxations.**LAMMPS**(*parameters*)
Bases: object

LAMMPS class for running LAMMPS on a single individual. Takes a dictionary, where the key: value are the parameters for running LAMMPs.

Parameters

- **min_style** (*str*) – The minimization scheme for running LAMMPS. See LAMMPS doc.
- **min_modify** (*str*) – Parameters for min_style energy minimization algorithm. See LAMMPS doc.
- **minimize** (*str*) – Convergence criteria for minimization algorithm. See LAMMPS doc.
- **pair_style** (*str*) – Type of potential used. See LAMMPS doc.
- **potential_file** (*str*) – The path to the potential_file. Should be absolute.
- **thermo_steps** (*int*) – How much output to print of thermodynamic information. If set to 0, only the last step is printed.See LAMMPS doc.
- **keep_file** (*bool*) – Will keep all of the LAMMPS input and output files for each individual. Use with caution.

- **repair** (*bool*) – Determines whether to run an algorithm to make sure no atoms are in "space". Atoms can be in space due to a mutation or crossover that results in a large force that shoots the atom outside of the particle.

get_command(*individual*)

relax(*individual*)
Relax an individual.

Args: individual (Individual): the individual to relax

(@parallel) Designed to run code that runs differently on different cores. The MPI functionality should be implemented inside these functions.

repair(*individual*, *generation*)

Repairs an individual. Currently takes isolated atoms moves them next to a non-isolated atom

(@parallel) Designed to run code that runs differently on different cores. The MPI functionality should be implemented inside these functions.

class structopt.common.individual.relaxations.**hard_sphere_cutoff**(*parameters*, *cutoff=0.7*)

Bases: object

A relaxation module to ensure atoms in an individual are not too close together. This is often a preliminary relaxation before LAMMPS for VASP to ensure the models do not explode.

relax(*individual*)
Relaxes the individual using a hard-sphere cutoff method. :param individual: the individual to relax :type individual: Individual

structopt.common.individual.generators

structopt.common.individual.generators.**read_xyz**(*filename*)

structopt.common.individual.fingerprinters

CHAPTER 4

Contributing

Bug fixes and error reports are always welcome. We accept PRs and will try to fix issues that have detailed descriptions and are reproducable in a timely fashion.

If you have a forward simulation module that you wish to contribute, please make an issue and the correct people will get email notifications so we can respond.

CHAPTER 5

License Agreement

StructOpt is distributed under the MIT license, reproduced below:

CHAPTER 6

Index and Search

- genindex
- modindex
- search

Index

A

add() (structopt.common.population.Population method), 36
adsorbate_info (structopt.common.individual.Individual attribute), 43
allgather() (in module structopt.tools.parallel), 35
allgather() (structopt.common.population.Population method), 36
append() (structopt.common.individual.Individual method), 43
apply_fingerprinters() (structopt.common.population.Population method), 37

B

bcast() (structopt.common.population.Population method), 37
best() (in module structopt.common.population.predators), 19
best() (in module structopt.common.population.selections), 18
best() (structopt.common.population.predators.Predators static method), 40
best() (structopt.common.population.selections.Selections static method), 42

C

calc (structopt.common.individual.Individual attribute), 43
calculate_fitness() (structopt.common.individual.fitnesses.Fitnesses method), 53
calculate_fitness() (structopt.common.individual.fitnesses.LAMMPS method), 53
calculate_fitness() (structopt.common.individual.Individual method), 43
calculate_fitnesses() (structopt.common.population.fitnesses.Fitnesses method), 39
calculate_fitnesses() (structopt.common.population.Population method), 37
cell (structopt.common.individual.Individual attribute), 43
center() (structopt.common.individual.Individual method), 43
chi2() (structopt.common.individual.fitnesses.FEMSIM method), 54
clear() (structopt.common.individual.Individual method), 44
clear() (structopt.common.population.Population method), 37
constraints (structopt.common.individual.Individual attribute), 44
copy() (structopt.common.individual.Individual method), 44
copy() (structopt.common.population.Population method), 37
crossover() (structopt.common.population.crossovers.Crossovers method), 38
crossover() (structopt.common.population.Population method), 37
Crossovers (class in structopt.common.population.crossovers), 38

E

edit() (structopt.common.individual.Individual method), 44
ellipsoid() (in module structopt.cluster.individual.generators), 14
enrich_bulk() (in module structopt.cluster.individual.mutations), 23
enrich_surface() (in module structopt.cluster.individual.mutations), 23
enrich_surface_defects() (in module structopt.cluster.individual.mutations), 24

enrich_surface_facets() (in module structopt.cluster.individual.mutations), 24
euler_rotate() (structopt.common.individual.Individual method), 44
extend() (structopt.common.individual.Individual method), 44
extend() (structopt.common.population.Population method), 37

F

fcc() (in module structopt.cluster.individual.generators), 15
FEMSIM (class in structopt.common.individual.fitnesses), 53
fitness (structopt.common.individual.Individual attribute), 44
fitness() (in module structopt.common.population.fitnesses.FEMSIM), 39
fitness() (in module structopt.common.population.fitnesses.LAMMPS), 39
Fitnesses (class in structopt.common.individual.fitnesses), 53
Fitnesses (class in structopt.common.population.fitnesses), 39
fits (structopt.common.individual.Individual attribute), 44
fromkeys() (structopt.common.population.Population method), 37
fuss() (in module structopt.common.population.predators), 20
fuss() (structopt.common.population.predators.Predators static method), 40

G

generate() (structopt.common.individual.Individual method), 44
get() (structopt.common.population.Population method), 37
get_all_distances() (structopt.common.individual.Individual method), 44
get_angle() (structopt.common.individual.Individual method), 44
get_angular_momentum() (structopt.common.individual.Individual method), 44
get_array() (structopt.common.individual.Individual method), 45
get_atom_indices_within_distance_of_atom() (structopt.common.individual.Individual method), 45
get_atomic_numbers() (structopt.common.individual.Individual method), 45
get_by_position() (structopt.common.population.Population method), 37
get_calculator() (structopt.common.individual.Individual method), 45
get_cell() (structopt.common.individual.Individual method), 45
get_cell_lengths_and_angles() (structopt.common.individual.Individual method), 45
get_celldisp() (structopt.common.individual.Individual method), 45
get_center_of_mass() (structopt.common.individual.Individual method), 45
get_charges() (structopt.common.individual.Individual method), 45
get_chemical_formula() (structopt.common.individual.Individual method), 45
get_chemical_symbols() (structopt.common.individual.Individual method), 45
get_command() (structopt.common.individual.fitnesses.LAMMPS method), 53
get_command() (structopt.common.individual.relaxations.LAMMPS method), 55
get_dihedral() (structopt.common.individual.Individual method), 45
get_dipole_moment() (structopt.common.individual.Individual method), 46
get_distance() (structopt.common.individual.Individual method), 46
get_distances() (structopt.common.individual.Individual method), 46
get_forces() (structopt.common.individual.Individual method), 46
get_initial_charges() (structopt.common.individual.Individual method), 46
get_initial_magnetic_moments() (structopt.common.individual.Individual method), 46
get_kinetic_energy() (structopt.common.individual.Individual method), 46
get_magnetic_moment() (structopt.common.individual.Individual method), 46
get_magnetic_moments() (structopt.common.individual.Individual method), 46

get_masses() (structopt.common.individual.Individual method), 46
get_momenta() (structopt.common.individual.Individual method), 46
get_moments_of_inertia() (structopt.common.individual.Individual method), 46
get_nearest_atom_indices() (structopt.common.individual.Individual method), 46
get_new_id() (structopt.common.population.Population method), 37
get_number_of_atoms() (structopt.common.individual.Individual method), 46
get_pbc() (structopt.common.individual.Individual method), 46
get_positions() (structopt.common.individual.Individual method), 47
get_potential_energies() (structopt.common.individual.Individual method), 47
get_potential_energy() (structopt.common.individual.Individual method), 47
get_reciprocal_cell() (structopt.common.individual.Individual method), 47
get_scaled_positions() (structopt.common.individual.Individual method), 47
get_spawn_args() (structopt.common.individual.fitnesses.FEMSIM method), 54
get_stress() (structopt.common.individual.Individual method), 47
get_stresses() (structopt.common.individual.Individual method), 47
get_tags() (structopt.common.individual.Individual method), 47
get_temperature() (structopt.common.individual.Individual method), 47
get_total_energy() (structopt.common.individual.Individual method), 47
get_velocities() (structopt.common.individual.Individual method), 47
get_vk_data() (structopt.common.individual.fitnesses.FEMSIM method), 54
get_volume() (structopt.common.individual.Individual method), 47

H

hard_sphere_cutoff (class in structopt.common.individual.relaxations), 55
has() (structopt.common.individual.Individual method), 47

I

Individual (class in structopt.common.individual), 43
items() (structopt.common.population.Population method), 37

K

keys() (structopt.common.population.Population method), 37
kill() (structopt.common.population.Population method), 37
kill() (structopt.common.population.predators.Predators method), 40

L

LAMMPS (class in structopt.common.individual.fitnesses), 26, 53
LAMMPS (class in structopt.common.individual.relaxations), 25, 54
load_modules() (structopt.common.individual.Individual method), 47
load_modules() (structopt.common.population.Population method), 37
logger_utils() (in module structopt.io), 35

M

move_atoms() (in module structopt.cluster.individual.mutations), 22
move_atoms() (structopt.common.individual.mutations.Mutations static method), 51
move_surface_atoms() (in module structopt.cluster.individual.mutations), 22
move_surface_defects() (in module structopt.cluster.individual.mutations), 23
mutate() (structopt.common.individual.Individual method), 48
mutate() (structopt.common.individual.mutations.Mutations method), 51
mutate() (structopt.common.population.mutations.Mutations method), 40
mutate() (structopt.common.population.Population method), 37
Mutations (class in structopt.common.individual.mutations), 51
Mutations (class in structopt.common.population.mutations), 40

N

new_array() (structopt.common.individual.Individual method), 48
normalize() (structopt.common.individual.fitnesses.LAMMPS method), 53
number_of_lattice_vectors (structopt.common.individual.Individual attribute), 48
numbers (structopt.common.individual.Individual attribute), 48

P

parallel() (in module structopt.tools.parallel), 35
parameters() (in module structopt.io), 35
pbc (structopt.common.individual.Individual attribute), 48
permutation() (in module structopt.common.individual.mutations), 22
permutation() (structopt.common.individual.mutations.Mutations static method), 51
poor2rich() (in module structopt.cluster.individual.mutations), 23
pop() (structopt.common.individual.Individual method), 48
pop() (structopt.common.population.Population method), 38
popitem() (structopt.common.population.Population method), 38
Population (class in structopt.common.population), 36
position() (structopt.common.population.Population method), 38
positions (structopt.common.individual.Individual attribute), 48
post_processing() (structopt.common.individual.fitnesses.Fitnesses method), 53
post_processing() (structopt.common.individual.mutations.Mutations method), 51
post_processing() (structopt.common.individual.relaxations.Relaxations method), 54
post_processing() (structopt.common.population.crossovers.Crossovers method), 38
post_processing() (structopt.common.population.fitnesses.Fitnesses method), 39
post_processing() (structopt.common.population.mutations.Mutations method), 40
post_processing() (structopt.common.population.predators.Predators method), 41
post_processing() (structopt.common.population.relaxations.Relaxations method), 39
post_processing() (structopt.common.population.selections.Selections method), 42
Predators (class in structopt.common.population.predators), 40

R

random_selection() (in module structopt.common.population.selections), 17
random_selection() (structopt.common.population.selections.Selections static method), 42
rank() (in module structopt.common.population.predators), 19
rank() (in module structopt.common.population.selections), 18
rank() (structopt.common.population.predators.Predators static method), 41
rank() (structopt.common.population.selections.Selections static method), 42
rattle() (in module structopt.common.individual.mutations), 22
rattle() (structopt.common.individual.Individual method), 48
rattle() (structopt.common.individual.mutations.Mutations static method), 51
read_inputs() (structopt.common.individual.fitnesses.FEMSIM method), 54
read_xyz() (in module structopt.common.individual.generators), 55
read_xyz() (in module structopt.io), 35
reference() (structopt.common.individual.fitnesses.LAMMPS method), 53
relax() (in module structopt.common.population.relaxations.hard_sphere_cutoff), 40
relax() (in module structopt.common.population.relaxations.LAMMPS), 39
relax() (structopt.common.individual.Individual method), 48
relax() (structopt.common.individual.relaxations.hard_sphere_cutoff method), 55
relax() (structopt.common.individual.relaxations.LAMMPS method), 55
relax() (structopt.common.individual.relaxations.Relaxations method), 54
relax() (structopt.common.population.Population method), 38
relax() (structopt.common.population.relaxations.Relaxations method), 39

Relaxations (class in structopt.common.individual.relaxations), 54
Relaxations (class in structopt.common.population.relaxations), 39
remove() (structopt.common.population.Population method), 38
repair() (structopt.common.individual.relaxations.LAMMPS method), 55
repeat() (structopt.common.individual.Individual method), 48
replace() (structopt.common.population.Population method), 38
rich2poor() (in module structopt.cluster.individual.mutations), 23
root() (in module structopt.tools.parallel), 35
rotate() (in module structopt.cluster.population.crossovers), 16
rotate() (structopt.common.individual.Individual method), 48
rotate() (structopt.common.population.crossovers.Crossovers static method), 38
rotate_all() (in module structopt.common.individual.mutations), 21
rotate_all() (structopt.common.individual.mutations.Mutations static method), 51
rotate_atoms() (in module structopt.common.individual.mutations), 21
rotate_atoms() (structopt.common.individual.mutations.Mutations static method), 52
rotate_cluster() (in module structopt.cluster.individual.mutations), 22
rotate_cluster() (structopt.common.individual.mutations.Mutations static method), 52
rotate_dihedral() (structopt.common.individual.Individual method), 49
rotate_euler() (structopt.common.individual.Individual method), 49
roulette() (in module structopt.common.population.predators), 19
roulette() (in module structopt.common.population.selections), 18
roulette() (structopt.common.population.predators.Predators static method), 41
roulette() (structopt.common.population.selections.Selections static method), 42
run_pso_moves() (structopt.common.population.Population method), 38

S

select() (structopt.common.population.Population method), 38
select() (structopt.common.population.selections.Selections method), 43
select_crossover() (structopt.common.population.crossovers.Crossovers method), 39
select_mutation() (structopt.common.individual.mutations.Mutations method), 52
select_predator() (structopt.common.population.predators.Predators method), 41
select_selection() (structopt.common.population.selections.Selections method), 43
Selections (class in structopt.common.population.selections), 42
set_angle() (structopt.common.individual.Individual method), 49
set_array() (structopt.common.individual.Individual method), 49
set_atomic_numbers() (structopt.common.individual.Individual method), 49
set_calculator() (structopt.common.individual.Individual method), 49
set_cell() (structopt.common.individual.Individual method), 49
set_celldisp() (structopt.common.individual.Individual method), 50
set_chemical_symbols() (structopt.common.individual.Individual method), 50
set_constraint() (structopt.common.individual.Individual method), 50
set_dihedral() (structopt.common.individual.Individual method), 50
set_distance() (structopt.common.individual.Individual method), 50
set_initial_charges() (structopt.common.individual.Individual method), 50
set_initial_magnetic_moments() (structopt.common.individual.Individual method), 50
set_masses() (structopt.common.individual.Individual method), 50
set_momenta() (structopt.common.individual.Individual method), 50
set_pbc() (structopt.common.individual.Individual method), 50
set_positions() (structopt.common.individual.Individual method), 50
set_scaled_positions() (structopt.common.individual.Individual method),

51
set_tags() (structopt.common.individual.Individual method), 51
set_velocities() (structopt.common.individual.Individual method), 51
setdefault() (structopt.common.population.Population method), 38
setup_individual_evaluation() (structopt.common.individual.fitnesses.FEMSIM method), 54
single_core() (in module structopt.tools.parallel), 35
sphere() (in module structopt.cluster.individual.generators), 15
swap_core_shell() (in module structopt.cluster.individual.mutations), 23
swap_positions() (in module structopt.common.individual.mutations), 21
swap_positions() (structopt.common.individual.mutations.Mutations static method), 52
swap_species() (in module structopt.common.individual.mutations), 21
swap_species() (structopt.common.individual.mutations.Mutations static method), 52

T

tournament() (in module structopt.common.population.predators), 20
tournament() (in module structopt.common.population.selections), 17
tournament() (structopt.common.population.predators.Predators static method), 41
tournament() (structopt.common.population.selections.Selections static method), 43
translate() (structopt.common.individual.Individual method), 51
twist() (in module structopt.cluster.individual.mutations), 23

U

update() (structopt.common.population.Population method), 38
update_parameters() (structopt.common.individual.fitnesses.FEMSIM method), 54

V

values() (structopt.common.population.Population method), 38
velocities (structopt.common.individual.Individual attribute), 51

W

wrap() (structopt.common.individual.Individual method), 51
write() (structopt.common.individual.Individual method), 51
write_paramfile() (structopt.common.individual.fitnesses.FEMSIM method), 54
write_xyz() (in module structopt.io), 35