structopt Documentation

Release 0.5

UW Madison, Materials Science

Dec 16, 2018

Contents

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 an atomic structure by, for example, moving atoms in the structure with the goal to minimize a function (such as system energy). 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, bond angle constraints, etc.

Different optimization algorithms can be used to minimize the different metrics including Monte Carlo, Genetic Algorithm, and Particle Swarm algorithms.

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. 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. However, parts of this documentation are likely lacking. If you have questions, please post them as an issue on github.

StructOpt serves the purpose of structure refinment for multiple different materials including nanoparticles and metallic glasses and is highly customizable and extendable to new structure types. There are many different types of simulations that can be set up, which requires getting to know the relevent parameters. Examples are included in the github repository and comments via issues on our github page are welcome.

CHAPTER 3

Contents

3.1 Core Concepts

Detailed information about StructOpt can be found in our paper: [in the submission process]

3.1.1 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 StructOpt an "individual" is an atomic model. A population of atomic models is first generated. Given this population, pairs of individuals are mated (aka crossed over) by selecting different volumes of different models and joining them. 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.

3.2 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.

3.2.1 Python Libraries

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

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.

3.2.2 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/v2-experiments-and-energy/structopt`) to your `PYTHONPATH` environment variable.

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

3.3 Input Parameters

The input parameters are defined in a JSON file as a single dictionary. Due to the modular nature of StructOpt, the input file is a dictionary of dictionaries where keys and values often relate directly to function names and kwargs.

The parameters for a simulation can be defined in the optimizer file using `structopt.setup(parameters)` where `parameters` is either a filename or a dictionary.

The parameters for a simple Au nanoparticle example that finds the lowest energy configuration using an EAM potential in LAMMPS is given blow. Each part of the parameters will be discussed in the following sections.

Example:

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

(continues on next page)

(continued from previous page)

```
        }
    },
    "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": 1.0}
    },
    "selections": {
        "rank": {"probability": 1.0}
    }
}
```

3.3.1 Global Parameters

Global parameters define parameters that pertain to the entire simulation.

structure_type

`structure_type (str)`: Defines the type of material being optimized. StructOpt currently supports the `periodic` and `aperiodic` structure types. The `periodic` structure type defines periodic boundary conditions in all three directions, and the `aperiodic` structure type has no periodic boundary conditions.

seed

`seed (int)`: The seed for the psuedo-random number generator. Two runs with identical input and seed should run identically (however, in rare cases rounding errors in the machine may cause some selection schemes to choose different individuals in runs that should be identical).

convergence

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

Example:

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

post_processing

`post_processing (dict)`: Determines the outputs of the simulation. 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 is the default behavior because saving every individual is both time and disk-space intensive).

Example:

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

3.3.2 Generators

Generators are functions for initializing the population. They 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 unless code is added to the optimizer to change the population size. *kwargs_i* are dictionaries that define the kwargs to the generator function being used. The kwargs are specific to the function and can be found in their help function, show below.

3.3.3 Crossovers

Crossovers are operations for combing 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 parents. Often the parents are chosen to be the 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 that will be used. 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 being used. These kwargs 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.

3.3.4 Selections

Selections are operations for choosing which individuals to "mate" when producing new individuals. Individuals are chosen based on their fitness, and different selection functions determine how which individuals will be mated. The selection scheme impacts the diversity of subsequent populations.

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.

3.3.5 Predators

Similar to selections, predators are selection processes that select 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 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.

3.3.6 Mutations

Mutations are operations applied to an individual that changes its structure and composition. It is a local search operation, although 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 being used. 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 being used. These will be specific to the function and can be found in their help function.

In addition to specifying the mutations 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 individuals themselves. This means the original individual is not changed during a mutation. `keep_original_best` applies `keep_original` to only the best individual.

The currently implemented mutations can be found in the `structopt/*/individual/mutations` folders depending on the structure typing being used. 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.

3.3.7 Relaxations

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

Relaxations differ from 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 being used. 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 being used. 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 the structure using a potential. Most of the kwargs can be found from the LAMMPS documentation.

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 a pull request to this repository and the potential will be added. Currently available potentials can be found in the `potentials/` directory.

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))

3.3.8 Fitnesses

Fitnesses evaluate the "goodness" of the individual, for example the simulated energy of the structure. Lower fitness values are better.

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. Note 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.

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 a pull request to this repository 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))

3.3.9 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,
        "use_mpi4py": true,
        "kwargs": {
            "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
        }
    }
}
```

3.4 Outputs

All outputs are contained in a log folder with the timestamp when the simulation started (with 1-second resolution).

3.5 Examples

Examples can be found in the `examples/` directory on github.

3.5.1 Running StructOpt

StructOpt can be run on a single processor or in parallel using MPI. Depending on the cluster/environment you are using, you may need to load the following modules:

```
module load lammps-31Jan14
module load compile/intel
module load mpi/intel/openmpi-1.10.2
```

StructOpt can be run serially using the following command:

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

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

3.6 Parallelism

In general, the parallelized parts of StructOpt are the fitness and relaxation modules.

StructOpt's fitness and relaxation modules allow 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 running 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 (see documentation online for `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 processed 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.

3.6.1 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

mpi4py needs to be installed from source against OpenMPI 1.10.2 because (at the time of developing this package) newer versions of OpenMPI had bugs in their implementation of `MPI_Comm_spawn_multiple`. Follow the instructions here under "3.3: Using distutils". In short:

```
# Setup modules so that `mpi/intel/openmpi` is loaded and `mpirun` 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
```

Depending on the installation and hardware, the following parameters may need to be added when running a StructOpt simulation: `-mca btl tcp,sm,self` forces the ethernet interfaces to use TCP rather than infiniband.

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 `mpirun` 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).

3.7 JobManager

3.7.1 Introduction

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

3.7.2 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.

3.7.3 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'

structopt_parameters = {
    "seed": 0,
    "structure_type": "cluster",
    ...
}

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

structopt_parameters = {
    "seed": 0,
    "structure_type": "cluster",
    ...
}

submit_parameters = {'system': 'PBS',
                     'queue': 'morgan2',
```

(continues on next page)

(continued from previous page)

```
                    '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
```

3.7.4 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
```

(continues on next page)

(continued from previous page)

```
calcdir = 'job_manager_examples/Au55-example'

structopt_parameters = {
    "seed": 0,
    "structure_type": "cluster",
    ...
}

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
```

3.7.5 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'

structopt_parameters = {
    "seed": 0,
    "structure_type": "aperiodic",
    ...
}

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

(continues on next page)

(continued from previous page)

```
optimizer = 'genetic.py'

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

3.8 Relaxation and Fitness Modules

3.8.1 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

3.8.2 VASP

VASP currently cannot be run within StructOpt. We have done quite a bit of work to get it running, but it isn't working yet. If you'd like to work on this, please post an issue on github.

3.8.3 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

3.8.4 STEM

The STEM software is included in the StructOpt repository.

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

3.8.5 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 a structure can be integrated into StructOpt. Contact the developers by making an issue on github to get in touch with us.

3.9 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. Machine learning techniques are being developed for optimization problems at an extremely fast rate. This requires research efforts to evolve equally quickly. 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.

3.10 Troubleshooting

For now, please see any issues on github.

3.11 StructOpt Package

3.11.1 Submodules

structopt.io

structopt.tools

structopt.postprocessing

3.11.2 The Optimizer

structopt.Optimizer

structopt.common.population.Population

structopt.common.population.crossovers

structopt.common.population.fitnesses

structopt.common.population.relaxations

structopt.common.population.mutations

structopt.common.population.predators

structopt.common.population.selections

structopt.common.individual.Individual

structopt.common.individual.mutations

structopt.common.individual.fitnesses

structopt.common.individual.relaxations

structopt.common.individual.generators

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.

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