GASP: The Genetic Algorithm for Structure and Phase Prediction

Will Tipton and Richard Hennig

Last updated: April 25, 2014
Abstract

The GASP code is contains a heuristic search algorithm to solve the atomic structure prediction problems. It includes most successful techniques described in the literature and a few important new ones. It can search for crystalline structures as well as those with periodicity in fewer than 3 dimensions. It is interfaced with a number of energy codes including GULP, VASP, and LAMMPS. This document describes the methodology behind the project as well as the practical aspects of the code’s implementation and use.
Chapter 1

Methodology

NB: Both the review of the literature and the discussion of our own methodology in this chapter are out of date. For a more recent literature review, please see [13], and for a description of our own work, please see [14]

1.1 Context and motivation

Many practical materials science problems are effectively the search for a material with certain properties. Determining the structure of a material is an important first step in finding its properties without experimental work. There have been successes in approaching the structure prediction problem through examination of known materials, both intuition-based and not [6]. However, there would be some advantage to a first-principles method. Such a method would work on a large class of materials without relying on intuition and would be able to predict structures completely different from those already known.

The stable structure of a material will be that which minimizes its quantum-mechanical free energy. So, the structure prediction problem is equivalent to the problem of finding the global minimum of the energy functional, which becomes our focus. Now, in general we have no analytical expression for the energy functional; we can only compute it for specific inputs. So we must perform some sort of guess-and-check type search over the space of solutions.

A number of search strategies have been implemented and tested including simulated annealing, minima hopping, and some monte-carlo methods. However, genetic algorithms modeled after natural evolution have shown much promise, and that is what we implement here. Glass et. al. have a good discussion of why the genetic strategy is particularly well suited to this problem.[8]

The evolutionary approach has also been tried a few times before [4] [5] [7] [9] [8] [12] [15] [16]. Here, we implement and evaluate techniques described in this literature as well as several improvements of our own.
1.2 The biological analogy

In nature, genetic information is carried in organisms. It is maintained in a population’s gene pool if it is passed through reproduction to successful offspring. New information can be introduced to the gene pool through mutation events, but these are rare (and usually lethal). The success that an organism has in passing on its genes is often given a value between zero and one and called the organism’s fitness.

The fitness of an organism is not universal but depends on its environment. A lion would do poorly in the Arctic and a great-white shark would do poorly in Kansas. More subtly, there is variance of traits within a single species. In some cases, these differences can lead to a difference in the organisms’ fitness. Thus, the relevant traits of the higher-fitness individuals are likely to be more common in subsequent generations. In this way populations (but not individuals) evolve to be well suited to their environment.

This assumes, of course, the well understood fact that relevant traits are passed on, to varying degrees, from parents to offspring. The correlation between a trait in a parent and that in an offspring is known as the heritability of a trait. In order for environmental pressure to cause quick evolution in a trait, that trait must have a high heritability.

1.3 Reframing the structure prediction problem

The evolutionary approach to structure prediction is modeled after the natural process. Each crystal structure is considered an organism. The representation described in 1.4.1 will be its “genotype.” In nature, the fitness of an organism is based on how well its phenotype is suited to its environment and, in particular, how successful it is in reproducing. We assign fitnesses to the organisms based on their energies and allow them to reproduce probabilistically based on those fitnesses. Pressures analogous to those which force species to adapt to their environments will thus lead to lower energy crystal structures.

We organize organisms into generations. The algorithm proceeds by creating successive generations. The methods by which an offspring generation is made from parents are called variation operations or variations. They include operations which are analogous to genetic mutation and crossover. Each time the algorithm wants to create a new organism using a variation, it must select parents using a selection method. Each offspring must meet some minimum standards to be considered viable, analogous to the “growing up” process in nature. We call this the development stage.

We also try to improve on the biological analogy when possible. In particular, we would rather not let the most optimal solution worsen from one algorithmic iteration to the next. So, we implement a promotion operation which promotes some number of the best organisms from one generation directly to the next. Also, mutations in nature are usually detrimental. We try to use mutation variations which are likely to introduce valuable new information to the gene pool.
1.4 Details of our approach

1.4.1 Structure representation

We consider primarily 3D-periodic or crystalline structures – atomic clusters, surfaces, etc. are generally dealt with by packaging them into a large box with an appropriate amount of vacuum spacing. So, a description of one cell determines the whole system. In particular, we will work with the six lattice parameters and the $3N$ atomic coordinates of a cell where $N$ is the number of atoms in the cell. The lattice parameters include three angles, usually represented in degrees, and three lengths, generally given in Angstroms. For most purposes, we will write the fractional coordinates of atoms in a cell.

In total, this gives $3N + 6$ variables needed to describe solutions. Additionally, in ab-initio applications, $N$ itself must usually be determined. However, these are not truly independent degrees of freedom: there are infinitely many ways to represent the same structure. In addition to differently-sized supercells, any affine transformation of the atomic positions or alternate choice of lattice vectors may produce an alternate representation of a single crystal structure.

In biological language, there are many genotypes which produce each phenotype. Since our solution space is somewhat more complicated than it has to be, our problem of searching that space is more complicated than it, in theory, has to be. In practice, we will try to focus our attention on one representative portion of the space. This way, variations which act on a particular representation will have high heritabilities.

1.4.2 Energy, value and fitness

There are subtle but important relationships between the values we will call the energy, value and fitness of an organism. The energy is the total energy as computed by some total energy code. The value of an organism is the number returned by the objective function. For example, if we are using EnergyPerAtom then the value of an organism will be its energy divided by the number of atoms in its cell.

An organism’s fitness is its value normalized in the context of its generation. In particular, an organism with value $value$ has fitness given by

$$f = \frac{value - worstValue}{bestValue - worstValue}$$

where $worstValue$ and $bestValue$ are the highest and lowest energies in the current generation. So, the lowest energy organism has a fitness of 1 and the highest energy organism has a fitness of 0.

1.4.3 The algorithm

Abstractly and without parallelization, the genetic algorithm proceeds as follows:

1. Create initial parents generation.
2. Create empty offspring generation.
3. Apply promotion and variation operations to create offspring.
   (a) If we have enough offspring, go to 4
(b) Apply a random variation to one or two randomly chosen organisms from the parent generation to create an offspring.

(c) Develop the offspring.

4. Relax each structure and evaluate its fitness.
5. Re-develop each offspring and, if successful, add it to the generation.
6. If converged, end.
7. Set offspring \(\rightarrow\) parents.
8. Go to 2.

See Appendix A for a complete code listing.

1.4.4 Selection

In nature, organisms of higher fitness are by definition more likely to successfully reproduce. In the genetic algorithm, we select those organisms to reproduce which we would like to be successful, that is, those with lower total energies. Preferential selection of lower energy organisms to act as parents is the only major evolutionary “force” that is acting. If we allowed all organisms equal probability of reproduction, the population as a whole would have no incentive to improve.

There are several commonly-used selection strategies: elitist, roulette, and tournament. Elitist is also known as simple and just means that the top \(numParents\) organisms are allowed to reproduce with equal probability [9]. In roulette selection, a random number \(d\) between 0 and 1 is chosen for each organism, and if \(d\) is less than the fitness of the organism, it is allowed to reproduce [4]. In this way, it is possible for any organism to reproduce (except the worst one which has fitness 0), but it is more likely for organisms of higher fitness. Finally, in tournament selection, all of the organisms in the parent generation are randomly grouped into pairs, and the better member of each pair is allowed to reproduce. Tournament and roulette methods have very similar results.

In order to test selection strategies, we use a method which is more or less a generalization of the three above: organisms are selected on the basis of a normalized probability distribution over their fitnesses. Two parameters are specified to describe the distribution: \(numParents\) and \(selectionPower\). \(numParents\) is the number of organisms we will allow the possibility of being parents. \(selectionPower\) determines the power-law which comprises the rest of the distribution. We will refer to different selection strategies with an ordered pair containing these two numbers:

\[
(numParents, selectionPower)
\]

So, for example, in a population of size 20, \((5, 0)\) selection is an elitist strategy, and \((20, 1)\) selection is a roulette strategy.

In more detail, we first set to zero the selection probabilities of organisms worse than the \(numParents+1^{th}\) best. We then recalculate the fitnesses of the remaining organisms with respect to the remaining “sub-generation” in the same way which fitnesses are normally calculated, except that we are only considering a portion of a generation. The selection probability of a remaining organism with renormalized fitness \(f_i\) is set to

\[
p_i = \frac{f_i^n}{\sum_j f_j^n}
\]
where $n = selectionPower$ and the sum is over the top $numParents$ organisms. Clearly, $\sum_i p_i = 1$, and our distribution is normalized. The renormalization was necessary so that the probability distribution is continuous.

Figure 1.1: Example selection probability distributions of a generation with 21 members with evenly-spaced fitnesses. In order of aggressiveness, we have a $(5, 0)$ elitist distribution in red, a $(15, 2)$ selection in green, a $(14, 1)$ selection in blue, and a $(18, 0.5)$ selection in cyan.

Figure 1.1 shows some examples. Elitist selection, as described above, can be achieved by choosing a small $numParents$ and setting $selectionPower$ to zero. Roulette is equivalent to using a $selectionPower$ of one and setting $numParents$ to the total generation size. Many other distributions are possible.

By varying these selection probabilities, we can change the amount of evolutionary pressure that we put on the population. For example, a $(5, 2)$ selection would favor the top members of a generation very strongly, and an algorithm using that selection would probably converge very quickly. However, when parametrizing for such quick convergence, it is more likely that the algorithm will converge prematurely to a non-global minimum. There is clearly a trade-off between speed of convergence and confidence in the solution.

1.4.5 Variations and their motivation

Variation operations are the primary way by which new organisms are made. All variations work with two generations: the parents and the offspring. A variation starts by selecting an appropriate number of organisms from the parent generation using a selection method (see 1.4.4). It then performs some operation on the parent structures to create a single new structure which may be added to the offspring generation. Descriptions of each of our variation methods follow.
Slicing crossover

The most energetically-important interactions in most materials come from species which are close to one another. This suggests that there is some amount of spatial separability in the energy-minimization problem. If regions of two structures are locally “good,” then somehow combining those two regions may result in a structure which is better than either of the original ones. This is the basic idea behind the slicing crossover variation, the primary way in which we use the genetic information in the parent generation to make offspring.

In more detail, the variation first selects two parents. The lattice parameters of the offspring are the average of those of its parents. To decide which atoms from the parent cells are copied to the child, it selects randomly from uniform distributions an axis, $A$, and a fractional coordinate, $s$. It selects from a Gaussian distribution a slice thickness, $t$, between 0 and 1. Then, all atoms in one parent whose fractional coordinate along $A$ is within $t/2$ of $s$ are copied into the offspring structure. Atoms in the other parent whose coordinate along $A$ is greater than $t/2$ from $s$ are also copied into the offspring. By “copied” we mean that each atom in the offspring has the same species-type and fractional coordinates as an atom in one of the parents. The result of this sort of variation applied to two artificial structures is shown in Fig. 1.2.

A couple of generalizations have been made to this general idea. The first, due to Oganov, involves shifting all the atoms in a cell by the same amount before crossover [8]. These shifts may happen with different probabilities along the axis where the cut is made and that which it is not. These probabilities are called the major and minor shift fractions, respectively. This removes any bias caused by the implicit correlation between the coordinate $s$ on the axis $A$ in one crystal with the coordinate $s$ on the axis $A$ in the other. In practice, this helps repeat good local structures to other parts of the cell. On the other hand, as discussed in 1.4.1, we do not necessarily want to encourage the replication of similar crystals with different representations in the population.

A second generalization is periodic slicing [4]. In this case, the value $s$ described above becomes a function of the coordinates of each atom under consideration. In particular, it is a cell-periodic function of the coordinates along the axes other than $A$. We use a sine curve whose amplitude and two frequencies are pulled from uniform distributions. The result of this sort of variation applied to two artificial structures is shown in Fig. 1.2.

Now, one can see that in the example crossovers, we have effectively transferred some of the local structure of each parent to the child, maintaining important things like periodicity and interatomic distances. However, in some sense we were just lucky that both the crystals represented by the parent cells as well as their representations in the computer were very similar.

In less ideal cases, there is no difficulty from the viewpoint of the crossover operation since it is performed primarily in fractional space. However, the offspring structure may be less successful. Unless the lattices of the parents were equal, the distances between atoms will be distorted in the child. In some cases, crystals which are physically very similar will create an offspring which has little in common with either of them if their representations (in particular, the lattice parameters or number of atoms in the cells) are sufficiently different.

Thus the crossover operation is most successful when the parents’ represented similarly in the computer. Biologically speaking, similar genotypes increase the
Figure 1.2: The slicing variation. The top two structures are used as parents. The bottom two structures are the result of the crossover using different parameters. 3x3x3 supercells of the child structures are shown.
heritability of important traits. Some solutions to this are discussed in 1.4.6.

**Mutation**

The mutation variation randomly modifies the genetic information of an organism. In particular, we perturb both the atomic positions (in Cartesian coordinates) and the lattice parameters of a parent crystal to create the child.

Once the mutation variation selects a parent, it considers each atom in the structure in turn. The chance that any particular atom is moved is variable. A perturbation to atomic position is done by adding a Gaussian random variable to each of the atom’s coordinates.

In order to mutate the lattice vectors, we apply to them a randomly generated strain matrix. In particular, if \( \mathbf{a} \) is a lattice vector of the parent crystal, then the corresponding lattice vector of the offspring is given by:

\[
\mathbf{a}' = (I + \epsilon_{ij}) \mathbf{a}.
\]

Here, \( I \) is the identity and the \( \epsilon_{ij} \) are Gaussian random variables constrained such that \( |\epsilon_{ij}| < 1 \). The same \( \epsilon_{ij} \) are used for each cell vector during a mutation event. Oganov claims that atomic perturbations are unnecessary but that lattice perturbations are important to avoid premature convergence 8.

**Permutation**

The permutation variation selects a single parent. It swaps some number of species’ spatial coordinates. The pairs of elements to consider for swaps as well as the Gaussian distribution describing the number of swaps can be specified. In oxides and ionic materials in general, swapping a cation with an anion usually costs a lot of energy. It is unlikely that the algorithm will keep any structures which would be improved by such a swap. However, it might make more sense to allow swapping of cations.

In metal alloys, permutation faults are generally very low energy. It is essential to use a permutation variation when studying these systems in order to effectively search the solution space. When studying these systems, an investigator might want to increase the probability of permutations in the endgame when the correct lattice has probably been found. Notice, though, that we do not intend the permutation variation to do the whole job of finding correct lattice site decorations. That would effectively be a random search of those degrees of freedom, an inefficient strategy.

**Basis size**

The number of atoms, \( N \), per cell of structures is important. If \( N \) is not at least a multiple of the size of the correct primitive cell of the material, the structure can not possibly relax to a global minimum. However, \( N \) is a hard parameter to search over. The local minimizer does not help us since it will (hopefully) not change the number of atoms in a cell. Furthermore, the energy hypersurface is not very “nice” with respect to this parameter. It is likely that values of \( N \) surrounding the

---

1More recently, USPEX includes “smart” mutation operators. See, e.g., the discussion in [11].
optimum will lead to structures quite high in energy while values of $N$ further from the ideal may lead to closer-to-ideal structures.

The crossover variation can introduce offspring with different $N$ than its parents, but this is an inefficient way to search over the parameter. In order to facilitate the search, there is the basis-size variation. This variation changes the number of atoms in a structure. It randomly adds or removes a nonzero number of stoichiometries worth of atoms to the cell of a parent structure. This should be used whenever the number of atoms in the primitive unit cell is unknown. Although $N$ is often an unknown in applications, most previous works have avoided the problem either by “guessing” correctly or working with large supercells [8] [15].

1.4.6 Constraints and reducing the size of the solution space

Ideally the crossover variation takes the small good parts from sub-optimal solutions and combines them to make something better. It can not really make new genetic information, it just rearranges what is already in parents. The mutation variation is capable of introducing new information into the gene pool, but it is inefficient in creating good structures. If we only used the mutation variation, we would be effectively performing a random search. The power of the evolutionary approach lies primarily in the slicing crossover variation and its ability to “intelligently” guide the search of the solution space. Our emphasis on the crossover variation is the reason we need to sample much of the solution space in our initial population. The algorithm works best if most of the necessary “raw genetic material” is in the population from the beginning. Unfortunately, it is not easy to sufficiently sample such a large configuration space.

Furthermore, the crossover algorithm works on the genotype of a crystal, that is, its particular representation in the computer. The crossover of a good organism with itself is likely to give the same structure back. However, the crossover of a crystal with a supercell or a rotation of the same crystal is likely to result in garbage. The problem is that discussed in 1.4.1, that there is a redundancy in our representation of structures. It is also likely that a crossover on two non-identical organisms which are represented “very differently” will not be very successful.

For both of these reasons, we implement some hard constraints on the types structures the algorithm considers. In particular, these are maximum and minimum limits on lattice parameters, number of atoms per cell, and interatomic distances. First of all, this constrains the space we must sample in the initial population and gives a scale on which to base the density of the sampling. Secondly, the crossover operation is likely to have more physical meaning between similar structures.

It is worth noting that there is a distinction between the uses of the hard constraints. We would certainly like to use them to remove as much redundancy as possible from the space of solutions. This makes the problem easier without limiting our set of possible answers or introducing any a priori assumptions as to the form of the solution. On the other hand, it is more dangerous to remove merely unlikely regions of the space from consideration. This brings into question both the validity of results and the claim to first-principles structure prediction. However, it is sometimes necessary to make some conservative assumptions. For example, a minimum interatomic distance constraint is usually necessary to ensure stability of energy codes.

Another strategy we use to reduce redundancy and attempt similar represen-
tation of cells is Niggli reduction [10]. There is a Niggli cell for any lattice which is both unique and has the shortest possible lattice lengths. We transform all structures into this representation during the development stage.

1.4.7 Local relaxation

During the energy calculation, structures are relaxed to the local minimum of the energy functional. The method is dependent on the energy code used, but the local minimization problem is a relatively well-understood problem and its solutions are generally stable.

This effectively divides the configuration space into regions. Each region is the basis of attraction for one local minimum. In order to find the global minimum, we must only find a solution in its basin of attraction, and the local minimizer will do the rest. This tremendously reduces the effective size of the space we are searching. A relatively sparse sampling of a region can find most of the local minima in it.

Two points by Oganov have some bearing here [8]. Firstly, due to the nature of the energy functional, the global minimum is generally surrounded by many low-lying local minima. So, sampling the area around the best local minima is likely to find the global minimum. Secondly, although it would be much quicker, running the genetic algorithm without local minimization would likely be ineffective since the correlation between the energy of an unrelaxed structure to that of its local minimum is very weak.

1.4.8 Development

The development stage of the algorithm comes between an offspring’s creation and its being added to the generation. Its biological analogy is an organism’s “growing up,” and it is possible for the organism to fail this process.

For the structure prediction problem, this stage is responsible for performing the constraint checks described in 1.4.6 and the redundancy checks described in 1.4.9. It might seem natural that crystal relaxation be performed here. However, relaxation is something we get “for free” when evaluating the objective function. Additionally, relaxation itself can change whether or not an organism satisfies the hard constraints. See Chapter 3 for details.

If the relevant option is specified, we maintain an estimate of the optimal density of structures independent of the population. In particular, this density is volume per atom, a number which is strictly only useful when we fix stoichiometry. The technique was suggested by Oganov [8]. His group optimized total volume, but they also fixed the number of atoms in the unit cell.

The density is optimized in the sense that we start with an initial guess and then update it each generation. The particular update scheme requires two parameters, a weight $w$ and a number $n$. Each generation, we find the average, $D_g$, of the densities of the top $n$ organisms in the generation. The new best density estimate is then given by

$$D_1 = w \cdot D_g + (1 - w) \cdot D_0$$

where $D_0$ is the previous best estimate.

---

2 This is no longer the case in newer versions of the USPEX code. It now supports variable-composition predictions.
Then, any time a new organism is made, it is scaled to this volume before relaxation. This serves two purposes. First it helps to standardize the representation of structures. Structures which are long and skinny tend to be scaled to a higher volume until, perhaps, they fail the maximum lattice length constraint. Other structures are scaled down in volume and might fail, e.g. minimum atomic distance constraints. Either way, if a structure fails constraints when scaled to the common density of the system, it is unlikely that the crossover operations involving the organism will result in successful offspring (see 1.4.6.

The second reason for the volume scaling is a practical one. Many minimization algorithms take a long time if the initial solution is far from a minimum. This scaling is an easy first pass at moving a solution towards a minimum.

1.4.9 Avoiding redundant calculations

The most computationally-expensive part of the algorithm, by far, is the energy computation. We would like to minimize the number of these computations. In particular, it is bad to run multiple energy calculations on a single structure.

Unfortunately, this can happen quite often. If a pair of structures mate more than once, they are likely to create similar offspring. If the set of best structures does not change from generation to generation due to promotion, the set of parents, and thus the resulting set of children, can be very similar also. In addition, as the generation as a whole converges to the global minimum, all of the organisms are likely to become more similar. What is worse: once a couple of low energy, often-selected organisms are in the population, they can reproduce and similar structures will effectively fill up the next generations. This leads to premature convergence which is in practice indistinguishable from convergence to the correct global minimum.

It is necessary to maintain genetic diversity to avoid this premature convergence. Some authors establish a “δValue” rule [7]. That is, they choose some interval in the energy and do not allow more than one organism in any generation with energies within that interval. However, the size of the interval is fairly arbitrary and system-dependent and, in fact, we would like many of structures close to the minimum as long as they are distinct.

Our solution to this issue is to keep a list of all structures the algorithm has seen and explicitly check against them when creating new structures. In fact, the algorithm works with two of these lists at any given time. The “perGen” list holds structures which are members of the current generation. The “wholePop” list holds all structures the algorithm has seen, both relaxed and unrelaxed.

The goal of the wholePop list is to prevent doing redundant work. If we have already computed the energy of a structure, there is no reason to do it again. If an unrelaxed structure has already been seen, it is just thrown away. The assumption is that if it was good enough to keep, it was promoted, and if not, there is no reason to spend more time on it. On the other hand, we never use the wholePop list to throw away relaxed structures; if we have already done their energy calculation, they may as well stick around.

The perGen list serves to prevent premature convergence due to multiple occurrences of a single structure in a generation. It works similarly to the wholePop list and, indeed, does not need to keep track of unrelaxed structures since the wholePop list does that. Only organisms which are added to the generation are added to the
perGen list. The difference between the two is that the perGen list throws away a matching structure even if it has been relaxed. Throwing away energy calculations is unfortunate but necessary to avoid both premature convergence and future redundant calculations. Seeing identical structures created in a single generation is much less common if we take these steps to maintain genetic diversity in the first place.

1.4.10 Endgame

Once the algorithm appears to have converged, the optimal strategy changes somewhat. It is advantageous to increase the probability of mutation and permutation variations while decreasing the magnitude of the perturbations these variations introduce. This will cause the algorithm to search the configuration space in the immediate vicinity of the current best solutions.

The algorithm’s progress is divided logically into two stages which it can switch between: normal and endgame. The algorithm starts in the normal stage and enters the endgame once it has gone a given number of generations without improving its best solution. Separate probabilities of use are given for each variation operation.
Chapter 2

Usage

The genetic algorithm may either be run from external Java code or manually from the command line. This section will focus primarily on the command line choice. However, the two methods are similar. If running it from code, the programmer should call the `crystalGA` method in `CrystalGA.java`. One of the arguments to this method is an input filename. This input file is the same as the one which would be passed using the “–f” flag on the command line.

2.1 Input

The genetic algorithm is packaged as a Java jar file called “ga.jar”. It can be run from the UNIX command line by typing something like

```
java -jar ga.jar
```

When run with no arguments, it prints out its usage statement: a list of tags which tell it how to run. These tags can be passed either on the command line or in an input file. The name of the input file itself must, of course, be passed on the command line. The format of commands is the same on the command line as in the input file except that they are prefaced on the command line with “--” whereas in the input file each tag is placed on a new line. Additionally, comments can be placed in the input file on lines whose first character is “#”.

In addition to the primary input file, other files may be necessary for the genetic algorithm to run, depending on the input options. For example, the energy methods require files which specify potentials, and the initial population may be read from disk.

2.1.1 The main input file

Here we list and discuss all of the parameters and options which might be specified in the main input file which is specified on the command line with the “–f” flag. Most are necessary for a proper run of the algorithm. However those which need special attention (those which are likely to change from system to system) are denoted with a star (*). Note that all options are case-insensitive with the exception of atomic symbols, and units are in Angstroms, degrees and eV unless otherwise noted. Arguments in brackets are meant to be filled in with the appropriate word
or number. “<d>” is used to indicate the argument should be a double, “<i>” for an integer, “<s>” for a string, and “<b>” for a Boolean.

See Appendix C for a sample input file.

- **help**
  Prints out a brief usage statement and exits.

- **verbosity <i, n>**
  The number n determines the amount of output to screen of the algorithm. It is an integer between 0 and 5. See 2.2.2.

- **runTitle <s>**
  Sets the title of the run.

- **outDir <s>**
  Specifies the directory where the output files will be stored. This directory should not exist already unless resuming from a previous run.

- **dryRun <b>**
  Setting this to true prevents the algorithm from writing anything to disk. Useful sometimes, but with most energy codes, the algorithm will not proceed normally.

- **keepTempFiles <b>**
  If this is set to false, the algorithm will delete the temp directory (see 2.2.1) at the end of the run.

- **saveStateEachIter <b>** TODO

- **popSize <i>**
  Specifies the number of offspring which will be added to each new generation for a serial run. See 3.9 for the subtleties of parallel runs. This option does not affect the first generation.

- **promotion <i>** TODO

- **parallelize <i> <i>**
  Specifies that energy calculations should be run in parallel. The first argument indicates the number of calculations that can be run concurrently. The second indicates the minimum number of organisms acceptable to constitute a generation. See 3.9 for the details of how these numbers affect offspring generation size.

- **compositionSpace** TODO

- **optimizeDensity <d, w> <i, n>**
  Used to specify that the cell density should be optimized independently as discussed in 1.4.8. The current optimal density estimate is updated by taking a weighted average (weight w) of it with the average of the densities of the top n organisms.
• **useRedundancyGuard** `<both|none|wholePopulation|perGeneration>` `<atomic misfit tolerance>` `<lattice misfit tolerance>` `<b>`

Specifies the use of the redundancy guard as described in 1.4.9. For the atomic and lattice misfit tolerances, 0.1 and 0.1 work well. The boolean indicates whether or not the comparison algorithm should assume PBCs.

• **useSurrogateModel**

  In development and unsupported.

• **endgameNumGens** `<i>`

  Specifies the number of generations without improvement the algorithm must go before it enters the endgame stage (see 1.4.10).

• **useNiggliReducedCell** `<b>`

  If true, reduce cell representation during development. See 1.4.6.

• **writeHartkeFile** `<b>`

  If true, write the hartke.txt output file which can be used to generate Hartke plots.

• **colorOutput** `<b>`

  If true, color output text according to the structure it refers to.

• **initialPopulation** `<i, N>` `<s>` `<...>`

  Used to specify an initial population creation method. Multiple such methods can be specified. `N` specifies how many organisms the method should create. Note that some methods will not be able to make an arbitrary number of organisms in which case `N` is an upper bound. The string specifies the type of method:

  - **initialPopulation** `<i>` **random randomVol** `<i>`
    Method creates structures with random lattice parameters (uniformly distributed within the hard constraints) and a given number of stoichiometries worth of atoms randomly placed.

  - **initialPopulation** `<i>` **random givenVol** `<d, v>`
    Method creates structures with random lattice parameters (uniformly distributed within the hard constraints). Structures are subsequently scaled to the given volume per atom, `v`.

  - **initialPopulation** `<i>` **poscarts** `<s, d>`
    Creates organisms for the initial population from POSCARS files in the given directory, `d`.

  - **initialPopulation** `<i>` **units**
    Used to create structures with coherent molecular units. Still under development.

  - **initialPopulation** `<i>` **manual**
    Used to tell the algorithm that other code will be providing structures for the initial generation. That other code should call `setSeedGeneration()` in `GAParameters` before beginning the algorithm.
This parameter specifies the function we want to minimize. Two objective functions have been implemented for the structure prediction problem. The first is energy per atom or “epa”, which is simply the energy of a structure divided by the number of atoms in its cell. The second is phase diagram or “pd” which specifies that the algorithm should build a convex hull and evaluate new structures based on their formation energies with respect to it.

Additionally, the “cluster” objective function can be used to search for atomic clusters. Call it by passing the words “cluster” and a number indicating a supercell size as the first arguments to the objectiveFunction command, and then follow those with another objective function and its arguments. Essentially, this causes the code to wrap the structure in a supercell and then evaluate it using the second objective function:

- \texttt{objectiveFunction cluster <d,padding length> <other objective function arguments>}

- \texttt{objectiveFunction <epa|pd> gulp <s> <s> <b> <s>\*}
  
  In order to use the Gulp energy code, the above syntax should be used. The first two arguments are the gulp header file and potential file, respectively, as described in 2.3.1. The truth of the Boolean argument specifies that the algorithm should throw away structures on which Gulp does not properly converge. This is highly recommended. The remaining string arguments are optional and are the symbols of those species which need to be given a shell in Gulp input files.

- \texttt{objectiveFunction <epa|pd> vasp <b> <s,KPOINTS> <s,INCAR> (<s,symbol> <s,POTCAR>)\*}
  
  In order to use the Vasp energy code, pass a boolean which indicates whether the GA should only accept the results of VASP cals which report “reached required accuracy” as well as KPOINTS, INCAR, and POTCAR files.

- \texttt{objectiveFunction <epa|pd> lammps <s> <b>}
  
  The first argument is a string giving the filename of a file which holds LAMMPS pair_coeff and pair_style lines that define the empirical potential to be used. The second argument is a boolean which, if true, indicates that the cell should be relaxed in the energy calculations.

- The interfaces to OHMMS, CASTEP, avogadro, dlpoly, MOPAC, and DFTPP are still experimental.

- \texttt{*variation <d> <d> <s> <...>}

This tag specifies variation methods which are used to create offspring from parents. At least one must be specified to run the genetic algorithm. The first two arguments specify the probability that this particular method will be used. The first probability is used when the algorithm is not in the endgame stage and the second is used when it is.

Note that either of these probabilities can be 0. This would be useful, for example, if the user wants to change the parameters of the mutation variation between the normal and endgame stages. Two separate mutation variations...
would then be specified, one which has 0 probability of being used during the normal stage and one which has 0 probability of use during the endgame. In order to not use the endgame functionality, the first two arguments should be equal.

- **variation** <d> <d> slicer <d> <d> <d> <d> <i> <b> <d>
  The first two arguments after “slicer” are the mean and sigma of the thickness of a slice in fractional coordinates, respectively. The second pair of arguments are the major and minor shifting probabilities. The last pair are the maximum amplitude (fractional coordinates) and frequency for periodic cuts. The frequency is an integer so that the the slice is cell-periodic. See 1.4.5. As examples, the offspring in [1.2] were created using the parameter sets “0.5 0 0 0 0 0” and “0.5 0 0 0 1 2.”
  The penultimate boolean argument indicates whether smaller parents should be grown to the approximate size of larger parents before crossover. The last argument gives a frequency with which one of the parents is doubled in size before crossover. Both of these options tend to make the average number of atoms in solutions in the population grow over time.

- **variation** <double> <double> structureMut <double> <double> <double>
  The arguments after “structureMut” specify, in order, the fraction of atoms whose position is perturbed, the average perturbation of an atomic coordinate (in Angstroms), and the average magnitude of the $\epsilon_{ij}$ in the lattice strain matrix (see 1.4.5).

- **variation** <d> <d> permutation <d> <d> <s> •
  The arguments after “permutation” specify, in order, the mean and sigma of the number of swaps a permutation will perform and the pairs of species which may be swapped. The pairs are given in the form A-B (e.g. Mn-O). See 1.4.5.

- **variation** <d> <d> numStoichsMut <d> <d>
  The arguments after “numStoichsMut” specify the mean and sigma of the number of stoichiometries which the variation should add or remove, respectively. The mean should probably be zero to prevent the variation from significantly shifting the average size of organisms over time.

- **selection probDist** <i> <d>
  The probDist selection is the only selection method implemented. The two parameters are `numParents` and `selectionPower`, respectively, as discussed in 1.4.4.

- **convergenceCriterion** <s> <...>
  Convergence criteria are used to decide that the algorithm should halt. They are checked at the end of each generation. The first argument specifies the type of method.

  - **convergenceCriterion** maxFunctionEvals <i>
Specifies that the algorithm should halt when it has exceeded a given number of energy calculations.

- `convergenceCriterion maxNumGens <i>`
  Specifies that the algorithm should halt when it has run for a given number of generations.

- `convergenceCriterion maxNumGensWOImpr <i, n> <d, t>`
  Specifies that the algorithm should halt when it has run for a given number of generations without improving its best solution. The improvement must be at least t to count.

- `convergenceCriterion valueAchieved <d>`
  Specifies that the algorithm should halt when it has found a solution with a value of at most the given number.

- `convergenceCriterion foundStructure <s>`
  Specifies that the algorithm should halt when it has found a given structure. The string parameter is the pathname of a Cif file.

- `minInteratomicDistance <d>`
  Specifies the minimum interatomic distance a structure can have.

- `maxLatticeLength <d>`
  Specifies the maximum lattice length a structure can have.

- `minLatticeLength <d>`
  Specifies the minimum lattice length a structure can have.

- `maxLatticeAngle <d>`
  Specifies the maximum lattice angle a structure can have.

- `minLatticeAngle <d>`
  Specifies the minimum lattice angle a structure can have.

- `maxNumAtoms <i>`
  Specifies the maximum number of atoms a structure can have.

- `minNumAtoms <i>`
  Specifies the minimum number of atoms a structure can have.

- `minNumSpecies <i>`
  Specifies the minimum number of species a structure can have. For example, setting this to 2 during a phase diagram search will avoid sampling elemental phases.

- `doNonnegativityConstraint <b>`
  Specifies whether or not to discard solutions with positive objective function values. (Generally true when using the “epa” objective functions and false when using “pda”).

- `dValue <d>`
  Specifies the size of the interval described in 14.9.
2.2 Output

2.2.1 Output to disk

The primary output of the algorithm is placed into a directory structure. The root of this structure can be specified in the input file with the `outDir` tag. The default output directory is called “garun_runTitle” where `runTitle` is the title of the run, specified by the `runTitle` tag and defaulting to “default”.

Inside this directory are a number of POSCAR files, each of which holds a single structure. They are named after their organism ID. These are written at the end of each generation, so each of the structures is relaxed, inside the hard constraints, and a full member of at least one generation.

Also in this directory are the files “parameters” and “index”. The parameters file holds the output of `GAParameters.toString()` and is written at the beginning of the algorithm. The index file is appended at the end of each generation with a block of lines of the form:

```
    generation g N
    id1 value1 filename1.cif
    id2 value2 filename2.cif
    ...
    idN valueN filenameN.cif
```

where each line after the first represents one organism in the generation. `N` is the number of organisms in that generation and `g` is the number of the current generation. The initial population is generation 0. The organisms within a particular generation are not guaranteed to be in any particular order. The filenames are given as paths relative to the directory from which the algorithm was called.

Finally, if the `keepTempFiles` option was set to `true`, there is a subdirectory of the output directory called “temp”. It contains temporary files used by the algorithm, most notably those used in communicating with external energy codes.

If the appropriate options are set, the algorithm may also output .save.tgz files which can be used to resume the algorithm, .pdb.tgz files which hold phase diagram data from a “pd” search, and a `hartke.txt` file.

The `hartke.txt` file holds a listing of all the structures which made it to the point of being sent to the objective function – one line for each structure. The first column is the structure ID, the second is the number of energy calculations. The third and fourth columns are the total energy and objective function value, respectively, and the fifth column indicates whether or not the organism was added to the generation in the end. Note that it is possible for the listing to include multiple structures with the same number of energy calculations indicated. The reason for this is that it is possible for a structure to be sent to the objective function but then for the objective function to decide it can’t evaluate it. For example, the LAMMPS code can’t handle structures whose unit cells are too oblique. In this case, the energy code does not run, and the energy calculations counter is not incremented, but both structures show up in the `hartke.txt` file.

2.2.2 Output to screen

Much important information about the progress of the algorithm is written to screen. This text is not saved elsewhere, but may be captured using e.g. the `tee` command.
utility on the UNIX command line. The amount of output is determined by the verbosity flag which may take on values between 0 and 5. The effect of this choice is described in the following table. The output of each level is a superset of the output of lower levels.

<table>
<thead>
<tr>
<th>Verbosity level</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>No output except for major errors.</td>
</tr>
<tr>
<td>1</td>
<td>Information at the beginning and after each generation.</td>
</tr>
<tr>
<td>2</td>
<td>Information about each variation being run.</td>
</tr>
<tr>
<td>3</td>
<td>Information about energy calculations and organisms which fail constraints.</td>
</tr>
<tr>
<td>4</td>
<td>More detailed information about variations, development, and energy calculations is displayed.</td>
</tr>
<tr>
<td>5</td>
<td>Output of energy codes is echoed. Structures are printed to screen after creation. Selection probabilities are displayed.</td>
</tr>
</tbody>
</table>

When testing the algorithm, a verbosity level of 4 is usually used. For important runs, it is probably best to choose as high a level of output as is understandable and to save it to disk for possible future analysis. Note that the order of output is unpredictable when using parallelization as each thread will print to the same screen independently. In this case, color-coded output can be helpful.

### 2.2.3 Visualizing output

It is often useful to visualize the results from the algorithm on several scales: the individual structures, a genetic algorithm’s worth of structures, and many runs worth of structures. The POSCAR files of individual structures are written to disk at the end of each generation as described in 2.2.1. These can be viewed using software such as vics-ii [3].

The progress of an entire run is often displayed in an evolutionary progress plot. Each dot represents an organism which the algorithm considered. Blue dots represent organisms promoted from the previous generation, and red dots represent those newly created. Along the horizontal axis is the generation, so each vertical line of dots represents one generation. Along the vertical axis is the organism’s value. Lower is better, so in evaluating the progress of the algorithm, we are particularly interested in the lowest value organism in each generation.

Visualizing the success of the algorithm over multiple runs is most useful for evaluating the success of certain parameter sets.

### 2.3 Energy code interfaces

Supported energy codes include Gulp for empirical potential models and Vasp for ab initio calculations. They are treated as black boxes by the algorithm as much as possible. See 2.1.1 for the input syntax.

The success of the local minimizer is critical to the success of the algorithm. Briefly, it is just important that it work well. This is dicussed further in 1.4.7.
2.3.1 Gulp

In order to run Gulp, the genetic algorithm runs the command

```bash
callgulp <input file>
```

The `callgulp` command is generally a small script and should be put on the current user's UNIX path. Its purpose is to avoid the need to hardcode details of the Gulp installation into the genetic algorithm. On most machines, the following `callgulp` script should work:

```bash
#!/bin/bash
gulp <$1
```

Gulp's full output is not saved but is possibly echoed to screen depending on the verbosity level (see Section 2.2.2).

Two files are necessary to perform Gulp calculations: the header file and the potential file. The Gulp input files for structures are generated in the format:

```text
<contents of gulp header file>
cell
  <a> <b> <c> <alpha> <beta> <gamma>
frac
  <atomic symbol 1> core <three fractional coordinates>
  <atomic symbol 2> core <three fractional coordinates>
  ...
<contents of gulp potential file>
```
where the tokens in brackets are not to be interpreted literally. Lines specifying the location of species’ shells are included if needed.

One should refer to the Gulp manual [1] for all the options available, but a Gulp header file might look like the following:

```
opti comp conj
switch_minimiser bfgs gnorm 0.2
```

This tells Gulp to optimize both the lattice parameters and atomic positions of the structure under constant pressure. It will initially use a conjugate-gradient minimization routine but will switch to the Newton-Raphson BFGS method after the gradient norm falls below 0.2. This strategy of switching minimizers is useful for many systems.

There are also many ways to give Gulp an empirical potential. As an example, though, consider

```
lennard
C core C core 100470 64.464 0.000 8.5125
```

This is the Lennard-Jones potential used by Abraham and Probert to model carbon [4].

### 2.3.2 Vasp

Similarly to the `callgulp` script, a `callvasp` script is used to move the details of a Vasp installation out of the Java code and into a shell script. The particular command run by the genetic algorithm is

```
callvasp <input directory>
```

The situation is slightly more complicated if the investigator wants to use e.g. the GridEngine job-queuing system. The script can not simply call `qsub` because the genetic algorithm assumes that the energy calculation is complete when `callvasp` returns. A solution is described in Appendix B. The user should check that the automatically generated Vasp input files are satisfactorily parametrized [2].

### 2.3.3 LAMMPS

Similarly to with the other codes, a `calllammps` script is used by the GA to interact with the LAMMPS code. The GA runs

```
calllammps <input directory>
```

where `<input directory>` is a directory holding lammps input files including one called `in.min`. The script should run LAMMPS and exit only once the calculation is finished. So, it probably looks something like
#!/bin/bash
cd $1
lmp < in.min

The LAMMPS input file used by the GA looks something like

```
units metal
dimension 3
atom_style atomic
boundary p p p
read_data data.in

<CONTENTS OF LAMMPS POTENTIAL FILE>

minimize 0.0 1.0e-8 1 1
fix 1 all box/relax tri 1e4 vmax 0.001
minimize 0.0 1.0e-8 10000 100000
dump myDump all atom 1000000000000000 dump.atom
dump_modify myDump sort 1 scale no
fix 1 all box/relax tri 0 vmax 0.001
minimize 0.0 1.0e-8 10000 100000
```

The contents of the potential file passed to the GA are inserted where indicated. Also, the two fix commands are omitted if the input to ObjectiveFunction indicates that the cell is not to be relaxed. We found that using an initial relaxation step with the system under small pressure is helpful for ensuring relaxation with many empirical potentials.

### 2.4 Resuming calculations

If the saveStateEachIter command is set to true, the algorithm will write a file called genX.save.tgz at the end of each generation to disk in the main output folder. These files essentially hold all of the state of the algorithm at the beginning of the generation X and can be used to resume the algorithm at that point when necessary. To do so, run the GA in the same directory it was originally run in and pass in the resume file with the --r option. For example, your command might look something like

```
java -jar ~/bin/ga.jar --r garun_runname/gen10.save.tgz
```

Again, none of the input files or structures are re-read from disk when resuming in this way. The .save.tgz file holds all of the state of the algorithm. So, there is currently no way to restart the algorithm while changing various parameters. The exception to this is the convergence criteria. You can restart the algorithm and clear all the convergence criteria by using --rc instead of --r above. This is useful, for example, if the run completes as expected but then you decide you want to run longer.
2.5 Strategies

A number of general strategies for successful genetic algorithm runs have been noticed. We will list and discuss some of them here.

There is a trade-off between the speed of convergence of the algorithm and the chance that it has found a global minimum since it is a search-based algorithm and needs time to sample the space and then investigate the most promising regions. The sampling of the space is very important, so it is often best to start with an initial population that is somewhat larger than subsequent generations.

The primary way we make a choice in the speed-certainty trade-off is by choosing a selection method. As mentioned previously, for a population of 20 organisms, a $(5, 2)$ selection strongly favors the best organisms in a generation and will lead to quick convergence. However, we will occasionally see test systems where such a parametrization improperly converges. A $(20, 1)$ selection is generally preferable for this reason.

A partial exception is for close-packed and metallic type structures. The algorithm generally finds the correct lattice for these quite quickly since the energy difference between lattices is much larger than differences in energy due to different site decorations. In this case the stronger selection may be appropriate as the genetic information of better organisms is likely to be strictly better than that in higher energy structures.

An important parameter to keep track of when varying the selection is numPromoted, the number of the best organisms which are automatically promoted from one generation to the next. In particular, if numPromoted is greater than numParents and there are no good organisms created in a generation, then the set of parent organisms can be the same in the next generation.

Then, especially if numParents is small, it is likely that many of the offspring will be similar to those which were unsuccessfully created from the same parents in the previous generation. It is likely that the same set of parents will be promoted to the next generation and so on. Progress will stagnate. To avoid this situation it is almost always best to use both redundancy guards and choose a numParents greater than numPromoted.

It is best to scale randomly created structures in the initial generation to a volume as close as possible to the optimal volume. This is, of course, an unknown, but can often be estimated. This makes the hard constraints more meaningful and significantly improves the speed and chances of the local minimization. For systems with many atoms, it may take a lot of tries to randomly generate structures, but it takes much longer to perform an energy calculation on a poorly formed structure. Density optimization and the strictest hard constraints that do not limit the set of possible solutions should generally be used for similar reasons.

A sample input file which should work as a template for most systems may be found in Appendix C.
Chapter 3

Implementation

3.1 General strategy

The algorithm is implemented in Java and packaged as a Jar file.

3.2 Randomness

The genetic algorithm is a stochastic search method and often needs random numbers. Here, we have used the PRNG that comes with Java. In particular, an instance of java.util.Random is created by GAParameters and is accessible to other objects through its getRandom() method. It is important that all methods use this single PRNG. If methods each created their own Random object using the standard method of seeding with the current time, repeated calls to fast algorithms which use randomness might have the same result.

Random numbers are often drawn from a distribution which is determined by user input. If the input specification asks for an upper and lower bound, then the corresponding distribution is uniform. If the input specification asks for a sigma and mean, then the corresponding distribution is Gaussian.

3.3 The main loop

The main loop lives in GeneticAlgorithm.doGeneticAlgorithm. The function takes a GAParameters object which should already be “filled out”. Eventually, it returns a Structure object containing the result.

So, until it has converged, the main loop is mostly concerned with turning parents into offspring using Variations. A full listing of the code is in Appendix A. Most of the steps will be explained more fully later, but a couple of things are worth noting now.

First, each organism with is added to the generation is developed, evaluated, and developed again. The primary task of development is to enforce the hard constraints. It is necessary to develop an organism before the evaluation to avoid expensive energy calculations on particularly poor candidate structures. However, it is also necessary to do certain things (e.g. redundancy checks, cell reduction) after the evaluation. A structure may relax out of the hard constraints during the
energy calculation. So, we end up developing a structure twice before it is added to the offspring generation.

Secondly, we do not specify that a certain number of organisms be produced from specific Variations each generation. Instead, we specify the probability that a particular Variation will be used. In this way, the algorithm avoids becoming stuck in the case that one of the Variations can not produce successful offspring. (This could happen, for example, if the population had mostly converged so that all organisms created by Slicer converge to a structure already in the generation and fail the redundancy check. We would want the mutation variations to pick up the slack.)

However, the offspring of some Variations might be perfectly capable of producing viable offspring but less likely to do so. For example, the offspring of a Permutation will never fail the interatomic distance, but that of a StructureMut will. In cases like this, we would like to stay as close as possible to the specified proportions. So, once a Variation is chosen, we use it to make new organisms until one is found which satisfies (at least) the pre-evaluation development.

3.4 ObjectiveFunction and Energy

ObjectiveFunction is an abstract class specifying, most notably, the evaluate method. This method returns a Thread which has been started. When the thread exits, the energy calculation is assumed to be complete. A new instance of the ObjectiveFunction is created for each energy calculation through a call to GAParameters. The abstract class also contains a counter to keep track of the number of function evaluations it does. Classes extending ObjectiveFunction should increment this counter when appropriate.

ObjectiveFunction is extended only by the EnergyPerAtom class, whose evaluate method returns the energy of a structure divided by the number of atoms in its cell. If the given StructureOrg already knows its value, this number is returned. Otherwise, the method must initiate a total energy calculation using an Energy.

The Energy interface specifies the getEnergy method and is implemented by GulpEnergy and VaspEnergy. These run the scripts callgulp and callvasp as described in 2.3.

3.5 Genetic operators

3.5.1 Selection

Selection is an interface specifying the single method doSelection. It takes an integer n and a Generation g and returns an array containing n members of g. It is implemented only by ProbDistSelection whose behavior is described in 1.4.4 and the lone instance of which is a private member variable of GAParameters.

The doSelection method of ProbDistSelection calls findProbabilities to get a mapping from structures to their normalized selection probabilities. It randomly selects and returns n structures based on the probabilities (assuming that n is not greater than the size of the generation). The calculation of the probabilities themselves is a straightforward implementation of the formulas described in the methodology, but special note should be taken of the fitnesses renormalization.
3.5.2 Variation

Variation is an interface implemented by Slicer, StructureMut, Permutation, and numStoichsMut. It specifies the doVariation method which takes two Generations and a Selection and creates an Organism which may be added to the offspring Generation. The variation objects themselves are straightforward implementations of the methods described in Chapter 1.

3.5.3 Promotion

There is one instance of Promotion. It is a private variable of the GAParameters singleton created when the “promotion” tag is read from the input file. Its doPromotion method is called at the beginning of each generation (line 22 in the main loop). At this point, it simply copies the top numPromoted organisms from the parent to the offspring generation.

All other times when parents are used to create children are implemented as Variation objects. This is not a Variation since Variations only create a single organism at a time. A promotion variation would have to keep some state information about which organisms it had previously promoted, and this state would need to be wiped clean each generation. This is possible, but it would fall somewhat outside the intended purpose of Variation. Variation objects have direct biological analogies, whereas with Promotion, we have tried to improve upon the natural process.

3.5.4 Development

Development is an interface specifying the single Boolean method doDevelop which has only been implemented by the StructureDev class. If specified, this class is used twice each time the algorithm creates a new organism.

The doDevelop method of the StructureDev class takes a single Organism and the Generation of which it may be a member. It takes the Niggli reduced cell. It then checks, in order, that the organism satisfies the various hard constraints, stoichiometry constraints, redundancy guard checks, and the $\delta$Value rule. The method returns true to indicate that the organism developed successfully and false otherwise.

The Organism will be modified by Niggli cell reduction. The Generation will be modified in two cases. If an Organism has already been relaxed but fails either the $\delta$Value rule or the perGen redundancy check, we compare its value to that of the Organism with which it conflicts. If the new Organism is better, the Development manually removes the old one from the Generation and adds the new one. It then returns false to tell the calling method that we did not, effectively, find a new structure which needs to be added to the Generation.

3.6 Organisms

3.6.1 Organism representation and StructureOrg

Organism is an abstract class which holds operations on a structure’s fitness, value, and ID. The ID is a unique number used to identify the Organism in the algorithm’s output.
StructureOrg is an Organism and has a Structure. Most of the logic in StructureOrg.java is for the input and output of Cif files.

3.6.2 Generation and Structures

Generation is an abstract class which contains a Vector of Organism. It implements the normalization of values into fitnesses as well as some basic statistical operations on its Organisms.

Structures is the extension of Generation used for the structure prediction problem. The Structures class holds the perGen RedundancyGuard if it is in use.

New Generations are created by the main loop (line 18) through calls to GAParameters since it knows the type of Generation to use.

3.6.3 Initial population

Organisms are made by objects which implement StructureOrgCreator which specifies the single method StructureOrg makeOrganism(Generation). A StructureOrgCreator is made by GAParameters when its input-file parser encounters the InitialPopulation tag. The new StructureOrgCreator and the number of organisms it is to be used to make are stored in a mapping, Map<StructureOrgCreator, Integer> initialOrgCreators in GAParameters. This map is used when the main loop calls getNewOrg during the first generation.

Notice that Organisms are developed, evaluated, and then developed again before being added to the Generation and that this is done inside the Structures constructor. Also notice that, although StructureOrgCreators only return one Organism at a time, they might need to know something about their previous actions. For example, FromCifsSOCreator should only return any particular Structure once. The general strategy in these cases is to create all possible StructureOrgs on the first call, store them in a list, and remove them as they are used.

The StructureOrgCreator used to create a StructureOrg is stored in a member variable of that organism. The only purpose for this has to do with initial generation creation using the parallelized algorithm. We want to use each StructureOrgCreator to create a given number of Organisms for the first Generation. However, we only want to count those Organisms which successfully relax and develop. We need to keep track of the method used to create an Organism so that we know which method to give the credit for its success.

3.6.4 Fitness and value

The difference between fitness and value is discussed in 1.4.2. Both numbers are stored as member variables of the Organism abstract class. They are initialized to null indicating that they have not yet been calculated. The value of an organism is calculated by the ObjectiveFunction on line 37 of the main loop (see 3.3). The calculation of fitness is done by the Generation.

An Organism knows whether or not its value and fitness have been calculated. The Boolean methods knowsFitness and knowsValue is used by objects (e.g. StructureDev) which need to act on this information.
3.7 Convergence

The decision whether or not to halt the algorithm is made at the end of each generation. It is mediated by classes implementing the ConvergenceCriterion interface which specifies the Boolean converged(Generation) method. By convention, all of these classes end in “CC”. They are, FoundStructureCC.java, NumFunctionEvalsCC.java, NumGensCC.java, NumGensWOImprCC.java and ValueAchievedCC.java.

ConvergenceCriteria are created by GAParameters during argument parsing. They are added to a list. At the end of each generation, the evaluate method of each object in this list is called by converged in GeneticAlgorithm.java, and if any return true, the algorithm is halted.

3.8 Input and parameters

GAParameters is a singleton class which is responsible for parsing most of the command line and input file options. It passes the ones which are particular to a certain algorithm on to that algorithm and stores the rest. Most of the dirty work is contained in GAParameters.setArgs. The command-line and input-file options are parsed into separate Map<String,String[]> and then combined such that the command-line arguments take precedence.

Next is a long chain of else if s in which there is one entry to handle each flag. For example, a command line option might be:

--objectiveFunction epa gulp gulpheader gulppotl true

An entry in the chain recognizes the first token, ‘-objectiveFunction’. It reads the next token, ‘epa,’ to decide that it should make an EnergyPerAtom objective function. The rest of the arguments, ‘gulp gulpheader gulppotl true,’ are passed to the EnergyPerAtom constructor. The EnergyPerAtom constructor reads the first token, ‘gulp,’ to see that it should use a GULP energy function, which happens to be called GulpEnergy. The rest of the arguments, ‘gulpheader gulppotl true,’ are passed on to GulpEnergy where they are parsed and stored. In this way, it is not necessary for GAParameters to be able to deal with all of the parameters of the whole program, but only the “top-level” ones.

Some common idioms are used repeatedly for parsing input options, especially in the “else-if chain” in GAParameters and in the constructors of algorithms which need parameters. See the code in 3.11.1 for examples.

3.9 Parallelization

Parallelization of the algorithm within a single generation is relatively straightforward. Variations can be used to quickly create a generation’s worth of offspring, and those offsprings’ energy calculations can be run concurrently in separate threads. See lines 37 through 43 in the main loop.

The only problem with this occurs when one offspring fails a development stage. In this case, we can end up having less organisms in a generation than we expected to. In the unlikely case that very few offspring are successful, the algorithm could fail to run normally. However, creating one or two more organisms after finding that some fail would be an inefficient use of resources.
Fortunately, having a slightly smaller offspring generation will rarely be a problem. So the parallelization code needs to take the minimum number of organisms acceptable in any generation as a parameter. If this number is, say, ninety percent of a full generation, then the algorithm will run normally and efficiently. If a new generation falls short of the given threshold, we simply compute more structures until we have enough.

We also take as a parameter the number of energy calculations to run concurrently. We start this many at a time and wait for them all to finish before beginning more.

### 3.10 Description of files

Descriptions of each of the files follow.

- **ConvergenceCriterion.java**
  ConvergenceCriterion is an interface specifying the Boolean method `converged(Generation)`. Each ConvergenceCriterion in use should be called by the algorithm exactly once per generation. It will return true if the population is determined to have converged at which point the algorithm is stopped. By convention, classes implementing ConvergenceCriterion end in “CC”.

- **CrystalGA.java**
  CrystalGA contains the methods that any user of the algorithm will call, whether that user is external code or the command line.

- **DatabaseSOCreator.java**
  DatabaseSOCreator implements StructureOrgCreator. It creates StructureOrgs from a database of prototype structures.

- **Development.java**
  The Development interface is implemented by methods which oversee the “growing up” of an organism by implementing `doDevelop`. If the organism is obviously unfit (hard constraints), `doDevelop` may return false, and the organism should not be considered any further. `doDevelop` may modify the Organism (e.g. structure relaxation), but should not modify the Generation.

- **Energy.java**
  Classes which implement the Energy interface can compute the total lattice energy of a StructureOrg. They’re generally wrappers which call an external code. They are most often not ObjectiveFunctions, but are used by ObjectiveFunctions (e.g. EnergyPerAtom).

- **EnergyPerAtom.java**
  EnergyPerAtom is an ObjectiveFunction. It uses an Energy object to compute the energy of a StructureOrg and then normalizes that by the number of atoms in the StructureOrg to get the Organisms’s value. Notice that there is no chemical potential involved, so this ObjectiveFunction is really only physically useful when we fix the stoichiometry.
• FoundStructureCC.java

FoundStructureCC is a ConvergenceCriterion which indicates convergence when there is a member of the current generation which is the same as a given Structure. The target Structure is given as a Cif file and the matching is done using a RedundancyGuard which calls a StructureFitter internally.

• FromCifsSOCreator.java

FromCifsSOCreator implements StructureOrgCreator. It is given a directory, and it creates StructureOrgs from all Cif files in the directory.

• GAParameters.java

GAParameters is a singleton class which is responsible for parsing and storing most of the command line and input file options. It passes the ones which are particular to a certain algorithm on to that algorithm and stores the rest. The main algorithms and the Population are also owned by GAParameters since GAParameters has all the info about what particular algorithms should be used. The result of this is that the algorithms and Population are kind of pseudo-singletons in that they should only need to be accessed through GAParameters’s getters.

• GARecord.java

GARecord is a singleton class which keeps a record of the status and progress of the genetic algorithm. Its function finishGen is called after each generation and is responsible for outputting results to console and disk. It takes care of other tasks which require a sense of history, e.g. counting the number of generations since the algorithm made improvement.

• GAUtils.java

GAUtils holds a variety of general utility functions as well as the usage statement.

• Generation.java

Generation contains a Vector of Organisms which are treated logically as a generation. It also implements a number of operations on that group such as calculation of fitnesses from values. A generation is not an evolving population: after its creation, it rarely gains or loses members.

• GeneticAlgorithm.java

This holds the main genetic algorithm. It contains the evolutionary strategy in abstract form. For crystal structure prediction, client code should call CrystalGA.crystalGA(). See also Appendix A.

• GulpEnergy.java

GulpEnergy computes the total energy of a StructureOrg using GULP and the given potential. It contains all of the methods and utilities that are specific to GULP.

• ManualSOCreator.java

ManualSOCreator allows for structures to be added by code which uses the genetic algorithm. That code should call GAParameters.setSeedStructures before beginning the algorithm.
• **NumFunctionEvalsCC.java**
  NumFunctionEvalsCC implements `ConvergenceCriterion`. It indicates that the algorithm has converged after a certain number of `ObjectiveFunction` evaluations have been done. Notice that keeping track of the number of evaluations is done by the `ObjectiveFunction` itself.

• **NumGensCC.java**
  NumGensCC implements `ConvergenceCriterion`. It indicates that the algorithm has converged when the algorithm has run for a given number of generations.

• **NumGensWOImprCC.java**
  NumGensWOImprCC implements `ConvergenceCriterion`. It indicates that the algorithm has converged when the algorithm has run for a given number of generations without improving its best solution.

• **NumStoichsMut.java**
  NumStoichsMut implements `Variation`. It creates an offspring `StructureOrg` from a parent by adding or removing a randomly chosen number of stoichiometries worth of species.

• **ObjectiveFunction.java**
  `ObjectiveFunction` is an abstract class which specifies the interface to function we want to minimize.

• **Organism.java**
  Organism is the abstract class which represents members of a Generation in the genetic algorithm. It is implemented by e.g. `StructureOrg`. It maintains the parts of an Organism not specifically related to the structure prediction problem such as value, fitness, and organism ID.

• **Permutation.java**
  Permutation is a `Variation` operation which creates an offspring from one parent by randomly swapping some of its atoms. The number of swaps is normally distributed with mean and standard deviation taken by the function as arguments.

• **ProbDistSelection.java**
  Contains the algorithm for random selection of Organisms as described in [1.4.3].

• **Promotion.java**
  Promotion “promotes” some number of the best organisms in one generation directly to the next.

• **RandomSOCreator.java**
  RandomSOCreator implements `StructureOrgCreator`. It creates `StructureOrgs` as described in [2.1.1].

• **RedundancyGuard.java**
  RedundancyGuard is used by the algorithm to avoid considering identical `StructureOrgs` more than once. It stores a `Map` of all `Structures”seen” to
their Organism IDs. Then, other algorithms (e.g. StructureDev) may check Organisms against the list of those already seen. Comparisons are done first by a direct comparison of lattice parameters and atomic positions and second, if necessary, by the slower but more accurate StructureFitter.

- **ResumeSOCreator.java**
  ResumeSOCreator implements StructureOrgCreator. It creates new StructureOrgs from the output of a previous GA run and is generally used for the limited resume-type functionality described in 2.1.1.

- **Selection.java**
  Selection is an interface to objects which take a Generation and an integer, $n$, and returns a vector of $n$ organisms from the Generation. It assumes that fitnesses have already been calculated.

- **Slicer.java**
  Slicer is a Variation operation. It implements the algorithm described in 1.4.5.

- **StructureDev.java**
  StructureDev implements Development and is the Development operation generally used with StructureOrgs. See 1.4.8.

- **StructureMut.java**
  StructureMut is a Variation operation which acts on StructureOrg organisms. See 1.4.5.

- **StructureOrg.java**
  StructureOrg extends Organism and has a Structure. It is the type of Organism generally used by the genetic algorithm for the crystal structure prediction problem.

- **StructureOrgCreator.java**
  StructureOrgCreator is an interface to objects which create StructureOrgs, generally for use in the initial population.

- **Structures.java**
  Structures extends Generation and has list of StructureOrg. It is the type of Generation generally used by the genetic algorithm for the crystal structure prediction problem.

- **ValueAchievedCC.java**
  ValueAchievedCC is a ConvergenceCriterion which indicates convergence when there is at least one member of the current generation with a value less than or equal to a given target value.

- **Variation.java**
  A Variation takes a pair of generations and considers them in creating an Organism to be added to the offspring generation. Neither Generation should be modified. Application of Variations, along with Promotion, are how a new Generation is created from a parent Generation.

- **VaspEnergy.java**
  VaspEnergy implements Energy. It computes the total energy of a StructureOrg using Vasp and the given pseudopotentials.
3.11 Modifying the code

3.11.1 Example: adding a parameter

Recall that the algorithm can change behavior when it thinks it has converged. Suppose we want to add the parameter `endGameNumGens` which indicates the number of generations without improvement the algorithm will run before it decides it has probably converged. The line we add to the input file will look something like:

```
endGameNumGens 6
```

This parameter applies to the algorithm as a whole rather than to any particular subroutine, so it will live in `GAParameters`. We add the following private variable to `GAParameters`:

```java
private int endGameNumGens = 50;
```

with a very conservative default value. Then, we add it to the chain of else-ifs in `GAParameters.setArgs`:

```java
else if (flag.equalsIgnoreCase('--endgameNumGens'))
    endGameNumGens = Integer.parseInt(getValues(flag)[0]);
```

The easiest way to do this part is probably to copy and paste the code from a pre-existing parameter. Notice the “--” which is necessary whether the flag was passed in on the command line or from the input file. We are not really validating the form of the input, but the default error that `Integer.parseInt` or `Double.parseDouble` will throw if given malformed input is satisfactory.

Now, the endGame check is implemented in `GARecord` which is responsible for recording what has happened over past generations. We need to make the parameter available to `GARecord`, so we add the following getter at the bottom of `GAParameters`:

```java
public int getEndGameNumGens() {
    return endGameNumGens;
}
```

We have adhered to the convention of naming getters as “get” plus the name of the relevant parameter. Now we may access the parameter from `GARecord` (or anywhere else) using

```
int n = GAParameters.getParams().getEndGameNumGens();
```

Finally, we are sure to add an entry to the usage statement in `GAUtils` and the `toString` method of `GAParameters`.

3.11.2 Example: testing new variations

It is possible and often helpful to test a `Variation` without performing a full run of the genetic algorithm. Furthermore, the code to do so is a good example of how many of the objects we have discussed fit together. The following is an adapted version of `Slicer`’s main method:
public static void main(String[] args) {
    GAParameters params = GAParameters.getParams();
    Structure p1 = StructureOrg.parseCif(new File("parent1.cif"));
    Structure p2 = StructureOrg.parseCif(new File("parent2.cif"));

    Generation parents = params.makeEmptyGeneration();
    parents.addOrganism(new StructureOrg(dense1));
    parents.addOrganism(new StructureOrg(dense2));

    String[] vArgs = {
        "0.5", "0.0", "0", "0.00", "1", "2"};
    Variation p = new Slicer(vArgs);
    String[] selArgs = {
        "2", "0"};
    Selection sel = new ProbDistSelection(selArgs);

    StructureOrg o = (StructureOrg)p.doVariation(parents, null, sel);
    GAUtils.writeStringToFile(o.getCIF(), new File("out.cif"), false);
}

The variation itself happens on line 15. It requires a parents Generation and a Selection to act. Setting the second argument to null indicates that no development will be done. The creation of the Variation object itself is on lines 10 and 11. The Selection is made on lines 12 and 13. Clearly, it needs to select from the top two organisms in the generation, but the selection power is irrelevant.

Now we must create the parents Generation. It will contain the two parent organisms which are loaded from files in lines 3 and 4. The Generation itself is created through the call on line 6 and the two parents are added to in on lines 7 and 8. Line 2 is an idiom which occurs many times in the code. We are accessing and storing the parameters singleton for repeated use.

Finally, on line 16, the Cif file of the offspring is written to disk.
Appendix A

Main loop

A partial listing of GeneticAlgorithm.java including the main loop follows. See 3.3 for a discussion.

```java
public static Organism doGeneticAlgorithm(GAParameters params) {
  GARecord record = GARecord.getRecord();
  int verbosity = params.getVerbosity();

  // Create the objects for the algorithms
  Selection sel = params.getSelection();
  Development dev = params.getDevelopment();
  Promotion pro = params.getPromotion();

  // Create the initial population
  Generation parents = null;
  Generation offspring = null;

  // the main algorithm loop:
  while (!converged(offspring)) {
    // the wheel of life turns
    parents = offspring;
    offspring = GAParameters.getParams().makeEmptyGeneration();

    // do promotions (except for the first generation)
    if (GARecord.getRecord().getGenNum() != 0)
      pro.doPromotion(parents, offspring);

    // make the offspring generation
    while (!madeEnough(offspring)) {
      List<Organism> organisms = new ArrayList<Organism>();
      List<Thread> threads = new ArrayList<Thread>();

      for (int i = 0; i < params.getNumCalcsInParallel(); i++) {
        // make a new organism
        Organism newOrg = getNewOrg(parents, offspring, sel);
        if (newOrg == null)
          break;

        organisms.add(newOrg);
        threads.add(new Thread(newOrg));
      }

      for (Thread t : threads)
        t.start();
continue;
organisms.add(newOrg);

// start the energy computation
threads.add(params.getObjectiveFunctionInstance(newOrg).evaluate());

// wait for the energy computations
try {
   for (Thread t : threads)
      t.join();
} catch (InterruptedException x) {
   if (verbosity >= 3)
      System.out.println("InterruptedException in energy calc thread: " + x.getMessage());
}
// re-develop each of the organisms and possibly
// add it to the offspring generation
for (Organism o : organisms) {
   if (dev != null && !dev.doDevelop(offspring, o))
      continue;
   offspring.addOrganism(o);
}

// find the organisms' fitnesses
if (verbosity >= 2)
   System.out.println("Starting fitness evaluations...");
offspring.findFitnesses();
// prints and saves status/progress info and whatnot
record.finishGen(offspring);
}
record.cleanup();
return offspring.getNthBestOrganism(1);
Appendix B

Using GridEngine and Vasp

To perform an energy calculation using Vasp, the genetic algorithm runs the command

    callvasp <input file directory>

It expects that when this script is done running, the *vasprun.xml* file is fully written and ready for parsing. That is, the *callvasp* script can not end until the Vasp calculation is complete.

In order to submit jobs to a GridEngine queue, the *qsub* command is used. However, *qsub* exits immediately. The following *callvasp* script is a work-around for this issue.

```
#!/bin/bash

cd $1
qsubandpoll /home/wtipton/bin/submitjob
```

where *qsubandpoll* is a script on the user’s path and the path to *submitjob* will change.

The heart of this solution is the *qsubandpoll* script. It gives each job a unique random name, submits the job to the queue, and then checks the output of *qstat* at regular intervals to determine when the job has finished. A listing of *qsubandpoll* follows:

```
#!/bin/bash

INTERVAL=1
TEMP='mktemp'
UNIQUENAME='echo $TEMP|sed ’s/tmptmp//’'

qsub -N $UNIQUENAME $1
while : ; do
    sleep $INTERVAL
    MATCH='qstat | grep -o $UNIQUENAME'
    if [ "$MATCH" == "" ]; then
        break
    fi
done
```
Notice that on lines 4, 5 and 10 the outer ' characters are backticks.

Finally, submitjob is a normal qsub script. Keeping in mind that the genetic algorithm expects vasprun.xml to be available in the output, submitjob might look something like

```bash
#$ -o vasp.out
/share/apps/aflow173 -vasp /share/apps/vasp/vasp.core2.64
gzip -d vasprun.xml.relax2.gz
mv vasprun.xml.relax2 vasprun.xml
```
Appendix C

Example input file

Please see the QuickStart guide located at [http://gasp.mse.cornell.edu/manual.html](http://gasp.mse.cornell.edu/manual.html).

It includes an input file for a Lennard-Jones Gulp run. It errs on the side of conservativism and may be used as a basic template for future runs. Recall that lines beginning with "#" are ignored, everything (except sometimes atomic symbols) is case-insensitive, and units are generally eV, Angstroms and degrees. Detailed descriptions of options are given in 2.1.1.
Appendix D

Scripts for testing parameters

In order to evaluate the dependence of the success of the algorithm on its parameters, three scripts were used to repeatedly run the algorithm while varying a parameter, to parse the output for the number of energy calculations needed, and to visualize the result. The first of these follows.

```bash
#!/bin/bash

NUMRUNSPERVALUE=20
GA_IN_TEMPLATE=/home/wtipton/testing/ljc_newt/input_template
INITIALVALUE=0.5
INCREMENT=0.1
GA="/home/wtipton/projects/ga_for_crystals/ga.jar"

#### Shouldn't be necessary to change below here ####
GA_IN_KEY_TO_CHANGE=CHANGEME
mkdir runs || exit
cd runs

for VALUE in 'seq -w $INITIALVALUE $INCREMENT $FINALVALUE'
do
    # make the input file for this value of the varied parameter
    GA_IN="ga_in_$VALUE"
    sed "s/$GA_IN_KEY_TO_CHANGE/$VALUE/" "$GA_IN_TEMPLATE" >"$GA_IN"
    # run the GA NUMRUNSPERVALUE times for this value
    for RUN in 'seq 1 $NUMRUNSPERVALUE'
do
        echo "Starting run $RUN of value $VALUE"
        #replace the outDirName
        sed -i 's/.*/outDirName.*/' "$GA_IN"
        echo outDirName "$VALUE.$RUN" >>$GA_IN
    done
done
```

43
This script runs the genetic algorithm repeatedly while varying a parameter. The user should only have to change the variables at the top of the script and the input template for any test. The input template should be a regular input file with the value that is going to be varied set to “CHANGEME.” The algorithm will be run with that value set to each of the values between the initial and final values given. Make sure to set reasonable convergence criteria.

The second script, parse_results.sh, should be run from the same directory as the first. It parses the directory structure created by the first script and outputs a file called “results.”

```bash
#!/bin/bash
	rm results
cd runs
ls output* | while read i ; do
    NUMCALCS='cat "$i" | tail -1 | cut -d ' ' -f 7'
    VALUE='echo $i|sed 's/output\.///' | sed 's/\.[^\./]*/$//' | sed 's/ //g''
    echo $VALUE $NUMCALCS >>../results
done
```

There is one line in the results file for each run of the genetic algorithm. On each line are two numbers: the value of the parameter and the number of energy calculations, respectively. The following Matlab script, testgraphs.m, takes the name of the results file, parses it and graphs the results.

```matlab
function testgraphs(filename)
%
% read in ourFileName
fid = fopen(filename);
C = textscan(fid, '%s', 'delimiter', '
');
fclose(fid);
lines = C{1};

% initialize data structures
values = [];
umcalcs = [];

% read in the data
for i = 1:length(lines)
    line = sscanf(lines{i}, '%f');
    if (numel(line) ~= 2)
        continue
    end
    values(end+1) = line(1);
umcalcs(end+1) = line(2);
end
```
% averages
avgCalcs=[];
stddevs=[];
for i = unique(values)
    currentCalcs = nonzeros(numcalcs .* (values == i));
    avgCalcs(end+1) = median(currentCalcs);
    %avgCalcs(end+1) = mean(currentCalcs);
    stddevs(end+1) = std(currentCalcs);
end

% plot it
hold on
errorbar(unique(values), avgCalcs, stddevs,'r');
plot(values, numcalcs, '.');
hold off
xlabel('value');
ylabel('num calcs');
title(['stuff']);
References