
The Otago Research Genetic Algorithm for Nanoclusters, Including Structural Methods and Similarity Documentation

Release 3

Geoffrey Weal

Mar 25, 2021

CONTENTS

1	What is this Documentation about?	3
2	What is Organisms	5
3	Try Organisms before you Clone/Pip/Conda (on Binder/Jupyter Notebooks)!	7
4	Installation	9
5	Table of Contents	11
5.1	How the Otago Research Genetic Algorithm for Nanoclusters, Including Structural Methods and Similarity (Organisms) Program Works	11
5.2	Installation: Setting Up the Organisms Program and Pre-Requisites Packages	13
5.3	How To Use The Organisms Program	19
5.4	<i>Run.py</i> - Running the Genetic Algorithm	20
5.5	Examples of Running the Organisms Program with <i>Run.py</i>	30
5.6	<i>RunMinimisation.py</i> - Writing a Local Minimisation Function for the Genetic Algorithm	30
5.7	<i>MakeTrials.py</i> - Creating Multiple, Repeated Genetic Algorithm Trials	37
5.8	Safely Finishing the Genetic Algorithm Midway through the Algorithm	48
5.9	Restarting the Genetic Algorithm	49
5.10	Common Issues of the Genetic Algorithm and Ways to Solve Them	50
5.11	Helpful Programs to Create and Run the Genetic Algorithm	52
5.12	Helpful Programs for Gathering data and Post-processing Data	55
5.13	Information about using the <i>make_energy_vs_similarity_results.py</i> script	58
5.14	Other Helpful Programs for Gathering data and Post-processing Data	66
5.15	Initialising a New Population	67
5.16	Using Predation Operators with the Genetic Algorithm	67
5.17	Using Fitness Operators with the Genetic Algorithm	73
5.18	The Structural Comparison Method (SCM)	79
5.19	Using the Memory Operator	80
5.20	Using Epoch Methods	80
5.21	Recording Clusters From The Genetic Algorithm	81
5.22	Using Databases with the Genetic Algorithm	84
5.23	Adding Surfaces	88
5.24	The Genetic Algorithm Python Files	88
5.25	Index	165
5.26	Python Module Index	165
6	Indices and tables	167
	Python Module Index	169

Section author: Geoffrey Weal <geoffrey.weal@gmail.com>

Group page: <https://blogs.otago.ac.nz/annagarden/>

Page to cite with work from: Development of a Structural Comparison Method to Promote Exploration of the Potential Energy Surface in the Global Optimisation of Nanoclusters; Geoffrey R. Weal, Samantha M. McIntyre, and Anna L. Garden; JCIM; in submission stage.

WHAT IS THIS DOCUMENTATION ABOUT?

This documentation is designed to guide the user to use the Otago Research Genetic Algorithm for Nanoclusters, Including Structural Methods and Similarity (Organisms) program.

WHAT IS ORGANISMS

The Otago Research Genetic Algorithm for Nanoclusters, Including Structural Methods and Similarity (Organisms) program is designed to perform a genetic algorithm global optimisation for nanoclusters. It has been designed with inspiration from the Birmingham Cluster Genetic Algorithm and the Birmingham Parallel Genetic Algorithm from the Roy Johnston Group (see J. B. A. Davis, A. Shayeghi, S. L. Horswell, R. L. Johnston, *Nanoscale*, 2015, 7, 14032 (<https://doi.org/10.1039/C5NR03774C> or link to Davis pdf here⁸) and R. L. Johnston, *Dalton Trans.*, 2003, 4193–4207 (<https://doi.org/10.1039/B305686D> or link to Johnston pdf here⁹).

This algorithm is designed to explore the potential energy surface of a cluster system, using the genetic algorithm, and to local the putative globally lowest energetic cluster. It was designed for obtaining low energy structures of clusters that could be catalytically interesting. The algorithm was designed by Dr Anna Garden and the Garden group at the University of Otago, Dunedin, New Zealand. See for more information about what the group does at blogs.otago.ac.nz/annagarden¹⁰. The Github page for this program can be found at github.com/GardenGroupUO/Organisms¹¹.

Genetic Algorithm

Operators

SCM-based predation operator

“Structure + energy” fitness operator

Recording clusters in databases

The genetic algorithm uses the ideas of Darwin’s theory of evolution to locate the global minimum.

This implementation of the genetic algorithm includes various predation, fitness and epoch operators. Also included is the SCM-based predation operator and a “structure + energy” fitness operator.

The SCM-based predation operator compares the structures of clusters together and excludes clusters from the population that are too similar to each other.

The “structure + energy” fitness operator is designed to include a portion of structural diversity into the fitness value as well as energy. The goal of this fitness operator is to guide the genetic algorithm around to unexplored areas of a cluster’s potential energy surface.

With the use of Atomic Simulation Environment, this algorithm has been designed so that you can record all the clusters you make, or just the important ones such as the lowest energy clusters that you make

⁸ <https://pubs.rsc.org/en/content/articlepdf/2015/nr/c5nr03774c>

⁹ <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.124.6813&rep=rep1&type=pdf>

¹⁰ <https://blogs.otago.ac.nz/annagarden/>

¹¹ <https://github.com/GardenGroupUO/Organisms>

TRY ORGANISMS BEFORE YOU CLONE/PIP/CONDA (ON BINDER/JUPTER NOTEBOOKS)!

If you are new to the Organisms program, it is recommended try it out by running Organisms live on our interactive Jupyter+Binder page before you download it. On Jupyter+Binder, you can play around with the Organisms program on the web. You do not need to install anything to try Organisms out on Jupyter+Binder.

Click the Binder button below to try Organisms out on the web! (The Binder page may load quickly or may take 1 or 2 minutes to load)

¹²

¹² https://mybinder.org/v2/gh/GardenGroupUO/Organisms_Jupyter_Examples/main?urlpath=lab

INSTALLATION

It is recommended to read the installation page before using the Organisms program. See *Installation: Setting Up the Organisms Program and Pre-Requisites Packages* for more information. Note that you can install Organisms through `pip3` and `conda`.

TABLE OF CONTENTS

5.1 How the Otago Research Genetic Algorithm for Nanoclusters, Including Structural Methods and Similality (Organisms) Program Works

5.1.1 How the Genetic Algorithm Works

The genetic algorithm is based on Darwin's Theory of Evolution. The algorithm implemented in this program will first create a population of randomly generated clusters. However, the population can be partially or fully populated with user-created clusters (more information given *Initialising a New Population*).

Individuals in the population are mated together and/or mutated to give a set of new offspring. These offspring can be removed if they are too similar to other offspring or clusters in the population if they are too similar to each other in some way. This is determined by a predation operator (more information is given in *Using Predation Operators*). The offspring are then added to the population.

The clusters are also assigned a fitness based on a fitness operator (more information is given in *Using Fitness Operators*). The fitness is used to determine how clusters will be used to pick parents and clusters for the mating and mutation procedures, in removing clusters during the predation procedure, and during the natural selection procedure (reference to *Using Fitness Operators* for more information).

Finally, a "Natural Selection" process will remove the least fit clusters from the population, while the fitter clusters will remain in the population. This process, from the creation of new offspring to the natural selection process is called a generation.

This process is repeated until the desired number of generations are reached.

5.1.2 How the Genetic Algorithm Works in Organisms

This implementation of the genetic algorithm has been designed with a few ideas in mind

1. The algorithm has been designed to warn the user if there is any chance the inputs the user has inputted will break the genetic algorithm or give nonsense results. The algorithm will not begin and purposely exit without running if it thinks that there will be a problem that would cause the algorithm to break or not function as intended. **It is recommended that you run your genetic algorithm at least once for at least 2 generations to check it will run without any obvious issues arising.**
2. This implementation has been designed as best as possible to allow for further modification of the algorithm to allow the user to play with the algorithm and add and modify functionalities of the algorithm as to research how to improve the efficiency of the genetic algorithm. This is especially the case for predation and fitness operators which were the focus of a PhD Thesis.

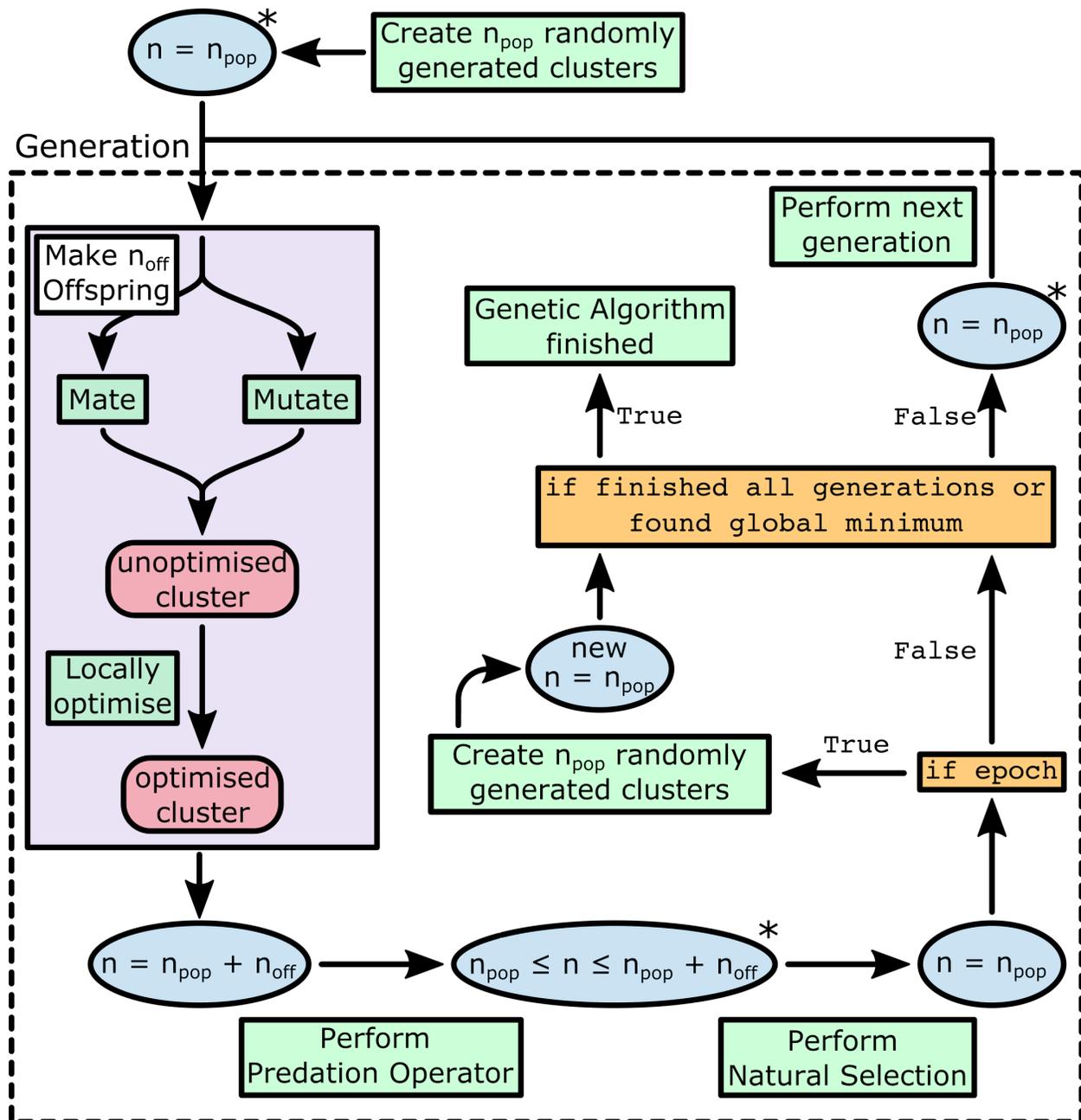


Fig. 1: This is a general schematic of the genetic algorithm applied to clusters.

3. This algorithm has been designed to include as many features as possible that have been mentioned or explained in the literature.
4. This algorithm has been designed to be restartable. This features that allow this algorithm to be restartable are explained further in *Restarting the Genetic Algorithm*.
5. This algorithm is the first genetic algorithm that has been created for globally optimising clusters in the Garden group. For this reason and because the initial goal of Geoff's PhD was to improving the efficiency of the genetic algorithm, making sure the algorithm works as intended with minimal bugs was more important than the speed of the algorithm (in regards to real time speed and not efficiency of the algorithm), and making sure the algorithm was written well was priority. Attempts have been made to increase the speed and decrease the running time of the algorithm have been made, such as parallelising the algorithm, however it is likely that this algorithms speed can not be improved until the generation procedure of this algorithm is written without algorithm checks and written in a faster language. The speed of the ASE based local optimisers may also be an issue, as these are also written in python.

5.2 Installation: Setting Up the Organisms Program and Pre-Requisites Packages

In this article, we will look at how to install the genetic algorithm and all requisites for this program.

5.2.1 Pre-requisites

Python 3 and pip3

This program is designed to work with **Python 3**. While this program has been designed to work with Python 3.6, it should work with any version of Python 3 that is the same or later than 3.6.

To find out if you have Python 3 on your computer and what version you have, type into the terminal

```
python3 --version
```

If you have Python 3 on your computer, you will get the version of python you have on your computer. E.g.

```
geoffreyweal@Geoffreys-Mini Documentation % python3 --version
Python 3.6.3
```

If you have Python 3, you may have pip3 installed on your computer as well. pip3 is a python package installation tool that is recommended by Python for installing Python packages. To see if you have pip3 installed, type into the terminal

```
pip3 list
```

If you get back a list of python packages install on your computer, you have pip3 installed. E.g.

```
geoffreyweal@Geoffreys-Mini Documentation % pip3 list
Package              Version
-----
alabaster             0.7.12
asap3                 3.11.10
ase                   3.20.1
Babel                 2.8.0
certifi               2020.6.20
```

(continues on next page)

(continued from previous page)

chardet	3.0.4
click	7.1.2
cycler	0.10.0
docutils	0.16
Flask	1.1.2
idna	2.10
imagesize	1.2.0
itsdangerous	1.1.0
Jinja2	2.11.2
kiwisolver	1.2.0
MarkupSafe	1.1.1
matplotlib	3.3.1
numpy	1.19.1
packaging	20.4
Pillow	7.2.0
pip	20.2.4
Pygments	2.7.1
pyparsing	2.4.7
python-dateutil	2.8.1
pytz	2020.1
requests	2.24.0
scipy	1.5.2
setuptools	41.2.0
six	1.15.0
snowballstemmer	2.0.0
Sphinx	3.2.1
sphinx-pyreverse	0.0.13
sphinx-rtd-theme	0.5.0
sphinx-tabs	1.3.0
sphinxcontrib-applehelp	1.0.2
sphinxcontrib-devhelp	1.0.2
sphinxcontrib-htmlhelp	1.0.3
sphinxcontrib-jsmath	1.0.1
sphinxcontrib-plantuml	0.18.1
sphinxcontrib-qthelp	1.0.3
sphinxcontrib-serializinghtml	1.1.4
sphinxcontrib-websupport	1.2.4
urllib3	1.25.10
Werkzeug	1.0.1
wheel	0.33.1
xlrd	1.2.0

If you do not see this, you probably do not have pip3 installed on your computer. If this is the case, check out [PIP Installation](#)¹³

¹³ <https://pip.pypa.io/en/stable/installing/>

Atomic Simulation Environment

This genetic algorithm uses the atomic simulation environment (ASE) for two purposes. The description of a cluster that has been designed for this algorithm is based on that given by ASE's Atoms class. This allows the genetic algorithm to take advantage of the features of ASE, such as the wide range of calculators that can be used to calculate the energy of the cluster, and the local optimisers available to optimise offspring created during the genetic algorithm. Furthermore, ASE also offers useful tools for viewing, manipulating, reading and saving clusters and chemical systems easily. Read more about [ASE here](https://wiki.fysik.dtu.dk/ase/)¹⁴. For this genetic algorithm, it is recommended that you **install a version of ase that is 3.19.1 or greater**.

The installation of ASE can be found on the [ASE installation page](https://wiki.fysik.dtu.dk/ase/install.html)¹⁵, however from experience if you are using ASE for the first time, it is best to install ASE using pip, the package manager that is an extension of python to keep all your program easily managed and easy to import into your python.

To install ASE using pip, perform the following in your terminal.

```
pip3 install --upgrade --user ase
```

Installing using pip3 ensures that ASE is being installed to be used by Python 3, and not Python 2. Installing ASE like this will also install all the requisite program needed for ASE. This installation includes the use of features such as viewing the xyz files of structure and looking at ase databases through a website. These should be already assessible, which you can test by entering into the terminal:

```
ase gui
```

This should show a gui with nothing in it, as shown below.

However, in the case that this does not work, we need to manually add a path to your `~/ .bashrc` so you can use the ASE features externally outside python. First enter the following into the terminal:

```
pip3 show ase
```

This will give a bunch of information, including the location of ase on your computer. For example, when I do this I get:

```
Geoffreys-Mini:~ geoffreyweal$ pip show ase
Name: ase
Version: 3.20.1
Summary: Atomic Simulation Environment
Home-page: https://wiki.fysik.dtu.dk/ase
Author: None
Author-email: None
License: LGPLv2.1+
Location: /Users/geoffreyweal/Library/Python/3.8/lib/python/site-packages
Requires: matplotlib, scipy, numpy
Required-by:
```

In the 'Location' line, if you remove the 'lib/python/site-packages' bit and replace it with 'bin'. The example below is for Python 3.6.

```
/Users/geoffreyweal/Library/Python/3.6/bin
```

This is the location of these useful ASE tools. You want to put this as a path in your `~/ .bashrc` as below:

¹⁴ <https://wiki.fysik.dtu.dk/ase/>

¹⁵ <https://wiki.fysik.dtu.dk/ase/install.html>

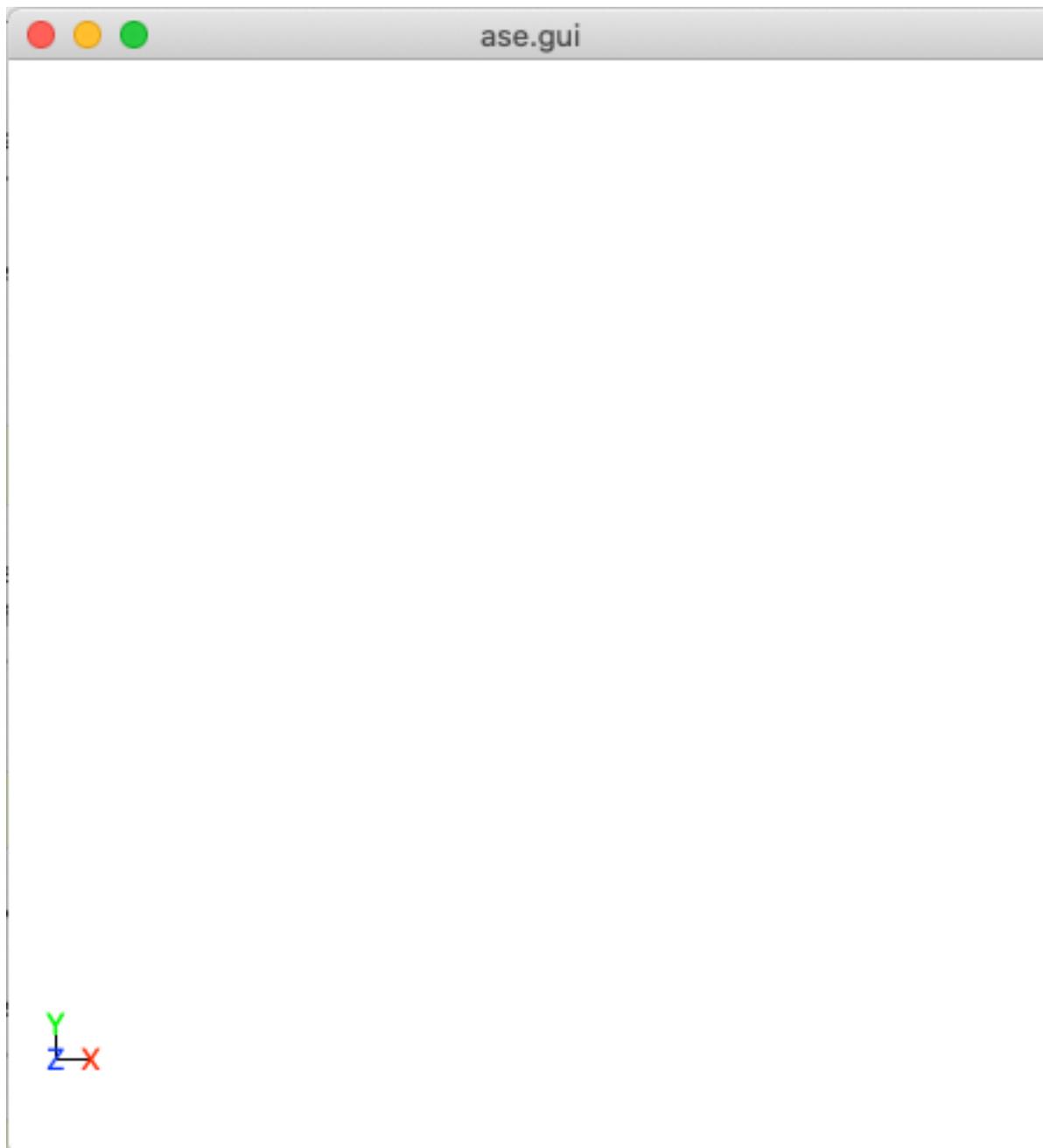


Fig. 2: This is a blank ase gui screen that you would see if enter ase gui into the terminal.

```
#####  
# For ASE  
export PATH=/Users/geoffreyweal/Library/Python/3.6/bin:$PATH  
#####
```

As Soon As Possible (ASAP)

As Soon As Possible (ASAP) is not a pre-requisite of this program, however it is a great source of empirical potentials that can be used to calculate the energies of clusters and chemical systems with ASE and with this program. It is required however if you want to use either the SCM-based predation operator or the structure + energy fitness operator (click here for more information on the *SCM-based predation operator* and the *structure + energy fitness operator*). You can read more about it at [ASAP¹⁶](#). You can find out how to install it at [ASAP Installation¹⁷](#), however I have found the best way to use it simply is using pip. To install using pip, perform the following in the terminal.

```
pip3 install --upgrade --user asap3==3.11.10
```

Where we install asap3 version 3.11.10. Generally, this program takes a bit of time to install. **NOTE: We require that you use asap3 version 3.11.10. This is because we have noticed a (core dump) issue that seems to occur during the genetic algorithm. Unfortunately, this error appears at seemingly random times so we don't know what the problem is, but it seems to be resolved if you use this version of asap3**

5.2.2 Setting up Organisms

There are two ways to install Organisms on your system. These ways are described below:

Install Organisms through pip3

To install the Organisms program using pip3, perform the following in your terminal.

```
pip3 install --upgrade --user Organisms
```

You should be able to access the genetic algorithm as well as run the scripts and commands described in *Helpful Programs to Create and Run the Genetic Algorithm*, *Helpful Programs for Gathering data and Post-processing Data*, and *Other Helpful Programs for Gathering data and Post-processing Data* in the terminal.

The website for Organisms on pip3 can be found by clicking the button below:

¹⁸

¹⁶ <https://wiki.fysik.dtu.dk/asap/>

¹⁷ <https://wiki.fysik.dtu.dk/asap/Installation>

¹⁸ <https://pypi.org/project/Organisms/>

Install Organisms through conda

You can also install Organisms through conda, however I am not as versed on this as using pip3. See docs.conda.io¹⁹ to see more information about this. Once you have installed anaconda on your computer, I believe you install the Organisms program using conda by performing the following in your terminal.

```
conda install ase
conda install asap3
conda install organisms
```

The website for Organisms on conda can be found by clicking the button below:

20

Manual installation

First, download Organisms to your computer. You can do this by cloning a version of this from Github, or obtaining a version of the program from the authors. If you are obtaining this program via Github, you want to cd to the directory that you want to place this program in on the terminal, and then clone the program from Github through the terminal as well

```
cd PATH/TO/WHERE_YOU_WANT_Organisms_TO_LIVE_ON_YOUR_COMPUTER
git clone https://github.com/GardenGroupUO/Organisms
```

Next, add a python path to it in your .bashrc to indicate its location. Do this by entering into the terminal where you cloned the Organisms program into pwd

```
pwd
```

This will give you the path to the Organisms program. You want to enter the result from pwd into the .bashrc file. This is done as shown below:

```
export PATH_TO_GA="<Path_to_Organisms>"
export PYTHONPATH="$PATH_TO_GA":$PYTHONPATH
```

where "<path_to_Organisms>" is the directory path that you place Organisms (Enter in here the result you got from the pwd command). Once you have run `source ~/.bashrc`, the genetic algorithm should be all ready to go!

Organisms contains many parts to it. You will see that there are six folders. These are GA, Subsidiary_Programs, Postprocessing_Programs, Helpful_Programs, Examples and Documentation.

The genetic algorithm is completely contained in the folder GA. If everything is working as it should, and you do not want to modify the genetic algorithm program, you shouldnt need to access it.

The folder called Examples contains all the files that one would want to used to use the genetic algorithm. This includes examples of the basic run code for the genetic algorithm, the Run.py and RunMinimisation.py files (in the Playground folder), as well as the files that can be used to create, execute, and coagulate data from multiple runs of the genetic algorithm (in the CreateSets folder).

This genetic algorithm contains many programs that can help you create and run the genetic algorithm (in Subsidiary_Programs), for postprocessing the data from your genetic algorithm(s) (Postprocessing_Programs), and other helpful programs for determining how the algorithm has run or for learning other things about your genetic algorithm runs (Helpful_Programs). To execute any of these programs contained within either the Subsidiary_Programs, Postprocessing_Programs, or Helpful_Programs folders, include the following in your ~/.bashrc:

¹⁹ <https://docs.conda.io/projects/conda/en/latest/user-guide/tasks/manage-pkgs.html>

²⁰ <https://anaconda.org/GardenGroupUO/organisms>

```
export PATH="$PATH_TO_GA"/Organisms/Subsidiary_Programs:$PATH
export PATH="$PATH_TO_GA"/Organisms/Postprocessing_Programs:$PATH
export PATH="$PATH_TO_GA"/Organisms/Helpful_Programs:$PATH
```

See *Helpful Programs to Create and Run the Genetic Algorithm, How Organisms Works* for more information about the programs that are available in the Subsidiary_Programs and Postprocessing_Programs folders respectively. You can also see *Other Helpful Programs for Gathering data and Post-processing Data* for information on how to use the scripts found in the Helpful_Programs folder.

Other Useful things to know before you start

- You may use `squeue` to figure out what jobs are running in slurm. For monitoring what genetic algorithm jobs are running, I have found the following alias useful. Include the following in your `~/ .bashrc` (see *How to execute all Trials using the JobArray Slurm Job Submission Scheme* for what is going on in the below line)

```
alias qme='squeue -o "%.20i %.9P %.5Q %.50j %.8u %.8T %.10M %.11l %.6D %.4C %.6b %.
↵20S %.20R %.8q" -u $USER --sort=+i'
```

Summary of what you want in the `~/ .bashrc` for the Organisms program if you manually installed the Organisms

You want to have the following in your `~/ .bashrc`:

```
#####
# Paths and Pythonpaths for the genetic algorithm

export PATH_TO_GA="<<Path_to_Organisms>"
export PYTHONPATH="$PATH_TO_GA":$PYTHONPATH

export PATH="$PATH_TO_GA"/Organisms/Subsidiary_Programs:$PATH
export PATH="$PATH_TO_GA"/Organisms/Postprocessing_Programs:$PATH
export PATH="$PATH_TO_GA"/Organisms/Helpful_Programs:$PATH

alias qme='squeue -o "%.20i %.9P %.5Q %.50j %.8u %.8T %.10M %.11l %.6D %.4C %.6b %.
↵20S %.20R %.8q" -u $USER --sort=+i'

#####
```

5.3 How To Use The Organisms Program

This program uses an input python file that contains all the information needed to run the Organisms program. This is called the **Run.py** file. You can read more about how to construct this file in *Run.py - Using the Organisms program*. Executing the Run.py file will perform the genetic algorithm for a cluster of your choosing. The genetic algorithm can also be run on a cluster on a surface if required. An example of the Run.py file can be found in Examples/Playground

However, it is likely that one genetic algorithm run may not be enough to confidently obtain the global minimum structure. This is because the genetic algorithm, or any global optimisation algorithm, is not guaranteed to locate the global minimum structure, and therefore the global minimum may not be found the first time the genetic algorithm is used. For this reason, this program comes a few extra scripts to make and simultaneously run multiple genetic algorithms easier, including setting up files to be used with the slurm job schedule manager. The Examples/CreateSets folder contains a file called **MakeTrials.py** that will create many repeated trials, using the same genetic algorithm

parameters. You can use this program to create and organise your directory to run multiple genetic algorithm runs. Read more about this feature at *MakeTrials.py - Creating Multiple, Repeated Genetic Algorithm Trials*.

There are two further pages that contain information that is useful for working with this program.

- *Helpful Programs to Create and Run the Genetic Algorithm* contains information about all the programs that can be used to create other files, such as randomly generated clusters, and run them on mass with as minimal work required by the user as possible. Note again that information on how to use make a number of repeated trials on mass can be found in *MakeTrials.py - Creating Multiple, Repeated Genetic Algorithm Trials*.
- *Helpful Programs for Gathering and Post-processing Data* contains information about programs designed for processing all the trials together, such as programs to check that all the trials had completed, determining the generations and number of minimisations required to obtain the lowest energy cluster each trial could find, the average number of generations and average number of minimisations required to obtain the lowest energy cluster from any of the trials performed, and more post-processing programs.

5.4 Run.py - Running the Genetic Algorithm

In this article, we will look at how to run the genetic algorithm. This program is run though the **Run.py** script, which includes all the information on what cluster to globally optimise and the genetic algorithm settings to use. You can find other examples of `Run.py` files at github.com/GardenGroupUO/Organisms²¹ under `Examples\Playground` and `Examples\Example_Run_Files`. Also, you can try out this program by running an example script through a Jupyter notebook. See *Examples of Running the Organisms Program with Run.py* to get access to examples of running Organisms through this Jupyter notebook!

5.4.1 Running the Genetic Algorithm Program

We will explain how the `Run.py` code works by running though the example shown below:

Listing 1: Run.py

```
1 from Organisms import GA_Program
2
3 # This details the elemental and number of atom composition of cluster that the user_
4 ↪would like to investigate
5 cluster_makeup = {"Cu": 37}
6
7 # Surface details
8 surface_details = None #{'surface': 'surface.xyz', 'place_cluster_where': 'center'}
9
10 # These are the main variables of the genetic algorithm that with changes could_
11 ↪affect the results of the Genetic Algorithm.
12 pop_size = 20
13 generations = 2000
14 no_offspring_per_generation = 16
15
16 # These setting indicate how offspring should be made using the Mating and Mutation_
17 ↪Procedures
18 creating_offspring_mode = "Either_Mating_and_Mutation"
19 crossover_type = "CAS_weighted"
20 mutation_types = [['random', 1.0]]
21 chance_of_mutation = 0.1
```

(continues on next page)

²¹ <https://github.com/GardenGroupUO/Organisms>

(continued from previous page)

```

20 # This parameter will tell the Organisms program if an epoch is desired, and how the_
    ↪user would like to proceed.
21 epoch_settings = {'epoch mode': 'same population', 'max repeat': 5}
22
23 # These are variables used by the algorithm to make and place clusters in.
24 r_ij = 3.4
25 cell_length = r_ij * (sum([float(noAtoms) for noAtoms in list(cluster_makeup.
    ↪values())]) ** (1.0/3.0))
26 vacuum_to_add_length = 10.0
27
28 # The RunMinimisation.py algorithm is one set by the user. It contain the def_
    ↪Minimisation_Function
29 # That is used for local optimisations. This can be written in whatever way the user_
    ↪wants to perform
30 # the local optimisations. This is meant to be as free as possible.
31 from RunMinimisation import Minimisation_Function
32
33 # This dictionary includes the information required to prevent clusters being placed_
    ↪in the population if they are too similar to clusters in this memory_operator
34 memory_operator_information = {'Method': 'Off'}
35
36 # This switch tells the genetic algorithm the type of predation scheme they want to_
    ↪place on the genetic algoithm.
37 #predation_information = {'Predation Operator':'Off'}
38 #predation_information = {'Predation Operator':'Energy', 'mode': 'simple', 'round_
    ↪energy': 2}
39 #predation_information = {'Predation Operator':'Energy', 'mode': 'comprehensive',
    ↪'minimum_energy_diff': 0.025, 'type_of_comprehensive_scheme': 'energy'}
40 #predation_information = {'Predation Operator':'Energy', 'mode': 'comprehensive',
    ↪'minimum_energy_diff': 0.025, 'type_of_comprehensive_scheme': 'fitness'}
41 #predation_information = {'Predation Operator': 'IDCM', 'percentage_diff': 5.0}
42 predation_information = {'Predation Operator': 'SCM', 'SCM Scheme': 'T-SCM', 'rCut_
    ↪high': 3.2, 'rCut_low': 2.9, 'rCut_resolution': 0.05}
43
44 # This switch tells the genetic algorithm the type of fitness scheme they want to_
    ↪place on the genetic algoithm.
45 energy_fitness_function = {'function': 'exponential', 'alpha': 3.0}
46 #SCM_fitness_function = {'function': 'exponential', 'alpha': 1.0}
47 fitness_information = {'Fitness Operator': 'Energy', 'fitness_function': energy_
    ↪fitness_function}
48 #fitness_information = {'Fitness Operator': 'SCM + Energy', 'Use Predation Information
    ↪': True, 'SCM_fitness_contribution': 0.5, 'normalise_similarities': False, 'Dynamic_
    ↪Mode': False, 'energy_fitness_function': energy_fitness_function, 'SCM_fitness_
    ↪function': SCM_fitness_function}
49 #fitness_information = {'Fitness Operator': 'SCM + Energy', 'SCM Scheme': 'T-SCM',
    ↪'rCut_high': 3.2, 'rCut_low': 2.9, 'rCut_resolution': 0.05, 'SCM_fitness_
    ↪contribution': 0.5, 'normalise_similarities': False, 'Dynamic Mode': False, 'energy_
    ↪fitness_function': energy_fitness_function, 'SCM_fitness_function': SCM_fitness_
    ↪function}
50 #fitness_information = {'Fitness Operator': 'SCM + Energy', 'SCM Scheme': 'T-SCM',
    ↪'rCut': 3.05, 'SCM_fitness_contribution': 0.5, 'normalise_similarities': False,
    ↪'Dynamic Mode': False, 'energy_fitness_function': energy_fitness_function, 'SCM_
    ↪fitness_function': SCM_fitness_function}
51
52 # Variables required for the Recording_Cluster.py class/For recording the history as_
    ↪required of the genetic algorithm.
53 ga_recording_information = {}

```

(continues on next page)

(continued from previous page)

```
54 ga_recording_information['ga_recording_scheme'] = 'Limit_energy_height' # float('inf')
55 ga_recording_information['limit_number_of_clusters_recorded'] = 5 # float('inf')
56 ga_recording_information['limit_energy_height_of_clusters_recorded'] = 1.5 #eV
57 ga_recording_information['exclude_recording_cluster_screened_by_diversity_scheme'] = 
↳True
58 ga_recording_information['record_initial_population'] = True
59 ga_recording_information['saving_points_of_GA'] = [3,5]
60
61 # These are last technical points that the algorithm is designed in mind
62 force_replace_pop_clusters_with_offspring = True
63 user_initialised_population_folder = None
64 rounding_criteria = 10
65 print_details = False
66 no_of_cpus = 2
67 finish_algorithm_if_found_cluster_energy = None
68 total_length_of_running_time = 70.0
69
70 ''' ----- '''
71 # This will execute the genetic algorithm program
72 GA_Program(cluster_makeup=cluster_makeup,
73            pop_size=pop_size,
74            generations=generations,
75            no_offspring_per_generation=no_offspring_per_generation,
76            creating_offspring_mode=creating_offspring_mode,
77            crossover_type=crossover_type,
78            mutation_types=mutation_types,
79            chance_of_mutation=chance_of_mutation,
80            r_ij=r_ij,
81            vacuum_to_add_length=vacuum_to_add_length,
82            Minimisation_Function=Minimisation_Function,
83            surface_details=surface_details,
84            epoch_settings=epoch_settings,
85            cell_length=cell_length,
86            memory_operator_information=memory_operator_information,
87            predation_information=predation_information,
88            fitness_information=fitness_information,
89            ga_recording_information=ga_recording_information,
90            force_replace_pop_clusters_with_offspring=force_replace_pop_clusters_with_
↳offspring,
91            user_initialised_population_folder=user_initialised_population_folder,
92            rounding_criteria=rounding_criteria,
93            print_details=print_details,
94            no_of_cpus=no_of_cpus,
95            finish_algorithm_if_found_cluster_energy=finish_algorithm_if_found_cluster_energy,
96            total_length_of_running_time=total_length_of_running_time)
97 ''' ----- '''
```

Lets go through each part of the Run.py file one by one to understand how to use it.

1) The elemental makeup of the cluster

The first part of Run.py specifies the type of cluster you will be testing. Here, the makeup of the cluster is described using a dictionary in the format, {element: number of that element in the cluster, ...}. An example of this is shown below:

```
3 # This details the elemental and number of atom composition of cluster that the user_
  ↳ would like to investigate
4 cluster_makeup = {"Cu": 37}
```

2) Details if the cluster lies on a surface

This feature allows the user to include a surface to place a cluster upon. This feature is still being developed and does not currently work.

```
6 # Surface details
7 surface_details = None #{'surface': 'surface.xyz', 'place_cluster_where': 'center'}
```

3) The main details of the genetic algorithm

Here, the components of the genetic algorithm are described below:

- **pop_size** (*int*): The number of clusters in the population.
- **generations** (*int*): The number of generations that will be carried out by the genetic algorithm.
- **no_offspring_per_generation** (*int*): The number of offspring generated per generation.

It is recommended that for a particular test case that one try a few variations for pop_size and no_offspring_per_generation. From the literature, an pop_size = 30 or 40 and no_offspring_per_generation set to 0.8*pop_size is common.

An example of these parameters in Run.py is given below:

```
9 # These are the main variables of the genetic algorithm that with changes could_
  ↳ affect the results of the Genetic Algorithm.
10 pop_size = 20
11 generations = 2000
12 no_offspring_per_generation = 16
```

4) Details concerning the Mating and Mutation Procedure

The following set of parameters are focused on settings that involve the Mating and Mutation Procedures of the genetic algorithm. These are processes that affect how new offspring are created during the genetic algorithm. There are four sets of parameters involving the Mating and Mutation Procedures. Firstly, below is a parameter that affects both the Mating and Mutation Procedures:

- **creating_offspring_mode** (*str*): This indicates how you want these procedures to work when making an offspring. There are two options:
 - "Either_Mating_and_Mutation" - the genetic algorithm will perform **either** a mating or mutation procedure to obtain the offspring.
 - "Both_Mating_and_Mutation" - the genetic algorithm will perform **both** a mating and mutation procedure to obtain the offspring. Here, the mating scheme will always occur, however there is only a chance that the mutation scheme will occur.

- **crossover_type** (*str*): The mating method will use the spatial information of two parent clusters to create a new cluster from the two of them. This variable determines which mating procedure the genetic algorithm will perform.
 - "CAS_weighted": *Deavon and Ho Weighted Cut and Splice Method* - The cut and splice method, weighted by fitness of parents.
 - "CAS_half": *Deavon and Ho Half Cut and Splice Method* - The cut and splice method, where both parents are in half (recommended from experience).
 - "CAS_random": *Deavon and Ho Random Cut and Splice Method* - The cut and splice method, where one parent is cut a random percent $x\%$, while the other parent is cut by $(100-x)\%$. The value of x changes each time it is used to a random number between 0% and 100% (recommended for LJ98 clusters).
 - "CAS_custom_XX": *Deavon and Ho Custom Cut and Splice Method* - The cut and splice method, where one parent cut a set percent $XX\%$ and the other parent is cut by $(100-XX)\%$. Here, XX is a value that is set by the user. To use this, set `crossType = CAS_custom_XX`, where XX is a float of your choice between 0 and 100.
- **mutation_types** (*[[str,float],...]*): The mutation method will change the structure of a cluster to give a new cluster as a result. The type of mutation method the user would like to use. This can be one of the following:
 - "random": This will completely erase the previous cluster and generate a new cluster, where atoms have been placed randomly within the cluster
 - "random_XX": This will randomly place XX percent of the atoms in the cluster to new positions in the cluster. (Make this more clear what is going on later)
 - "move": This will move all the atoms in the cluster from their original positions by a maximum default distance equal to $r_{ij} * 0.5 \text{ \AA}$.
 - "move_XX": This will move all the atoms in the cluster from their original positions by a maximum distance $XX \text{ \AA}$.
 - "homotop": This will swap the positions of two atoms that are of a different element. This mutation method can not be used for monometallic clusters.

It is possible for more than one mutation method to be used. For this reason, the format for `mutTypes` is *[[Type of Mutation,chance of mutation],...]*. Here, all the chances of mutations should add up to 1.0. For example, "`mutTypes = [[random, 0.45], [move, 0.225], [random_33.33, 0.325]]`" is acceptable as all the chances of mutations add up to 1.0 ($0.45 + 0.225 + 0.325 = 1.0$).
- **chance_of_mutation** (*float*): The chance that a mutation will occur. How the genetic algorithm uses this variable depends on the input for `creating_offspring_mode`. If:
 - `creating_offspring_mode = "Either_Mating_and_Mutation"` - `chance_of_mutation` is the chance that a mutation will occur rather than a mating scheme.
 - `creating_offspring_mode = "Both_Mating_and_Mutation"` - `chance_of_mutation` is the chance that a mutation will occur. The mating scheme will always occur when `creating_offspring_mode = "Both_Mating_and_Mutation"`.

An example of these parameters used in the `Run.py` file is given below:

```
14 # These setting indicate how offspring should be made using the Mating and Mutation_
    ↪ Procedures
15 creating_offspring_mode = "Either_Mating_and_Mutation"
16 crossover_type = "CAS_weighted"
17 mutation_types = [['random', 1.0]]
18 chance_of_mutation = 0.1
```

5) Epoch Settings

It is possible to include a epoch in this version of the genetic algorithm. An epoch is a feature that allows the population to be reset with new, randomly generated clusters. See *Using Epoch Methods* for more information on epoch methods, including the various types of epoches and settings.

An example of the epoch parameters used in the Run.py file is given below:

```
20 # This parameter will tell the Organisms program if an epoch is desired, and how the_
    ↪user would like to proceed.
21 epoch_settings = {'epoch mode': 'same population', 'max repeat': 5}
```

6) Other Details

There are three other variables which are important to include in your Run.py file. These are:

- **r_ij (float)**: This is the maximum bond distance that we would expect in this cluster. This parameter is used when clusters are created using either or both the mating or mutation schemes. This parameter is used to determine if a cluster has stayed in one piece after the local minimisation, as it is possible for the cluster to break into multiple pieces. This should be a reasonable distance, but not excessively large. For example, for Au, which has a FCC lattice constant of 4.078 . Therefore it has a first nearest neighbour of 2.884 and a second nearest neighbour of 4.078 . Therefore r_ij should be set to some value between 2.884 and 4.078 . For example, r_ij = 3.5 or r_ij = 4.0 would probably be appropriate, however I have been able to get away with r_ij = 3.0 . r_ij is given in .
- **cell_length (float)**: If you are wanting to create randomly generated clusters, either at the start of the genetic algorithm or using the 'random' mutation method, then you will want to specify the length of the box that you want to add atoms to. boxtoplacelength is the length of this box. Don't make this too big, or else it is likely atoms will be too far apart and a cluster will be broken into multiple pieces. cell_length is given in .
- **vacuum_to_add_length (float)**: The length of vacuum added around the cluster. This variable is only used to aesthetics reasons. How the genetic algorithm records clusters for databases is to take the *locally optimised* cluster and measure its radius. A new cell is then created with cell length that is twice the radius of the cluster. The cluster is then placed in this new cell, and a vacuum of vacuum_to_add_length is then added to this new cell. The reason for taking the radius of the cluster to make a cell is so that no matter if and how you rotate the cluster for post processing, you can be assured that the cluster will always have a vacuum of at least vacuum_to_add_length . This is important especially if you reoptimise the cluster with periodic potential such as VASP where it is vital that a minimum vacuum is given to prevent reoptimisation issues from occurring. vacuum_to_add_length is given in .

An example of these parameters used in the Run.py file is given below:

```
23 # These are variables used by the algorithm to make and place clusters in.
24 r_ij = 3.4
25 cell_length = r_ij * (sum([float(noAtoms) for noAtoms in list(cluster_makeup.
    ↪values())]) ** (1.0/3.0))
26 vacuum_to_add_length = 10.0
```

7) Minimisation Scheme

This component of Run.py focuses on the function/method that the genetic algorithm uses for performing local minimisations. This is used by the genetic algorithm as a def type (i.e. as a function). This means that, rather than a variable being passed into the algorithm, a function is passed into the algorithm.

One can write this function into the Run.py file, however it is usually easier and nicer to view this function in a different python file. I typically call this something like RunMinimisation.py, and the function in this file is called Minimisation_Function. Minimisation_Function will contain the algorithm for performing a local optimisation.

Because of the flexibility, it is possible to use any type of calculator from ASE, ASAP, GWAP, LAMMPS, etc. It is even possible for the user to design this to use with non-python user-interface based local optimisers, such as VASP or Quantum Espresso!

To see an example of how to write Minimisation_Function, see [Writing a Local Minimisation Function for the Genetic Algorithm](#).

The algorithm is imported into Run.py as follows:

```
28 # The RunMinimisation.py algorithm is one set by the user. It contain the def_
    ↳Minimisation_Function
29 # That is used for local optimisations. This can be written in whatever way the user_
    ↳wants to perform
30 # the local optimisations. This is meant to be as free as possible.
31 from RunMinimisation import Minimisation_Function
```

8) The Memory Operator

This operator is designed to prevent clusters from being in the population that resemble any cluster in this memory operator in some way. This operator uses the SCM to determine how structurally similar cluster are. See [Using the Memory Operator](#) for more information on how to use the memory operator. An example of how the memory operator is written in the Run.py file is shown below.

```
33 # This dictionary includes the information required to prevent clusters being placed_
    ↳in the population if they are too similar to clusters in this memory_operator
34 memory_operator_information = {'Method': 'Off'}
```

9) Predation Operators

This component of Run.py specifies all the information concerning the predation operator. You can see more about how the predation operators works at [Using Predation Operators with the Genetic Algorithm](#).

In terms of the Run.py file, there is only one variable that we need to deal with, predation_information. This variable is a dictionary type, {}, which holds all the information that would be needed for the predation operator that the user wishes to use. For example:

```
predation_information = {'Predation_Switch': 'SCM', 'SCM Scheme': 'T-SCM', 'rCut_high
    ↳': 3.2, 'rCut_low': 2.9, 'rCut_resolution': 0.05}
```

There are a variety of predation operators that are inbuilt currently into the genetic algorithm. You can find out more about what they do, and how to use them in your Run.py file, at [Using Predation Operators with the Genetic Algorithm](#).

Below are some other examples of the inputs required for the other types of predation operators available. **Please see [Using Predation Operators with the Genetic Algorithm](#) before using these predation operators.**

```
36 # This switch tells the genetic algorithm the type of predation scheme they want to_
    ↪ place on the genetic algorithm.
37 #predation_information = {'Predation Operator':'Off'}
38 #predation_information = {'Predation Operator':'Energy', 'mode': 'simple', 'round_
    ↪ energy': 2}
39 #predation_information = {'Predation Operator':'Energy', 'mode': 'comprehensive',
    ↪ 'minimum_energy_diff': 0.025, 'type_of_comprehensive_scheme': 'energy'}
40 #predation_information = {'Predation Operator':'Energy', 'mode': 'comprehensive',
    ↪ 'minimum_energy_diff': 0.025, 'type_of_comprehensive_scheme': 'fitness'}
41 #predation_information = {'Predation Operator': 'IDCM', 'percentage_diff': 5.0}
42 predation_information = {'Predation Operator': 'SCM', 'SCM Scheme': 'T-SCM', 'rCut_
    ↪ high': 3.2, 'rCut_low': 2.9, 'rCut_resolution': 0.05}
```

10) Fitness Operators

This component of Run.py specified all the information required by the fitness operators. You can find more information about how the fitness operators works at *Using Fitness Operators with the Genetic Algorithm*.

In the Run.py file, all the setting for the fitness operator are contained in the dictionary called fitness_information. For example:

```
fitness_information = {'Fitness Operator': 'Energy', 'fitness_function': energy_
    ↪ fitness_function}
```

There are a variety of fitness scheme available to be used in this implementation of the genetic algorithm. You can find all the information about all the available fitness schemes in *Using Fitness Operators with the Genetic Algorithm*.

An example of how the fitness scheme is written in the Run.py file is shown below. **Please see *Using Fitness Operators with the Genetic Algorithm* before using these fitness operators.**

```
44 # This switch tells the genetic algorithm the type of fitness scheme they want to_
    ↪ place on the genetic algorithm.
45 energy_fitness_function = {'function': 'exponential', 'alpha': 3.0}
46 #SCM_fitness_function = {'function': 'exponential', 'alpha': 1.0}
47 fitness_information = {'Fitness Operator': 'Energy', 'fitness_function': energy_
    ↪ fitness_function}
48 #fitness_information = {'Fitness Operator': 'SCM + Energy', 'Use Predation Information
    ↪ ': True, 'SCM_fitness_contribution': 0.5, 'normalise_similarities': False, 'Dynamic_
    ↪ Mode': False, 'energy_fitness_function': energy_fitness_function, 'SCM_fitness_
    ↪ function': SCM_fitness_function}
49 #fitness_information = {'Fitness Operator': 'SCM + Energy', 'SCM Scheme': 'T-SCM',
    ↪ 'rCut_high': 3.2, 'rCut_low': 2.9, 'rCut_resolution': 0.05, 'SCM_fitness_
    ↪ contribution': 0.5, 'normalise_similarities': False, 'Dynamic Mode': False, 'energy_
    ↪ fitness_function': energy_fitness_function, 'SCM_fitness_function': SCM_fitness_
    ↪ function}
50 #fitness_information = {'Fitness Operator': 'SCM + Energy', 'SCM Scheme': 'T-SCM',
    ↪ 'rCut': 3.05, 'SCM_fitness_contribution': 0.5, 'normalise_similarities': False,
    ↪ 'Dynamic Mode': False, 'energy_fitness_function': energy_fitness_function, 'SCM_
    ↪ fitness_function': SCM_fitness_function}
```

11) Recording Clusters from the Genetic Algorithm

This input in the Run.py file indicates how the user would like to record clusters that are created during the genetic algorithm. The information is contained in the dictionary called `ga_recording_information`. There are six parameters that the user can set. These are:

- `ga_recording_scheme`
- `limit_number_of_clusters_recorded`
- `limit_energy_height_of_clusters_recorded`
- `exclude_recording_cluster_screened_by_diversity_scheme`
- `saving_points_of_GA`
- `record_initial_population`

Not all of these parameters need to be entered. If the user does not enter in any of these parameters, the genetic algorithm will not keep a record of the clusters that were obtained.

More information on how to record clusters made during the genetic algorithm can be found at [Recording Clusters From The Genetic Algorithm](#).

An example of the '`ga_recording_information`' variable in the Run.py file is shown below. **Please see [Recording Clusters From The Genetic Algorithm](#) before using this feature to record clusters obtained during the genetic algorithm.**

```
52 # Variables required for the Recording_Cluster.py class/For recording the history as_  
↪required of the genetic algorithm.  
53 ga_recording_information = {}  
54 ga_recording_information['ga_recording_scheme'] = 'Limit_energy_height' # float('inf')  
55 ga_recording_information['limit_number_of_clusters_recorded'] = 5 # float('inf')  
56 ga_recording_information['limit_energy_height_of_clusters_recorded'] = 1.5 #eV  
57 ga_recording_information['exclude_recording_cluster_screened_by_diversity_scheme'] =_  
↪True  
58 ga_recording_information['record_initial_population'] = True  
59 ga_recording_information['saving_points_of_GA'] = [3,5]
```

12) Other details of the Genetic algorithm

These last set of parameters are important, but there is no good appropriate place to put them in the Run.py file. These last parameters are:

- **`force_replace_pop_clusters_with_offspring` (*bool*):** In the genetic algorithm, the predation operator may find that the an offspring is “identical” to a cluster in the population, but that offspring is more fit than the cluster in the population. In this case, the genetic algorithm can replace the less fit cluster in the population with the “identical” more fit offspring. Set this variable to `True` if you want this to happen. Set this variable to `False` if you don’t want this to happen. Default: `True`.
- **`user_initilised_population_folder` (*str*):** This is the name, or the path to, the folder holding the initialised population that you would like to use instead of the program creating a set of randomly generated clusters. If you do not have, or do not want to use, an initialised population, set this to `None` or `''`.
- **`rounding_criteria` (*int*):** This is the round that will be enforced on the value of the cluster energy. Default: 2
- **`print_details` (*bool*):** Will print the details of the genetic algorithm, like a verbose.
- **`no_of_cpus` (*int*):** This is the number of cpus that you would like the algorithm to run on. These extra cores will be used to create the offspring as well as used by the predation and fitness operators if beneficial to use extra cores for the chosen operators.

- **finish_algorithm_if_found_cluster_energy** (*dict.*): This parameter will stop the algorithm if the desired global minimum is found. This parameter is to be used if the user would like to test the performance of the algorithm and knows beforehand what the energy of the global minimum is. This parameter is set as a dictionary as two parameters. **'cluster energy'** is a float that states the energy of the global minimum. **'round'** is an interger that you want to set to the same rounding that you gave for the 'cluster energy' input. This will round the energy of clusters made, and compare this energy to your 'cluster energy' input. An example of this for Au38 using Cleri Gupta parameters are `finish_algorithm_if_found_cluster_energy = {'cluster energy': -130.54, 'round': 2}`. If you are not testing the performance of the algorithm, or dont know the global minimum of the cluster you are testing, set `finish_algorithm_if_found_cluster_energy = None`. Default: None
- **total_length_of_running_time** (*int*): This is the maximum amount of time (in hours) that the algorithm is allow to run for. This variable is useful if you are running on a remote computer system like slurm that finishes once a certain time limit is reached. To prevent the algorithm from being incorrectly cancelled when running, set this value to a time limit less than your maximum time limit on slurm. I have been setting this to the slurm job time minus 2 hours. For example, if the genetic algorithm is submitted to slurm for 72 hours, set `total_length_of_running_time=70.0`. While this algorithm is designed to be able to be restarted even if the program is cancelled during a generation, it is best to prevent any issues from occurring by using this variable to cancel the algorithm safety so that there are absolutely no issues when restarting the genetic algorithm. Is None is given, no time limit will be set. Default: None

An example of how they are written in the Run.py file are show below:

```
61 # These are last technical points that the algorithm is designed in mind
62 force_replace_pop_clusters_with_offspring = True
63 user_initialised_population_folder = None
64 rounding_criteria = 10
65 print_details = False
66 no_of_cpus = 2
67 finish_algorithm_if_found_cluster_energy = None
68 total_length_of_running_time = 70.0
```

The Genetic Algorithm!

You have got to the end of all the parameter setting stuff! Now on to the fun stuff! The next part of the Run.py file tells the genetic algorithm to run. This is written as follows in the Run.py:

```
71 # This will execute the genetic algorithm program
72 GA_Program(cluster_makeup=cluster_makeup,
73            pop_size=pop_size,
74            generations=generations,
75            no_offspring_per_generation=no_offspring_per_generation,
76            creating_offspring_mode=creating_offspring_mode,
77            crossover_type=crossover_type,
78            mutation_types=mutation_types,
79            chance_of_mutation=chance_of_mutation,
80            r_ij=r_ij,
81            vacuum_to_add_length=vacuum_to_add_length,
82            Minimisation_Function=Minimisation_Function,
83            surface_details=surface_details,
84            epoch_settings=epoch_settings,
85            cell_length=cell_length,
86            memory_operator_information=memory_operator_information,
87            predation_information=predation_information,
```

(continues on next page)

(continued from previous page)

```
88     fitness_information=fitness_information,
89     ga_recording_information=ga_recording_information,
90     force_replace_pop_clusters_with_offspring=force_replace_pop_clusters_with_
    ↪offspring,
91     user_initialised_population_folder=user_initialised_population_folder,
92     rounding_criteria=rounding_criteria,
93     print_details=print_details,
94     no_of_cpus=no_of_cpus,
95     finish_algorithm_if_found_cluster_energy=finish_algorithm_if_found_cluster_energy,
96     total_length_of_running_time=total_length_of_running_time)
```

5.5 Examples of Running the Organisms Program with *Run.py*

Provided in the Organisms Github repository are examples of *Run.py* (and *RunMinimisation.py*, see *RunMinimisation.py - Writing a Local Minimisation Function for the Genetic Algorithm* for more information about this file) scripts that you can try out. Find these various examples at <https://github.com/GardenGroupUO/Organisms/tree/main/Examples>.

We have also developed a Jupyter notebook with some examples of various *Run.py* (and *RunMinimisation.py*) that you can play with and muck around with. The Github repository for this Jupyter notebook can be found at https://github.com/GardenGroupUO/Organisms_Jupyter_Examples.

Along with this Jupyter notebook, we have also implemented this Jupyter notebook into Binder. Binder (<https://mybinder.org/>) is an interactive online platform that allows you to use Jupyter notebooks on a web browser without having to set up anything. It does all the setting up on a virtual computer for you. If you want to play around with the Organisms program before you download it on your computer or if you need help when things go wrong using Organisms on your computer, Binder+Jupyter is the best way to do this. **It is recommended that you try out the Organisms program on Binder if you are interested or intending on using the Organisms program.**

The Binder webpage can be accessed by clicking the binder button below:

²² This will load a Binder page that will allow you to play about with the Organisms program interactively in Binder. This Binder page may load quickly, or it may take 1 to 2 minutes to load. Don't refresh the page as Binder takes a good amount of time to load. Get a coffee or a cup of tea while you wait.

Once this is done you will see a Jupyter notebook that you can interact with. Mess around with it as much as you want!

5.6 *RunMinimisation.py* - Writing a Local Minimisation Function for the Genetic Algorithm

In this article, we will look at how to write the local optimisation method for the genetic algorithm.

²² https://mybinder.org/v2/gh/GardenGroupUO/Organisms_Jupyter_Examples/main?urlpath=lab

5.6.1 What is the Minimisation_Function

The `Minimisation_Function` is a definition that tells the genetic algorithm how to perform a local optimisation. This is used by the genetic algorithm as a `def` (i.e. as a function). This means that, rather than a variable being passed into the algorithm, a function is passed into the algorithm.

The implementation of the local minimisation process in this genetic algorithm program has been designed to be as free as possible, so that the user can use whatever local optimisation algorithm or program they want to use. In general, this algorithm will import a cluster in an ASE format from the genetic algorithm. The user can locally optimise it before sending it back to the genetic algorithm again in the ASE format.

Because of the flexibility, it is possible to use any type of calculator from ASE, ASAP, GWAP, LAMMPS, etc. It is even possible for the user to design this to use with non-python user-interface based local optimisers, such as VASP or Quantum Espresso! See at the bottom of this page, *How to write the Minimisation_Function for non-ASE Implemented Calculator*.

In the following documentation we will describe how the `Minimisation_Function` method is designed in a `RunMinimisation.py` file, and how you can make your own. Examples of `RunMinimisation.py` files can be found in github.com/GardenGroupUO/Organisms²³ in the directory path `Examples/Set_of_RunMinimisation_Files` (this should be found in github.com/GardenGroupUO/Organisms/tree/main/Examples/Set_of_RunMinimisation_Files²⁴).

5.6.2 Where to write the Minimisation_Function

The `Minimisation_Function` can be written into the `Run.py` file. However, as a personal preference and also to make the code cleaner to read, write and use, I put it into another python file. This file I have called `RunMinimisation.py`. This does not need to be the name of this file. For example, I have named this file `RunMinimisation_AuPd.py` when I wanted to keep a record that this minimisation python file contained the Gupta parameters and code for locally minimising a cluster using the Gupta potential for a cluster containing Au and Pd atoms.

Furthermore, the `def Minimisation_Function` does not even need to be called `Minimisation_Function`. It could be called `TheGuptaFunction`, `the_local_minimisation_function`, or `The_Electric_Eel_Function`. Again, I have just always called it `Minimisation_Function` for simplicity and for ease when using different `Run.py` files with different `Minimisation_Function` codes.

However, it is important that this code is referenced somehow in the `Run.py` program. The algorithm is imported into `Run.py` as follows (You can also see this in *Run.py - Using the Genetic Algorithm: Minimisation Scheme*):

```
# The RunMinimisation.py algorithm is one set by the user. It contain the def_
↳Minimisation_Function
# That is used for local optimisations. This can be written in whatever way the user_
↳wants to perform
# the local optimisations. This is meant to be as free as possible.
from RunMinimisation import Minimisation_Function
```

where, in the above code, `RunMinimisation` is the name of the file the local minimisation code is found in (This file is called `RunMinimisation.py`), and `Minimisation_Function` is the name of the function that is found in the `RunMinimisation.py`. If you do it like this, make sure that your `RunMinimisation.py` file is in the same folder as your `Run.py` file.

²³ <https://github.com/GardenGroupUO/Organisms>

²⁴ https://github.com/GardenGroupUO/Organisms/tree/main/Examples/Set_of_RunMinimisation_Files

5.6.3 How to write the Minimisation_Function

The `Minimisation_Function` must be written with the following requirements:

- **cluster** (*ase.Atoms*): This is the unoptimised version of the cluster.
- **collection** (*str*): This indicates if the cluster is apart of which instance of `Population` or `Offspring_pool`.
- **cluster_name** (*int*): This is the name of the cluster to be optimised. This should be the number of the cluster that was made during this genetic algorithm run.

NOTE: The **collection** and **cluster_name** variables do not need to be used in your `RunMinimisation.py` script if you are using an ASE or ASE implemented calculator and local optimiser. This information may be useful if you want the cluster to be locally optimised using an external program that can not be easily used with python (for example with VASP, see *How to write the Minimisation_Function for non-ASE Implemented Calculator*).

returns:

- **cluster** (*ase.Atoms*) - This is the optimised version of the cluster.
- **converged** (*bool*) - This boolean indicates if the local optimisation converged (`True` for converged, `False` for did not converge).
- **Info** (*dict*) - This sends information back to the genetic algorithm about how the local minimisation ran.

An example of a `RunMinimisation.py` file for a Gupta potential involving only Cu atoms is given below:

Listing 2: `RunMinimisation.py`

```
1  '''
2  RunMinimisation.py, GRW, 8/6/17
3
4  This python program is designed to input the POSCAR file and use it
5  with Atomic Simulation Environment (ASE) to minimise the given structure
6  using an empirical potential.
7
8  The program will take the output translate it into a OUTCAR file which
9  the bpga will use.
10
11  Required inputs: BeforeOpt
12  Required to outputs: AfterOpt.traj, INFO.txt
13  Other outputs: Trajectory file.
14
15  '''
16  import sys
17  import time
18  from asap3.Internal.BuiltinPotentials import Gupta
19  from ase.optimize import FIRE
20
21  def Minimisation_Function(cluster,collection,cluster_name):
22      cluster.pbc = False
23      #####
24      ↪#####
25      # Perform the local optimisation method on the cluster.
26      # Parameter sequence: [p, q, a, xi, r0]
27      Gupta_parameters = {'Cu': [10.960, 2.2780, 0.0855, 1.224, 2.556]}
28      cluster.set_calculator(Gupta(Gupta_parameters, cutoff=1000, debug=False))
29      dyn = FIRE(cluster,logfile=None)
30      startTime = time.time(); converged = False
31      try:
32          dyn.run(fmax=0.01,steps=5000)
```

(continues on next page)

(continued from previous page)

```

32     converged = dyn.converged()
33     if not converged:
34         errorMessage = 'The optimisation of cluster ' + str(cluster_name) + ' did_
↳not optimise completely.'
35         print(errorMessage, file=sys.stderr)
36         print(errorMessage)
37     except Exception:
38         print('Local Optimiser Failed for some reason.')
39     endTime = time.time()
40     #####
↳#####
41     # Write information about the algorithm
42     Info = {}
43     Info["INFO.txt"] = ''
44     Info["INFO.txt"] += ("No of Force Calls: " + str(dyn.get_number_of_steps()) + '\n
↳')
45     Info["INFO.txt"] += ("Time (s): " + str(endTime - startTime) + '\n')
46     #Info["INFO.txt"] += ("Cluster converged?: " + str(dyn.converged()) + '\n')
47     #####
↳#####
48     return cluster, converged, Info

```

We will explain the components of this example below:

Importing external code

To begin, you will need to import all the external files that you will need so that you have the descriptor of the potential you want to use, and the local optimiser that you would like to use. In this example, the Gupta potential is used as the descriptor for the potential, while FIRE is the local optimiser that will be used to locally optimise the cluster.

```

16 import sys
17 import time
18 from asap3.Internal.BuiltinPotentials import Gupta
19 from ase.optimize import FIRE

```

Preparing the cluster

First, it is usually a good idea to tell ase if you want the calculator to calculate the cluster with periodic boundary conditions pbc or not. In the case of the Gupta potential, we will include the line `cluster.pbc = False` to make sure that there are no boundary conditions on upon the cluster, since we do not want this and the Gupta potential does not need this turned on. For your potential, you may want to include this, or not.

```

22 cluster.pbc = False

```

Preparing the Potential, and setting up the local optimiser.

We would like to set up the parameters needed for the descriptor of the potential, attach the descriptor as a calculator to the `cluster`, and set up the local optimiser. In this example, Gupta is called a calculator. It contains a description of the Gupta potential that can be used to calculate the energy of `cluster`. We do this in the line `cluster.set_calculator(Gupta(Gupta_parameters, cutoff=1000, debug=True))`. For more information on how this works, see [Tutorial on Using Calculators in ASE²⁵](#).

The last line, `dyn = FIRE(cluster)`, sets up the local optimiser, FIRE, to be used to locally minimise the cluster `cluster` (see [Tutorial on Structure Optimization in ASE²⁶](#)).

See below for an example:

```
24 # Perform the local optimisation method on the cluster.
25 # Parameter sequence: [p, q, a, xi, r0]
26 Gupta_parameters = {'Cu': [10.960, 2.2780, 0.0855, 1.224, 2.556]}
27 cluster.set_calculator(Gupta(Gupta_parameters, cutoff=1000, debug=False))
28 dyn = FIRE(cluster, logfile=None)
```

Executing the local optimiser

We would like to now get the definition to run a local optimisation. This is done by performing `dyn.run(fmax=0.01, steps=5000)`. However I have found that if something breaks for some reason during the optimisation, this can completely stop the genetic algorithm in its tracks, and cause it to finish with a fatal error. You may want this to happen so that you can address issues when they arise, but sometimes it is hard to continue to work when it keeps happening. If you would like, you can make sure the genetic algorithm does not fail entirely by adding a Error Handling block, as shown in the example below:

```
29 startTime = time.time(); converged = False
30 try:
31     dyn.run(fmax=0.01, steps=5000)
32     converged = dyn.converged()
33     if not converged:
34         errorMessage = 'The optimisation of cluster ' + str(cluster_name) + ' did not_
↳optimise completely.'
35         print(errorMessage, file=sys.stderr)
36         print(errorMessage)
37 except Exception:
38     print('Local Optimiser Failed for some reason.')
39 endTime = time.time()
```

You can also see that I have placed an if statement to determine if the local optimisation actually converged. I have found that it is useful to include a way of noting if the optimisation was able to converge or not. See more about [How to perform a local optimisation in ASE here²⁷](#), or refer to the manual of the local optimiser you are using for more information on how to do this.

²⁵ <https://wiki.fysik.dtu.dk/ase/ase/calculators/calculators.html>

²⁶ <https://wiki.fysik.dtu.dk/ase/ase/optimize.html>

²⁷ <https://wiki.fysik.dtu.dk/ase/ase/optimize.html>

Writing the Info variable

I have found it useful for benchmarking in the past to give back to the genetic algorithm information on Force calls and times, and whether the optimisation converged or not. For this we include it in the INFO.txt file for that optimisation as shown below. FEATURE IS CURRENTLY DISABLED.

```
41 # Write information about the algorithm
42 Info = {}
43 Info["INFO.txt"] = ''
44 Info["INFO.txt"] += ("No of Force Calls: " + str(dyn.get_number_of_steps()) + '\n')
45 Info["INFO.txt"] += ("Time (s): " + str(endTime - startTime) + '\n')
46 #Info["INFO.txt"] += ("Cluster converged?: " + str(dyn.converged()) + '\n')
```

Return the Optimised Cluster and Info

Remember to return the `cluster` and `Info` to the genetic algorithm so that it can use this information, as well as the optimiser `cluster`, to proceed to explore the potential energy surface of the cluster you wish to explore.

```
48 return cluster, converged, Info
```

5.6.4 How to write the Minimisation_Function for a ASE Implemented Calculator

If the descriptor for the potential you would like to use is implemented in ASE, it is very easy to implement this into your `Minimisation_Function` definition. You can use the example of `RunMinimisation.py` above, where the only component you need to change is the `set_calculator` function used by `Opt_cluster`. This is the bit of the code above that looks like this:

```
Gupta_parameters = {'Cu': [10.960, 2.2780, 0.0855, 1.224, 2.556]}
cluster.set_calculator(Gupta(Gupta_parameters, cutoff=1000, debug=True))
```

Instead of this, you can include all the parameters that you need for your potential before the `set_calculator` line. For example:

```
Potential_Parameters = ...
cluster.set_calculator(Potential(Potential_Parameters))
```

Where `Potential` is the potential you would like to use, and `Potential_Parameters` are all the parameters that `Potential` needs to work. Please consult the manual of the potential you would like to use to learn how to use that potential.

5.6.5 How to write the Minimisation_Function for non-ASE Implemented Calculator

In the previous section of this page we have been performing a local optimisation using ASE implemented calculators. However, you may want to use a calculator to locally optimise your cluster. This may only be possible by allowing the program you wish to use to itself completely locally optimise the cluster. For example, VASP contains its own local optimisation functions since it is not implemented in ASE. This is no issue for us! We just need to be careful to implement the local optimisation using your own program, and make sure that the `RunMinimisation.py` file is constructed as follows:

Input into `Minimisation_Function`:

- **cluster** (*ASE.Atoms*): This is the unoptimised version of the cluster.
- **collection** (*str*): This indicates if the cluster is apart of the population or an offspring.

- **cluster_name** (*int.*): This is the name of the cluster to be optimised. This should be the number of the cluster that was made during this genetic algorithm run.

returns:

- **cluster** (*ASE.Atoms*) - This is now the optimised version of the cluster.
- **converged** (*bool.*) - This boolean indicates if the local optimisation converged (True for converged, False for did not converge).
- **Info** (*dict.*) - This sends information back to the genetic algorithm about how the local minimisation ran.

A general script for locally optimising however you want to is given below:

Listing 3: RunMinimisation_General.py

```
1 import time
2 from copy import deepcopy
3 from asap3.Internal.BuiltinPotentials import Gupta
4 from ase.optimize import FIRE
5 from subprocess import Popen
6
7 def Minimisation_Function(cluster, collection, cluster_name):
8     #####
9     ↪#####
10    cluster.pbc = What_you_want # make sure that the periodic boundry conditions are_
11    ↪set off
12    #####
13    ↪#####
14    # Perform the local optimisation method on the cluster.
15    startTime = time.time();
16    #Pre-calculation
17    try:
18        Popen(['run', 'external', 'program'])
19    except Exception:
20        pass
21    #Post-calculation
22    endTime = time.time()
23    #####
24    ↪#####
25    # Write information about the algorithm
26    Info = {}
27    Info["INFO.txt"] = ''
28    Info["INFO.txt"] += ("No of Force Calls: " + str(number_of_force_calls) + '\n')
29    Info["INFO.txt"] += ("Time (s): " + str(endTime - startTime) + '\n')
30    Info["INFO.txt"] += ("Cluster converged?: " + str(dyn.converged()) + '\n')
31    #####
32    ↪#####
33    return cluster, converged, Info
```

For example, this is how you might want to set your RunMinimisation.py script for locally optimising using VASP.

Listing 4: RunMinimisation_VASP.py

```
1 import os, time
2 from ase.io import write as ase_write
3 from ase.io import read as ase_read
4 from shutil import copyfile
5 from subprocess import Popen
6
```

(continues on next page)

(continued from previous page)

```

7 def Minimisation_Function(cluster, collection, cluster_name):
8     #####
9     ↪#####
10    cluster.pbc = True # make sure that the periodic boundry conditions are set off
11    #####
12    ↪#####
13    # Perform the local optimisation method on the cluster.
14    original_path = os.getcwd()
15    offspring_name = str(cluster_name)
16    clusters_to_make_name = 'clusters_for_VASP'
17    if not os.path.exists(clusters_to_make_name):
18        os.mkdir(clusters_to_make_name)
19    copyfile('VASP_Files/INCAR', clusters_to_make_name+'/'+offspring_name+'/INCAR')
20    copyfile('VASP_Files/POTCAR', clusters_to_make_name+'/'+offspring_name+'/POTCAR')
21    copyfile('VASP_Files/KPOINTS', clusters_to_make_name+'/'+offspring_name+'/KPOINTS')
22    os.chdir(clusters_to_make_name+'/'+offspring_name)
23    ase_write(cluster, 'POSCAR', 'vasp')
24    startTime = time.time();
25    try:
26        Popen(['srun', 'vasp'])
27    except Exception:
28        pass
29    endTime = time.time()
30    cluster = ase_read('OUTCAR')
31    os.chdir(original_path)
32    #####
33    ↪#####
34    # Write information about the algorithm
35    Info = {}
36    Info["INFO.txt"] = ''
37    #Info["INFO.txt"] += ("No of Force Calls: " + str(dyn.get_number_of_steps()) + '\n
38    ↪')
39    Info["INFO.txt"] += ("Time (s): " + str(endTime - startTime) + '\n')
40    #Info["INFO.txt"] += ("Cluster converged?: " + str(dyn.converged()) + '\n')
41    #####
42    ↪#####
43    return cluster, converged, Info

```

5.7 MakeTrials.py - Creating Multiple, Repeated Genetic Algorithm Trials

Typically, it is common that one will not run just one genetic algorithm run, but multiple genetic algorithm runs with the same parameters. This can a pain to set up all the files to make the multiple runs and run them all. It can also be very hard to analyse all the data together, since there is just so much! Therefore, we have developed a set of scripts to make this experience less painful for the user.

In this article, we will look at how to use some of the tools that have been developed to create and run many genetic algorithm runs on slurm. Slurm (Slurm Workload Manager) is a Linux resource management system for running a computer cluster system. See [Slurm Workload Manager](https://slurm.schedmd.com/documentation.html)²⁸ for more information about slurm.

In the next article (*Helpful Programs for Gathering data and Post-processing Data*) we describe a set of scripts to analyse the data from the multiple genetic algorithms.

²⁸ <https://slurm.schedmd.com/documentation.html>

5.7.1 What to make sure is done before running the MakeTrials.py program.

If you installed Organisms through pip3

If you installed the Organisms program with pip3, these scripts will be installed in your bin. You do not need to add anything into your ~/.bashrc. You are all good to go.

If you performed a Manual installation

If you have manually added this program to your computer (such as cloning this program from Github), you will need to make sure that you have included the Helpful_Programs folder into your PATH in your ~/.bashrc file. All of these program can be found in the Helpful_Programs folder. To execute some of these programs from the Helpful_Programs folder, you must include the following in your ~/.bashrc:

```
export PATH_TO_GA="<Path_to_Organisms>"
```

where *<Path_to_Organisms>*" is the path to get to the genetic algorithm program. Also include somewhere before this in your ~/.bashrc:

```
export PATH="$PATH_TO_GA/Organisms/Helpful_Programs:$PATH"
```

See more about this in *Installation of the Genetic Algorithm*.

5.7.2 How does MakeTrials.py work?

MakeTrials.py is a script which uses the MakeTrialsProgram class in Organisms.SubsidiaryPrograms.MakeTrialsProgram (found in Organisms/SubsidiaryPrograms/MakeTrialsProgram.py) to make all the files that one would need to make to perform a set of repeated the genetic algorithm upon a cluster system. This will create lots of the same genetic algorithm files (Run.py and RunMinimisation.py) and put them into folders called Trials.

You can find another example of a MakeTrials.py file and other associated files at github.com/GardenGroupUO/Organisms²⁹ under Examples\CreateSets.

5.7.3 Setting up MakeTrials.py

MakeTrials.py is designed to make all the trials desired for a specific cluster system. This is designed to be as customisable as possible. A typical MakeTrials.py script will look as follows:

Listing 5: MakeTrials.py

```
1 from Organisms import MakeTrialsProgram
2
3 # This details the elemental and number of atom composition of cluster that the user_
  ↳ would like to investigate
4 cluster_makeup = {"Cu": 37}
5
6 # Surface details
7 surface_details = None #{'surface': 'surface.xyz', 'place_cluster_where': 'center'}
8
9 # These are the main variables of the genetic algorithm that with changes could_
  ↳ affect the results of the Genetic Algorithm.
```

(continues on next page)

²⁹ <https://github.com/GardenGroupUO/Organisms>

(continued from previous page)

```

10 pop_size = 20
11 generations = 2000
12 no_offspring_per_generation = 16
13
14 # These setting indicate how offspring should be made using the Mating and Mutation_
  ↳Proceedures
15 creating_offspring_mode = "Either_Mating_and_Mutation"
16 crossover_type = "CAS_weighted"
17 mutation_types = [['random', 1.0]]
18 chance_of_mutation = 0.1
19
20 # This parameter will tell the Organisms program if an epoch is desired, and how the_
  ↳user would like to proceed.
21 epoch_settings = {'epoch mode': 'same population', 'max repeat': 5}
22
23 # These are variables used by the algorithm to make and place clusters in.
24 r_ij = 3.4
25 cell_length = r_ij * (sum([float(noAtoms) for noAtoms in list(cluster_makeup.
  ↳values())]) ** (1.0/3.0))
26 vacuum_to_add_length = 10.0
27
28 # The RunMinimisation.py algorithm is one set by the user. It contain the def_
  ↳Minimisation_Function
29 # That is used for local optimisations. This can be written in whatever way the user_
  ↳wants to perform
30 # the local optimisations. This is meant to be as free as possible.
31 from RunMinimisation import Minimisation_Function
32
33 # This dictionary includes the information required to prevent clusters being placed_
  ↳in the population if they are too similar to clusters in this memory_operator
34 memory_operator_information = {'Method': 'Off'}
35
36 # This dictionary includes the information required by the predation scheme
37 predation_information = {'Predation Operator': 'SCM', 'SCM Scheme': 'TC-SRA', 'rCut_
  ↳high': 3.2, 'rCut_low': 2.9, 'rCut_resolution': 0.05}
38
39 # This dictionary includes the information required by the fitness scheme
40 energy_fitness_function = {'function': 'exponential', 'alpha': 3.0}
41 SCM_fitness_function = {'function': 'exponential', 'alpha': 1.0}
42 fitness_information = {'Fitness Operator': 'SCM + Energy', 'Use Predation Information
  ↳': True, 'SCM_fitness_contribution': 0.5, 'Dynamic Mode': False, 'energy_fitness_
  ↳function': energy_fitness_function, 'SCM_fitness_function': SCM_fitness_function}
43
44 # Variables required for the Recording_Cluster.py class/For recording the history as_
  ↳required of the genetic algorithm.
45 ga_recording_information = {}
46 ga_recording_information['ga_recording_scheme'] = 'Limit_energy_height' # float('inf')
47 ga_recording_information['limit_number_of_clusters_recorded'] = 5 # float('inf')
48 ga_recording_information['limit_energy_height_of_clusters_recorded'] = 1.5 #eV
49 ga_recording_information['exclude_recording_cluster_screened_by_diversity_scheme'] =
  ↳True
50 ga_recording_information['record_initial_population'] = True
51 ga_recording_information['saving_points_of_GA'] = [3,5]
52
53 # These are last technical points that the algorithm is designed in mind
54 force_replace_pop_clusters_with_offspring = True
55 user_initialised_population_folder = None

```

(continues on next page)

(continued from previous page)

```
56 rounding_criteria = 10
57 print_details = False
58 no_of_cpus = 2
59 finish_algorithm_if_found_cluster_energy = None
60 total_length_of_running_time = None
61
62 # These are the details that will be used to create all the Trials for this set of
63 ↪genetic algorithm experiments.
64 dir_name = 'ThisIsTheFolderThatScriptsWillBeWrittenTo'
65 NoOfTrials = 100
66 Condense_Single_Mention_Experiments = True
67 making_files_for = 'slurm_JobArrays_full'
68 finish_algorithm_if_found_cluster_energy = None
69 total_length_of_running_time = 6.0
70
71 # These are the details that are used to create the Job Array for slurm
72 JobArraysDetails = {}
73 JobArraysDetails['mode'] = 'JobArray'
74 JobArraysDetails['project'] = 'uoo00084'
75 JobArraysDetails['time'] = '8:00:00'
76 JobArraysDetails['nodes'] = 1
77 JobArraysDetails['ntasks_per_node'] = no_of_cpus
78 JobArraysDetails['mem'] = '1G'
79 JobArraysDetails['email'] = "geoffreywealslurmnotifications@gmail.com"
80 JobArraysDetails['python version'] = 'Python/3.6.3-gimkl-2017a'
81
82 ''' ----- '''
83 # Write all the trials that the user desires
84 MakeTrialsProgram(cluster_makeup=cluster_makeup,
85                  pop_size=pop_size,
86                  generations=generations,
87                  no_offspring_per_generation=no_offspring_per_generation,
88                  creating_offspring_mode=creating_offspring_mode,
89                  crossover_type=crossover_type,
90                  mutation_types=mutation_types,
91                  chance_of_mutation=chance_of_mutation,
92                  r_ij=r_ij,
93                  vacuum_to_add_length=vacuum_to_add_length,
94                  Minimisation_Function=Minimisation_Function,
95                  surface_details=surface_details,
96                  epoch_settings=epoch_settings,
97                  cell_length=cell_length,
98                  memory_operator_information=memory_operator_information,
99                  predation_information=predation_information,
100                  fitness_information=fitness_information,
101                  ga_recording_information=ga_recording_information,
102                  force_replace_pop_clusters_with_offspring=force_replace_pop_clusters_with_
103 ↪offspring,
104                  user_initialised_population_folder=user_initialised_population_folder,
105                  rounding_criteria=rounding_criteria,
106                  print_details=print_details,
107                  no_of_cpus=no_of_cpus,
108                  dir_name=dir_name,
109                  NoOfTrials=NoOfTrials,
110                  Condense_Single_Mention_Experiments=Condense_Single_Mention_Experiments,
111                  JobArraysDetails=JobArraysDetails,
112                  making_files_for=making_files_for,
```

(continues on next page)

(continued from previous page)

```
111     finish_algorithm_if_found_cluster_energy=finish_algorithm_if_found_cluster_energy,  
112     total_length_of_running_time=total_length_of_running_time)  
113     ''' ----- '''
```

We will now explain the components of this script. Many of the variables have been explained in *Run.py - Using the Genetic Algorithm*. These are `cluster_makeup`, `surface_details`, `pop_size`, `generations`, `no_offspring_per_generation`, `creating_offspring_mode`, `crossover_type`, `mutation_types`, `chance_of_mutation`, `epoch_settings`, `r_ij`, `cell_length`, `vacuum_to_add_length`, `Minimisation_Function`, `memory_operator_information`, `predation_information`, `fitness_information`, `ga_recording_information`, `force_replace_pop_clusters_with_offspring`, `user_initilised_population_folder`, `rounding_criteria`, `print_details`, `no_of_cpus`, `finish_algorithm_if_found_cluster_energy` and `total_length_of_running_time`.

Here, we will cover the meaning for variables `dir_name`, `NoOfTrials`, `Condense_Single_Mention_Experiments`, `making_files_for` and `JobArraysDetails`.

1) Details to create all the desired trials

These include the details needed for this program to make the trials desired. These variables are:

- **`dir_name`** (*str.*): This is the name of the folder to put the trials into.
- **`NoOfTrials`** (*int.*): This is the number of trials you would like to create.
- **`Condense_Single_Mention_Experiments`** (*bool.*): This program is designed to place the trials in a ordered system. If this is set to true, the trials will be put in a folder called `XN_P_p_O_o`, where `XN` is the cluster makeup, `p` is the size of the population, and `o` is the number of offspring make per generations. If this is set to False, the directory that will be made will be `X/XN/Pop_p/Off_o`, where `X` are the elements that make up the cluster, `XN` is the cluster makeup, `p` is the size of the population, and `o` is the number of offspring make per generations.
- **`making_files_for`** (*str.*): This tells how the `MakeTrials` program will write files for performing multiple genetic algorithm trials. See *How files are created for running multiple genetic algorithm trials* for more information about this.
- **`no_of_packets_to_make`** (*int.*): If `making_files_for = 'slurm_JobArrays_packets'`, then this tells the `MakeTrials` program how to split the **`NoOfTrials`** number of genetic algorithm trials into packets. See *How files are created for running multiple genetic algorithm trials* for more information about this.

An example of these parameters in `MakeTrials.py` is given below:

```
62 # These are the details that will be used to create all the Trials for this set of_  
63 ↪genetic algorithm experiments.  
64 dir_name = 'ThisIsTheFolderThatScriptsWillBeWrittenTo'  
65 NoOfTrials = 100  
66 Condense_Single_Mention_Experiments = True  
67 making_files_for = 'slurm_JobArrays_full'  
68 finish_algorithm_if_found_cluster_energy = None  
69 total_length_of_running_time = 6.0
```

1.1) making_files_for: How files are created for running multiple genetic algorithm trials

This option is designed to write the `submit.sl` or `mass_submit.sl` scripts that you need for running multiple genetic algorithm trial jobs on slurm. There are three options for this setting. These options for `JobArraysDetails['mode']` are `'individual'`, `'slurm_JobArrays_full'`, and `'slurm_JobArrays_packets'`.

If `'individual'`, `MakeTrials` will create a `slurm.sl` file for each individual genetic algorithm trial.

If `'slurm_JobArrays_full'`, `MakeTrials` will create a `mass_slurm.sl` file will submit an array job that performs `NoOfTrials` genetic algorithm trials. An example of this shown below.

Listing 6: `mass_submit_full.sl`

```
1  #!/bin/bash -e
2  #SBATCH -J Data_fitness_changed_in_epoch_max_repeat_5_try2_Epoch_D_Energy_F_lrCut_SCM_
   ↪ alpha_3_fitness_normalised_F_SCM_0.0_lrCut_Ne38_P20_O16
3  #SBATCH -A uoo00084          # Project Account
4
5  #SBATCH --array=1-1000
6
7  #SBATCH --time=72:00:00      # Walltime
8  #SBATCH --nodes=1
9  #SBATCH --ntasks-per-node=1
10 #SBATCH --mem=300MB
11
12 #SBATCH --partition=large
13 #SBATCH --output=arrayJob_%A_%a.out
14 #SBATCH --error=arrayJob_%A_%a.err
15 #SBATCH --mail-user=geoffreywealslurmnotifications@gmail.com
16 #SBATCH --mail-type=ALL
17
18 #####
19 # Begin work section #
20 #####
21
22 # Print this sub-job's task ID
23 echo "My SLURM_ARRAY_JOB_ID: "${SLURM_ARRAY_JOB_ID}
24 echo "My SLURM_ARRAY_TASK_ID: "${SLURM_ARRAY_TASK_ID}
25
26 module load Python/3.6.3-gimkl-2017a
27
28 if [ ! -d Trial${SLURM_ARRAY_TASK_ID} ]; then
29     mkdir Trial${SLURM_ARRAY_TASK_ID}
30 fi
31 cp Run.py Trial${SLURM_ARRAY_TASK_ID}
32 cp RunMinimisation_LJ.py Trial${SLURM_ARRAY_TASK_ID}
33 cd Trial${SLURM_ARRAY_TASK_ID}
34 python Run.py
35 cd ..
36 cp arrayJob_${SLURM_ARRAY_JOB_ID}_${SLURM_ARRAY_TASK_ID}.out Trial${SLURM_ARRAY_TASK_
   ↪ ID}
37 cp arrayJob_${SLURM_ARRAY_JOB_ID}_${SLURM_ARRAY_TASK_ID}.err Trial${SLURM_ARRAY_TASK_
   ↪ ID}
```

If `'slurm_JobArrays_packets'`, `MakeTrials` will create a `mass_slurm.sl` file will submit an array job that performs `NoOfTrials` genetic algorithm trials. However, these will be run as `no_of_packets_to_make` number of packets that are running on slurm, where each packet is made up of $\frac{\text{NoOfTrials}}{\text{no_of_packets_to_make}}$ genetic algorithm trials that

are run in series. Use the 'slurm_JobArrays_packets' setting if you think each individual genetic algorithm trial will run for only a short amount of time (less than 5-10 minutes on average). The reason for using this setting rather than 'slurm_JobArrays_full' is because running lots of short jobs on slurm can cause issues for the slurm controller that controls the queue on slurm.

IMPORTANT: Make sure that the amount of time you have given for JobArraysDetails['time'] is much greater than the maximum amount of time you think each trial will be completed in times no_of_packets_to_make, i.e.

$$\text{maximum amount of time to run one GA trial} \times \frac{\text{NoOfTrials}}{\text{no_of_packets_to_make}} \ll \text{JobArraysDetails['time']}$$

If the below is not true, then it is recommended to use a partition on slurm that allows you to set :math:\{\text{rm}\{\text{JobArraysDetails['time']}\}\} to as long as possible so that the above equation is true. If you cant do this, consider breaking up running all your GA trials with a greater value of no_of_packets_to_make (up to :math:\{\text{rm}\{\text{no_of_packets_to_make}\} = \frac{\{\text{rm}\{\text{NoOfTrials}\}\{2\}}). If you cant do this as well, lower your value of NoOfTrials and do all your trials bit by bit (for example, if performing 1,000,000 trials, first run trials 1-1000, then trials 1001-2000, then trials 2001-3000, so on ...).

Avoid using 'slurm_JobArrays_full' if possible as performing lots of small jobs on slurm can break slurm. If you cant avoid it, use 'slurm_JobArrays_full' with caution.

As a guideline, set JobArraysDetails['time'] to as large a wall time as possible for the partition you are using on your slurm cluster.

An example of this shown below.

Listing 7: mass_submit_packets.sl

```

1  #!/bin/bash -e
2  #SBATCH -J Data_fitness_changed_in_epoch_max_repeat_5_try2_Epoch_D_Energy_F_1rCut_SCM_
   ↪ alpha_3_fitness_normalised_F_SCM_0.0_1rCut_Ne38_P20_O16
3  #SBATCH -A uoo00084          # Project Account
4
5  #SBATCH --array=1-50
6
7  #SBATCH --time=72:00:00      # Walltime
8  #SBATCH --nodes=1
9  #SBATCH --ntasks-per-node=1
10 #SBATCH --mem=300MB
11
12 #SBATCH --partition=large
13 #SBATCH --output=arrayJob_%A_%a.out
14 #SBATCH --error=arrayJob_%A_%a.err
15 #SBATCH --mail-user=geoffreywealslurmnotifications@gmail.com
16 #SBATCH --mail-type=ALL
17
18 #####
19 # Begin work section #
20 #####
21
22 # Print this sub-job's task ID
23 echo "My SLURM_ARRAY_JOB_ID: "${SLURM_ARRAY_JOB_ID}
24 echo "My SLURM_ARRAY_TASK_ID: "${SLURM_ARRAY_TASK_ID}
25
26 module load Python/3.6.3-gimkl-2017a
27
28 number_of_divides=20
29 for i in $( eval echo {1..${number_of_divides}} ); do

```

(continues on next page)

(continued from previous page)

```
30
31 trial_no=$(( ((${SLURM_ARRAY_TASK_ID} - 1) * ${number_of_divides} )) + $i ))
32 echo Currently performing calculation on trial: $trial_no
33
34 if [ ! -d Trial${trial_no} ]; then
35     mkdir Trial${trial_no}
36 fi
37 cp Run.py Trial${trial_no}
38 cp RunMinimisation_LJ.py Trial${trial_no}
39 cd Trial${trial_no}
40 python Run.py
41 cd ..
42 cp arrayJob_${SLURM_ARRAY_JOB_ID}_${SLURM_ARRAY_TASK_ID}.out Trial${trial_no}/
43 ↪arrayJob_${SLURM_ARRAY_JOB_ID}_${trial_no}.out
44 cp arrayJob_${SLURM_ARRAY_JOB_ID}_${SLURM_ARRAY_TASK_ID}.err Trial${trial_no}/
45 ↪arrayJob_${SLURM_ARRAY_JOB_ID}_${trial_no}.err
46 echo -n "" > arrayJob_${SLURM_ARRAY_JOB_ID}_${SLURM_ARRAY_TASK_ID}.out
47 echo -n "" > arrayJob_${SLURM_ARRAY_JOB_ID}_${SLURM_ARRAY_TASK_ID}.err
48
49 done
```

2) Slurm Details

The `JobArraysDetails` dictionary contains all the information that will be needed to write the `submit.sl` or `mass_submit.sl` scripts that you need for running multiple genetic algorithm trial jobs on slurm. The parameters to be entered into the `JobArraysDetails` dictionary are:

- **project** (*str*): The name of the project to run this on.
- **partition** (*str*): The partition to run this on.
- **time** (*str*): The length of time to give these jobs. This is given in 'HH:MM:SS', where HH is the number of hours, MM is the number of minutes, and SS is the number of seconds to run the genetic algorithm for.
- **nodes** (*int*): The number of nodes to use. Best to set this to 1.
- **ntasks_per_node** (*int*): The number of cpus to run these jobs across on a node. It is best to set this to the same value as `no_of_cpus` in the `MakeTrials.py` file.
- **mem** (*str*): This is the memory that is used in total by the job.
- **email** (*str*): This is the email address to send slurm messages to about this job. If you do not want to give an email, write here either `None` or `' '`.
- **python version** (*str*): This give the submit script the version of python to load when submitting this job on slurm. The default is `'Python/3.6.3-gimkl-2017a'`. However, if instead you want to use Python 3.7, you could write here `JobArraysDetails['python version'] = 'Python/3.7.3-gimkl-2018b'`. In slurm write module `avail python` to find out what versions of python you have available on your computer cluster system.

See [sbatch - Slurm Workload Manager - SchedMD³⁰](https://slurm.schedmd.com/sbatch.html) and [The Slurm job scheduler³¹](http://www.arc.ox.ac.uk/content/slurm-job-scheduler) to learn more about these parameters in the `submit.sl` script for slurm.

An example of these parameters in `MakeTrials.py` is given below:

³⁰ <https://slurm.schedmd.com/sbatch.html>

³¹ <http://www.arc.ox.ac.uk/content/slurm-job-scheduler>

```
70 # These are the details that are used to create the Job Array for slurm
71 JobArraysDetails = {}
72 JobArraysDetails['mode'] = 'JobArray'
73 JobArraysDetails['project'] = 'uoo00084'
74 JobArraysDetails['time'] = '8:00:00'
75 JobArraysDetails['nodes'] = 1
76 JobArraysDetails['ntasks_per_node'] = no_of_cpus
77 JobArraysDetails['mem'] = '1G'
78 JobArraysDetails['email'] = "geoffreywealslurmnotifications@gmail.com"
79 JobArraysDetails['python version'] = 'Python/3.6.3-gimkl-2017a'
```

3) Time to Run the MakeTrials Program

Now all there is to do is to run this program!

```
82 # Write all the trials that the user desires
83 MakeTrialsProgram(cluster_makeup=cluster_makeup,
84                  pop_size=pop_size,
85                  generations=generations,
86                  no_offspring_per_generation=no_offspring_per_generation,
87                  creating_offspring_mode=creating_offspring_mode,
88                  crossover_type=crossover_type,
89                  mutation_types=mutation_types,
90                  chance_of_mutation=chance_of_mutation,
91                  r_ij=r_ij,
92                  vacuum_to_add_length=vacuum_to_add_length,
93                  Minimisation_Function=Minimisation_Function,
94                  surface_details=surface_details,
95                  epoch_settings=epoch_settings,
96                  cell_length=cell_length,
97                  memory_operator_information=memory_operator_information,
98                  predation_information=predation_information,
99                  fitness_information=fitness_information,
100                 ga_recording_information=ga_recording_information,
101                 force_replace_pop_clusters_with_offspring=force_replace_pop_clusters_with_
102                 ↳offspring,
103                 user_initialised_population_folder=user_initialised_population_folder,
104                 rounding_criteria=rounding_criteria,
105                 print_details=print_details,
106                 no_of_cpus=no_of_cpus,
107                 dir_name=dir_name,
108                 NoOfTrials=NoOfTrials,
109                 Condense_Single_Mention_Experiments=Condense_Single_Mention_Experiments,
110                 JobArraysDetails=JobArraysDetails,
111                 making_files_for=making_files_for,
112                 finish_algorithm_if_found_cluster_energy=finish_algorithm_if_found_cluster_energy,
113                 total_length_of_running_time=total_length_of_running_time)
```

5.7.4 Can I make writing trials easily for many types of system.

The answer is yes. This program has been designed to be as flexible as possible. For example, the following MakeTrials script will make 100 trials for various types of cluster systems, and for various population and offspring per generation sizes.

Listing 8: MakeTrials_Multiple.py

```
1 from Organisms import MakeTrialsProgram
2
3 from RunMinimisation_Cu import Minimisation_Function as Minimisation_Function_Cu
4 from RunMinimisation_Au import Minimisation_Function as Minimisation_Function_Au
5 from RunMinimisation_AuPd import Minimisation_Function as Minimisation_Function_AuPd
6
7 cluster_makeups = [{"Cu": 37}, Minimisation_Function_Cu), {"Au": 55}, Minimisation_
8 ↪Function_Au), {"Au": 21, "Pd": 17}, Minimisation_Function_AuPd)]
9
10 genetic_algorithm_systems = [(20,16), (100,80), (50,1)]
11
12 for cluster_makeup, Minimisation_Function in cluster_makeups:
13     for pop_size, no_offspring_per_generation in genetic_algorithm_systems:
14         # Surface details
15         surface_details = {}
16
17         # These are the main variables of the genetic algorithm that with changes_
18 ↪could affect the results of the Genetic Algorithm.
19         generations = 2000
20
21         # These setting indicate how offspring should be made using the Mating and_
22 ↪Mutation Proceedures
23         creating_offspring_mode = "Either_Mating_and_Mutation"
24         crossover_type = "CAS_weighted"
25         mutation_types = [['random', 1.0]]
26         chance_of_mutation = 0.1
27
28         # This parameter will tell the Organisms program if an epoch is desired, and_
29 ↪how the user would like to proceed.
30         epoch_settings = {'epoch mode': 'same population', 'max repeat': 5}
31
32         # These are variables used by the algorithm to make and place clusters in.
33         r_ij = 3.4
34         cell_length = r_ij * (sum([float(noAtoms) for noAtoms in list(cluster_makeup.
35 ↪values())]) ** (1.0/3.0))
36         vacuum_to_add_length = 10.0
37
38         # The RunMinimisation.py algorithm is one set by the user. It contain the def_
39 ↪Minimisation_Function
40         # That is used for local optimisations. This can be written in whatever way_
41 ↪the user wants to perform
42         # the local optimisations. This is meant to be as free as possible.
43
44         # This dictionary includes the information required to prevent clusters being_
45 ↪placed in the population if they are too similar to clusters in this memory_operator
46         memory_operator_information = {'Method': 'Off'}
47
48         # This dictionary includes the information required by the predation scheme
49         predation_information = {'Predation Operator':'Energy', 'mode': 'comprehensive
50 ↪', 'minimum_energy_diff': 0.025}
51
```

(continues on next page)

(continued from previous page)

```

42     # This dictionary includes the information required by the fitness scheme
43     energy_fitness_function = {'function': 'exponential', 'alpha': 3.0}
44     SCM_fitness_function = {'function': 'exponential', 'alpha': 1.0}
45     fitness_information = {'Fitness Operator': 'SCM + Energy', 'Use Predation_
↳Information': False, 'SCM_fitness_contribution': 0.5, 'Dynamic Mode': False,
↳'energy_fitness_function': energy_fitness_function, 'SCM_fitness_function': SCM_
↳fitness_function}

46
47     # Variables required for the Recording_Cluster.py class/For recording the_
↳history as required of the genetic algorithm.
48     ga_recording_information = {}
49     ga_recording_information['ga_recording_scheme'] = 'Limit_energy_height' #_
↳float('inf')
50     ga_recording_information['limit_number_of_clusters_recorded'] = 5 # float('inf
↳')
51     ga_recording_information['limit_energy_height_of_clusters_recorded'] = 1.5 #eV
52     ga_recording_information['exclude_recording_cluster_screened_by_diversity_
↳scheme'] = True
53     ga_recording_information['record_initial_population'] = True
54     ga_recording_information['saving_points_of_GA'] = [3,5]
55
56     # These are last technical points that the algorithm is designed in mind
57     force_replace_pop_clusters_with_offspring = True
58     user_initialised_population_folder = None
59     rounding_criteria = 10
60     print_details = False
61     no_of_cpus = 2
62     finish_algorithm_if_found_cluster_energy = None
63     total_length_of_running_time = None
64
65     ''' ----- '''
66     # These are the details that will be used to create all the Trials for this_
↳set of genetic algorithm experiments.
67     dir_name = 'ThisIsTheFolderThatScriptsWillBeWrittenTo'
68     NoOfTrials = 100
69     Condense_Single_Mention_Experiments = True
70     making_files_for = 'slurm_JobArrays_full'
71
72     ''' ----- '''
73     # These are the details that are used to create the Job Array for slurm
74     JobArraysDetails = {}
75     JobArraysDetails['mode'] = 'JobArray'
76     JobArraysDetails['project'] = 'uoo00084'
77     JobArraysDetails['time'] = '8:00:00'
78     JobArraysDetails['nodes'] = 1
79     JobArraysDetails['ntasks_per_node'] = no_of_cpus
80     JobArraysDetails['mem'] = '1G'
81     JobArraysDetails['email'] = "geoffreywealslurmnotifications@gmail.com"
82     JobArraysDetails['python version'] = 'Python/3.6.3-gimkl-2017a'
83
84     ''' ----- '''
85     # Write all the trials that the user desires
86     MakeTrialsProgram(cluster_makeup=cluster_makeup,
87                       pop_size=pop_size,
88                       generations=generations,
89                       no_offspring_per_generation=no_offspring_per_generation,
90                       creating_offspring_mode=creating_offspring_mode,

```

(continues on next page)

(continued from previous page)

```
91         crossover_type=crossover_type,
92         mutation_types=mutation_types,
93         chance_of_mutation=chance_of_mutation,
94         r_ij=r_ij,
95         vacuum_to_add_length=vacuum_to_add_length,
96         Minimisation_Function=Minimisation_Function,
97         surface_details=surface_details,
98         epoch_settings=epoch_settings,
99         cell_length=cell_length,
100        memory_operator_information=memory_operator_information,
101        predation_information=predation_information,
102        fitness_information=fitness_information,
103        ga_recording_information=ga_recording_information,
104        force_replace_pop_clusters_with_offspring=force_replace_pop_clusters_with_
↳offspring,
105        user_initialised_population_folder=user_initialised_population_folder,
106        rounding_criteria=rounding_criteria,
107        print_details=print_details,
108        no_of_cpus=no_of_cpus,
109        dir_name=dir_name,
110        NoOfTrials=NoOfTrials,
111        Condense_Single_Mention_Experiments=Condense_Single_Mention_Experiments,
112        JobArraysDetails=JobArraysDetails,
113        making_files_for=making_files_for,
114        finish_algorithm_if_found_cluster_energy=finish_algorithm_if_found_
↳cluster_energy,
115        total_length_of_running_time=total_length_of_running_time)
116        ''' ----- '''
```

5.8 Safely Finishing the Genetic Algorithm Midway through the Algorithm

In some cases, if you are running the genetic algorithm, or have submitted the genetic algorithm trials through slurm, you may want to cancel the genetic algorithm from running before the genetic algorithm had completed. The algorithm is designed to be cancelled at any point during the algorithm and should be able to be restarted, but if possible it is recommended that this program is safely finished, meaning the algorithm completes the generation it is currently performing before finishing.

This program can be finished safely by making a file called *finish* in the same directory that the *Run.py* is in. Nothing needs to be added to this finish file. If the algorithm finds this *finish* file it will finish once the current generation has completed.

If you are running a mass number of genetic algorithms that you would like to safely finish, there is a program called *make_finish_files.py* that will add a *finish* file to any subdirectories from where you ran the *make_finish_files.py* program that also contain a *Run.py* file.

To run this program, type *make_finish_files.py* into the terminal. This program will deposit a *finish* file in any subdirectory that contains a *Run.py* file.

Do you think that the genetic algorithm will not finish before your job times out on slurm or whatever computer cluster job scheduler cancels your genetic algorithm job? Ultimately, this should not be an issue as the genetic algorithm has been designed to be cancelled unsafely. However, safely cancelling the genetic algorithm is preferred just in case there is a further bug in the program. Read up about the *total_length_of_running_time* parameter in *Other details of the Genetic algorithm* to learn about how to tell the genetic algorithm to automatically safely finish after a

certain period of time.

5.9 Restarting the Genetic Algorithm

The algorithm has been designed to record all the relevant information required to restart the algorithm from the last previously successful generation. The genetic algorithm by default records all the data it needs to disk, however, making files for backing up can be turned off or change to record only after so many generations if desired. If you do need to restart the genetic algorithm, you only need to run your `Run.py` file by running `python3 Run.py` in the terminal. You do not need to change any files and it is **recommended you do not change any genetic algorithm files** (unless the genetic algorithm does not restart properly, which in that case you may want to play around with files for debugging reasons. However, if the genetic algorithm is working properly you should not need to do this). The genetic algorithm will change or update any files that it needs to. Just sit back and let the python program do all the work for you.

5.9.1 Requirements for Restarting the Genetic Algorithm

To be able to restart this genetic algorithm, the algorithm will write the following files to disk after each generation:

- `Population/Population.db` - This is a ASE database that contains all the structural and energy information about clusters in the population. See *Recording Clusters From The Genetic Algorithm* for more information about recording clusters created during the genetic algorithm.
- `Population/current_population_details.txt` - This contain information about the current generation, and the names and energies of the clusters in the population at that generation.
- `epoch_data` - This contains information that is needed to know about epoching. This file is formatted differently for each epoch method. See *Using Epoch Methods* for more information about this file.
- `Population/EnergyProfile.txt` - This file contains all the information of the clusters that were created at any point during the genetic algorithm, whether they were placed into the population or if they were excluded because they violated the predation operator
- `Population/Population_history.txt` - This file contains all the information about the population after every generation.
- `Recorded_Data/GA_Recording_Database.db` - This contains the clusters that were created during a genetic algorithm. This file will only be created if the user wants to record clusters created during the genetic algorithm. See *Recording Clusters From The Genetic Algorithm* for more information about recording clusters created during the genetic algorithm.

These six files are required and must have been created at the same generation for the Genetic Algorithm to proceed. The algorithm will not proceed if any of these files is missing or any information about these files causes issues and confusions for the genetic algorithm.

Also, backups of the `Population/Population.db`, `Population/current_population_details.txt`, and the `epoch_data` files are created before beginning a generation. This means that if anything goes wrong that these backup files are available for the genetic algorithm to use. Generally, this genetic algorithm will first look for these backup files and get information from them, since it is known that these files should contain all the required information and should not be corrupted. The non-backup files are generally created at different points during the generation, so can be corrupt or contain information that is inconsistent with each other if the algorithm was abruptly cancelled during a generation. However, if a generation finished successfully without any issues, these backup files are removed after every generation. In these cases there will be no backup files, and the genetic algorithm will look for the non-backup files.

5.9.2 Files that are updated when restarting the genetic algorithm

There are three other files that are updated by the genetic algorithm if they contain information about clusters obtained from an unsuccessful generation. These are:

- `Population/EnergyProfile.txt` - This file contains all the information of the clusters that were created at any point during the genetic algorithm, whether they were placed into the population or if they were excluded because they violated the predation operator
- `Population/Population_history.txt` - This file contains all the information about the population after every generation.
- `Recorded_Data/GA_Recording_Database.db` - This contains the clusters that were created during a genetic algorithm. See *Recording Clusters From The Genetic Algorithm* for more information about recording clusters created during the genetic algorithm.

The `GA_Recording_Database.db` is not backed-up after every generation. This is because this file can get quite large and would continuously increase the computational time for performing each generation. For this reason, this file is not continuously backed-up. There are rarely problems with this database file.

5.10 Common Issues of the Genetic Algorithm and Ways to Solve Them

5.10.1 Randomly generated clusters all explode when creating the initial population

Here, the genetic algorithm keeps giving you the message `Cluster exploded`. Will obtain a new `cluster`. Here, the genetic algorithm keeps creating a single cluster that continuously explodes over and over while you are initially setting up the initial population with randomly generated clusters. An exploded cluster means that the cluster does not form a single structure when it forms. Instead, the cluster locally minimises into two or more separate smaller clusters that are not attached to each other. This is not what we want, as we want to make a single cluster with the specifications given in the `cluster_makeup` variable in the `Run.py` file

If this occurs, first check that your `Minimisation` method in your `RunMinimisation.py` is set up for a cluster with an element that is different to the elemental makeup of the cluster you have specified in the `cluster_makeup` variable in your `Run.py` file. The genetic algorithm proceeds but your clusters do not locally optimise because of this mismatch. Check your `RunMinimisation.py` and make sure that it includes the elements that you need to locally optimise, i.e. matches the `cluster_makeup` variable in your `Run.py` file.

If your `RunMinimisation.py` is correct and you still have this issue, this issue may be occurring because the value that you have set for `cell_length` is too high. Look at systematically reducing this to a point that this doesn't happen. If you don't do this, this will likely not affect the rest of the genetic algorithm, it will just take longer than necessary to make the initial population. However, if you are using the `random` or `random_XX` mutation methods, this will cause issues during the genetic algorithm. If you are using either of these mutation methods, it is recommended that you try to set a value of `cell_length` where explosions are minimised.

5.10.2 My cluster does not locally minimise properly or there is an error with ASAP

If your cluster is not locally minimising properly or you find there is an error with ASAP, it is likely that you have an incorrect setting in your `RunMinimisation.py` script. For example, you may be try to locally minimise a Cu cluster, but the calculator you are using in the `RunMinimisation.py` script is incorrectly set to give the potential of Au atoms only. Check your `RunMinimisation.py` script and then try running the genetic algorithm again.

5.10.3 I have tried to run my genetic algorithm, but it won't run because it says that a file called `ga_running.lock` exists in the genetic algorithms directory

When the genetic algorithm begins, it will create a blank file called `ga_running.lock`. This file is a lock that will prevent the user from running the genetic algorithm if it exists in the genetic algorithm's directory, i.e. preventing the user from running the genetic algorithm twice simultaneously, as this will cause major issues for your genetic algorithm if you do this by accident. This file should only exist while the genetic algorithm is running.

This is to prevent the user from accidentally running the genetic algorithm while it is already running. However, if you had to stop the genetic algorithm without finishing it safely (see *Safely Finishing the Genetic Algorithm Midway* to see what "safely" means), then this file will still be in the genetic algorithm's directory. If this is the case, and you are sure that the genetic algorithm is not running, remove the `ga_running.lock` file from the directory and try running the genetic algorithm again.

5.10.4 The ase database website is not formatted correct/format is hard to use

This is a problem in most recent versions of ASE. Formatting does not seem to be an issue for any version of ASE version 3.19, but ASE database does not seem to be formatted properly for any version of ASE version 3.20. The current solution for this is to change the version of ase to 3.19. To do this, first uninstall ASE by typing `pip3 uninstall ase` in the terminal, then running `pip3 install --user ase==3.19.3` in the terminal.

See *Using Databases with the Genetic Algorithm* for more information on how databases work in ASE and for the genetic algorithm.

5.10.5 I am having issues that some jobs are not submitted to slurm every so often when running `Run_mass_submitSL_slurm.py` and it is having to resubmit jobs

Every so often I have found that slurm does not submit a job. This may be because too many jobs are being submitting consecutive in a short period of time, or because there is an internet issue or a slurm hang that only lasts for a few tens of seconds.

`Run_mass_submitSL_slurm.py` has been designed to wait for 10 seconds before attempting to resubmit the job to slurm. If `Run_mass_submitSL_slurm.py` has attempted to submit the current job to slurm a maximum of 20 times, then `Run_mass_submitSL_slurm.py` will exit and tell the user what jobs have not been submitted due to this issue.

The amount of time that `Run_mass_submitSL_slurm.py` will wait before attempting to resubmit a job to slurm is given by the variable `time_to_wait_before_next_submission_due_to_temp_submission_issue` in `Subsidiary_Programs/Run_mass_submitSL_slurm.py`, while the number of times that `Run_mass_submitSL_slurm.py` will attempt to resubmit a job to slurm before it will give up is given the variable `number_of_consecutive_error_before_exitting` in `Subsidiary_Programs/Run_mass_submitSL_slurm.py`.

See [Run_mass_submitSL_slurm.py - How to execute all Trials using the JobArray Slurm Job Submission Scheme](#) for more information about how `Run_mass_submitSL_slurm.py` is designed and works and the settings that you may need to change in `Run_mass_submitSL_slurm.py` so that it works for you in slurm.

5.10.6 I found that when the fitness operator gave a `ZeroDivisionError` error when trying to obtain the `CNA_fitness_contribution` or the energy fitness value of `rho_i`

An example of this type of error is given below for a `ZeroDivisionError` error for the `CNA_fitness_contribution` value.

```
"/.../GA/Fitness_Operators/CNA_Fitness_Contribution.py", line 71, in get_CNA_fitness_
↪parameter_normalised
CNA_fitness_contribution = (CNA_most_similar_average - min_minimarity)/(max_
↪similarity - min_minimarity)
ZeroDivisionError: float division by ZeroDivisionError: float division by zero
```

Here, either the population has no similarity span (or no energy span). In this example, this is because the cluster in the population with the highest similarity value has the same similarity as the cluster in the population with the lowest similarity value (i.e. `max_similarity` is equal to `min_minimarity`). There has been an update that should prevent this from occur. However, if this problem does arise, the easiest way to potentially solve this problem is to lower your input value for `rounding_criteria`

5.11 Helpful Programs to Create and Run the Genetic Algorithm

As well as the genetic algorithm, we have included a bunch of programs that can be used to create all the scripts that are needed to run the genetic algorithm. In this articles, we will introduce all of the program that can be used to create these scripts and to run them all in mass. Example files for running many of these programs can be found in `Examples/CreateSets`. Some of these programs can also be run by typing the program you want to run into the terminal from whatever directory you are in.

The scripts and programs that we will be mentioned here are:

- `get_newly_initilised_population.py`
- `Run_submitSL_slurm.py`
- `Run_mass_submitSL_slurm.py`
- `make_finish_files.py`
- `remove_finish_files.py`
- `remove_lock_files.py`

There is also information on how to generate multiple trials of your genetic algorithm experiment on mass at [Make-Trials.py - Creating Multiple, Repeated Genetic Algorithm Trials](#).

5.11.1 What to make sure is done before running any of these scripts.

If you installed Organisms through pip3

If you installed the Organisms program with pip3, these scripts will be installed in your bin. You do not need to add anything into your `~/ .bashrc`. You are all good to go.

If you performed a Manual installation

If you have manually added this program to your computer (such as cloning this program from Github), you will need to make sure that you have included the `Subsidiary_Programs` folder into your `PATH` in your `~/ .bashrc` file. All of these program can be found in the `Subsidiary_Programs` folder. To execute these programs from the `Subsidiary_Programs` folder, you must include the following in your `~/ .bashrc`:

```
export PATH_TO_GA="<Path_to_Organisms>"
```

where `<Path_to_Organisms>`" is the path to get to the genetic algorithm program. Also include somewhere before this in your `~/ .bashrc`:

```
export PATH="$PATH_TO_GA"/Organisms/Subsidiary_Programs:$PATH
```

See more about this in *Installation of the Genetic Algorithm*.

5.11.2 `get_newly_initilised_population.py` - Creating an initial population of randomly generated clusters

More information on creating a newly initialised population and how to use it in the genetic algorithm can be found in *Initialising a New Population*.

5.11.3 `Run_submitSL_slurm.py` - How to execute all the Run.py files for all Trial folders in slurm

NOT ADD/DEVELOPED until I know if this is necessary.

5.11.4 `Run_mass_submitSL_slurm.py` - How to execute all Trials using the JobArray Slurm Job Submission Scheme

If you have use the `JobArraysDetails['mode'] = 'JobArray'` scheme in your `MakeTrials.py` script, an easy way to run them all is to run the script `Run_mass_submitSL_slurm.py` from the same directory as you have your `MakeTrials.py` script. This script will walk through all subdirectories from this point and find and run all your `mass_submit.sl` scripts for you automatically. All you need to do is enter `Run_mass_submitSL_slurm.py` into the terminal, press enter, and watch all our genetic algorithm trials be submitted to slurm.

There is one input that you can pass into the program. The `Run_mass_submitSL_slurm.py` program is designed to wait one minute between submitting jobs. This is because you can cause slurm issues if too many jobs are submitted at once. However, you can override this if you enter in `False` as the first argument. This will tell `Run_mass_submitSL_slurm.py` to keep submitting jobs without waiting one minute between them.

Note that you will notice that if you submit too many jobs, or if you submit too many jobs too quickly, you will get problems with slurm that you are submitting too many jobs or submitting them too quickly. This is all dependent on how slurm is set up. This has been covered by the program, and will tell you how long it will be waiting when it

is waiting. Do not stop the program while it is submitting jobs. If you do quit before it has finished, it will make resubmitting jobs difficult. If you find there is an error somewhere, or you know that you have entered in the wrong variables into the “MakeTrials.py”, then feel free to quit the program.

The following variables can be different for different people. Feel free to change the following variables in your `Subsidiary_Programs/Run_mass_submitSL_slurm.py` as you need

- **Max_jobs_in_queue_at_any_one_time** (*int*): This is the maximum number of jobs that slurm allows you to have in your queue. This is usually set by default to 1000. I personally have set this to 10,000 and this is what is current set in `Run_mass_submitSL_slurm.py`. Default: 10,000
- **time_to_wait_before_next_submission** (*float*): This is the amount of time that this program waits after submitting a job, before continuing on. This is given in seconds. **Do not set this to less than 10 seconds.** Default: 20.0 (seconds)
- **time_to_wait_max_queue** (*float*): This is the amount of time that this program waits after it has found that the maximum number of jobs have been submitted to the queue. `Run_mass_submitSL_slurm.py` will wait for this amount of time before continuing again. This is given in seconds. **Do not set this to less than 10 seconds.** Default: 60.0 (seconds)

Problems can occur every so often when submitting jobs to slurm, but these are generally internet connectivity problems or slurm hanging problems that resolve themselves after a few tens of seconds. There are two other variables that determine how `Run_mass_submitSL_slurm.py` will deal with issues.

- **time_to_wait_before_next_submission_due_to_temp_submission_issue** (*float*): This is the amount of time that this program waits after it has experienced an error in submitting a job. This is given in seconds. **Do not set this to less than 10 seconds.** Default: 10.0 (seconds)
- **number_of_consecutive_error_before_exitting** (*int*): This is the number of times that `Run_mass_submitSL_slurm.py` will attempt to resubmit a job to slurm before it will give up. After this many consecutive errors arising, some systematic error is likely occurring. In this case, `Run_mass_submitSL_slurm.py` will print the directories of all the jobs that were not submitted and then close.

Hopefully running `Run_mass_submitSL_slurm.py` will submit all your genetic algorithm trials.

The names of the jobs can be quite big, only because of how these are made by `MakeTrials.py`. When looking in `squeue` to see how things are going, it is sometimes useful to expand the names in the `squeue` output. This can be done as shown below:

```
squeue -o "%.20i %.9P %.5Q %.50j %.8u %.8T %.10M %.11l %.6D %.4C %.6b %.20S %.20R %.8q  
↪" -u $USER --sort=+i
```

Here, after `-o`, `i` specifies the job ID and `j` specifies the job name. You can change this number to the number of characters this will display. Here `%.20i` indicates that `squeue` will dedicate 20 characters to displaying the job ID and `%.50j` indicates that `squeue` will dedicate 50 characters to displaying the name of the job.

5.11.5 *make_finish_files.py* - How to safely exit a genetic algorithm that not completed all generations

This program is designed to create a *finish* file in directories that contain a *Run.py* file. This *finish* file does not contain anything, but is a flag for the genetic algorithm to tell it to safely exit the algorithm once it has finished running its current generation if you would like to finish the algorithm before it has completed.

To run this program, type `make_finish_files.py` into the terminal. This program will deposit a *finish* file in any subdirectory that contains a `Run.py` file.

To read more about how the *finish* file work in the genetic algorithm, see [Safely Finishing the Genetic Algorithm Midway through the Algorithm](#)

5.11.6 *remove_finish_files.py* - Removing all `finish` files from many directories

This program is designed to remove all `finish` files in directories that contain a `Run.py` file. This `finish` file does not contain anything, but is a flag for the genetic algorithm to tell it to safely exit the algorithm once it has finished running its current generation if you would like to finish the algorithm before it has completed.

To run this program, type `remove_finish_files.py` into the terminal. This program will remove all `finish` file in any subdirectory that contains a `Run.py` file.

To read more about how the `finish` file work in the genetic algorithm, see *Safely Finishing the Genetic Algorithm Midway through the Algorithm*

5.11.7 *remove_lock_files.py* - Removing all `ga_running.lock` files from many directories

This program is designed to remove all `ga_running.lock` files in directories that contain a `Run.py` file. This `ga_running.lock` file does not contain anything, but is a flag that prevents the user from running the genetic algorithm if the genetic algorithm is already actively running. This prevents the user running the genetic algorithm twice simultaneously. However, if you cancel the genetic algorithm unsafely, this file will be left in the directory. If you know that you are not currently running the genetic algorithm, you can remove the `ga_running.lock` file from the genetic algorithm's directory. If you have many of these to remove, use this program to help you do this in one click of the button.

To run this program, type `remove_lock_files.py` into the terminal. This program will remove all `ga_running.lock` file in any subdirectory that contains a `Run.py` file.

To read more about how the `ga_running.lock` file work in the genetic algorithm, see *Common error issues using involving ``ga_running.lock``*

5.12 Helpful Programs for Gathering data and Post-processing Data

As well as including programs for creating and running mass numbers of genetic algorithm, we have also included scripts and programs to gather and process the data across all the set of repeated genetic algorithm runs. In this article, we will introduce all of the program that can be used to run these programs. These programs can be run by typing the program you want to run into the terminal from whatever directory you are in.

The scripts and programs that we will be mentioned here are:

- `Did_Find_LES.py`
- `Did_Complete.py`
- `GetLESOAllTrials.py`
- `Postprocessing_Database.py`

5.12.1 What to make sure is done before running any of these scripts.

If you installed Organisms through pip3

If you installed the Organisms program with pip3, these scripts will be installed in your bin. You do not need to add anything into your `~/ .bashrc`. You are all good to go.

If you performed a Manual installation

If you have manually added this program to your computer (such as cloning this program from Github), you will need to make sure that you have included the `Postprocessing_Programs` folder into your `PATH` in your `~/ .bashrc` file. All of these program can be found in the `Postprocessing_Programs` folder. To execute these programs from the `Postprocessing_Programs` folder, you must include the following in your `~/ .bashrc`:

```
export PATH_TO_GA="<Path_to_Organisms>"
```

where `<Path_to_Organisms>`" is the path to get to the genetic algorithm program. Also include somewhere before this in your `~/ .bashrc`:

```
export PATH="$PATH_TO_GA"/Organisms/Postprocessing_Programs:$PATH
```

See more about this in *Installation of the Genetic Algorithm*.

5.12.2 *Did_Complete.py* - Have all your genetic algorithm trials completed?

This program is the first program that you should use before continuing on with any analysis. It is a quick program that will scan through all the trials, and check to see if they have completed.

To use this program, you want to enter into into the terminal

```
Did_Complete.py
```

in the directory that you ran your `MakeTrials.py` script from. You can also enter `Did_Complete.py` into the terminal within any folders, as long as at some point it will find the `Trials` in the subdirectories that you ran.

5.12.3 *Did_Find_LES.py* - Did all your genetic algorithm trials find the global minimum?

This program is designed to determine which of the trials you can found the global minimum that you were searching for. To run this program, enter `Did_Find_LES.py` into the terminal at any directory you want. This program will go through all subdirectories in search for folders that start with `Trial`, and look through the result to see if the global minimum you are looking for has been found for each trial run.

More specifically, this algorithm looks for any entries in the `EnergyProfiles.txt` files for each trial of clusters that are of a certain energy to a certain number of decimal places.

This program will ask the user what the energy is of the cluster that the user wants to locate in each trial, the number of decimal places that the user wants to round the energy to, and the number of generations the user wants to means the genetic algorithm trials up to (If this is not given, the algorithm will look through every generation).

Each set of trials is measured individually for different genetic algorithms in different folders.

5.12.4 *GetLESOAllTrials.py* - Get information of generations and number of minimisations performed

This program is designed to obtain information about the generation and the number of minimisations performed to first obtain the lowest energy clusters each trial had found. This algorithm will also report the average number of generations and average number of minimisation performed across all the trials that had found the lowest of the lowest energy clusters those trials had found. For example, if 5 of 20 genetic algorithm trials found the a cluster with the same energy and this cluster was lower in energy than the lowest energy clusters found from the other 15 trials, then the average number of generations and minimisations is taken for those 5 that had found the lowest of the lowest energy clusters.

You can run this program by typing `GetLESOAllTrials.py` in the terminal in any folder. This program will search through all subdirectories for folders that start with the name `Trial`, and report on those genetic algorithm trials found in the same folder (being apart of the same set of genetic algorithm trials). The algorithm will ask for two pieces of information:

- The generation you would like to search up to (Default: The full genetic algorithm until the LES has been found or the genetic algorithm has successfully finished).
- The number of decimal places to round the energy to (Default: 2 decimal places).

You can also enter this in the terminal when you type in `GetLESOAllTrials.py`:

```
GetLESOAllTrials.py maximum_generation_to_sample_up_to
```

where the number of decimal places to run the genetic algorithm to is given as 2 decimal places (this is the default), or you can enter into the terminal

```
GetLESOAllTrials.py maximum_generation_to_sample_up_to number_of_decimal_places_to_
↪round_the_energy_to
```

Each set of trials is measured individually for different genetic algorithms in different folders. This program should be run **after all genetic algorithm trials have successfully finished**.

5.12.5 *Postprocessing_Database.py* - For breaking a large database into smaller chunks

If a database (such as the storage databse in `Recorded_Data/GA_Recording_Database.db`) is too big to process with `ase db`, this program is designed to break up the database into smaller databases which can be better handled by `ase db` and your computer. This program will sort these clusters before placing them in the separate, potentially smaller databases. This program will also rotate the cluster so that the principle axis of inertia points along the z axis.

This program is run by the user moving into the `Recorded_Data` folder in the terminal and running the `Postprocessing_Database.py` program. There are two parameters that need to be entered. These are:

- **number_of_clusters_per_database** (*int*): This is the maximum number of clusters you would like in each database.
- **sort_clusters_by** (*str*): This tells the program how you would like clusters sorted in this(these) database(s).

You can also enter this in the terminal when you type in `Postprocessing_Database.py`:

```
Postprocessing_Database.py number_of_clusters_per_database
```

where the number of decimal places to run the genetic algorithm to is given as 2 decimal places (this is the default), or you can enter into the terminal

```
Postprocessing_Database.py number_of_clusters_per_database sort_clusters_by
```

5.12.6 *database_viewer.py* - Viewing GA databases with ASE database website viewer with metadata

The databases that are created by the Organisms program has metadata that allows the clusters to be organised in the database by their energy. The metadata also contains information about all the variables included in the database for the users convenience. However, in recent versions of ASE the metadata is not included when using the website. *database_viewer* allows the metadata to be included in the ASE website viewer.

This program is run by the user moving into the `Recorded_Data` folder in the terminal and running the `database_viewer.py` program. There is one parameter that need to be entered. This is:

- **name_of_the_database** (*str*): This is the name of the database that you want to view.

Enter this into the terminal when you type in `database_viewer.py`:

```
database_viewer.py name_of_the_database
```

5.12.7 *make_energy_vs_similarity_results.py* - For analysing the genetic algorithm under-the-hood

It is often useful to understand how the genetic algorithm procedure during the global optimisation of a cluster. This is especially useful if you are wanting to analyse the efficiency of the genetic algorithm. We have created a program that can help to get under the hood of the Organisms program and understand what clusters the genetic algorithm was obtaining. This creates a series of energy vs similarity plots that act as a way of observing clusters created on the potential energy surface. See more information about the *make_energy_vs_similarity_results.py* program at [Information about using the make_energy_vs_similarity_results.py script](#).

5.13 Information about using the *make_energy_vs_similarity_results.py* script

This page gives a description how to analysed the clusters that have been created. This page is an extention of [make_energy_vs_similarity_results.py - For analysing the genetic algorithm under-the-hood](#)

One of the bits of information that you may like to obtain from the genetic algorithm, especially if you are accessing the efficiency of the genetic algorithm program, is what happened when various genetic algorithms ran. One way of assessing what happened during a genetic algorithm is to observe what clusters were created as the genetic algorithm ran. This is practically very hard to do if you are just looking at the clusters that are made. However, if you can assess the clusters that you make with various parameters, you can get a better idea of the type of clusters that were created during your genetic algorithm.

The best way to monitor the types of clusters that are created is using a potential energy surface (PES) plot, which describes the energy as a function of the spatial positions of the atoms in your cluster. This type of plot can not be visualise as this would require a plot with $3N - 6$ spatial axes and one energy axis (where N is the number of atoms in your cluster). Instead, we (and others in the literature) have found it best to replace the $3N - 6$ spatial axes with one or two axis that describe the structural similarity between clusters.

In Organisms, we have a function that is able to describe the structural similarity between various clusters based on the common neighbour analysis. This function is called the Structural Comparison Method (SCM). Using this similarity parameter, it is possible to plot a PES plot, where:

- the x axis is the similarity between clusters made during your genetic algorithm against a reference cluster, and
- the y axis is the energy of the cluster.

We have created the `make_energy_vs_similarity_results.py` program that processes the data from your genetic algorithm and gives you energy vs. similarity plots that describes the types of cluster that were created during the genetic algorithm and plots them in the style of a PES plot. In this page, we will describe how to use this program, as well as describe the types of plots that you will obtain from this program.

5.13.1 Requirements for using the `make_energy_vs_similarity_results.py` Program

To use this program, you must record all the clusters that were created during the genetic algorithm. This requires your `ga_recording_information` dictionary to be set to record all clusters created during the genetic algorithm. Do to this, include the following in the `ga_recording_information` variable in your `Run.py` file:

```
ga_recording_information = {}
ga_recording_information['ga_recording_scheme'] = 'All'
ga_recording_information['exclude_recording_cluster_screened_by_diversity_scheme'] = 
↪False
```

Note that you can still set the `saving_points_of_GA`, `record_initial_population`, and `show_GA_Recording_Database_check_percentage` variables how ever you like.

5.13.2 Running the `make_energy_vs_similarity_results.py` Program

Due to the complexity of the information required to run this program, this program is executed from a python script rather than accessing it directly unlike most of the other side programs available in Organisms.

An example of the `Run_energy_vs_similarity_script.py` script that is used to execute the `make_energy_vs_similarity_results.py` program for a global optimisation of a 98-atom Lennard-Jones cluster is given below:

Listing 9: `Run_energy_vs_similarity_script.py`

```
1 from ase.io import read
2 from asap3.Internal.BuiltinPotentials import LennardJones
3 from Organisms import make_energy_vs_similarity_results
4
5 # 
↪-----
6 # Information for processing data
7 #
8 # The path to where the genetic algorithm was run from (the directory where Run.py is 
↪found).
9 path_to_ga_trial = '.'
10 # The rCut value for the common neighbour analysis, in units of Angstroms
11 rCut = 1.355 # Angstroms
12 # The ase.Atoms object of the cluster to compare all clusters from the genetic 
↪algorithm to.
13 # In this example, we are comparing all clusters to the LJ98 global minimum.
14 clusters_to_compare_against = read('LJ98_GM.xyz')
15 # This is the calculator used to initially locally minimise ONLY the clusters_to_ 
↪compare_against cluster
16 # just to make sure this cluster is a local minimum.
17 elements = [10]; epsilon = [1]; sigma = [1]; rCut_for_LJ_potential = 1000
18 calculator = LennardJones(elements, epsilon, sigma, rCut=rCut_for_LJ_potential, 
↪modified=True)
```

(continues on next page)

(continued from previous page)

```
19 # Specify the number of cores you want to use.
20 no_of_cpus = 1
21 #_
22 ↪=====
23 #_
24 ↪=====
25 # Information for plotting data
26 #
27 # This setting will create plots that plot the energy vs similarity over generations,
28 ↪including generation plots of the eras between epochs.
29 # This setting requires process_over_generations = True to be used.
30 make_epoch_plots = True
31 # This setting will create a video that shows how clusters were created per_
32 ↪generations.
33 # This setting requires process_over_generations = True to be used.
34 get_animations = True
35 # This setting will create a video that shows only the energies and similarities of_
36 ↪clusters in the population over generation.
37 # This setting requires process_over_generations = True to be used.
38 get_animations_do_not_include_offspring = True
39 # You can customise the unit that you give for the energy scale. For example, if you_
40 ↪are wanting to obtain energy vs similarity plots for Lennard-Jones clusters, you_
41 ↪may want to set this to 'LJ energy units'
42 energy_units = 'LJ energy units'
43 # This setting will indicate if you want to make svg files along with the png files_
44 ↪that are made during this program
45 make_svg_files = False
46 # This is the number of generations that will be shown per second (gps) in your_
47 ↪animation (if you choose to make animations of your genetic algorithm run.)
48 gps = 60
49 # You can also set the maximum amount of time that you would like your movie to run_
50 ↪in minutes. You only need to give a value either for gps or max_time.
51 max_time = None
52 # You can include a label in your animations that will count the number of generations_
53 ↪that have past
54 label_generation_no = True
55 # You can include a label in your animations that will count the number of times an_
56 ↪epoch occurs, i.e. will indicate the era value during the genetic algorithm
57 label_no_of_epochs = True
58 # place all these settings into the plotting_settings dictionary.
59 plotting_settings = {'make_epoch_plots': make_epoch_plots, 'get_animations': get_
60 ↪animations, 'get_animations_do_not_include_offspring': get_animations_do_not_
61 ↪include_offspring, 'energy_units': energy_units, 'make_svg_files': make_svg_files,
62 ↪'gps': gps, 'max_time': max_time, 'label_generation_no': label_generation_no,
63 ↪'label_no_of_epochs': label_no_of_epochs}
64 #_
65 ↪=====
66 #_
67 ↪=====
68 # Run the make_energy_vs_similarity_results program
69 make_energy_vs_similarity_results(path_to_ga_trial, rCut, clusters_to_compare_against,
70 ↪calculator, no_of_cpus=no_of_cpus, plotting_settings=plotting_settings)
71 #_
72 ↪=====
```

Lets go through each part of the `Run_energy_vs_similarity_script.py` file one by one to understand how to use it.

1) Variables for processing data

There are three variables required that determine how data from the genetic algorithm will be processed for making energy vs similarity plots. These are:

- **path_to_ga_trial** (*str.*): This is the path to the genetic algorithm data that you want to process. This will have the same path as your `Run.py` file.
- **rCut** (*float*): **This is the rCut value for the common neighbour analysis in Angstroms. This value determines which pairs of atoms are considered neighbours/bonded.**
 - If a pair of atoms have an interatomic distance less than or equal to rCut, that pair of atoms is considered neighbours/bonded.
 - If a pair of atoms have an interatomic distance greater than rCut, that pair of atoms is not considered neighbours/bonded.
- **clusters_to_compare_against** (*ase.Atoms or [list of ase.Atoms]*): These are all the clusters that you want to compare clusters to. Generally, you will only want to compared to genetic algorithm clusters to one reference cluster, such as the global minimum. However, if you need to do some checks, you can compare your genetic algorithm clusters to a few reference clusters. If you only want to give one reference cluster, assign **clusters_to_compare_against** to the `ase.Atoms` object for your cluster. If you have several reference clusters, put their `ase.Atoms` objects into a list. You can import most types of files as `ase.Atoms` object using the `ase.io.read` function. See [File input and output](#)³² to read more about the read function in ASE.
- **calculator** (*ase calculator*): If you want to locally minimise the cluster you gave for **clusters_to_compare_against** before this algorithms begins, set the calculator to the calculator you used in your genetic algorithm. This will not locally minimise any of the clusters that you created during the genetic algorithm, it will only be used to locally minimise **clusters_to_compare_against**. If you set the calculator to `None`, **clusters_to_compare_against** will not be locally optimised and will be used as is in this program. Default: `None`
- **no_of_cpus** (*int*): This is the number of cpus that are used to gather information that is used for making these energy vs similarity plots. Default: 1

An example of these parameters in `Run.py` is given below:

```
5 #_
6 # Information for processing data
7 #
8 # The path to where the genetic algorithm was run from (the directory where Run.py is_
  ↳ found).
9 path_to_ga_trial = '.'
10 # The rCut value for the common neighbour analysis, in units of Angstroms
11 rCut = 1.355 # Angstroms
12 # The ase.Atoms object of the cluster to compare all clusters from the genetic_
  ↳ algorithm to.
13 # In this example, we are comparing all clusters to the LJ98 global minimum.
14 clusters_to_compare_against = read('LJ98_GM.xyz')
15 # This is the calculator used to initially locally minimise ONLY the clusters_to_
  ↳ compare_against cluster
16 # just to make sure this cluster is a local minimum.
```

(continues on next page)

³² <https://wiki.fysik.dtu.dk/ase/ase/io/io.html?highlight=read#ase.io.read>

(continued from previous page)

```
17 elements = [10]; epsilon = [1]; sigma = [1]; rCut_for_LJ_potential = 1000
18 calculator = LennardJones(elements, epsilon, sigma, rCut=rCut_for_LJ_potential,
↳modified=True)
19 # Specify the number of cores you want to use.
20 no_of_cpus = 1
21 #
↳=====
```

2) Variables for plotting data

There are several variables required that determine how data from the genetic algorithm will be processed for making energy vs similarity plots. These are placed in the `plotting_settings` dictionary. These variables are:

- **make_epoch_plots** (*bool*): This plots the genetic algorithm over generations, as well as making plots over generations that are divided into era between epochs. Default = `False`.
- **get_animations** (*bool*): This will make a movie file of your energy vs similarity plots as they were made over generations if you set this to `True`. The offspring are included in this video. If you dont want these videos, set this to `False`. Default = `False`.
- **get_animations_do_not_include_offspring** (*bool*): This will make a movie file of your energy vs similarity plots as they were made over generations if you set this to `True`. The offspring are not included in this video. If you dont want these videos, set this to `False`. Default = `False`.
- **energy_units** (*str*): This variable allows you to give a custom unit for the energy the energy of your clusters are recorded in a units that is not eV. For example, if you are performing these plots on Lennard-Jones clusters, you may want to set this value to `'LJ energy units'`. Default: `'eV'`
- **make_svg_files** (*bool*): If this is set to `True`, this program will make svg files of plots that are created. These svg files allow the plots to be customised using programs like inkscape. If this is set to `False`, svg files of plots will not be created. Note that png files of plots are always created by this program no matter what you choose this setting to be. Default = `False`.

You can also set the animation variables in the `plotting_settings` dictionary. You only need to set either **gps** or **max_time** in this dictionary.

- **gps** (*int*): This is the number of generations that are filmed per second. This is equivalent to the frame per second or the rate rate of the animation. Default = 1.
- **max_time** (*float*): This is the maximum amount of time that your animations will run for. Default = `None`.
- **label_generation_no** (*bool*): If `label_generation_no` is set to `True`, the number of generations that have past will be shown. Default = `False`.
- **label_no_of_epochs** (*bool*): If `label_no_of_epochs` is set to `True`, the era value and the number of epochs that have occur will be labelled in your animation. Default = `False`.

IMPORTANT NOTE: In you give a value for `max_time` that is not `None`, this program will make sure that your movies only run for at most this amount of time. If you do not give a value for `max_time`, it will be set to `None` by default, which will tell the program to take your value of `gps` for the equivalent of the frames per second that the movie will run at.

An example of these parameters in `Run.py` is given below:

```
23 #
↳=====
24 # Information for plotting data
25 #
```

(continues on next page)

5.13.3 Data files that are made during this program

Within the `Similarity_Investigation_Data` folder that is created by this program are three txt files. There are:

- `energy_and_GA_data.txt`: This contains all the information about the clusters created during the genetic algorithm, including the generation when the cluster was created.
- `CNA_Profile_data.txt`: This contains all the total CNA profiles for each cluster created during the genetic algorithm as measured with the `rCut` value you gave
- `similarity_data_cluster_NNN.txt`: This contains all the similarity data for each cluster as compared to the cluster you gave in `clusters_to_compare_against`. There are a number of files given for each cluster that you are comparing in this program. `NNN` is the number given to each of your inputted reference clusters. This number is given in the order that you placed clusters in the `clusters_to_compare_against` list. If you only gave an `ase.Atoms` object, `NNN` will just be given as 1.

It is possible to restart this program if it fails midway through, or if you want to change one of the setting of your plots. These files are needed if you want to restart your program. **Do not delete these files if you want to restart this program.**

5.13.4 What plots do you get from the `make_energy_vs_similarity_results.py` program?

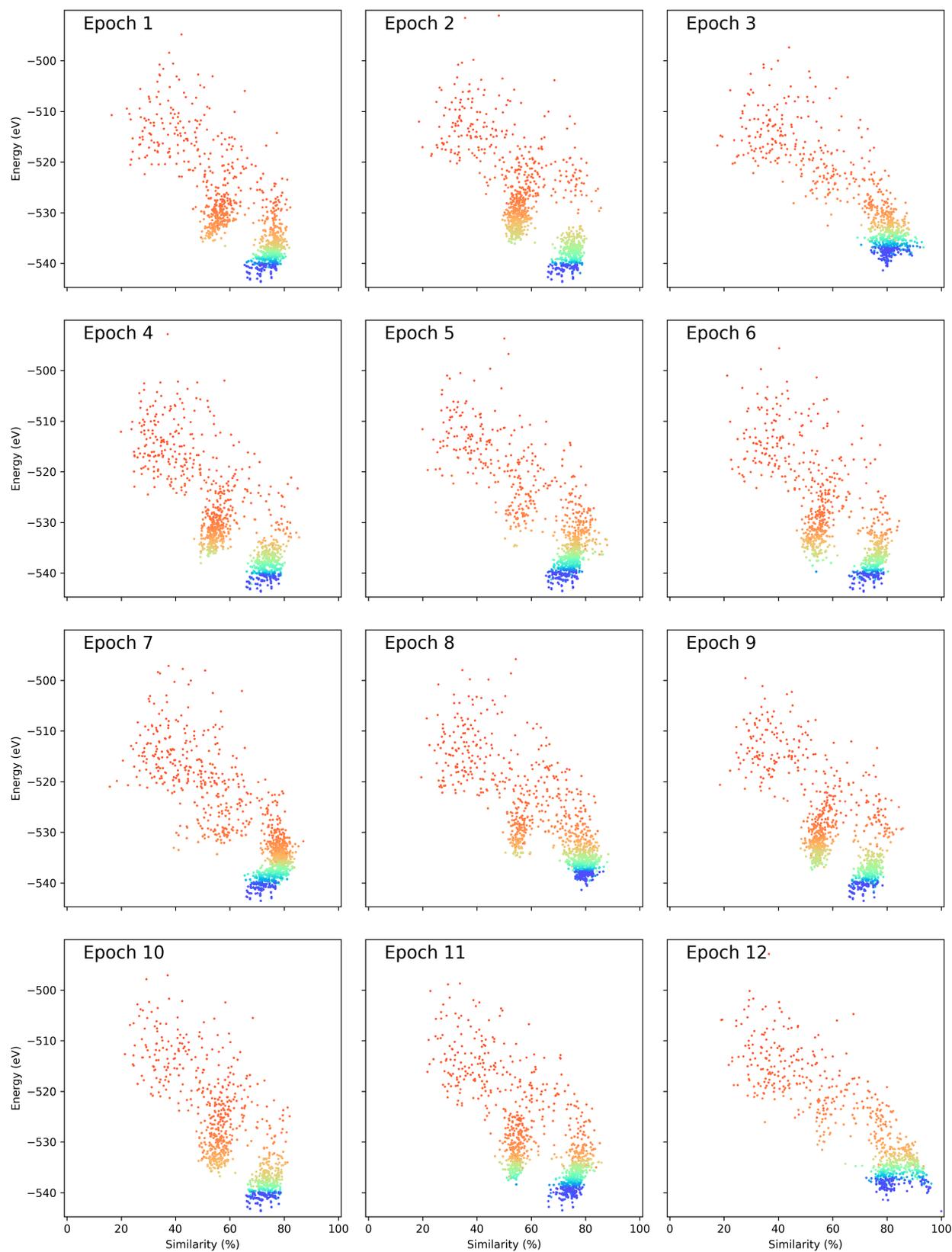
As well as the data files described above, this program also gives plots and movie files of your genetic algorithm in a folder inside the `Similarity_Investigation_Data` folder called `Ref_Cluster_NNN`, where `NNN` is the number given to each of your inputted reference clusters. This number is given in the order that you placed clusters in the `clusters_to_compare_against` list. If you only gave an `ase.Atoms` object, `NNN` will just be given as 1.

In this folder you will find the following plots and movies (depending on the settings you gave in the `plotting_settings` dictionary). There are (with examples for locally optimising a LJ98 cluster using the energy predation operator, energy fitness operator, and the population epoch operator where `nepoch = 1`):

```
if make_epoch_plots = True:
```

You will get two types of plots that contain each individual era (between epochs). Some examples are given below:

You will also get a collection of eras that are placed together on the same page so you can print them on a page together:



```
if get_animations = True:
```

This will create a video file that shows how the clusters changed over time in the population, including the offspring that are created in orange.

```
if get_animations_do_not_include_offspring = True:
```

This will create a video file that shows how the clusters changed over time in the population. This video only shows the change in the population over time without including offspring created in the video.

5.13.5 Troubleshooting possible issues that can arise

Here are some of the troubleshooting issue that have occurred in the past and possible troubleshooting solution to problem:

While ANALYSE CLUSTERS AND CNA PROFILES, each cluster are taking a very long time to process

I have found that sometimes the **Cluster Analysed:** does not show up while the program is running for some reason. However, something to check is what is being written to **energy_and_GA_data.txt** and **CNA_Profile_data.txt** in your **Similarity_Investigation_Data** folder and keep opening it up multiple time to see if new clusters are being written to these files.

If it is taking a while for a cluster to be written, take a look at the CNA profile that are being written to **CNA_Profile_data.txt**. The CNA profiles should contains many entries of signatures, like for example:

```
{'CNA_profile': [Counter({(4, 3, 3): 78, (4, 2, 2): 66, (5, 5, 5): 61, (3, 2, 2): 49, (2, 1, 1): 42, (5, 4, 4): 39, (3, 1, 1): 36, (2, 0, 0): 27, (4, 2, 1): 22, (3, 0, 0): 4, (1, 0, 0): 3, (4, 1, 1): 3})], 'name': 1}
```

If you have entries that look more like this

```
{'CNA_profile': [Counter({(113, 225, 208): 4753})], 'name': 1}
```

This is a sign that you have set your `rCut` value way to high, or somewhere in your `Run_energy_vs_similarity_script.py` script you have accidentally changed your `rCut` value to a very high value. Make sure that you set `rCut` to a value somewhere in between the first and second nearest neighbour values (ideally to half way between these values).

5.14 Other Helpful Programs for Gathering data and Post-processing Data

There are also a few other programs that have been designed that may be helpful for various reasons, but are not absolutely necessary for Postprocessing. In this article, we will introduce all these scripts, indicating what they do and how to use them. Some of these programs can be run by typing the program you want to run into the terminal from whatever directory you are in, but some of them you may need to copy to where you need to use them.

The scripts and programs that we will be mentioned here are:

- The `delALL.sh` command

5.14.1 What to make sure is done before running any of these scripts.

If you installed Organisms through pip3

If you installed the Organisms program with pip3, these scripts will be installed in your bin. You do not need to add anything into your `~/ .bashrc`. You are all good to go.

If you performed a Manual installation

If you have manually added this program to your computer (such as cloning this program from Github), you will need to make sure that you have included the `Helpful_Programs` folder into your `PATH` in your `~/ .bashrc` file. All of these program can be found in the *Helpful_Programs* folder. To execute some of these programs from the `Helpful_Programs` folder, you must include the following in your `~/ .bashrc`:

```
export PATH_TO_GA="<Path_to_Organisms>"
```

where `<Path_to_Organisms>`" is the path to get to the genetic algorithm program. Also include somewhere before this in your `~/ .bashrc`:

```
export PATH="$PATH_TO_GA"/Organisms/Helpful_Programs:$PATH
```

See more about this in *Installation of the Genetic Algorithm*.

5.14.2 The delALL command

The `delALL.sh` command will remove all the files that are created during a genetic algorithm. This command only removes the following files (see bash code below)

```
rm -rf GA_Run_Details.txt epoch_data epoch_data.backup ga_running.lock
rm -rf Population Recorded_Data Initial_Population Saved_Points_In_GA_Run Memory_
↳Operator_Data Diversity_Information
rm -rf __pycache__
```

Run this command by typing `delALL.sh` into the terminal where you have run your genetic algorithm trial (in the path where the `Run.py` file is).

5.15 Initialising a New Population

5.16 Using Predation Operators with the Genetic Algorithm

The predation operator is designed to remove offspring, or even swap offspring in for particular clusters in the population, that are too similar to other offspring or clusters in the population in some way. This is to prevent duplicates from entering into the population and prevent it from loosing predation.

5.16.1 Types of Predation Operators Available and How to Use Them

Several different predation operators have been implemented into this genetic algorithm. The predation operators that are available are:

- **Off:** No predation operator will be performed.
- **Energy Predation:** If two clusters have similar energies, one of those clusters will be removed.
- **IDCM-based Predation:** This operator will determine if two clusters are structurally identical.
- **SCM-based Predation:** This operator will determine if two clusters are structurally similar based on the structural comparison method, developed by the Garden group to improve the efficiency of global optimisation algorithms.

No Predation Operator

Yes, I know, why is “No Predation Operator” a predation operator. Its just a title. This option will not invoke any predation operator into your genetic algorithm run.

To use this, in your Run.py or MakeTrials.py script, set

```
Diversity_Information = {'Predation Operator': 'Off'}
```

Energy Predation Operator

This predation operator is designed to prevent the population containing two clusters with the same energy.

There are two different implementation of this predation operator written into this program. These are the:

- **Simple Energy Predation Operator**
- **Comprehensive Energy Predation Operator**

This is set in the `predation_information` dictionary with the 'mode' input. These are described below:

Simple Energy Predation Operator

This predation operator is designed to prevent the population containing two clusters with the same energy to some decimal place. For example, if cluster 1 has an energy of -186.2537935 eV and cluster 2 has an energy of -186.2482194 eV, and you have set the energy predation operator is set to round energies to 2 d.p., then these two clusters will be considered the same energy, and one of these clusters will be removed from the genetic algorithm.

To use this in your Run.py or MakeTrials.py script, will want to add two setting in the `predation_information` variable is required:

- **mode** (*str*): This variable should be set to 'simple'.
- **round_energy** (*int*): This is the decimal place to which energies will be compared to in the Energy Predation operator.

An example of how these settings are implimented into your Run.py or MakeTrials.py scripts is shown below:

```
Predation_Information = {'Predation Operator': 'Energy', 'mode': 'simple', 'round_
↪energy': 2}
```

Comprehensive Energy Predation Operator

This energy predation operator allows you to specify the minimum energy difference between clusters in the population. All clusters in the population must have energies that differ by greater or equal to this minimum energy difference.

This operator works as follows:

- **Initialisation of Population:** Every cluster in the population must differ in energy by `energy_difference`. If two or more clusters have the same energy, one of these will be kept and the others removed. This will give vacant sites in the population for the genetic algorithm to repopulate with other clusters more suited to the comprehensive energy predation operator.
- **During the Genetic Algorithm:** After all the offspring are created, the offspring are:
 - compared to other offspring to see if they have an energy difference less than the minimum energy difference. The offspring that are removed will again depend either on the energy or the fitness value of the offspring.
 - compared to clusters in the population to see if they have an energy difference less than the minimum energy difference. Here, the offspring may be removed, or swapped with a cluster from the population. This will depend either on the energy or the fitness value of the clusters.

The clusters that are kept or removed will depend on the setting given for the variable `'type_of_comprehensive_operator'`. This can be set to either `'energy'`, or `'fitness'`.

- If `type_of_comprehensive_operator = 'energy'`, clusters will be removed or replaced based on their energy. Clusters are more likely to be kept if they have a lower energy, and clusters with higher energies are more likely to be removed or replaced.
- If `type_of_comprehensive_operator = 'fitness'`, clusters will be removed or replaced based on their fitness value. Clusters are more likely to be kept if they have a higher fitness, and clusters with lower fitnesses are more likely to be removed or replaced.

To use this predation operator in your `Run.py` or `MakeTrials.py` script, you will want to add three setting in the `predation_information` variable:

- **mode** (*str*): This variable should be set to `'comprehensive'`.
- **minimum_energy_diff** (*float*): This is the difference in energy that any two clusters in the population can be between each other (in eV).
- **type_of_comprehensive_operator** (*str*): This variable determines how clusters are kept and removed from the genetic algorithm.
 - Set `type_of_comprehensive_operator = 'energy'` if you want clusters to be kept, replaced, or removed based on their energy, or
 - Set `type_of_comprehensive_operator = 'fitness'` if you want clusters to be kept, replaced, or removed based on their fitness values.

An example of how these settings are implemented into your `Run.py` or `MakeTrials.py` scripts is shown below:

```
Predation_Information = {'Predation Operator': 'Energy', 'mode': 'comprehensive',  
↪ 'minimum_energy_diff': 0.025, 'type_of_comprehensive_operator': 'energy'}
```

IDCM-based Predation Operator

The Interatomic Distance Comparison Method (IDCM) based predation operator is designed to remove clusters that are structurally identical to other clusters in the population or in the offspring set. The implementation of this predation operator will measure all the distances between every atom in a cluster to give a list of distances between atoms in the cluster. This list is sorted from shortest to longest distance. If all elements of both lists differ by $< X \%$, then the clusters are considered structurally identical. This predation operator is based on the predation operator from J. A. Vargas, F. Buendía, M. R. Beltrán, *J. Phys. Chem. C*, 2017, 121, 20, 10982-10991³³.

This operator works as follows:

- **Initialisation of Population:** Every cluster in the population must not be structurally identical to one another. If this is the case, the fitter cluster will be kept while the less fitter clusters will be removed.
- **During the Genetic Algorithm:** After all the offspring are created, the offspring are:
 - compared to other offspring to see if they structurally identical to one another. The fittest offspring is kept and the other less fit offspring are removed.
 - compared to cluster in the population to see if they structurally identical to one another. If the cluster in the population has the higher fitness, all the less fit offspring will be removed. If the offspring is the fitter cluster, it will be swapped into the population at the expense of the less fit cluster in the population.

To use this predation operator in your Run.py or MakeTrials.py script, you will want to add three setting in the `predation_information` variable is required:

- **percentage_diff** (*float*): This is the value $X \%$ in the description above. If all elements of both lists differ by $< \text{'percentage_diff' } \%$, then the clusters are considered structurally identical.

An example of how these settings are implemented into your Run.py or MakeTrials.py scripts is shown below:

```
predation_information = {'Predation Operator': 'IDCM', 'percentage_diff': 5.0}
```

SCM-Based Predation Operator

The Structural Comparison Method (SCM) based predation operator is based on the structural comparison method (SCM), that is designed to identify if two clusters are structurally similar. Two clusters are classed in to one of three similarity classes: Class I (structurally identical or geometrically similar), class II (structurally different, but are of the same structural motif) or class III (structurally different, and are of different motifs). See more about how the SCM works at *The Structural Comparison Method*. This method works as follows:

This operator works as follows:

- **Initialisation of Population:** Every cluster in the population must not be structurally identical or geometrically similar to one another (of class I similarity). If this is the case, the fitter cluster will be kept while the less fitter clusters will be removed.
- **During the Genetic Algorithm:** After all the offspring are created, the offspring are:
 - compared to other offspring to see if they structurally identical or geometrically similar to one another. The fittest offspring is kept and the other less fit offspring are removed.
 - compared to cluster in the population to see if they structurally identical or geometrically similar to one another. If the cluster in the population has the higher fitness, all the less fit offspring will be removed. If the offspring is the fitter cluster, it will be swapped into the population at the expense of the less fit cluster in the population.

There are two forms of the SCM that can be used in this implementation of the genetic algorithm. These are:

³³ <https://pubs.acs.org/doi/10.1021/acs.jpcc.6b12848>

- **The Total Structural Comparison Method (T-SCM):** This method will tally up the abundances of all the CNA signatures, across all the atoms in a cluster. The method will then compare the total abundances of two clusters using the Jaccard similarity index to get the structural similarity between these two clusters.
- **The Atomic Structural Comparison Method (A-SCM):** This method will compare the number of atoms that have an equal number of the same atomic CNA signatures between two clusters. The similarity between the clusters is based on the number of CNA equivalent atoms between the two clusters.

To use this predation operator in your Run.py or MakeTrials.py script, you will want to add three settings in the `predation_information` variable is required:

- **CNA scheme (str):** This is the type of CNA scheme you would like to use, be it the The Total Structural Comparison Method (T-SCM) or the The Atomic Structural Comparison Method (A-SCM).

The CNA required the user to input a value of `rCut`, a cutoff value that specifies the maximum distance between atoms to be considered neighbours or “bonded”. There are two ways that this can be specified in the `predation_information` variable. If you want to sample just one value of `rCut`, the variable you want to add is:

- **rCut (float):** This is a single cutoff value to be used by the SCM to get the similarity between two clusters. Given in .

If you want the similarity between two clusters to be sampled over a range of `rCut` values, use the following inputs:

- **rCut_low (float):** This is the minimum cutoff distance that the SCM will sample. Given in .
- **rCut_high (float):** This is the maximum cutoff distance that the SCM will sample. Given in .
- **rCut_resolution (float or int):** This specifies the cutoff distances that the SCM will sample. If this is given as a *float*, then this value describes the distance between the consecutive `rCut` values that will be sampled. E.g. if `rCut_low = 2.1`, `rCut_high = 3.4`, and `rCut_resolution = 0.2`, then the cutoff values that will be sampled are 2.1, 2.3, 2.5, 2.7, 2.9, 3.1 and 3.3. If this is given as a *int*, then this value will describe the number of `rCut` values that will be sampled. E.g. if `rCut_low = 2.4`, `rCut_high = 3.4`, and `rCut_resolution = 101`, then the cutoff values that will be sampled are 2.1, 2.11, 2.12, 2.13, 2.14, . . . , 3.37, 3.38, 3.39, 3.4.

You can also give the `rCut` settings in terms of the nearest neighbour distances relative to the lattice constant. In this case you must give the `lattice_constant`:

- **lattice_constant (float):** This is the lattice constant of your metal/element in the bulk. Given in .

If you want to sample the CNA at one value, give that single value in terms of nearest neighbour units:

- **single_nn_measurement (float):** This is a single nearest neighbour value to be used by the SCM to get the similarity between two clusters. The `rCut` value is then given as `fnn_distance * single_nn_measurement`. This value must be between 1.0 and 2.0. Given in nearest neighbour distance units.

Note that `fnn_distance` is the first nearest neighbour distance, given as `fnn_distance = lattice_constant / (2.0 ** 0.5)`. If you want the similarity between two clusters to be sampled over a range of `rCut` values, use the following inputs:

- **nn_low (float):** This is the minimum nearest neighbour distance that the SCM will sample. The minimum `rCut` value that will be sampled is then given as `fnn_distance * single_nn_measurement`. This value must be between 1.0 and 2.0. Given in nearest neighbour distance units.
- **nn_high (float):** This is the maximum nearest neighbour distance that the SCM will sample. The maximum `rCut` value that will be sampled is then given as `fnn_distance * single_nn_measurement`. This value must be between 1.0 and 2.0. Given in nearest neighbour distance units.
- **nn_resolution (int):** This specifies the number of `rCut` values you would like to sample. For example, if you set `nn_low = 1.2`, `nn_high = 1.6`, and `nn_resolution = 41`, then the cutoff values that will be sampled are 1.2, 1.21, 1.22, 1.23, . . . , 1.58, 1.59, 1.60.

An example of how these settings are implemented into your Run.py or MakeTrials.py scripts is shown below:

```
predation_information = {'Predation Operator': 'SCM', 'CNA scheme': 'T-SCM', 'rCut_
↪high': 3.2, 'rCut_low': 2.9, 'rCut_resolution': 0.05}
```

If you want to perform your SCM predation operator on gold (with a lattice constant of 4.07) sampling 78 points between the $1 + 1/3$ n.n.d and $1 + 2/3$ n.n.d (where n.n.d is the nearest neighbour distance), This is how you would enter this into your Run.py or MakeTrials.py script:

```
predation_information = {'Predation Operator': 'SCM', 'CNA scheme': 'T-SCM', 'lattice_
↪constant': 4.07, 'nn_high': 1.0 + (2.0/3.0), 'n_low': 1.0 + (1.0/3.0), 'nn_
↪resolution': 78}
```

5.16.2 Writing Your Own Predation Operators for the Genetic Algorithm

It is possible to write your own predation operators to incorporate into this genetic algorithm program. How fun is that! (I am writing this while on a plane jetlagged, apologies for my enthusiasm). To do this, you will need to write a python script that starts with the following:

```
from Organisms.GA.Predation_Operators.Predation_Operator import Predation_Operator

class Sample_Predation_Operator(Predation_Operator):
    def __init__(self, predation_information, population, print_details):
        super().__init__(predation_information, population, print_details)

    def check_initial_population(self, return_report=False):
        # algorithm to check the initial population
        if return_report:
            return clusters_to_remove, report
        else:
            return clusters_to_remove

    def assess_for_violations(self, offspring_pool, force_replace_pop_clusters_with_
↪offspring):
        # algorithm to check for violations between clusters in the_
↪population and the offspring
        return offspring_to_remove, force_replacement
```

In this Sample_Predation_operator, you will want to enter the following for each definition.

- `__init__(self, predation_information, population)`: This is the initialisation function.
 - `predation_information (dict)`: contains all the information that the predation operator needs.
 - `population (Organisms.GA.Population)`: is the population that the predation operator will focus on monitoring.
 - `print_details (bool)`: This indicates if the user wants the algorithm to print out the details of what the predation operator is doing during the genetic algorithm.
- `check_initial_population(self, return_report=False)`: This definition is responsible for making sure that the initialised population obeys the predation operator.
 - `return_report (bool)`: indicates if a report on the clusters that were removed is needed.
 - `clusters_to_remove (list)`: indicated which clusters to remove from the population. This is given as a list in the form: [index position of cluster in the population, the name of the cluster].

- `report (dict)`: This indicates what clusters have violated the predation operator, and the clusters in the population that it is similar to. This is given as a dictionary in the form: {name of cluster to remove: [names of all the other cluster that this cluster is similar to (i.e. why this cluster violates the predation operator)]}
- `assess_for_violations (self, offspring_pool, force_replace_pop_clusters_with_offspring)`: This definition is designed to determine which offspring (and the clusters in the population) violate the predation operator during a generation. It will not remove or change any clusters in the offspring or population, but instead will record which offspring violate the predation operator. It will also recommend if it is beneficial to force replace a cluster in the population with a higher fitness offspring.
 - `offspring_pool (Organisms.GA.Offspring_Pool)`: The offspring to check against the population for violations against this predation operator
 - `offspring_to_remove (list)`: This gives a list of all the offspring to be removed from the `offspring_pool` due to violating the predation operator. This is a list with the form: [name of offspring to be remove, index of the offspring in the `offspring_pool` to be remove]
 - `force_replace_pop_clusters_with_offspring (bool)`: This will tell the genetic algorithm whether to swap clusters in the population with offspring if the predation operator indicates they are the same but the predation operator has a better fitness value than the cluster in the population.
 - `force_replacement (list)`: This gives a list of clusters in the offspring that, while violating the predation operator, have a higher fitness than their counterpart cluster in the population. Therefore, it is recommended to replace the cluster in the population with the offspring. This is a list with the form: (name of cluster in the population to remove, name of offspring to replace with)

5.17 Using Fitness Operators with the Genetic Algorithm

The fitness operator is designed to assign the fitness values to each cluster created during the genetic algorithm. This allows one to invoke different ways for the genetic algorithm to explore the potential energy surface for a particular cluster. The fitness is used for the following reasons:

- To determine the likelihood of clusters being mated together during the crossover procedure (during the offspring creation process).
- To determine which clusters are removed during the predation process.
- To determine which clusters survive during the natural selection process.

5.17.1 Types of Fitness Operators Available and How to Use Them

Two types of fitness operators have been implemented into this genetic algorithm. The fitness operators that are available are:

- **Energy**: Here, the fitness of a cluster is determined by its energy as well as the energy of the lowest energetic and highest energetic cluster in the population and offspring pool.
- **Structure + Energy**: In this operator, the fitness of a cluster is obtained from the energy of the cluster (obtained using the **Energy** fitness operator) as well as from the similarity of the cluster, as obtained using the structural comparison method (SCM).

Energy Fitness Operator

This operator gives each cluster a fitness value by taking the energy of the cluster, and the energy of the highest and lowest energetic clusters in the current population and offspring pool. The only input require are the settings for the 'fitness_function', which will be explained in *Fitness Functions*. An example of how these settings are implimented into your Run.py or MakeTrials.py scripts is shown below:

```
energy_fitness_function = {'function': 'exponential', 'alpha': 3.0}
fitness_information = {'Fitness Switch': 'Energy', 'fitness_function': energy_fitness_
↳function}
```

Structure + Energy Fitness Operator

This operator works by taking the energy of the cluster (as well as the energy of the highest and lowest energetic clusters in the current population and offspring pool) and the representative similarity of the cluster, and converting these into a fitness value for the cluster. Read more about how the SCM works at the section on *The Structural Comparison Method*.

The representative similaity is obtained as follows: Every cluster in the population and in the offspring pool will have an associated average similarity associated to them. For each cluster, all the of the average similarities that cluster has with every other cluster are placed in a list, and the maximum average similarity from the list is taken as that clusters representative similaity.

The input parameters required for this fitness operator are:

- **SCM scheme** (*str*): This is the type of SCM scheme you would like to use. This can be either Total Comparison Structural Comparison Method (T-SCM) or the Atom-by-Atom Comparison Structural Comparison Method (A-SCM). Read more about this at the *SCM Based Predation Operator*.
- **SCM_fitness_contribution** (*float*): This is the relative contribution of the SCM fitness value to the overall value. The energetic fitness contribution is 1 - 'SCM_fitness_contribution'.
- **Take from the collection of a clusters similarities** (*str*): This determines how you want to process all the similarities that are associated with a cluster in the population and offspring. This can be set to either 'Maximum' or 'Average'. See *Take from the collection of a cluster's similarities* for more information. Default: 'Maximum'.
- **normalise_similarities** (*bool*): This will tell the SCM + Energy Fitness Operator whether to normalise the similarity or not. Normalising the similarities means taking the structural similarity, subtracting it from the minimum similarity of clusters in the population , and dividing it by the maximum similarity minus minimum similarity of clusters in the population. True means to normalise, False means do not normalise the similarities. See *Normalise Similarities* for more information. Default: False.
- **Dynamic Mode** (*bool*): This will set the operator into dynamic mode. See more about this at *Dynamic Mode*.
- **energy_fitness_function** (*Organisms.GA.Fitness_Scripts.Fitness_Function*): This describes how the fitness is obtained for the energetic fitness value. More about fitness functions can be read at *Fitness Functions*.
- **SCM_fitness_function** (*Organisms.GA.Fitness_Scripts.Fitness_Function*): This describes how the fitness is obtained for the SCM-based fitness value. More about fitness functions can be read at *Fitness Functions*.

If you are using the SCM Based Diversity operator as well, you can also include the following inputs:

- **Use Predation Information** (*bool*): If this is set to True, the Structure + Energy fitness operator will use the same rCut parameters as the SCM-based predation operator. Do not include this in the *predation_information*, or set this to False, if you want to use different values of rCut for the SCM + Energy fitness operator or are not using the SCM-based Predation Operator. default: False

If you need to set the rCut values, you can enter this in two ways. If you want to sample just one value of rCut, the variable you want to add is:

- **rCut** (*float*): This is a single cutoff value to be used by the SCM to get the similarity between two clusters. Given in .

If you want the similarity between two clusters to be sampled over a range of rCut values, use the following inputs:

- **rCut_low** (*float*): This is the minimum cutoff distance that the SCM will sample. Given in .
- **rCut_high** (*float*): This is the maximum cutoff distance that the SCM will sample. Given in .
- **rCut_resolution** (*float*): This specifies the cutoff distances that the SCM will sample. If this is given as a float, then this value describes the distance between the consecutive rCut values that will be sampled. E.g. if rCut_low = 2.1, rCut_high = 3.4, and rCut_resolution = 0.2, then the cutoff values that will be sampled are 2.1, 2.3, 2.5, 2.7, 2.9, 3.1 and 3.3. If this is given as a int, then this value will describe the number of rCut values that will be sampled. E.g. if rCut_low = 2.4, rCut_high = 3.4, and rCut_resolution = 101, then the cutoff values that will be sampled are 2.1, 2.11, 2.12, 2.13, 2.14, ..., 3.37, 3.38, 3.39, 3.4.

You can also give the rCut settings in terms of the **nearest neighbour distances relative to the lattice constant**. In this case you must give the lattice_constant:

- **lattice_constant** (*float*): This is the lattice constant of your metal/element in the bulk. Given in .

If you want to sample the CNA at one value, give that single value in terms of nearest neighbour units:

- **single_nn_measurement** (*float*): This is a single nearest neighbour value to be used by the SCM to get the similarity between two clusters. The rCut value is then given as fnn_distance * single_nn_measurement. This value must be between 1.0 and 2.0. Given in nearest neighbour distance units.

Note that fnn_distance is the first nearest neighbour distance, given as `fnn_distance = lattice_constant / (2.0 ** 0.5)`. If you want the similarity between two clusters to be sampled over a range of rCut values, use the following inputs:

- **nn_low** (*float*): This is the minimum nearest neighbour distance that the SCM will sample. The minimum rCut value that will be sampled is then given as fnn_distance * single_nn_measurement. This value must be between 1.0 and 2.0. Given in nearest neighbour distance units.
- **nn_high** (*float*): This is the maximum nearest neighbour distance that the SCM will sample. The maximum rCut value that will be sampled is then given as fnn_distance * single_nn_measurement. This value must be between 1.0 and 2.0. Given in nearest neighbour distance units.
- **nn_resolution** (*int*): This specifies the number of rCut values you would like to sample. For example, if you set nn_low = 1.2, nn_high = 1.6, and nn_resolution = 41, then the cutoff values that will be sampled are 1.2, 1.21, 1.22, 1.23, ..., 1.58, 1.59, 1.60.

Three examples of how these settings are implemented into your Run.py or MakeTrials.py scripts are shown below. First, if you have not used the SCM-based predation operator, or you are using the SCM-based predation operator but sampling different values of rCut, an example of fitness_information is given below.

```
fitness_information = {'Fitness Operator': 'Structure + Energy', 'CNA scheme': 'T-SCM
↪', 'rCut_high': 3.2, 'rCut_low': 2.9, 'rCut_resolution': 0.05, 'SCM_fitness_
↪contribution': 0.5, 'normalise_similarities': False, 'Dynamic Mode': False, 'energy_
↪fitness_function': energy_fitness_function, 'SCM_fitness_function': SCM_fitness_
↪function}
```

If you want to perform your SCM fitness operator on gold (with a lattice constant of 4.07) sampling 78 points between the 1 + 1/3 n.n.d and 1 + 2/3 n.n.d (where n.n.d is the nearest neighbour distance), This is how you would enter this into your Run.py or MakeTrials.py script:

```
fitness_information = {'Fitness Operator': 'Structure + Energy', 'CNA scheme': 'T-SCM
↪', 'lattice_constant': 4.07, 'nn_high': 1.0 + (2.0/3.0), 'nn_low': 1.0 + (1.0/3.0)
↪', 'nn_resolution': 78, 'SCM_fitness_contribution': 0.5, 'normalise_similarities':
↪False, 'Dynamic Mode': False, 'energy_fitness_function': energy_fitness_function,
↪SCM_fitness_function': SCM_fitness_function}
```

(continued from previous page)

If you are using the SCM-based predation operator and sampling the same values of rCut, you can set Use Predation Information = True and negate writing in the same values for rCut. An example is given below:

```
fitness_information = {'Fitness Operator': 'Structure + Energy', 'CNA scheme': 'T-SCM
↪', 'Use Predation Information': True, 'SCM_fitness_contribution': 0.5, 'normalise_
↪similarities': False, 'Dynamic Mode': False, 'energy_fitness_function': energy_
↪fitness_function, 'SCM_fitness_function': SCM_fitness_function}
```

Take from the collection of a cluster's similarities

When obtaining the value of $\sigma_{SCM}(x)$ for cluster x , you take the collection of all σ values between cluster x and every other cluster in the population and offspring, and you perform some sort of mathematical operation upon this collection of σ values to obtain $\sigma_{SCM}(x)$. There are two settings for this:

If you set 'Take from the collection of a clusters similarities' in the fitness_information dictionary to 'Maximum', then you will take the maximum value of σ_{xy} between the x^{th} cluster and every other cluster in the population (including offspring)

$$\sigma_{SCM}(x) = \max\{\sigma_{xy} | y = 1, \dots, n_{total}, y \neq x\}$$

where n_{total} is the total number of clusters in the population (including offspring). 'Maximum' is the default setting for this setting in the fitness_information dictionary.

If you set 'Take from the collection of a clusters similarities' in the fitness_information dictionary to 'Average', then you will take the mean value of σ_{xy} between the x^{th} cluster and every other cluster in the population (including offspring)

$$\sigma_{SCM}(x) = \text{mean}\{\sigma_{xy} | y = 1, \dots, n_{total}\}$$

where n_{total} is the total number of clusters in the population (including offspring).

Normalise Similarities

The similarity obtained from the SCM is used to obtain the structural fitness values for the clusters in the population. To do this, the algorithm obtains the $\rho_{SCM}(x)$ for the x^{th} cluster in the population, which is the translated into the structural fitness value, $f_{SCM}(x)$ for the x^{th} cluster. The value of $\rho_{SCM}(x)$ can be obtained in two ways.

First, the unnormalised similarity can be used, where the x^{th} cluster's similarity is divided by 100 to give the similarity as a decimal, which is between 0 and 1.

$$\rho_{SCM}(x) = \frac{\sigma_{SCM}(x)}{100}$$

Second, the similarity can be normalised. Here, the maximum and minimum similarities of all cluster in the population, including offspring, are obtained (referred to as $\sigma_{SCM,max}$ and $\sigma_{SCM,min}$). $\rho_{SCM}(x)$ for the x^{th} cluster is then obtained as below

$$\rho_{SCM}(x) = \frac{\sigma_{SCM}(x) - \sigma_{SCM,min}}{\sigma_{SCM,max} - \sigma_{SCM,min}}$$

Dynamic Mode

To be developed.

Fitness Functions

In this implementation of the genetic algorithm, there are a few different functions that one can use to convert an energy or a similarity value into a fitness value. You can find more information about these fitness functions in R. L. Johnston, Dalton Trans., 2003, 4193-4207³⁴

Exponential Function

This will use an exponential function to obtain the fitness value.

$$f(i) = e^{-\alpha\rho(i)}$$

The input required is the value of α

An example of the input for this function is shown below.

```
energy_fitness_function = {'function': 'exponential', 'alpha': 3.0}
```

Hyperbolic Tangent Function

This will use a hyperbolic tangent function to obtain the fitness value.

$$f(i) = \frac{1}{2}[1 - \tanh(2\rho(i) - 1)]$$

An example of the input for this function is shown below.

```
energy_fitness_function = {'function': 'tanh'}
```

Linear Function

This will use a linear function to obtain the fitness value.

$$f(i) = \text{gradient} \times \rho(i) + \text{constant}$$

The input required is the value of gradient and constant

An example of the input for this function is shown below.

```
energy_fitness_function = {'function': 'linear', 'gradient': 0.5, 'constant': 0.5}
```

³⁴ <https://pubs-rsc-org.ezproxy.otago.ac.nz/en/content/articlelanding/2003/dt/b305686d#!divAbstract>

Direct Function

This will use a direct function to obtain the fitness value.

$$f(i) = \rho(i)$$

An example of the input for this function is shown below.

```
energy_fitness_function = {'function': 'direct'}
```

5.17.2 Writing Your Own Fitness operators for the Genetic Algorithm

It is possible to write your own fitness operators to incorporate into this genetic algorithm program. To do this, you will need to write a python script that has the following:

```
from Organisms.GA.Fitness_Operators.Fitness_Operator import Fitness_Operator
from Organisms.GA.Fitness_Operators.Fitness_Function import Fitness_Function

class Sample_Fitness_Operator(Fitness_Operator):

    def __init__(self, fitness_information, predation_operator, population, print_
↳details):

        def assign_initial_population_fitnesses(self):

            def assign_resumed_population_fitnesses(self, resume_from_generation):

                def assign_all_fitnesses_before_assess_against_predation_operator(self, all_
↳offspring_pools, current_generation_no):

                    def assign_all_fitnesses_after_assess_against_predation_operator(self, all_
↳offspring_pools, current_generation_no, offspring_to_remove):

                        def assign_all_fitnesses_after_natural_selection(self, current_generation_no):
```

In this `Sample_Fitness_Operator`, you will want to enter the following for each definition.

- `__init__(self, fitness_information, predation_operator, population, print_details)`: This is the initialisation function.
 - `fitness_information (dict)`: Contains all the information that the fitness operator needs.
 - `predation_operator (Organisms.GA.Predation_Operators.Predation_Operator)`: This is the predation operator that is being used in the genetic algorithm.
 - `population (Organisms.GA.Population)`: Is the population that the predation operator will focus on monitoring.
 - `print_details (bool)`: This indicates if the user wants the algorithm to print out the details of what the predation operator is doing during the genetic algorithm.
- `assign_initial_population_fitnesses(self)`: This assigns the fitnesses to the clusters in the initial population.
- `assign_resumed_population_fitnesses(self, resume_from_generation)`: This assigns the fitnesses to the clusters in the population that has been resumed.
 - `resume_from_generation (int)`: The number of the generation that the genetic algorithm is being resumed from.

- `assign_all_fitnesses_before_assess_against_predation_operator(self, all_offspring_pools, current_generation_no)`: This will assign fitness to the clusters in the population and the offspring before the predation operator has been performed for this generation.
 - `all_offspring_pools` (*Organisms.GA.Offspring_Pool* or a list of *Organisms.GA.Offspring_Pool*): The offspring_pool
 - `current_generation_no` (*int*): The current generation.
- `assign_all_fitnesses_after_assess_against_predation_operator(self, all_offspring_pools, current_generation_no, offspring_to_remove)`: This will assign fitness to the clusters in the population and the offspring after the predation operator has been performed for this generation.
 - `all_offspring_pools` (*Organisms.GA.Offspring_Pool* or a list of *Organisms.GA.Offspring_Pool*): The offspring_pool
 - `current_generation_no` (*int*): The current generation.
 - `offspring_to_remove` (*list of ints*): This is a list of the names of the clusters that will be removed. This is currently not needed, but kept as an input variable just in case it is needed in the future.
- `assign_all_fitnesses_after_natural_selection(self, current_generation_no)`: This will assign all the fitnesses to all clusters in the population after performing the natural selection process
 - `current_generation_no` (*int*): The current generation.

5.18 The Structural Comparison Method (SCM)

The structural comparison method (SCM) was created and designed to identify the structural similarity between various clusters. There are two versions of the implementation of this algorithm. These are the Total Structural Comparison Method (T-SCM), and the Atom-by-Atom Structural Comparison Method (A-SCM).

Both of these methods use the Common Neighbour Analysis (CNA) to provide a description of the structure of a cluster. The CNA is a tool designed to describe all the local structural environments about a cluster. It does this by assigning signatures that describe the number of neighbouring atoms between two neighbouring (or bonded) atoms. More information about the CNA can be found at D Faken, H Jónsson, *Comput. Mater. Sci.*, 1994, 2, 279-286³⁵, while examples of the CNA can be found at N. Lümmen and T. Kraska, *Model. Simul. Mater. Sci. Eng.*, 2007, 15, 3,³⁶.

An important note to take about the CNA is that the user is required to provide a value of `rCut`, which is the maximum distance between two atoms for those two atoms to be considered neighbours (or bonded).

These provide a list of CNA signatures, and the amount of each CNA signature in total or on each atom. A secondary method compares these lists (of CNA signatures) to provide a value of the structural similarity between two clusters. Performing the SCM over a range of `rCut` values allows the SCM to assign a similarity class to the comparison of those two clusters. The three classes are:

- **Class I**: Two structures are structurally identical or geometrically similar.
- **Class II**: Two structures are structurally different, but have the same motif.
- **Class III**: Two structures are structurally different, and are of different motifs.

For more information about the SCM, see XXX

³⁵ <https://notendur.hi.is/hj/papers/paperCNaanal.pdf>

³⁶ <https://iopscience.iop.org/article/10.1088/0965-0393/15/3/010>

5.18.1 The Total Structural Comparison Method (T-SCM)

This method will tally the number of each CNA signature across the whole cluster.

5.18.2 The Atomic Structural Comparison Method (A-SCM)

5.19 Using the Memory Operator

The following options for this operator are:

- **perform_memory_operator** (*bool*): Do you want to use this operator. Either `True` or `False`.
- **rCut** (*float*):
- **cut_off_similarity** (*float*):
- **SCM Type** (*str*):

5.20 Using Epoch Methods

An epoch method is designed to reset the population with a set of randomly generated clusters if the epoch method believes that the population has stagnated.

In this article, we describe the types of epoch schemes that are available, as well as other settings that are available that allow the epoch method to do other things

5.20.1 Types of Epoch Methods

There are three types of Epoch methods available. These are no epoch method, the mean energy epoch method, and the same population epoch method. For no epoch method, set `epoch_settings = {'epoch mode': None}`

Mean Energy Epoch Method

This method is designed to reset the population is the mean energy of clusters in the population does not decrease after a generation. To use this method, set `'epoch mode': 'mean energy'` in your `epoch_settings` dictionary.

To use this method, you need to also include the following setting in your `epoch_settings` dictionary.

- `'mean energy difference'`: The mean energy must decrease by at this this amount after every generation to avoid an epoch.

Same Population Epoch Method

This method is designed to reset the population if no offspring replace any of the clusters in the population after so many generations. To use this method, set 'epoch mode': 'same population' in your `epoch_settings` dictionary.

To use this method, you need to also include the following setting in your `epoch_settings` dictionary.

- 'max repeat': This is the number of generations that the population can stay the same without being epoched. If the population does not change after this many generations, the population will be epoched.

5.20.2 Other Settings

There are other settings that you can set in the epoch method.

- **first epoch changes fitness operator to energy fitness operator** (*bool*): The epoch method is designed to reset the population with a set of randomly generated clusters once the population stagnates. However, there is another option. This other option works as follows: When the population stagnates for the first time, the population does not epoch but instead the fitness operator changes to the energy fitness operator. It is only once the population stagnates for a second time that the epoch method will reset the population. This method only works if you have not set the fitness operator to the energy fitness operator or if you have selected the structure + energy fitness operator but have set `SCM_fitness_contribution` to `0.0`. (Default: `False`)

5.20.3 Epoch Method files

The epoch method will make a file called `epoch_data`, which contains all the information about the epoch for the current generation. There is also a file called `epoch_data.backup` which contains the epoch information about the last successful generation. The first line of these files is the generation that that epoch data is valid for. For the other lines, this information depends on the epoch method chosen.

Mean Energy Epoch Method

The second line gives the mean energy of the population from the last successful generation.

Same Population Epoch Method

The second line contains the list of the names of all the clusters in the last unique population. The third line indicates the recent number of generations that have had the same population.

5.21 Recording Clusters From The Genetic Algorithm

It is possible to record the types of clusters that are created during the genetic algorithm. In many cases, the user doesn't want to record all the clusters that are created by the genetic algorithm, but instead record the more important ones, such as the lowest energetic clusters. This algorithm is even designed to prevent recording replicas of clusters that are the same with respect to the diversity scheme you have chosen.

All of the parameters for this components are gathered together in a dictionary, called `ga_recording_information`. This is passed into a class called the `Recording_Cluster`.

These parameters are:

- `ga_recording_scheme` (*str*):

- 'None': Do not record any clusters
- 'All': Record all clusters that are made (however, the highest energy clusters may still be removed if you give a input for `limit_number_of_clusters_recorded` or `limit_size_of_database`).
- 'Limit_energy_height': Here, all clusters will be recorded, expect for those that have an energy that is higher than $Energy(LowestEnergyCluster) + EnergyHeight$ (or if clusters need to be removed based your input for `limit_number_of_clusters_recorded` or `limit_size_of_database`. In this case, the highest energy clusters will be removed). This scheme also requires the user to also input into the `ga_recording_information` dictionary:
 - * 'limit_energy_height_of_clusters_recorded': This is the value for *EnergyHeight*.
- 'Set_higher_limit': All clusters will be recorded if that have an energy lower than 'Set_higher_limit' (or if clusters need to be removed based your input for `limit_number_of_clusters_recorded` or `limit_size_of_database`. In this case, the highest energy clusters will be removed). This scheme also requires the user to also input into the `ga_recording_information` dictionary:
 - * 'upper_energy_limit': This is the upper energy limit. Only clusters with an energy lower than this will be recorded.
- 'Set_energy_limits': Clusters will be recorded if they have an energy between 'lower_energy_limit' and 'upper_energy_limit' (or if clusters need to be removed based your input for `limit_number_of_clusters_recorded` or `limit_size_of_database`. In this case, the highest energy clusters will be removed). This scheme also requires the user to also input into the `ga_recording_information` dictionary:
 - * 'lower_energy_limit': This is the lower energy limit. Only clusters with an energy higher than this will be recorded.
 - * 'upper_energy_limit': This is the upper energy limit. Only clusters with an energy lower than this will be recorded.
- **exclude_recording_cluster_screened_by_diversity_scheme** (*bool*): Once a cluster is created by the genetic algorithm, it will be put through a Diversity Scheme. After this, it may be deleted as it is found to be too simiar to another cluster (based on the criteria of that Diversity Scheme). This setting will determine weither to still record the cluster, even if it removed by the diversity scheme, or not to bother. If `True`, then we do not bother with recording it. If `False`, we will still consider recording if, even if it is removed by the Diversity Scheme. Default: `True`
- **limit_number_of_clusters_recorded** (*int*): This is the number of clusters that the user would like to limit themselves to recording. The clusters recorded will be the lowest energetic clusters obtained from that genetic algorithm run. This can be useful as to prevent GBs of data from being written, that might not necessary be useful. For example, if `limit_number_of_clusters_recorded = 50`, the 50 lowest energetic clusters obtained from the genetic algorithm will be recorded. Default: `None`
- **limit_size_of_database** (*str*): This is the maximum size that the database can get to in KB, MB, GB or TB. For example, enter in '250MB' if you want the maximum size to be 250 MB. If the database gets bigger than this, the higher energetic clusters in the database is deleted.
- **saving_points_of_GA** (*[int,...]*): This parameter is not so much for recording individual clusters, but to save the state of the genetic algorithm after a certain number of generations. This can be useful if the user would like to benchmarking the genetic algorithm in some way, by using the data from the genetic algorithm after some numbers of generations has been performed. The data that will be recorded are the population, the PoolProfile, the EnergyProfile, and the Recorded_Clusters folders. This variable is a list of integers, where each entry in the list tell the algorithm to record all the data of the genetic algorithm after that number of genetations. For example, `saving_points_of_GA = [2, 5, 10, 23]` indicates the genetic algorithm should record the all the data once 2, 5, 10 and 23 generations have occurred. Default: `None` (i.e. do not bother recording anything.)

- **record_initial_population** (*bool.*): If `True` (set if not specified) -> Create a folder called “Initial_Population” which contains the folders (including structural files) of the initial population). If `False` -> Do not do this. Default: `True`.
- **show_GA_Recording_Database_check_percentage** (*bool.*): if `True`: This will show a loading bar when the `GA_Recording_Database` is being checked to make sure that all cluster in the database were made before the current generation. This can sometimes take a while so this progress bar is to show that something is happening. If `False`, no process bar will be shown, but the `GA_Recording_Database` checking procedure will still run. Set to `True` if you are debugging the `GA_Recording_Database` component of this program. Set to `False` if you are running this through slurm or another job scheduling manager as this can fill up the `stderr` file and make it very large, hard to read and hard to open (because it is so large). Default: `False`.

TIME/EFFICIENCY ISSUES TO NOTE IF YOU USE `ga_recording_scheme = Limit_energy_height` OR SET `limit_number_of_clusters_recorded` TO SOME VALUE OR SET `limit_size_of_database` TO SOME VALUE.: If you choose to use the '`Limit_energy_height`' recording scheme and/or if you set `limit_number_of_clusters_recorded` to some value and/or set `limit_size_of_database` to some value, you may find that your Genetic algorithm slows down. This may occur especially if you set `limit_number_of_clusters_recorded` to some value and/or set `limit_size_of_database` to some value. Here, clusters can be removed from the cluster recording database. If the database gets quite large, this can take some time to do and can be prohibitive. Ultimately, if possible you should avoid using these settings to prevent any time issues that could arise. It is best to use `ga_recording_information['ga_recording_scheme'] = 'None' or 'All' or 'Set_higher_limit' or 'Set_energy_limits'`, and do not set `ga_recording_information['limit_number_of_clusters_recorded']` and `ga_recording_information['limit_size_of_database']` (or set `ga_recording_information['limit_number_of_clusters_recorded'] = None` and `ga_recording_information['limit_size_of_database'] = None`).

An example of how to enter these parameters into '`ga_recording_information`' for your `Run.py` or `Make-Trials.py` file is shown below:

```
ga_recording_information = {}
ga_recording_information['ga_recording_scheme'] = 'Limit_energy_height' # float('inf')
ga_recording_information['limit_number_of_clusters_recorded'] = 70 # float('inf')
ga_recording_information['limit_energy_height_of_clusters_recorded'] = 1.5 #eV
ga_recording_information['exclude_recording_cluster_screened_by_diversity_scheme'] = True
ga_recording_information['saving_points_of_GA'] = [50, 100, 150, 200]
ga_recording_information['record_initial_population'] = True
ga_recording_information['show_GA_Recording_Database_check_percentage'] = False
```

Note: If `ga_recording_information = {}` is all that is entered, the instance of `Recording_Cluster` will not record any clusters. The clusters that are recorded will be the lowest energetic structures that can be recorded. This will not take the fitness of the cluster into account, only the energy of the cluster.

Note that the `Recorded_Data/GA_Recording_Database.db` may become very big and hard to process on your computer with `ase db`. There is a program called `Postprocessing_Database.py` that is designed to break up the `GA_Recording_Database.db` database into smaller chunks if needed. See **Postprocessing_Database.py** - *For breaking a large database into smaller chunks* for more information on how to use this program.

See *Using Databases with the Genetic Algorithm* for more information about how to use databases in ASE.

5.22 Using Databases with the Genetic Algorithm

The genetic algorithm has been designed to create database files that are designed to hold data about the clusters that are created during the genetic algorithm. See [Databases in ASE](#)³⁷ for information about how databases generally work in ASE.

5.22.1 Databases in Organisms

There are a few databases that can be created during the genetic algorithm. First is the `Population\Population.db` database. This database is created to store clusters that are in the current population. This database is required by the genetic algorithm in case the user needs to resume the genetic algorithm. The user can also create the `Recorded_Data/GA_Recording_Database.db` database which is designed to save clusters that were created during the genetic algorithm. See *Recording Clusters From The Genetic Algorithm* for more information on how to record clusters during the genetic algorithm.

5.22.2 Opening ASE database website

ASE allows the user to view the database in a number of ways. See [ase db](#)³⁸ to see how to use this. One of these ways is using the database website viewer, which allows the user to view the clusters in the database in a very nice graphical user interface. To use this, write into the terminal `ase db -w name_of_database.db` and open up the website `http://0.0.0.0:5000/`.

In Organisms, metadata is saved to these databases that allows the information given in the ASE database website to be viewed easier. Due to a bug in ASE versions above 3.20.0, the metadata is not shown in ASE versions above 3.20.0.

Opening ASE database website in ASE versions above and equal to 3.20.0

We have created a program called `database_viewer.py` that allows the metadata to be included in all versions of ASE. This program is run by the user moving into the `Recorded_Data` folder in the terminal and running the `database_viewer.py` program. There is one parameter that need to be entered. This is:

- **name_of_database** (*str*): This is the name of the database that you want to view.

Enter this into the terminal when you type in `database_viewer.py`:

```
database_viewer.py name_of_database
```

The bug is not that the metadata is not included in the database, but the ASE database website does not use that metadata to make the website easier to read.

³⁷ <https://wiki.fysik.dtu.dk/ase/ase/db/db.html>

³⁸ <https://wiki.fysik.dtu.dk/ase/ase/db/db.html#ase-db>

Opening ASE database website in ASE versions below and equal to 3.19.3

There is no bug in ASE database website viewer. You can open databases with these versions of ASE by typing into the terminal

```
ase db -w name_of_database.db
```

where `name_of_database.db` is the database you want to include. The metadata is included in this database.

How to use the ASE database website

When you use `database_viewer.py` or `ase db -w` to open a ASE database website, you will see the database can be organised by Name, Cluster Energy, and ID. By default, the database in `Recorded_Data/GA_Recording_Database.db` is ordered by cluster energy

Genetic Algorithm Database

Search formula e.g. MoS2

Help with constructing advanced search queries ...
Toggle list of keys ...

Displaying rows 1-25 out of 2000

Name	Cluster Energy	ID
14339	-105.598	1
12660	-105.598	2
18718	-105.598	3
14622	-105.598	4
13319	-105.598	5
17500	-105.598	6
21199	-105.598	7
22131	-105.598	8
3576	-105.598	9
6171	-105.598	10
11281	-105.598	11
1479	-105.598	12
3218	-105.598	13
2772	-105.598	14
21547	-105.598	15

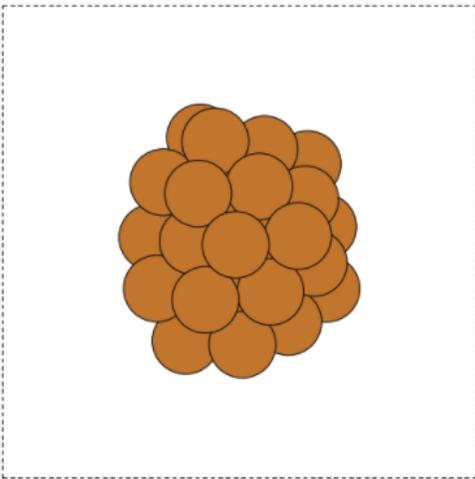
You can click on these to change the ordering. You can also click on add columns to add other pieces of data recorded in the database to the website. The `Toggle list of keys ...` indicates what all the data means.

If you click on one of the cluster, you can view all the information about that cluster and can view that cluster with a picture. You can also click on the **Miscellaneous** button to get more information about the cluster other than what is shown directly on the page.

If you click on **Open ASE's GUI**, you can get a interactive viewer for that cluster. You can also download the cluster as a `.xyz` file by clicking on **Download**. If you have ASE 3.20.0 or higher, you can also view this cluster on the website with `jmol`.

Cu₃₇

Basic properties



Open ASE's GUI

Download ▾ Unit cell ▾

Axis	x	y	z	Periodic
1	18.706	0.000	0.000	False
2	0.000	18.706	0.000	False
3	0.000	0.000	18.706	False
Lengths:	18.706	18.706	18.706	
Angles:	90.000	90.000	90.000	

Key	Value
Unique row ID	1
Mass	2351.202 au
Age	66.112 hours

Miscellaneous

5.22.3 Make a custom metadata file (*meta.py*)

You can also make a custom format for the ASE database website. To do this you will want to run the following command in the terminal `ase db -w name_of_database.d -M meta.py`. The `meta.py` file is a python file that can be used to custom format the ASE database website. The `meta.py` file is designed to give the database title, give definitions for all the variables given to the database, and the format of the database website. The `meta.py` file contains the following variables:

- **title** (*str*): This is the title of the database
- **default_columns** (*dict*): This contains information about each variable in the database
- **key_descriptions** (*list of str*): This is a list that specifies the order variable given in the database.

An example is a `meta.py` file is given in `Helpful_Programs` and an example is also given below

Listing 10: `meta.py`

```

1 metadata = {}
2 metadata['title'] = 'Genetic Algorithm Database'
3 metadata['default_columns'] = ['name', 'cluster_energy', 'id']
4 # -----

```

(continues on next page)

(continued from previous page)

```
5 metadata['key_descriptions'] = {}
6 metadata['key_descriptions']['name'] = ('Name', 'Name of the cluster.', '')
7 metadata['key_descriptions']['gen_made'] = ('Generation Created', 'The generation the
↳ cluster was created.', '')
8 metadata['key_descriptions']['cluster_energy'] = ('Cluster Energy', 'Potential energy
↳ of the cluster.', 'energy_units')
9 metadata['key_descriptions']['ever_in_population'] = ('ever_in_population', 'This
↳ variable indicates if the cluster was ever in the population. If an offspring was
↳ made and was not ever accepted into the population, this variable will be False. If
↳ the cluster had been in the population for even one generation, this variable will
↳ be True.', 'bool')
10 metadata['key_descriptions']['excluded_because_violates_predation_operator'] = (
↳ 'excluded_because_violates_predation_operator', 'This variable will determine if a
↳ cluster was excluded from the population because it violated the predation operator.
↳ ', 'bool')
11 metadata['key_descriptions']['initial_population'] = ('initial_population', 'This
↳ variable will indicate if the cluster was apart of a newly created population.',
↳ 'bool')
12 metadata['key_descriptions']['removed_by_memory_operator'] = ('removed_by_memory_
↳ operator', 'This variable will indicate if a cluster is removed because it resembles
↳ a cluster in the memory operator.', 'bool')
13 # -----
14
15 title = metadata['title']
16 default_columns = metadata['default_columns']
17 key_descriptions = metadata['key_descriptions']
```

The format for the database as specified in this example is: name cluster_energy id.

The following variables are included in the database:

- **name** (*int*): Name of the cluster.
- **gen_made** (*int*): The generation the cluster was created.
- **cluster_energy** (energy units, *float*): Potential energy of the cluster.
- **ever_in_population** (*bool*): This variable indicates if the cluster was ever in the population. If an offspring was made and was not ever accepted into the population, this variable will be False. If the cluster had been in the population for even one generation, this variable will be True.
- **excluded_because_violates_predation_operator** (*bool*): This variable will determine if a cluster was excluded from the population because it violated the predation operator.
- **initial_population** (*bool*): This variable will indicate if the cluster was apart of a newly created population.
- **removed_by_memory_operator** (*bool*): This variable will indicate if a cluster is removed because it resembles a cluster in the memory operator.

5.23 Adding Surfaces

This has not been constructed yet, functionality to be built if desired.

5.24 The Genetic Algorithm Python Files

Below are all the files that are used by the genetic algorithm to run.

5.24.1 Table of Contents

GA_Program.py

This is the main component of the genetic algorithm.

```
class Organisms.GA.GA_Program.GA_Program(cluster_makeup, pop_size, generations,
                                         no_offspring_per_generation, creating_offspring_mode, crossover_type, mutation_types, chance_of_mutation, r_ij, vacuum_to_add_length, Minimisation_Function, surface_details=None, epoch_settings={'use epoch': 'off'}, cell_length='default', memory_operator_information={'perform_memory_operator': 'Off'}, predation_information={'Predation_Switch': 'Off'}, fitness_information={'Fitness_Switch': 'Energy'}, ga_recording_information={}, force_replace_pop_clusters_with_offspring=True, user_initialised_population_folder=None, rounding_criteria=2, print_details=False, no_of_cpus=1, finish_algorithm_if_found_cluster_energy=None, total_length_of_running_time=None)
```

The Otago Research Genetic Algorithm for Nanoclusters, Including Structural Methods and Similarity (Organisms) program has been designed to perform a genetic algorithm for a nanoparticle of any composition and any size.

It has been designed for that it can be modified by others in the Garden group after I have left. Hopefully the code is readable and can be tuned, however I have tried to note what is happening during the code and why. More notes can be found in the

Parameters

- **cluster_makeup** (*{str: int, ..}*) – This contains the information on the makeup of the cluster you would like to optimise for. Format is a dictionary in the form of: {element: number of that element}
- **pop_size** (*int*) – The size of the population
- **generations** (*int*) – The number of generations that are run
- **no_offspring_per_generation** (*int*) – The number of offspring that are created per generation
- **creating_offspring_mode** (*str.*) – This indicates how the offspring are created, either via the mating method ‘followed’ by the mutation method, or by only perform the

mating method 'or' mutation method, or (i.e. either mating and/or mutation). See manual for how to set this.

- **crossover_type** (*str.*) – This is the type of crossover that you would like to use. See the manual for more information.
- **mutation_types** (*list of (str., float)*) – This is a list that contains all the information about the mutation methods you would like to use.
- **chance_of_mutation** (*float*) – This indicates the change of a mutation occurring. See the manual on specifically how this works.
- **r_ij** (*float*) – This is the maximum bond distance that we would expect in this cluster. See the manual for more information.
- **vacuum_to_add_length** (*float*) – This is the amount of vacuum to place around the cluster.
- **Minimisation_Function** (*__func__*) – This is a function that determines how to locally minimise clusters. See manual for more information.
- **surface_details** (*None*) – This functionality has not been designed yet. Default: None
- **epoch_settings** (*dict.*) – This is designed to hold the information about the epoch method.
- **cell_length** (*float*) – This is the length of the cubic unit cell to construct clusters in. See manual for more information. Default: 'default'
- **predation_information** (*dict.*) – This holds all the information about the predation operator. Default: {'Predation Operator': 'Off'}
- **fitness_information** (*dict.*) – This holds all the information about the fitness operator. Default: {'Fitness Operator': 'Off'}
- **ga_recording_information** (*dict.*) – Default: {}
- **force_replace_pop_clusters_with_offspring** (*bool.*) – This will tell the genetic algorithm whether to swap clusters in the population with offspring if the predation operator indicates they are the same but the predation operator has a better fitness value than the cluster in the population.
- **user_initialised_population_folder** (*str. or None*) – This is the directory to a folder containing any custom made clusters you would like to include in the initial population. Set this to None if you do not have any initial clusters to add into the population. Default: None
- **rounding_criteria** (*int*) – The number of decimal places to round the energies of clusters made during the genetic algorithm to. Default: 2
- **print_details** (*bool.*) – Verbose for this algorithm.
- **no_of_cpus** (*int*) – The number of cpus that the algorithm can use via multiprocessing. Default: 1
- **finish_algorithm_if_found_cluster_energy** (*dict. or None*) – If desired, the algorithm can finish if the LES is located. This is useful to use for methods testing. The algorithm will determine that the LES is found when the genetic algorithm locates the energy of the LES. Read the manual on how to use this. Default: None

- **total_length_of_running_time** (*int or None*) – The total amount of time to run the genetic algorithm for. If the algorithm is still running after this time, the algorithm will safety finish. Time given in hours. None means no limit on time, Default: None.

get_tasks (*previous_cluster_name, generation_number*)

This provides a generator that contains the inputs needed to create the offspring via parallelisation.

natural_selection (*offspring_pool, generation_number*)

This definition allows the genetic algorithm to perform the natural selection procedure.

The algorithm works as follows:

- The fitness of the population and the offspring are assessed.
- The clusters in the population are ordered from lowest to highest fitness.
- The offspring are ordered from highest to lowest fitness.
- Replace the clusters in the population with the lowest fitnesses with the offspring with the highest fitnesses.
- Once done, the Natural Selection Procedure is finished.

Parameters

- **generation_number** (*int*) – This is the current generation of the genetic algorithm run. This may not be 0 if you are restarting the algorithm
- **offspring_pool** (*Offspring*) – This is the offspring pool the GA will be using.

This Natural Selection procedure will update self.population

run_GA ()

This definition is the base of the algorithm. Once the algorithm has set itself up, this will run the genetic algorithm

GA_Program_Details.py

This class will write the details of the genetic algorithm run.

```
class Organisms.GA.GA_Program_Details.GA_Program_Details (GA_Program,  
                                                         is_new_ga=True)
```

This class is designed to keep track of the process of the genetic algorithm run.

Parameters

- **GA_Program** (*Organisms.GA.GA_Program:*) – This is the main genetic algorithm program this class will be writing about
- **is_new_ga** (*bool.*) – This determines if to write all the details about this genetic algorithm to a file. You only need to do this if the genetic algorithm is just beginning from generation 0. Default: True.

close ()

This will close the GA_Run_Details text file that is being recorded to.

This is designed to be a private definition

create (*is_new_ga*)

This definition will create a file containing the details of this genetic algorithm on the disk.

Parameters `is_new_ga` (*bool.*) – This determines if to write all the details about this genetic algorithm to a file. You only need to do this if the genetic algorithm is just beginning from generation 0.

end_clock (*generation*)

Get the end time taken for a generation to run.

Parameters `generation` (*int*) – The generation that was run.

returns `time_taken`: The time taken for the generation to run in seconds. `rtype time_taken`: float

ending_details ()

This def will write the finishing remarks of the genetic algorithm to GA_Run_Details

open (*read_type*)

This will open the GA_Run_Details text file that is being recorded to.

This is designed to be a private definition

Parameters `read_type` (*char*) – This is the way to open the file, either read (r), write (w), or append (a).

start_clock ()

Get the Starting time for this genetic algorithm, which will be used to time each generation.

Cluster.py

This describe a cluster, where the system information is stored in the Genetic Algorithm

Cluster.py, 12/04/2017, Geoffrey R Weal

This class is specifically designed to hold cluster objects for the Genetic Algorithm program. This object will hold a reference to the Atoms class from ASE for the cluster, the energy of the cluster and the dir tag for the cluster

```
class Organisms.GA.Cluster.Cluster (symbols=None, positions=None, numbers=None,
tags=None, momenta=None, masses=None, mag-
moms=None, charges=None, scaled_positions=None,
cell=None, pbc=None, celldisp=None, constraint=None,
calculator=None, info=None, surface=None)
```

This class is designed to clude all the information that the genetic algorithm needs to know from the chemical system.

Parameters

- **symbols** (*str (formula) or list of str., or ASE.Atoms*) – This can be a list of all the chemical symbols of atoms in the clusters, OR, the ASE.Atoms object that this object will be base on. For writing this as a list of element symbols, see See <https://wiki.fysik.dtu.dk/ase/ase/atoms.html?highlight=atoms#ase.Atoms> for more information. Default: None
- **positions** (*list of xyz-positions*) – A list of all the (x,y,z) values for the cluster. See <https://wiki.fysik.dtu.dk/ase/ase/atoms.html?highlight=atoms#ase.Atoms> for more information. Default: None
- **numbers** (*list of int*) – See <https://wiki.fysik.dtu.dk/ase/ase/atoms.html?highlight=atoms#ase.Atoms> for more information. Default: None
- **tags** (*list of int*) – See <https://wiki.fysik.dtu.dk/ase/ase/atoms.html?highlight=atoms#ase.Atoms> for more information. Default: None
- **momenta** (*list of xyz-momenta*) – See <https://wiki.fysik.dtu.dk/ase/ase/atoms.html?highlight=atoms#ase.Atoms> for more information. Default: None

- **masses** (*list of float*) – See <https://wiki.fysik.dtu.dk/ase/ase/atoms.html?highlight=atoms#ase.Atoms> for more information. Default: None
- **magnoms** (*list of float or list of xyz-values*) – See <https://wiki.fysik.dtu.dk/ase/ase/atoms.html?highlight=atoms#ase.Atoms> for more information. Default: None
- **charges** (*list of float*) – See <https://wiki.fysik.dtu.dk/ase/ase/atoms.html?highlight=atoms#ase.Atoms> for more information. Default: None
- **scaled_positions** (*list of scaled-positions*) – See <https://wiki.fysik.dtu.dk/ase/ase/atoms.html?highlight=atoms#ase.Atoms> for more information. Default: None
- **cell** (*3x3 matrix or length 3 or 6 vector*) – See <https://wiki.fysik.dtu.dk/ase/ase/atoms.html?highlight=atoms#ase.Atoms> for more information. Default: None
- **pbcc** (*one or three bool*) – See <https://wiki.fysik.dtu.dk/ase/ase/atoms.html?highlight=atoms#ase.Atoms> for more information. Default: None
- **celldisp** (*Vector*) – See <https://wiki.fysik.dtu.dk/ase/ase/atoms.html?highlight=atoms#ase.Atoms> for more information. Default: None
- **constraint** (*constraint object(s)*) – See <https://wiki.fysik.dtu.dk/ase/ase/atoms.html?highlight=atoms#ase.Atoms> for more information. Default: None
- **calculator** (*calculator object*) – See <https://wiki.fysik.dtu.dk/ase/ase/atoms.html?highlight=atoms#ase.Atoms> for more information. Default: None
- **info** (*dict of key-value pairs*) – See <https://wiki.fysik.dtu.dk/ase/ase/atoms.html?highlight=atoms#ase.Atoms> for more information. Default: None
- **surface** (*ASE.Atoms/str.*) – This is the surface that the cluster is modelled on. This is given either as a string or the ASE.Atoms object. Default: None

centre_cluster_at_centre_of_cell (*vacuumAdd*)

This method will center the cluster in the middle of the cell. This method has been employed to make it easier to look at clusters that are made during the Organisms program.

Inputs: vacuumAdd (float): The amount of value to add around the cluster.

custom_verify_cluster (*name, gen_made, cluster_energy, ever_in_population, excluded_because_violates_predation_operator, initial_population*)

This method allows you to custom verify the cluster. To be used when resuming the genetic algorithm.

Parameters

- **name** (*int/str.*) – The name that the cluster is referenced in this genetic algorithm program. This should be a integer that is based on when the cluster was made.
- **gen_made** (*int*) – This is the generation when the cluster was created
- **ever_in_population** (*bool.*) – Was this cluster ever in the population.
- **excluded_because_violates_predation_operator** (*bool.*) – Was the cluster removed from the offspring pool because it violated the predation operator
- **initial_population** (*bool.*) – Was the cluster apart of the initial population, either at the beginning of the population or after an epoch.

deepcopy ()

Will create a copy of the Cluster. This copy does not include a copy of the calculator as this has the potential to cause issues.

deepcopy_skeleton ()

This will create a copy of the cluster that does not contain any atoms, but will contain all the other settings for this cluster.

get_elemental_makeup ()

This gives a list which indicates the types of elements, and the number of those elements, in the cluster

Returns A list which indicates the types of elements, and the number of those elements, in the cluster

Return type {str: int, ..} (old output was [[str.,int],..])

get_total_cluster_energy (rounding_criteria=None)

Obtain the energy of the cluster to a predefined decimal place.

Parameters **rounding_criteria** (*int*) – The number of decimal places that the cluster's energy is rounded to.

returns energy: This is the energy of the cluster rtype energy: float

remove_calculator ()

This method will remove the calculator for this Cluster

sortZ ()

Sort the cluster from most positive to most negative value of z. This sorting by z axis is required to allow the algorithm to easily split itself into two sides of a dividing plane.

verify_cluster (name, gen_made, vacuum_length, rounding_criteria)

This method will verify that the cluster contains all the other information that it needs to run during the genetic algorithm, providing the name, generation made, and the energy of the cluster, as well as centering in a unit cell. This method is performed after the cluster has been locally optimised.

Parameters

- **name** (*int/str.*) – The name that the cluster is referenced in this genetic algorithm program. This should be a integer that is based on when the cluster was made.
- **gen_made** (*int*) – This is the generation when the cluster was created
- **vacuum_length** (*float*) – This is the amount of vacuum to give the cluster.
- **rounding_criteria** (*int*) – The number of decimal places that the cluster's energy is rounded to.

view ()

Allow the user to visually look at the cluster using the ASE gui. This is a debugging method.

`Organisms.GA.Cluster.import_surface (surface)`

This method is designed to import the surface into the genetic algorithm

Parameters **surface** (*str/ASE.Atoms.*) – This is the surface that the cluster is optimised upon.

Surface.py

This class is designed to record the lowest energetic clusters that the genetic algorithm run generates.

```
class Organisms.GA.Surface.MyConstraint (a, direction)
    Constrain an atom to move along a given direction only.
```

Collection.py

This describes a collection of clusters, such as that stored as a population or a set of offspring.

```
class Organisms.GA.Collection.Collection (name, size, path=None, have_database=False,
                                          write_collection_history=False,
                                          write_cluster_in_RAM=True)
```

This is the foundation of the object used to store clusters in the population, the offspring, and for recording clusters made during the genetic algorithm using the GA_Recording_System.py

Parameters

- **path** (*str.*) – The path that clusters will be written to disk. Default: None, meaning will write to same path as your execution Run.py file.
- **name** (*str.*) – Name of the Collection. This can be any name you like, it is just to note what the collection is. All collections should have a unique name if possible to prevent confusion, however this will not break the program.
- **size** (*int*) – This is the number of clusters that should be in the collection. In this version of the Organisms program, the number of clusters in the population and made during Creation of Offspring should be consistent throughout the genetic algorithm process.
- **write_collection_history** (*bool.*) – This will tell the collection to record a txt file of what clusters were in the collection over the generations Organisms performs. Default: False
- **write_files_as** (*str.*) – This tells the collection if and how to write clusters to the disk (Default: “database”). There are three options:
 - “database” if you want the Collection to make a database
 - “xyz” if you want the Collection to make xyz files
 - None, “None” or “none” if you do not want the Collection to make any cluster files

```
add (index, cluster)
```

Adds a cluster to the Collection.

Index: *index* (int/str.): the index of the *i*th cluster in the Collection. If “End” is inputed, the cluster will be append to the end of the Collection list. *cluster* (Organisms.GA.Cluster): The cluster to add at the *i*th position in the Collection.

```
add_clusters_into_RAM (cluster_dict, cluster_names)
```

This method adds clusters into the RAM

Parameters

- **cluster_dict** (*{int: ASE.Cluster}*) – This is a dictionary of all the clusters from the database, given as {*cluster_name*: Cluster}
- **cluster_names** (*list of int*) – list of the names of the clusters that are needed for the collection

add_metadata ()

Due to an issue with some versions of ASE, this method will write metadata to the ASE database, if a database is being used to store cluster information.

add_to_database (cluster, center=False)

Allows the user to write a cluster in the collection to a ASE database

Inputs: cluster (Organisms.GA.Cluster): The cluster to add to the database.

add_to_history_file (generation_number, is_epoch=False, epoch_due_to_population_energy_convergence=None)

This definition will add the information of the population. This is suppose to be used after each generation has completed.

Parameters

- **generation_number** (*int*) – The current generation that the genetic algorithm run has just performed.
- **is_epoch** (*bool.*) – Has an epoch just occurred
- **epoch_due_to_population_energy_convergence** (*bool.*) – If an epoch occurred, was it because the energy of the clusters converged.

assess_clusters_in_database (database_path)

This algorithm will check to make sure that there are no technical issues with the database, whether that is the original or the backup database.

Parameters database_path (*str.*) – The path to the database

returns is_database_working: Is true if there are no technical issues with the database, False if there are technical issues. rtype is_database_working: bool. returns reason_for_issue: Return a None object if everything is all good, otherwise returns the exception detailing the issues with the database or the name of the cluster that caused issues. rtype is_database_working: None or str.

backup_database ()

This method will make a backup of all the clusters in the Collection as a ASE database

check_PoolProfileTXT_exists ()

This definition checks to see if the PoolProfile folder exists

check_clusters_in_database (cluster_dict, cluster_names, cluster_energies, decimal_place)

This method will check that the database contains all the clusters you need and does not have any issues.

Parameters

- **cluster_dict** (*{int: ASE.Cluster}*) – This is a dicionary of all the clusters from the database, given as {cluster_name: Cluster}
- **cluster_names** (*list of int*) – list of the names of the clusters that are needed for the collection
- **cluster_energies** (*list of float*) – list of the energies of the clusters that are needed for the collection

returns is_database_all_good: True means the database is all good and contains all the clusters needed to restore the collection, as well as comfirming they are of the correct energy. False means something is not working with the database, the database does not contain a required cluster, or there is an issue with clusters not having the energy that they should have. rtype is_database_all_good: bool.

check_database_and_determine_if_to_use_backup ()

This method will remove any journal or lock files associated with the database, as well as check that either the database or the backup is functional and can be used to allowing your genetic algorithm trial to resume.

returns `did_use_backup_database`: True means the backup database was used. False means the original database was used. `rtype did_use_backup_database`: bool.

check_historyfile (*resume_from_generation*)

This method will check the history file to make sure that it does not contain any information from a failed generation.

If it does contain information from failed generations, it will delete those lines from the history file.

Parameters `generation_number` (*int*) – The current generation that the genetic algorithm run has just performed.

close ()

This closes the history text file.

create_collection_history ()

This definition will create the history file.

This includes the folder, contents of the folder and beginning to write the history file.

It also included information about the clusters in the collection file when it was first created.

delete_collection_database ()

This method will remove the whole database for this Collection from the disk.

does_contain_database (*backup*)

Does the collection contain a backup file.

Parameters `backup` (*bool.*) – If true, look for the backup database file. If false, look for the original database file.

returns `does_database_exist`: Returns if either the database file or the backup database file exists. `rtype does_database_exist`: bool.

get_cluster_energies ()

Returns all the clusters that the collection contains in a list

Returns all the clusters that the collection (list)

get_cluster_from_name (*name*)

This method will return the cluster in the Collection with the name “name”

Inputs: `name` (*int*): The name of the cluster you want to obtain from the Collection.

Returns The cluster in the Collection with the name “name” (Organisms.GA.Cluster)

get_cluster_names (*order=False*)

Will provide a list of all the names of all the clusters in the Collection

Inputs: `order` (*bool.*): This tag will tell this method whether the user would like the list of names given in order.

Returns List of the names of all the clusters in the Population

get_clusters ()

Returns all the clusters that the collection contains in a list

Returns all the clusters that the collection (list)

get_history_path ()

Return the path to the history file

Returns the path to the history file

get_index (*name_to_find*)

This method will provide the index of the cluster that has the name “name_to_find” in the Collection

Inputs: name_to_find (int): the name of the cluster in the Collection to obtain the index for

Returns the index of the cluster in the Collection with the name “name_to_find”

Exceptions: Will break if the cluster with the name “name_to_find” can not be found in this method.

get_max_mean_min_energies ()

The maximum, mean, and minimum energy of the clusters in the population.

returns maximum_energy: This is the maximum energy of the cluster rtype maximum_energy: float returns

mean_energy: This is the mean energy of the cluster rtype mean_energy: float returns minimum_energy:

This is the minimum energy of the cluster rtype minimum_energy: float

history_file_name (*end_name=None*)

Get the name of the history file for this Collection.

Inputs: end_name (str): The suffix of the name for the history gfile.

import_clusters_from_database_to_memory (*current_generation,* *clusters_in_resumed_population,* *clusters_in_resumed_population_energies,* *decimal_place*)

This method will attempt at obtaining the clustrs from the database and placing them in the collection in the RAM.

This method is currently set up for reading the population, but if needed it can be reworked for general purpose.

Parameters

- **current_generation** (*float*) – The current generation
- **clusters_in_resumed_population** (*list of int*) – The names of the clusters in the current population
- **clusters_in_resumed_population_energies** (*list of floats*) – The energies of the clusters in the current population
- **decimal_place** (*int*) – The number of decimal places that energies are rounded to in your genetic algorithm run

returns did_clusters_come_from_backup: Did the imported clusters come from the backup database or the original. True if from the backup database, False if from the original backup. rtype did_clusters_come_from_backup: bool.

is_there_an_energy_range (*rounding*)

Determines if there is a range of energies in the collection

Parameters **rounding** (*float*) – The rounding of the energy of the cluster

returns Is there a range of energies in the collection rtype bool

make_collection_folder ()

Will create the directory for self.path if it does not exist.

max_energy ()

The maximum energy of the clusters in the population.

returns maximum_energy: This is the maximum energy of the cluster rtype maximum_energy: float

mean_energy ()

The mean energy of the clusters in the population.

returns mean_energy: This is the mean energy of the cluster rtype mean_energy: float

min_energy ()

The minimum energy of the clusters in the population.

returns minimum_energy: This is the minimum energy of the cluster rtype minimum_energy: float

move_backup_database_to_normal_backup ()

This method will remove the original database file and replace it with the backup database file.

open (w_or_a)

This opens the profile pool text file

Inputs: w_or_a = 'Indicates how to open the file, whether to open it as a new file ("w") or to append information to the history file ("a").

pop (ith)

Pops the cluster at the ith position of the Collection and returns it.

Inputs: ith (int): The index for the cluster that you want to get from the Collection

Returns The cluster that you want to obtain from the Collections (Organisms.GA.Cluster)

read_collection_database (database_path, current_generation=None)

This method will read the clusters in the database. Furthermore, this method is also designed to repair the collection database by removing any clusters that were created after the current generation if desired.

Parameters

- **database_path (str.)** – The path to the database.
- **current_generation (int)** – The current generation that your genetic algorithm trial is being resumed from

returns clusters: This is a list of all the clusters from the database rtype clusters: list of Organisms.GA.Clusters

remove (index)

Removes a cluster to the Collection.

Index: index (int): the index of the ith cluster in the Collection

remove_backup_database ()

This method will remove the backup of all the clusters in the Collection, which will be in the format of a ASE database.

remove_backup_database_if_exists ()

This method will remove the backup of all the clusters in the Collection, which will be in the format of a ASE database.

remove_clusters_from_database_that_are_from_unsuccessful_generations (database_path, current_generation=None)

This method will go through the database and delete any clusters of unsuccessful generations.

Parameters

- **database_path (str.)** – The path to the database.
- **current_generation (int)** – The current generation that your genetic algorithm trial is being resumed from

remove_to_database (*cluster*)

Allows the user to remove a cluster in the collection from the ASE database

Inputs: cluster (Organisms.GA.Cluster): The cluster to remove from the database.

replace (*index, new_cluster*)

Will replace the *ith* cluster in the Collection with a new cluster. Uses the self.remove and self.add methods.

Inputs: index (int): the index of the *ith* cluster in the Collection new_cluster (Organisms.GA.Cluster): The new cluster to add at the *ith* position in the Collection

sort_by_energy ()

This method will sort the clusters in the list by their energy (from lowest energy to highest energy).

sort_by_name ()

This method will sort the clusters in the list by their name.

view_cluster (*ith*)

Allow the user to visually look at the cluster using the ASE gui. This is a debugging method.

Inputs: *ith* (int): the index of the *ith* cluster in the Collection to view in the ASE gui.

Collections_iterator.py

This describes a collection of clusters, such as that stored as a population or a set of offspring.

```
class Organisms.GA.Collections_Iterator.Collections_Iterator (population, offspring_pools=None, pass_first_cluster=False)
```

This iterator is designed to iterate through multiple collections together, in such a way that we sample every cluster across a range of collections. This method is designed to work in place so that it take up minimal amount of space on ram.

Parameters

- **population** (*Organisms.GA.Population*) – This is the population being used for this Genetic Run
- **offspring_pools** (*Organisms.GA.Offspring_Pool* or [*Organisms.GA.Offspring_Pool, ..*]) – These are all the offspring_pools that are being used. This can be inputs as one offspring_pool, or a list of multiple offspring_pools
- **pass_first_cluster** (*bool.*) – Start with the first cluster in the population, or the second.

Population.py

This describe the population in the genetic algorithm

```
class Organisms.GA.Population.Population (name, size, user_initialised_population_folder=None, write_data=True)
```

This class stores all the clusters that are in the population.

Parameters

- **name** (*str.*) – Name of the Collection. This can be any name you like, it is just to note what the collection is. All collections should have a unique name if possible to prevent confusion, however this will not break the program.

- **size** (*int*) – This is the number of clusters that should be in the collection. In this version of the Organisms program, the number of clusters in the population and made during Creation of Offspring should be constant throughout the genetic algorithm process.
- **user_initialised_population_folder** (*str.*) – Indicates the directory to obtain clusters to place in the initial population. If None, there are no user created clusters to be obtained. Default: None.
- **write_data** (*bool.*) – Write the clusters to disk in a database. Default: True.

backup_files ()

Backup the database and the current state file

backup_state_file ()

This method will make a backup of all the clusters in the Collection as a ASE database

current_state_file (*generation_number*)

Write the current state file for the population for this generation.

Parameters *generation_number* (*int*) – The current generation.

get_current_generation_from_state_file ()

This method is used to get the current generation from the Organisms program if it had already run previously. This method is used at the beginning of the Organisms program to get the current generation if the Organisms program has been restarted.

Returns The current generation of the Organisms program if it has already run previously and is being restarted. The names of the clusters in the population at that generation.

get_data_from_current_state_file (*current_state_file*)

Get data about the current population from the state file on disk.

Parameters *current_state_file* (*str.*) – The path to the current state file.

:returns the generation the state file was made, a list of the names of the clusters in the population, and a list of the energies of the clusters in the population. :rtype float, list of int, and list of float

get_details ()

Debugging tool for the Population Class.

This definition is designed to print all the details about the population.

get_pool_folder_size (*folder_to_look_at=False*)

This definition will count the number of clusters in the population that the user places into the GA before it runs. It does this by counting the number of numbered folders (these in this program hold information about each cluster created during a GA Run). This method is only needed before the GA begins.

Inputs: *folder_to_look_at* (False/None/str.): Indicate the number of clusters in the population as stored on the disk.

returns: *len(clusters_in_population)*: The size of the population, measured by counting the number of folders named by a number (which this program will interpret as a cluster in the population). rtypes: int

move_backup_to_current_files ()

This method remove the last state file if one exists, and replace it with the backup.

print_clusters ()

Debugging tool for the Population Class.

This definition is designed to print all the details about the clusters in the population.

remove_backup_files ()

Remove the backup the database and the current state file

remove_backup_state_file()

This method will remove the backup of all the clusters in the Collection, which will be in the format of a ASE database.

remove_backup_state_file_if_exists()

This method will remove the backup of all the clusters in the Collection, which will be in the format of a ASE database.

repair_current_state_file(*generation_number, cluster_names, cluster_energies*)

Write the current state file for the population for this generation. This method is used if the state file or a back up could not be found or were incomplete. This method will get the data from the information from the energyprofile.txt file

Parameters

- **generation_number** (*int*) – The current generation.
- **cluster_names** (*list of ints.*) – The names of the clusters in the population.
- **cluster_energies** (*list of floats*) – The energies of the clusters in the population.

EnergyProfile.py

This class is designed to record the energies of the clusters during the genetic algorithm.

class Organisms.GA.EnergyProfile.**EnergyProfile** (*collection, end_name=None*)

This class is designed to record the energies of the clusters during the genetic algorithm.

Parameters

- **collection** (*Organisms.GA.Collection*) – This is the specific collection of the genetic algorithm this class will be recording for.
- **end_name** (*str.*) – Include a suffix to the EnergyProfile.txt filename

GA_Starts()

This will write the energies of the collection into the recording EnergyProfile text file.

add_collection(*collection, generation_number*)

This will add information about the collection to the EenergyProfile.txt

Inputs: *collection* (Organisms.GA.collection): This is the collection to be added to the EnergyProfile.txt, should be an Offspring_Pool. *generation_number* (int): This is the current generation number of the running Organisms program.

add_epoch_note()

Add a note to the energyprofile to say that an epoch event occurred at this point

add_epoch_note_due_to_population_energy_convergence()

Add a note to the energyprofile to say that an epoch event occurred at this point due to the energies in the cluster having converged.

add_found_LES_note()

Add a note to the energyprofile to say that the LES was located, such that your genetic algorithm will now finish.

add_to(*cluster, generation_number*)

This definition will add the information of a cluster. Designed to be used when offspring are created.

Inputs: cluster (Organisms.GA.Cluster): This is the cluster to write information about in the EnergyProfile.txt generation_number (int): This is the current generation number of the running Organisms program.

check (*resume_from_generation, no_offspring_per_generation*)

Check the EnergyProfile to make sure it is all good to go. As well as if the LES has been located by the genetic algorithm, and to remove any information from the end of the EnergyProfile document that attains to future generations that the algorithm did not complete successfully.

Parameters

- **resume_from_generation** (*int*) – The generation that the algorithm is resuming from
- **no_offspring_per_generation** (*int*) – The number of offspring generated per generation.

close ()

This will close the EnergyProfile text file that is being recorded to.

This is designed to be a private definition

create ()

This definition will create the folders and text files for

get_current_generation_and_last_cluster_generated_from_EnergyProfile ()

Get the number of the last generation and the name of the last cluster that was recorded in the EnergyProfile

:returns The last recorded generation, and the name of the last recorded cluster. :rtype int, int

is_LES_note_in_EnergyProfile ()

Look through the EnergyProfile if it exists for a note saying that the LES had be found.

:returns True if the EnergyProfile states the LES has been found, otherwise return False. :rtype bool.

open (*read_type*)

This will open the EnergyProfile text file that is being recorded to.

This is designed to be a private definition

Inputs: read_type (str.): This is the way to open the file, either read ('r'), write ('w'), or append ('a').

Organisms.GA.EnergyProfile.**remove_end_lines_from_text** (*file, number_of_lines_to_remove*)

This method removes the last number of lines from the document.

For more information, see: <https://superuser.com/questions/127786/efficiently-remove-the-last-two-lines-of-an-extremely-large-t>

Parameters

- **file** (*str.*) – This is the file path to the path you want to read the end of
- **number_of_lines_to_remove** (*int*) – This is the number of lines to remove from the document.

Organisms.GA.EnergyProfile.**tail** (*f, n, offset=0*)

This method will read the last lines from a text file

Parameters

- **f** (*str.*) – This is the file path to the path you want to read the end of
- **n** (*int*) – This is the last lines to read from the document.
- **offset** (*int*) – This is an offset number of line that should at least have been read, but this can be kept as 0 and given in n. For the user to decide.

returns lines: These are the line to be read from the end of the document rtype lines: list of str.

Offspring_Pool.py

This class is designed to hold all offspring clusters that are created and perform any requirement for them.

class Organisms.GA.Offspring_Pool.**Offspring_Pool** (*name*, *offspring_pool_size*)

This class is designed to hold the offspring that are made during the Organisms program.

Parameters

- **name** (*str.*) – The name of the Offspring_Pool. The names of all the collections should be different to prevent confusion, but this shouldn't affect how this program works.
- **offspring_pool_size** (*int*) – The maximum number of clusters in the collection

clean ()

Remove all the clusters in the collection.

sort_by_fitness ()

Sort the offspring in the Offspring_Pool by fitness

GA_Setup.py

This class is designed

Organisms.GA.GA_Setup.**GA_Setup** (*self*, *cluster_makeup*, *pop_size*, *generations*,
no_offspring_per_generation, *creating_offspring_mode*,
crossover_type, *mutation_types*, *chance_of_mutation*,
r_ij, *vacuum_to_add_length*, *Minimisation_Function*,
surface_details, *epoch_settings*, *cell_length*, *mem-*
ory_operator_information, *predation_information*,
fitness_information, *ga_recording_information*,
force_replace_pop_clusters_with_offspring,
user_initialised_population_folder, *round-*
ing_criteria, *print_details*, *no_of_cpus*, *fin-*
ish_algorithm_if_found_cluster_energy, *to-*
tal_length_of_running_time)

This method will set up the genetic algorithm.

Parameters

- **cluster_makeup** (*{str: int, ..}*) – This contains the information on the makeup of the cluster you would like to optimise for. Format is a dictionary in the form of: {element: number of that element}
- **pop_size** (*int*) – The size of the population
- **generations** (*int*) – The number of generations that are run
- **no_offspring_per_generation** (*int*) – The number of offspring that are created per generation
- **creating_offspring_mode** (*str.*) – This indicates how the offspring are created, either via the mating method 'followed' by the mutation method, or by only perform the mating method 'or' mutation method, or (i.e. either mating and/or mutation). See manual for how to set this.
- **crossover_type** (*str.*) – This is the type of crossover that you would like to use. See the manual for more information.

- **mutation_types** (*list of (str., float)*) – This is a list that contains all the information about the mutation methods you would like to use.
- **chance_of_mutation** (*float*) – This indicates the change of a mutation occurring. See the manual on specifically how this works.
- **r_ij** (*float*) – This is the maximum bond distance that we would expect in this cluster. See the manual for more information.
- **vacuum_to_add_length** (*float*) – This is the amount of vacuum to place around the cluster.
- **Minimisation_Function** (*__func__*) – This is a function that determines how to locally minimise clusters. See manual for more information.
- **surface_details** (*None*) – This functionality has not been designed yet. Default: None
- **epoch_settings** (*dict.*) – This is designed to hold the information about the epoch method.
- **cell_length** (*float*) – This is the length of the cubic unit cell to construct clusters in. See manual for more information. Default: 'default'
- **predation_information** (*dict.*) – This holds all the information about the predation operator. Default: {'Predation Operator': 'Off'}
- **fitness_information** (*dict.*) – This holds all the information about the fitness operator. Default: {'Fitness Operator': 'Off'}
- **ga_recording_information** (*dict.*) – Default: {}
- **force_replace_pop_clusters_with_offspring** (*bool.*) – This will tell the genetic algorithm whether to swap clusters in the population with offspring if the predation operator indicates they are the same but the predation operator has a better fitness value than the cluster in the population.
- **user_initialised_population_folder** (*str. or None*) – This is the directory to a folder containing any custom made clusters you would like to include in the initial population. Set this to None if you do not have any initial clusters to add into the population. Default: None
- **rounding_criteria** (*int*) – The number of decimal places to round the energies of clusters made during the genetic algorithm to. Default: 2
- **print_details** (*bool.*) – Verbose for this algorithm.
- **no_of_cpus** (*int*) – The number of cpus that the algorithm can use via multiprocessing. Default: 1
- **finish_algorithm_if_found_cluster_energy** (*dict. or None*) – If desired, the algorithm can finish if the LES is located. This is useful to use for methods testing. The algorithm will determine that the LES is found when the genetic algorithm locates the energy of the LES. Read the manual on how to use this. Default: None
- **total_length_of_running_time** (*int or None*) – The total amount of time to run the genetic algorithm for. If the algorithm is still running after this time, the algorithm will safety finish. Time given in hours. None means no limit on time, Default: None.

GA_Initiate.py

This class is designed

Organisms.GA.GA_Initiate.**GA_Initiate** (*self*)

This definition provides the details to how to run the genetic algorithm. The program has the ability to restart if required.

Organisms.GA.GA_Initiate.**Initate_New_GAProgram** (*self*)

Set up the Organisms program to perform a new genetic algorithm run.

returns: the name of the last cluster that was created. rtype: int

Organisms.GA.GA_Initiate.**Initial_ProgramChecking** (*self*)

This definition will check data from files and from that obtained self.get_resume_from_generation() and check to whether the program can continue or not.

Organisms.GA.GA_Initiate.**Resume_GAProgram** (*self*, *resume_from_generation*, *clusters_in_resumed_population*, *clusters_in_resumed_population_energies*)

This definition initialises all python variables and rewrites and modifies all files to the point when the generation self.resume_from_generation finished and the next generation began.

Parameters

- **resume_from_generation** (*int*) – the generation the genetic algorithm is resuming from.
- **clusters_in_resumed_population** (*list of int*) – list of the name of the clusters in the population to resume from.
- **clusters_in_resumed_population_energies** (*list of float*) – list of the name of the clusters in the population to resume from

returns: the name of the last cluster that was created. rtype: int

Organisms.GA.GA_Initiate.**add_metadata** (*self*)

This is included as this seems to not be added to the collections database when it is first created. Fixing an issue with ASE.

Organisms.GA.GA_Initiate.**get_resume_from_generation** (*self*)

Will determine what the clusters in the population are, and the current generation, if the Organisms program is being restarted or if the number of generations is being increased.

Will get this information for files in the Population folder.

Returns current_generation: The current generation that the Organisms program is being restarted from, or otherwise None if this is a new Organisms program run.

Return type int or None

Returns clusters: The clusters in the population. In the formation of [(name of cluster, energy of cluster)]

Return type [(int, float), ..]

GA_Introducing_Remarks.py

This class is designed

`Organisms.GA.GA_Introducing_Remarks.GA_Program_Logo()`

Provides the opening logo for the Organisms program.

`Organisms.GA.GA_Introducing_Remarks.Introducing_Remarks(self)`

Provides information about the settings for your Organisms run.

`Organisms.GA.GA_Introducing_Remarks.version_no()`

Will provide the version of the Organisms program

Initialise_Population.py

This class is designed

`Organisms.GA.Initialise_Population.Check_Population_against_predation_operator(population, predation_operator)`

Checks that all the clusters in this newly initialised population obey the predation operator

Parameters

- **population** (*Organisms.GA.Population*) – The population
- **predation_operator** (*Organisms.GA.Predation_Operator*) – This is the predation operator

:returns a list of all the clusters that do not obey the predation operator, as well as a report about the issues.

:rtype A list of int, and a string

`Organisms.GA.Initialise_Population.Initialise_Population(population, cluster_makeup, surface, Minimisation_Function, memory_operator, predation_operator, fitness_operator, epoch, cell_length, vacuum_to_add_length, r_ij, rounding_criteria, no_of_cpus, previous_cluster_name=0, generation=0, get_already_created_clusters=True, is_epoch=False, epoch_due_to_population_energy_convergence=1)`

This method will initialise the Population by

1. Placing clusters that the user would like in the initial population
2. Generate a number of extra clusters so that the population has the desired number of clusters in it.

This method is used when the population is first created when the genetic algorithm is just starting, and when an epoch method is resetting the population with a new population of randomly generated clusters.

Parameters

- **population** (*Organisms.GA.Population*) – The population

- **cluster_makeup** (*dict.*) – The makeup of the cluster
- **surface** (*Organisms.GA.Surface*) – This is the surface that the cluster is placed on. None means there is no surface.
- **Minimisation_Function** (*__func__*) – The minimisation function
- **memory_operator** (*Organisms.GA.Memory_Operator*) – The memory operator
- **epoch** (*Organisms.GA.Epoch*) – The epoch method
- **predation_operator** (*Organisms.GA.Predation_Operator*) – This is the predation operator
- **fitness_operator** (*Organisms.GA.Fitness_Operator*) – This is the fitness operator
- **cell_length** (*float*) – This is the length of the square unit cell the cluster will be created in.
- **vacuum_to_add_length** (*float*) – The amount of vacuum to place around the cluster
- **r_ij** (*float*) – The maximum distance that should be between atoms to be considered bonded. This value should be as large a possible, to reflected the longest bond possible between atoms in the cluster.
- **rounding_criteria** (*int*) – The number of decimal places to round the energy of clusters to.
- **no_of_cpus** (*int*) – The number of cpus available to create clusters
- **previous_cluster_name** (*int*) – This is the name of the last cluster created in the genetic algorithm. Default: 0
- **generation** (*int*) – The number of generations that have been performed. Default: 0
- **get_already_created_clusters** (*bool.*) – Are there clusters that the user created in the population. True if yes, False if no. Default: True
- **is_epoch** (*bool.*) – Has the genetic algorithm just epoched. Default: False
- **epoch_due_to_population_energy_convergence** (*bool.*) – Did the genetic algorithm epoch because the energies of the clusters in the last populatino converge. Default: None

Returns previous_cluster_name This is the name of the last cluster created by this method

Rtype previous_cluster_name int

Organisms.GA.Initialise_Population.**Initialise_Population_with_Randomly_Generated_Clusters** (*p*

This method will place a number of randomly generated clusters into the population until it is at the desired size.

Parameters

- **population** (*Organisms.GA.Population*) – The population
- **cluster_makeup** (*dict.*) – The makeup of the cluster
- **surface** (*Organisms.GA.Surface*) – This is the surface that the cluster is placed on. None means there is no surface.
- **Minimisation_Function** (*__func__*) – The minimisation function
- **cell_length** (*float*) – This is the length of the square unit cell the cluster will be created in.
- **vacuum_to_add_length** (*float*) – The amount of vacuum to place around the cluster
- **r_ij** (*float*) – The maximum distance that should be between atoms to be considered bonded. This value should be as large a possible, to reflected the longest bond possible between atoms in the cluster.
- **rounding_criteria** (*int*) – The number of decimal places to round the energy of clusters to.
- **no_of_cpus** (*int*) – The number of cpus available to create clusters
- **predation_operator** (*Organisms.GA.Predation_Operator*) – This is the predation operator
- **fitness_operator** (*Organisms.GA.Fitness_Operator*) – This is the fitness operator
- **memory_operator** (*Organisms.GA.Memory_Operator*) – The memory operator

- **previous_cluster_name** (*int*) – This is the name of the last cluster created in the genetic algorithm.

returns The name of the most recently created cluster by this method in the population. rtype int

Organisms.GA.Initialise_Population.**Place_Already_Created_Clusters_In_Population** (*population*, *cluster_makeup*, *Minimisation_Function*, *vacuum_to_add_length*, *r_ij*, *rounding_criteria*, *surface*, *memory_operator*, *predation_operator*, *fitness_operator*, *previous_cluster_name*)

This method will place any user created clusters into the population.

Parameters

- **population** (*Organisms.GA.Population*) – The population
- **cluster_makeup** (*dict.*) – The makeup of the cluster
- **Minimisation_Function** (*__func__*) – The minimisation function
- **vacuum_to_add_length** (*float*) – The amount of vacuum to place around the cluster
- **r_ij** (*float*) – The maximum distance that should be between atoms to be considered bonded. This value should be as large a possible, to reflected the longest bond possible between atoms in the cluster.
- **rounding_criteria** (*int*) – The number of decimal places to round the energy of clusters to.
- **surface** (*Organisms.GA.Surface*) – This is the surface that the cluster is placed on. None means there is no surface.
- **memory_operator** (*Organisms.GA.Memory_Operator*) – The memory operator
- **predation_operator** (*Organisms.GA.Predation_Operator*) – This is the predation operator
- **fitness_operator** (*Organisms.GA.Fitness_Operator*) – This is the fitness operator
- **previous_cluster_name** (*int*) – This is the name of the last cluster created in the genetic algorithm.

:returns The name of the most recently created cluster by this method in the population. :rtype int

Organisms.GA.Initialise_Population.**create_a_cluster**(*input_data*)

This will create a cluster, allowing the user to create cluster via multiprocessing.

Parameters **input_data** (*list of Any*) – This tuple contains all the information required to create a new randomly generated cluster.

returns tuple of the index to place the cluster into the population, and the optimised cluster rtype (int,Organisms.GA.Cluster)

Organisms.GA.Initialise_Population.**get_tasks**(*population, clusters_to_create, cell_length, vacuum_to_add_length, cluster_makeup, surface, r_ij, rounding_criteria, Minimisation_Function, memory_operator*)

This is a generator that will allow python to create clusters with multiprocessing.

Parameters

- **population** (*Organisms.GA.Population*) – The population
- **clusters_to_create** (*list of int*) – This is a list of all the names for the clusters to be given.
- **cell_length** (*float*) – This is the length of the square unit cell the cluster will be created in.
- **vacuum_to_add_length** (*float*) – The amount of vacuum to place around the cluster
- **cluster_makeup** (*dict.*) – The makeup of the cluster
- **surface** (*Organisms.GA.Surface*) – This is the surface that the cluster is placed on. None means there is no surface.
- **r_ij** (*float*) – The maximum distance that should be between atoms to be considered bonded. This value should be as large a possible, to reflected the longest bond possible between atoms in the cluster.
- **rounding_criteria** (*int*) – The number of decimal places to round the energy of clusters to.
- **Minimisation_Function** (*__func__*) – The minimisation function
- **memory_operator** (*Organisms.GA.Memory_Operator*) – The memory operator

returns list of all the information needed to create a cluster rtype list of Any

Get_Offspring.py

This class will write the details of the genetic algorithm run.

Organisms.GA.Get_Offspring.**Create_An_Offspring**(*input_data*)

This method is used to obtain an offspring

Parameters **input_data** (*tuple of many inputs*) – A tuple of all the information needed to make a new offspring

Returns The offspring and a string containing any information that would be useful for the user to see about how this cluster was made

Return type Organisms.GA.Cluster and str.

`Organisms.GA.Get_Offspring.Create_An_Unoptimised_Offspring` (*cluster_number*,
chance_of_mutation,
cell_length, *vacuum_to_add_length*,
population, *creating_offspring_mode*,
crossover_procedure,
mutation_procedure)

This definition provides the methodology for how to create an offspring.

Inputs: *cluster_number* (int): the name of the cluster *chance_of_mutation* (float): the change than a mutation will occur, either as well as or instead of crossover, depending on input for *creating_offspring_mode*. *cell_length* (float): The size of the unit cell. *vacuum_to_add_length* (float): The amount of vacuum to add to the unit cell” *population* (`Organisms.GA.Population`): This is the population that clusters will be taken from for mating and mutation schemes. *creating_offspring_mode* (str.): This tag indicates how the offspring are created.

- *creating_offspring_mode* == ‘Mating_and_Mutation_Together’ - The mating scheme is run first, followed by a mutation scheme with some probability..
- *creating_offspring_mode* == ‘Either_Mating_and_Mutation’ - Either the mating ‘or’ the mutation scheme will run, depending on a probability value.

crossover_procedure (`Organisms.GA.Crossover`): *mutation_procedure* (`Organisms.GA.Mutation`):

Returns the newly created offspring cluster.

Return type `Organisms.GA.Cluster`

`Organisms.GA.Get_Offspring.Will_Mutation_Occur` (*choice_mate_or_mutate*,
chance_of_mutation)

This method will determine if the algorithm should perform a mutation.

Parameters

- ***choice_mate_or_mutate*** (*float*) – This is a random value between 0 and 1. If this value is less than `self.chance_of_mutation`, this indicates to perform a mutation
- ***chance_of_mutation*** (*float*) – This a decimal value between 0 and 1 that determines the chance of a mutation.

Returns If True; mutate. If False; do not mutate (mate).

Return type `bool`.

Crossover.py

This describes the

`MatingProcedure.py`, 13/04/2017, Geoffrey R. Weal

This program is designed to run the mating procedure of the Genetic Algorithm.

class `Organisms.GA.Crossover.Crossover` (*crossover_type*, *r_ij*, *vacuumAdd*, *size_of_clusters*)

This class is designed to perform the mating procedure of the genetic algorithm. This will produce a cluster that is an outcome of the mating procedure.

Parameters

- ***crossover_type*** (*str.*) – This is the type of mating procedure the user would like to use. There are currently a few options implemented into this mating procedure:

- "CAS_weighted": Cut and Splice - Deavon and Ho - weighted by fitness of parents (CAS_weighted)
- "CAS_random": Cut and Splice - Deavon and Ho - cut a random percent x% of parent 1 and (100-x)% of parent 2 (CAS_random)
- "CAS_half": Cut and Splice - Deavon and Ho - cut both parents by half (CAS_half)
- "CAS_custom_XX": Cut and Splice - Deavon and Ho - cut a random percent XX% of parent 1 and (100-XX)% of parent 2. To use this set crossType = CAS_custom_XX, where XX is a float of your choice between 0 and 100.
- **r_ij** (*float*) – the maximum bond distance between atoms in the cluster. This should be the largest value possible for your cluster.
- **vacuumAdd** (*float*) – The vacuum around the cluster
- **size_of_clusters** (*int*) – The number of atoms in the cluster.

Cut_and_Splice_Devon_and_Ho (*parents*)

This definition is designed to perform the Cut and Splice Method as specified by Devon and Ho to mate the parents to give the offspring. The method works as follows:

1. Rotate each parent by some random amount in the theta and phi directions.
2. Sort the assignment of atoms in the cluster from most positive to most negative z value.
3. Perform a Cut and Splice procedure, where the parents are cut in some way and spliced together to give the offspring.

Parameters parents (*[Organisms.GA.Cluster, Organisms.GA.Cluster]*) – This is the population to choose clusters from to mate together.

Returns Returns an offspring that has been created from mating two parent clusters together.

Rtypes Organisms.GA.Cluster

centre_offspring_at_centre_of_cell (*offspring*)

This method will center the offspring as required.

Parameters offspring (*Organisms.GA.Cluster*) – The offspring cluster.

centre_parents_about_origin (*parents*)

This method will center the parents about the zero point. This is to make sure that their two halves will align correctly when they are cut and spliced to form a new offspring

Parameters parents (*[Organisms.GA.Cluster, Organisms.GA.Cluster]*) – This is the population to choose clusters from to mate together.

Returns The parents which have been centered about the (0,0,0) point of the unit cell

Rtypes [Organisms.GA.Cluster, Organisms.GA.Cluster]

half_index_custom_method (*cross_type*)

This method will determine how to cut the cluster based on the atom to divide from, where the atoms have been numbered in order of the z axis.

This version of the method divides the parents based on a percentage of atoms to take from parent 1

Here, percentage of parent1 is taken, while 1.0-percentage of parent2 two is taken.

Parameters crossover_type (*str.*) – This is the type of mating procedure the user would like to use. This should be set to "CAS_custom_XX", where XX is the percentage of parent 1 to be cut.

Returns The atom number in the cluster to cut the atom.

Return type int

half_index_half_method ()

This method will determine how to cut the cluster based on the atom to divide from, where the atoms have been numbered in order of the z axis.

This version of the method divides the parents in half.

Returns The atom number in the cluster to cut the atom.

Return type int

half_index_random_method ()

This method will determine how to cut the cluster based on the atom to divide from, where the atoms have been numbered in order of the z axis.

This version of the method divides the parents at some random atom indice

Returns The atom number in the cluster to cut the atom.

Return type int

half_index_weighted_method (*parents*)

This method will determine how to cut the cluster based on the atom to divide from, where the atoms have been numbered in order of the z axis.

This version of the method divides the parents based on their relative fitnesses.

Parameters **parents** (*[Organisms.GA.Cluster, Organisms.GA.Cluster]*) – This is the population to choose clusters from to mate together.

Returns The atom number in the cluster to cut the atom.

Return type int

mate_Cut_and_Splice (*parents*)

This method will perform the mating procedure as specified by the user. This can be designed to be used with multiple parents, however currently it is set up for 2 parents. It has also been developed for mating multi-metallic (multi-elemental) clusters together.

Parameters **parents** (*[Organisms.GA.Cluster, Organisms.GA.Cluster]*) – This is the population to choose clusters from to mate together.

Returns Returns an offspring that has been created from mating two parent clusters together.

Rtypes Organisms.GA.Cluster

mate_Cut_and_Splice_error_checking_1 (*parents*)

This is a method for checking if there are any issues with the Cut_and_splice method.

This method is only needed when you are developing or modifying a mating procedure, to help with debugging. Otherwise, you dont need to use this algorithm

Parameters **parents** (*[GA.Cluster, GA.Cluster]*) – This is the population to choose clusters from to mate together.

Returns A list which indicates the types of elements, and the number of those elements, in the cluster

Return type {str: int, ..}

mate_Cut_and_Splice_error_checking_2 (*elemental_makeup_of_parents, offspring*)

This is a method for checking if there are any issues with the Cut_and_splice method.

This method is only needed when you are developing or modifying a mating procedure, to help with debugging. Otherwise, you don't need to use this algorithm

Parameters

- **elemental_makeup_of_parents** (*{str: int, ...}*) – A list which indicates the types of elements, and the number of those elements, in the cluster
- **offspring** (*Organisms.GA.Cluster*) – The offspring cluster.

mating (*parents*)

This definition is designed as a switch to choose the mating method the user wishes to use for the genetic algorithm.

Parameters **parents** (*[Organisms.GA.Cluster, Organisms.GA.Cluster]*) – This is the population to choose clusters from to mate together.

Returns Returns an offspring that has been created from mating two parent clusters together.

Rtypes *Organisms.GA.Cluster*

pickParentsFromThePopulation (*population*)

This definition will pick the parents for mating.

Parameters **population** (*Organisms.GA.Population*) – This is the population to choose clusters from to mate together.

Returns two clusters from the population that will be mated together to give the offspring.

Rtypes (*Organisms.GA.Cluster, Organisms.GA.Cluster*)

rotate (*cluster*)

Rotate the cluster by some random angle in the theta and phi directions.

Parameters **cluster** (*Organisms.GA.Cluster*) – The cluster to be randomly rotated

roulette (*population*)

Performed the roulette wheel method to obtain the parents for a Mating procedure

Reference: <https://pubs.rsc.org/en/content/articlepdf/2003/dt/b305686d>

Parameters **population** (*Organisms.GA.Population*) – This is the population to choose clusters from to mate together.

Returns Will return a tuple of two integers that represent the indices of clusters in the population to use as parent to mate together to give a new offspring.

Rtypes [*int, int*]

run (*run_input*)

Run this the Mating Procedure and gives an offspring.

Parameters **population** (*Organisms.GA.Population*) – This is the population to choose clusters from to mate together.

Returns offspring

Rtypes *Organisms.GA.Cluster*

tournament (*population*)

Performed the tournament method to obtain the parents for a Mating procedure

Reference: <https://pubs.rsc.org/en/content/articlepdf/2003/dt/b305686d>

Parameters **population** (*Organisms.GA.Population*) – This is the population to choose clusters from to mate together.

Returns Will return a tuple of two integers that represent the indices of clusters in the population to use as parent to mate together to give a new offspring.

Rtypes [int, int]

Mutation.py

This describe a cluster, where the system information is stored in the Genetic Algorithm

class Organisms.GA.Mutation.**Mutation** (*mutation_types, r_ij, vacuum_to_add_length*)

This class contains all the information and procededures required to perform a Mutation.

Parameters

- **mutation_types** (*[(str, float), ...]*) – Contains the information about the mutations that the user would like to perform. See Manual for more information about this.
- **r_ij** (*float*) – The maximum distance that should be between atoms to be considered bonded. This value should be as large a possible, to reflected the longest bond possible between atoms in the cluster.
- **vacuum_to_add_length** (*float*) – The amount of vacuum to place around the cluster

change_mutation_chances ()

This method will change the format of self.mutation_types into a format that can be used by this Mutation class.

The format of self.mutation_types changes as follows with this example. [] => []

check (*r_ij*)

This method will check that the inputs for the mutation_types has been done correctly, without any violating issues.

Parameters **r_ij** (*float*) – The maximum distance that should be between atoms to be considered bonded. This value should be as large a possible, to reflected the longest bond possible between atoms in the cluster.

get_mutation_type ()

This definition will pick the type of mutation.

Returns mutation_type

Rtypes str.

mutation (*mutation_method, cluster_to_mutate, cell_length=None, vacuum_length=None*)

This method will perform the mutation and return the mutant.

Inputs: mutation_method (str.): The type of mutation to use for the mutation cluster_to_mutate (GA>Cluster): This is the cluster that we would like to perform a mutation upon. cell_length (float): This is the cell length of the unit cell. vacuum_length (float): The amount of vacuum to add to the unit cell.

Returns mutant

Rtypes Atom

pickClusterFromThePopulation (*population*)

This definition will pick the cluster from the population to mutate.

Inputs: population (GA.Population): The population to choose the cluster to mutate from.

Returns cluster_to_mutate - The list of parents for mating.

Rtypes Cluster

run (*run_input*, *cell_length=None*, *vacuum_length=None*)

This definition will run the Mutation Procedure.

Inputs: *run_input* (GA.Population/GA.Cluster/ase.Atoms): This is the input data about the cluster to mutate, or the population to take a cluster to mutate. *cell_length* (float): This is the cell length of the unit cell. *vacuum_length* (float): The amount of vacuum to add to the unit cell.

Returns *mutant* (GA.Cluster) - The mutated cluster *mutation_method* (str.) - The type of mutation that was picked to use for this mutation.

Organisms.GA.Mutation.**isfloat** (*element*)

Is the input a float.

Parameters *element* (*Any*) – some input

:returns True if the element can be considered a float, False if not. :rtype ebool.

Types_Of_Mutations.py

This class is designed

Organisms.GA.Types_Of_Mutations.**homotopMutate** (*cluster_to_mutate*)

This definition is designed to swap the positions of two elementally different atoms in a cluster

Parameters *cluster_to_mutate* (GA.Cluster) – The cluster to move the atoms in the cluster.

:returns The newly created mutated cluster :rtype GA.Cluster

Organisms.GA.Types_Of_Mutations.**moveMutate** (*cluster_to_mutate*, *dist_to_move*)

This method randomly displaces all the atoms within the structure *cluster_to_mutate* from its original position. This def depends on the value of the maximum bond length of the cluster, *self.r_ij*. The def will cause all atoms in the cluster to move randomly by up to a 1/2 of *self.r_ij* from their original position.

Parameters

- **cluster_to_mutate** (GA.Cluster) – The cluster to move the atoms in the cluster.
- **dist_to_move** (float) – The distance to move clusters by

:returns The newly created mutated cluster :rtype GA.Cluster

Organisms.GA.Types_Of_Mutations.**randomMutate** (*boxtoplaceinlength*, *vacuumAdd*,
cluster_makeup=None, *cluster_to_mutate=None*, *percentage_of_cluster_to_randomise=None*)

This definition provides the random method for the mutation procedure. In this method, a cluster is designed by randomly placing the designed atoms of elements into a predefined unit cell.

Parameters

- **boxtoplaceinlength** (float) – This is the length of the box you would like to place atoms in to make a randomly constructed cluster.
- **vacuumAdd** (float) – The length of vacuum added around the cluster. Written in Å.
- **cluster_makeup** (*{'Element': int(number of that 'element' in this cluster), ..}*) – check this

- **cluster_to_mutate** (*ASE.Atoms*) – If the user desired, they can tell this definition what cluster they want to mutate. Default: None.
- **percentage_of_cluster_to_randomise** (*float*) – This is the percentage of the number of atoms in the cluster to randomise

Returns mutant The description of the mutant cluster.

Rtypes ase.Atoms or GA.Cluster

Get_Predation_and_Fitness_Operators.py

Meow

```
Organisms.GA.Get_Predation_and_Fitness_Operators.check_asap3_version(self,  
                                                                    preda-  
                                                                    tion_information,  
                                                                    fit-  
                                                                    ness_information)
```

This method is required because there seems to be subtle issues with asap3 3.12.2. Require that asap3 3.11.10 be used.

Parameters

- **predation_information** (*dict.*) – All the information about the predation operator.
- **fitness_information** (*dict.*) – All the information about the fitness operator

```
Organisms.GA.Get_Predation_and_Fitness_Operators.get_fitness_operator(fitness_information,  
                                                                    pre-  
                                                                    da-  
                                                                    tion_operator,  
                                                                    pop-  
                                                                    ulation,  
                                                                    gen-  
                                                                    era-  
                                                                    tions,  
                                                                    no_of_cpus,  
                                                                    print_details)
```

This def will set up the fitness operator to be by the genetic algorithm. This is dependent on the entry for the 'Predation_Switch' given in the fitness_information.

Parameters

- **fitness_information** (*dict.*) – All the information about the fitness operator
- **predation_information** (*dict.*) – All the information about the predation operator.
- **population** (*Organisms.GA.Population*) – The population
- **generations** – The number of generations being run
- **generations** – int
- **no_of_cpus** (*int*) – The number of cpus that are to be used by your genetic algorithm run.
- **print_details** (*bool.*) – Print information to the terminal

Returns The fitness operator object

Return type Organisms.GA.Fitness_Operators.Fitness_Operator

`Organisms.GA.Get_Predation_and_Fitness_Operators.get_predation_and_fitness_operators` (*predation_infor-*
fitness_infor-
pop-
u-
la-
tion,
gen-
er-
a-
tions,
no_of_cp-
print_de

This def will set up the predation and fitness operators to be by the genetic algorithm.

Parameters

- **predation_information** (*dict.*) – All the informatino about the predation operator.
- **fitness_information** (*dict.*) – All the information about the fitness operator
- **population** (*Organisms.GA.Population*) – The population
- **generations** – The number of generations being run
- **generations** – int
- **no_of_cpus** (*int*) – The number of cpus that are to be used by your genetic algorithm run.
- **print_details** (*bool.*) – Print information to the terminal

Returns The predation operator object and the fitness operator object

Return type A `Organisms.GA.Predation_Operators.Predation_Operator` object and a `Organisms.GA.Fitness_Operators.Fitness_Operator` object

`Organisms.GA.Get_Predation_and_Fitness_Operators.get_predation_operator` (*predation_infor-*
fit-
ness_infor-
pop-
u-
la-
tion,
no_of_cpus,
print_details)

This def will set up the predation operator to be by the genetic algorithm. This is dependent on the entry for the 'Predation_Switch' given in the predation_information.

Parameters

- **predation_information** (*dict.*) – All the informatino about the predation operator.
- **fitness_information** (*dict.*) – All the information about the fitness operator
- **population** (*Organisms.GA.Population*) – The population
- **no_of_cpus** (*int*) – The number of cpus that are to be used by your genetic algorithm run.
- **print_details** (*bool.*) – Print information to the terminal

Returns The predation operator object

Return type `Organisms.GA.Predation_Operators.Predation_Operator`

Predation Operators

Predation Operator

Meow

```
class Organisms.GA.Predation_Operators.Predation_Operator.Predation_Operator (predation_informati  
pop-  
u-  
la-  
tion,  
print_details)
```

This is an abstract class to act as the skeleton of the `Predation_Operator` class.

Parameters

- **Predation_Information** (*dict.*) – This contains all the information needed by the Predation Operator you want to use to run.
- **population** (*Organisms.GA.Population*) – This is the population that this Operator will be controlling to make sure that no two clusters in the population have the same energy.
- **print_details** (*bool*) – Print details of the predation operator, like verbose

abstract add_to_database (*collection*)

Add clusters similarities to the CNA database to be stored for future generations.

Parameters *collection* (*Organisms.GA.Collection.Collection*) – update the fitnesses of clusters in the collection.

abstract assess_for_violations (*offspring_pool, force_replace_pop_clusters_with_offspring*)

This definition is designed to determine which offspring (and the clusters in the population) violate the Predation Operator. It will not remove or change any clusters in the offspring or population, but instead will record which offspring violate the Predation Operator.

It will also recommend which clusters in the population should be removed and be replaced by which offspring prior to the natural selection process. Here, the cluster in the population and the offspring will be in violation of each other, however it may be advantageous to keep the offspring rather than the cluster in the population as the offspring is fitter than the cluster in the offspring

Parameters

- **offspring_pool** (*Organisms.GA.Offspring_Pool.Offspring_Pool*) – This is the collection of offspring to assess for violations to the Predation Operator.
- **force_replace_pop_clusters_with_offspring** (*bool.*) – This will tell the genetic algorithm whether to swap clusters in the population with offspring if the predation operator indicates they are the same but the predation operator has a better fitness value than the cluster in the population.

Returns A list of the names of the offspring to be removed * **force_replacement** (*tuple of (int, int)*): A list of the clusters in the population that should be replaced, and the offspring they should be replaced by.

Return type

- **offspring_to_remove** (*tuple of ints*)

check ()

This method will check to make sure that the predation operator is available.

abstract check_initial_population (*return_report=False*)

This definition is responsible for making sure that the initialised population obeys the Predation Operator of interest.

Parameters **return_report** (*bool.*) – Will return a dict with all the information about what clusters are similar to what other clusters in the population.

Returns a list of the clusters to remove from the population as they violate the Predation Operator. Format is [(index in population, name fo cluster),...] * **CNA_report** (*dict.*): a dictionary with information on the clusters being removed and the other clusters in the population which have caused the violation to the SCM Predation Operator. This information is only used to display information so they know why there are violations to the Predation Operator when they occur. For is {removed cluster: [list of clusters that this cluster is similar to in the population.]}

Return type

- **clusters_to_remove** (*list of ints*)

abstract remove_from_database (*cluster_names_to_remove*)

Clusters to remove from the CNA database

Parameters **cluster_names_to_remove** (*list of ints*) – A list of the names of all the clusters to remove from the CNA database

remove_offspring_and_replace_with_population_that_violate_predation_operator (*population, offspring_pool, offspring_to_remove, population_to_be_replaced_by_offspring*)

This method will remove similar offspring in the offspring_pool, and replace any offspring with clusters in the population (if desired).

Parameters

- **population** (*Organisms.GA.Population.Population*) – This is the population.
- **offspring_pool** (*Organisms.GA.Offspring_Pool.Offspring_Pool*) – This is all the offspring.
- **offspring_to_remove** (*list of int*) – These are all the offspring to remove from the offspring_pool.
- **population_to_be_replaced_by_offspring** (*list of (int, int)*) – These are the names of the clusters in the population to be replaced by offspring in the offspring_pool. This is a list of (name of cluster in population to be replaced, name of offsrng to replaced that cluster).

remove_offspring_that_violate_the_predation_operator (*offspring_pool, offspring_to_remove*)

Remove offspring from the Offspring_Pool

Parameters

- **offspring_pool** (`Organisms.GA.Offspring_Pool.Offspring_Pool`) – This is all the offspring.
- **offspring_to_remove** (*list of int*) – These are all the offspring to remove from the `offspring_pool`.

replace_population_with_offspring (*population, offspring_pool, population_to_be_replaced_by_offspring*)

Replace clusters in the population with offspring in the `offspring_pool`. These population clusters are replaced by similar offspring that have higher fitnesses than the offspring in the `Offspring_Pool`.

Parameters

- **population** (`Organisms.GA.Population.Population`) – This is the population.
- **offspring_pool** (`Organisms.GA.Offspring_Pool.Offspring_Pool`) – This is all the offspring.
- **population_to_be_replaced_by_offspring** (*list of (int, int)*) – These are the names of the clusters in the population to be replaced by offspring in the `offspring_pool`. This is a list of (name of cluster in population to be replaced, name of offspring to replace that cluster).

abstract reset ()

Reset the CNA database with no inputs

No_Predation_Operator.py

This class is designed to allow the user to not use a predation operator during the genetic algorithm run.

No_Predation_Operator.py, Geoffrey Weal, 28/10/2018

This is one of the Predation Operators that can be used by the genetic algorithm program.

```
class Organisms.GA.Predation_Operators.No_Predation_Operator.No_Predation_Operator (Predation_
                                                                    pop-
                                                                    u-
                                                                    la-
                                                                    tion,
                                                                    print_detai)
```

This is one of the Predation Operators that can be used by the genetic algorithm program.

This Predation Operator will not do anything. It is the option to pick if you do not want to remove offspring or clusters from the population due to being energetically or structurally similar.

Parameters

- **Predation_Information** (*dict.*) – This contains all the information needed by the Predation Operator you want to use to run.
- **population** (`Organisms.GA.Population`) – This is the population that this Operator will be controlling to make sure that no two clusters in the population have the same energy.
- **print_details** (*bool*) – Print details of the predation operator, like verbose

add_to_database (*collection*)

Add clusters similarities to the CNA database to be stored for future generations.

This does not do anything since this predation method does nothing

Parameters `collection` (`Organisms.GA.Collection.Collection`) – update the fitnesses of clusters in the collection.

assess_for_violations (`offspring_pool, force_replace_pop_clusters_with_offspring`)

This definition is designed to determine which offspring (and the clusters in the population) violate the predation Operator. It will not remove or change any clusters in the offspring or population, but instead will record which offspring violate the predation Operator.

Since this ‘Operator’ does nothing, this def will not do anything.

It will return two tuples with nothing in them, as required by this def.

Parameters

- **offspring_pool** (`Organisms.GA.Offspring_Pool.Offspring_Pool`) – This is the collection of offspring to assess for violations to the Predation Operator.
- **force_replace_pop_clusters_with_offspring** (`bool.`) – This will tell the genetic algorithm whether to swap clusters in the population with offspring if the predation operator indicates they are the same but the predation operator has a better fitness value than the cluster in the population.

Returns A list of the names of the offspring to be removed * **force_replacement** (*tuple of (int, int)*): A list of the clusters in the population that should be replaced, and the offspring they should be replaced by.

Return type

- **offspring_to_remove** (*tuple of ints*)

check_initial_population (`return_report=False`)

This definition is responsible for making sure that the initialised population obeys the Energy Predation Operator.

Since this ‘Operator’ does nothing, this def will not do anything. It will report back that it has not removed any cluster

Parameters **return_report** (`bool.`) – Will return a dict with all the information about what clusters are similar to what other clusters in the population.

Returns a list of the clusters to remove from the population as they violate the Predation Operator. Format is [(index in population, name fo cluster),...] * **CNA_report** (*dict.*): a dictionary with information on the clusters being removed and the other clusters in the population which have caused the violation to the SCM Predation Operator. This information is only used to display information so they know why there are violations to the Predation Operator when they occur. For is {removed cluster: [list of clusters that this cluster is similar to in the population.]}

Return type

- **clusters_to_remove** (*list of ints*)

remove_from_database (`cluster_names_to_remove`)

Clusters to remove from the CNA database

This does not do anything since this predation method does nothing

Parameters **cluster_names_to_remove** (*list of ints*) – A list of the names of all the clusters to remove from the CNA database

reset ()

Reset the CNA database with no inputs

This does not do anything since this predation method does nothing

Energy_Predation_Operator.py

This class is designed to allow the user to implement the Energy Predation Operator into your genetic algorithm run.

Energy_Predation_Operator.py, Geoffrey Weal, 28/10/2018

This is one of the Predation Operators that can be used by the genetic algorithm program.

```
class Organisms.GA.Predation_Operators.Energy_Predation_Operator.Energy_Predation_Operator
```

This is one of the Predation Operators that can be used by the genetic algorithm program.

This class will allow the genetic algorithm to procedure without any Predation Operator. Instead, this class provide a way to use the energetic fitness to obtain a fitness for each cluster. This Operator will not remove any clusters from the population or the offspring pool.

Parameters

- **Predation_Information** (*dict.*) – This contains all the information needed by the Predation Operator you want to use to run.
- **population** (*Organisms.GA.Population*) – This is the population that this Operator will be controlling to make sure that no two clusters in the population have the same energy.
- **print_details** (*bool*) – Print details of the predation operator, like verbose

Energy_Predation_Operators_Options (*predation_information*)

This method is designed to add and check the options that you have placed in to the predation_information dictionary for the Energy Predation Operator

Parameters **predation_information** (*dict.*) – This contains all the information needed by the Predation Operator you want to use to run.

add_to_database (*collection*)

Add clusters similarities to the CNA database to be stored for future generations.

Parameters **collection** (*Organisms.GA.Collection.Collection*) – update the fitnesses of clusters in the collection.

assess_for_violations (*offspring_pool, force_replace_pop_clusters_with_offspring*)

This definition is designed to determine which offspring (and the clusters in the population) violate the predation Operator. It will not remove or change any clusters in the offspring or population, but instead will record which offspring violate the predation Operator.

It will also recommend which clusters in the population should be removed and be replaced by which offspring prior to the natural selection process. Here, the cluster in the population and the offspring will be in violation of each other, however it may be advantageous to keep the offspring rather than the cluster in the population as the offspring is fitter than the cluster in the offspring

Parameters

- **offspring_pool** (*Organisms.GA.Offspring_Pool.Offspring_Pool*) – This is the collection of offspring to assess for violations to the Predation Operator.
- **force_replace_pop_clusters_with_offspring** (*bool.*) – This will tell the genetic algorithm whether to swap clusters in the population with offspring if the predation

operator indicates they are the same but the predation operator has a better fitness value than the cluster in the population.

Returns A list of the names of the offspring to be removed * **force_replacement** (*tuple of (int, int)*): A list of the clusters in the population that should be replaced, and the offspring they should be replaced by.

Return type

- **offspring_to_remove** (*tuple of ints*)

check_initial_population (*return_report=False*)

This definition is responsible for making sure that the initialised population obeys the Predation Operator of interest.

Parameters **return_report** (*bool.*) – Will return a dict with all the information about what clusters are similar to what other clusters in the population.

Returns a list of the clusters to remove from the population as they violate the Predation Operator. Format is [(index in population, name fo cluster),...] * **CNA_report** (*dict.*): a dictionary with information on the clusters being removed and the other clusters in the population which have caused the violation to the SCM Predation Operator. This information is only used to display information so they know why there are violations to the Predation Operator when they occur. For is {removed cluster: [list of clusters that this cluster is similar to in the population.]}

Return type

- **clusters_to_remove** (*list of ints*)

get_energy_predation_methods ()

This method is designed to import the methods needed to run this Predation Operator, using with the 'simple' or 'comprehensive' mode.

remove_from_database (*cluster_names_to_remove*)

Clusters to remove from the CNA database

Parameters **cluster_names_to_remove** (*list of ints*) – A list of the names of all the clusters to remove from the CNA database

reset ()

Reset the CNA database with no inputs

Energy Predation Operator Scripts

Simple_Energy_Predation_Operator.py

Meow

Energy_Diversity_Scheme.py, Geoffrey Weal, 28/10/2018

This is one of the Diversity schemes that can be used by the genetic algorithm program. This is the Energy Diversity Scheme.

This scheme works by preventing the population from having clusters with the same energy existing in the population at any one time.

Organisms.GA.Predation_Operators.Energy_Predation_Operator_Scripts.Simple_Energy_Predation

This method allows us to check that the collection does not violate the simple diversity scheme after it has been perform on the collection.

The collection is either the Population of Offspring_Pool.

Parameters collection (`Population` or `Offspring_Pool`) – This is the collection to record. This is either the instance of the Population or the Offspring_Pool

`Organisms.GA.Predation_Operators.Energy_Predation_Operator_Scripts.Simple_Energy_Predation`

This will assess which offspring to remove before the natural selection process.

The offspring are assessed against clusters in the population. Offspring are removed from the offsprings if: * They have an energy equal to another offspring * They have an energy equal to another cluster in the population.

Parameters offsprings (`Offspring_Pool`) – This is the offspring that you want to add eventually using the natural selection process to the population.

`Organisms.GA.Predation_Operators.Energy_Predation_Operator_Scripts.Simple_Energy_Predation`

This definition is responsible for making sure that the initialised population obeys the Energy Diversity Scheme.

Here, the clusters in the population are checked to see if they have the same energy. This is needed for after the population has been initially populated with randomly generated clusters and clusters the user has specified.

Parameters return_report (`bool.`) –

Returns Contains a list of [the index of cluster in pop, the dir of the cluster] that have been removed from the population as they violate the Energy Diversity Scheme. * **same_energies_report** (`{int: [int,...]}`): Contain a more detailed report of what this method has done by returning a dictionary in the format of {dir of kept cluster: [list of dirs of clusters that have been removed because they have the same energy as the cluster that was kept.]}

Return type

- **clusters_to_remove** (`[[int,str],...]`)

Comprehensive_Energy_Predation_Operator_energy.py

Meow

Energy_Diversity_Scheme.py, Geoffrey Weal, 28/10/2018

This is one of the Diversity schemes that can be used by the genetic algorithm program. This is the Energy Diversity Scheme.

This scheme works by preventing the population from having clusters with the same energy existing in the population at any one time.

`Organisms.GA.Predation_Operators.Energy_Predation_Operator_Scripts.Comprehensive_Energy_Pre`

This method allows us to check that the collection does not violate the comprehensive diversity scheme after it has been perform on the collection.

The collection is either the Population of Offspring_Pool.

Parameters collection (`Population` or `Offspring_Pool`) – This is the collection to record. This is either the instance of the Population or the Offspring_Pool

class Organisms.GA.Predation_Operators.Energy_Predation_Operator_Scripts.Comprehensive_Ene

This is used by the Remove_Cluster_Due_To_Diversity_Violation definition to store information in an easy way to help the user understand what is going on in this method

Parameters

- **collection** (*Population or Offspring_Pool*) – This is the collection to record. This is either the instance of the Population or the Offspring_Pool
- **collection_type** (*str.*) – This describes if the cluster recorded is in the population (given as ‘pop’) or the offspring (given as ‘off’).
- **index** (*int*) – The position/index of the cluster in the collection.

Organisms.GA.Predation_Operators.Energy_Predation_Operator_Scripts.Comprehensive_Energy_Pr

This will assess which offspring to remove before the natural selection process.

The offspring are assessed against clusters in the population. Offspring are removed from the offspring_pool if: * They have an energy equal to another offspring * They have an energy equal to another cluster in the population.

Parameters offspring_pool (*Offspring_Pool*) – This is the offspring that you want to add eventually using the natural selection process to the population.

Organisms.GA.Predation_Operators.Energy_Predation_Operator_Scripts.Comprehensive_Energy_Pr

This will assess which offspring to remove before the natural selection process.

The offspring are assessed against clusters in the population. Offspring are removed from the offspring_pool if: * They have an energy equal to another offspring * They have an energy equal to another cluster in the population.

Parameters offspring_pool (*Offspring_Pool*) – This is the offspring that you want to add eventually using the natural selection process to the population.

Organisms.GA.Predation_Operators.Energy_Predation_Operator_Scripts.Comprehensive_Energy_Pr

This definition is responsible for making sure that the initialised population obeys the Energy Diversity Scheme.

Here, the clusters in the population are checked to see if they have the same energy. This is needed for after the population has been initially populated with randomly generated clusters and clusters the user has specified.

Parameters return_report (*bool.*) – This indicates if the user want to return a report on which clusters were removed and why (i.e. what clusters it was similar to energetically.)

Returns Contains a list of [the index of cluster in pop, the name of the cluster] that have been removed from the population as they violate the Energy Diversity Scheme. * **Energy_Diversity_report** (*{int: [int,...]}*): Contain a more detailed report of what this method has done by returning a dictionary in the format of {name of kept cluster: [list of names of clusters that have been removed because they have the same energy as the cluster that was kept.]}

Return type

- **clusters_to_remove** (*[[int,str],...]*)

Comprehensive_Energy_Predation_Operator_fitness.py

Meow

Energy_Diversity_Scheme.py, Geoffrey Weal, 28/10/2018

This is one of the Diversity schemes that can be used by the genetic algorithm program. This is the Energy Diversity Scheme.

This scheme works by preventing the population from having clusters with the same energy existing in the population at any one time.

```
Organisms.GA.Predation_Operators.Energy_Predation_Operator_Scripts.Comprehensive_Energy_Pre
```

This method allows us to check that the collection does not violate the comprehensive diversity scheme after it has been perform on the collection.

The collection is either the Population or Offspring_Pool.

Parameters `collection` (`Population` or `Offspring_Pool`) – This is the collection to record. This is either the instance of the `Population` or the `Offspring_Pool`

```
class Organisms.GA.Predation_Operators.Energy_Predation_Operator_Scripts.Comprehensive_Ene
```

This is used by the `Remove_Cluster_Due_To_Diversity_Violation` definition to store information in an easy way to help the user understand what is going on in this method

Parameters

- **collection** (`Population` or `Offspring_Pool`) – This is the collection to record. This is either the instance of the `Population` or the `Offspring_Pool`
- **collection_type** (`str.`) – This describes if the cluster recorded is in the population (given as 'pop') or the offspring (given as 'off').
- **index** (`int`) – The position/index of the cluster in the collection.

```
Organisms.GA.Predation_Operators.Energy_Predation_Operator_Scripts.Comprehensive_Energy_Pre
```

This will assess which offspring to remove before the natural selection process.

The offspring are assessed against clusters in the population. Offspring are removed from the `offspring_pool` if: * They have an energy equal to another offspring * They have an energy equal to another cluster in the population.

Parameters `offspring_pool` (`Offspring_Pool`) – This is the offspring that you want to add eventually using the natural selection process to the population.

```
Organisms.GA.Predation_Operators.Energy_Predation_Operator_Scripts.Comprehensive_Energy_Pre
```

This will assess which offspring to remove before the natural selection process.

The offspring are assessed against clusters in the population. Offspring are removed from the `offspring_pool` if: * They have an energy equal to another offspring * They have an energy equal to another cluster in the

population.

Parameters `offspring_pool` (`Offspring_Pool`) – This is the offspring that you want to add eventually using the natural selection process to the population.

`Organisms.GA.Predation_Operators.Energy_Predation_Operator_Scripts.Comprehensive_Energy_Predation_Operator`

This definition is responsible for making sure that the initialised population obeys the Energy Diversity Scheme.

Here, the clusters in the population are checked to see if they have the same energy. This is needed for after the population has been initially populated with randomly generated clusters and clusters the user has specified.

Parameters `return_report` (`bool`.) – This indicates if the user want to return a report on which clusters were removed and why (i.e. what clusters it was similar to energetically.)

Returns Contains a list of [the index of cluster in pop, the name of the cluster] that have been removed from the population as they violate the Energy Diversity Scheme. * **Energy_Diversity_report** (`{int: [int,...]}`): Contain a more detailed report of what this method has done by returning a dictionary in the format of {name of kept cluster: [list of names of clusters that have been removed because they have the same energy as the cluster that was kept.]}

Return type

- `clusters_to_remove` (`[[int,str],...]`)

IDCM_Predation_Operator.py

hello

IDCM_Predation_Operator.py, Geoffrey Weal, 25/12/2019

This is one of the Predation Operators that can be used by the genetic algorithm program. This is the IDCM Predation Operator.

This Operator works by preventing the population from having clusters with the same ...

```
class Organisms.GA.Predation_Operators.IDCM_Predation_Operator.Cluster_Block(collection,  
                                                                           col-  
                                                                           lec-  
                                                                           tion_type,  
                                                                           in-  
                                                                           dex)
```

This is used by the `Remove_Cluster_Due_To_Predation_Violation` definition to store information in an easy way to help the user understand what is going on in this method

Parameters

- **collection** (`Population` or `Offspring_Pool`) – This is the collection to record. This is either the instance of the `Population` or the `Offspring_Pool`
- **collection_type** (`str.`) – This describes if the cluster recorded is in the population (given as ‘pop’) or the offspring (given as ‘off’).
- **index** (`int`) – The position/index of the cluster in the collection.

class Organisms.GA.Predation_Operators.IDCM_Predation_Operator.IDCM_Predation_Operator (*Pred*

*pop-
u-
la-
tion,
no_o.
print*

This is a Predation Operator for identifying if two clusters are identical using a style of a euclidean distance matrix.

This is based on the method given in MEGA to describe the IDCM Operator.

MEGA is described in more detail in <https://pubs.acs.org/doi/abs/10.1021/acs.jpcc.6b12848>

Parameters

- **Predation_Information** (*dict.*) – This contains all the information needed by the Predation Operator you want to use to run.
- **population** (*Organisms.GA.Population*) – This is the population that this Operator will be controlling to make sure that no two clusters in the population have the same energy.
- **print_details** (*bool*) – Print details of the predation operator, like verbose

add_to_database (*collection*)

Add clusters similarities to the CNA database to be stored for future generations.

Parameters collection (*Organisms.GA.Collection.Collection*) – update the fitnesses of clusters in the collection.

assess_for_violations (*offspring_pool, force_replace_pop_clusters_with_offspring*)

The offspring are assessed against clusters in the population. Offspring are removed from the offspring_pool if: 1) They are geometrically the same as another offspring 2) They are geometrically the same as another cluster in the population.

Must return the clusters that have been removed from population and offspring_pool lists.

Parameters

- **offspring_pool** (*Organisms.GA.Offspring_Pool.Offspring_Pool*) – This is the collection of offspring to assess for violations to the Predation Operator.
- **force_replace_pop_clusters_with_offspring** (*bool.*) – This will tell the genetic algorithm whether to swap clusters in the population with offspring if the predation operator indicates they are the same but the predation operator has a better fitness value than the cluster in the population.

Returns A list of the names of the offspring to be removed * **force_replacement** (*tuple of (int, int)*): A list of the clusters in the population that should be replaced, and the offspring they should be replaced by.

Return type

- **offspring_to_remove** (*tuple of ints*)

assess_for_violations_force_replacement (*offspring_pool*)

The offspring are assessed against clusters in the population. Offspring are removed from the offspring_pool if: 1) They are geometrically the same as another offspring 2) They are geometrically the same as another cluster in the population.

Must return the clusters that have been removed from population and offspring_pool lists.

Parameters `offspring_pool` (`Organisms.GA.Offspring_Pool.Offspring_Pool`) – This is the collection of offspring to assess for violations to the Predation Operator.

Returns A list of the names of the offspring to be removed * **force_replacement** (*tuple of (int, int)*): A list of the clusters in the population that should be replaced, and the offspring they should be replaced by.

Return type

- **offspring_to_remove** (*tuple of ints*)

assess_for_violations_message (*swap_cluster_in_pop_with_offspring, offspring_to_remove_as_too_many_pop_in_similar_common, offspring_to_remove_due_to_pop, offspring_to_remove_due_to_other_offspring*)

This method will tell the information that offspring are removed or replaced and why.

Parameters

- **swap_cluster_in_pop_with_offspring** (*tuple of (Organisms.GA.SCM_Predation_Operator.Cluster_Block, Organisms.GA.SCM_Predation_Operator.Cluster_Block)*) – The clusters in the population to swap out, and the offspring to swap in its place. These will be replaced because the offspring has a higher fitness than the cluster in the population, even though they are similar.
- **offspring_to_remove_as_too_many_pop_in_similar_common** (*tuple of (Organisms.GA.SCM_Predation_Operator.Cluster_Block, Organisms.GA.SCM_Predation_Operator.Cluster_Block)*) – Remove offspring because there are too many similar clusters to it in the population already.
- **offspring_to_remove_due_to_pop** (*tuple of (Organisms.GA.SCM_Predation_Operator.Cluster_Block, Organisms.GA.SCM_Predation_Operator.Cluster_Block)*) – Remove offspring because they are similar to a cluster in the population. These offspring are less fit than their similar counterparts in the population.
- **offspring_to_remove_due_to_other_offspring** (*tuple of (Organisms.GA.SCM_Predation_Operator.Cluster_Block, Organisms.GA.SCM_Predation_Operator.Cluster_Block)*) – Remove offspring because they are similar to other clusters in the Offspring_Pool. This offspring is less fit than their similar counterparts in the offspring_pool.

assess_for_violations_no_force_replacement (*offspring_pool*)

The offspring are assessed against clusters in the population. Offspring are removed from the offspring_pool if: 1) They are geometrically the same as another offspring 2) They are geometrically the same as another cluster in the population.

Must return the clusters that have been removed from population and offspring_pool lists.

Parameters `offspring_pool` (`Organisms.GA.Offspring_Pool.Offspring_Pool`) – This is the collection of offspring to assess for violations to the Predation Operator.

Returns A list of the names of the offspring to be removed * **force_replacement** (*tuple of (int, int)*): A list of the clusters in the population that should be replaced, and the offspring they should be replaced by.

Return type

- **offspring_to_remove** (*tuple of ints*)

check_database (*collections*)

This method will check to make sure that all the clusters comparisons in the self.LoD_comparison_database are false

I.e. check that all the clusters in the collections are not identical

Parameters *collections* (*list of Organisms.GA.Collection.Collection*)

– This is a list of all the Populations and Offspring_Pool in your genetic algorithm to check.

check_initial_population (*return_report=False*)

This definition is responsible for making sure that the initialised population obeys the Predation Operator of interest.

Parameters *return_report* (*bool.*) – Will return a dict with all the information about what clusters are similar to what other clusters in the population.

Returns a list of the clusters to remove from the population as they violate the Predation Operator. Format is [(index in population, name fo cluster),...] * **CNA_report** (*dict.*): a dictionary with information on the clusters being removed and the other clusters in the population which have caused the violation to the SCM Predation Operator. This information is only used to display information so they know why there are violations to the Predation Operator when they occur. For is {removed cluster: [list of clusters that this cluster is similar to in the population.]}

Return type

- *clusters_to_remove* (*list of ints*)

get_identical_structures_initial_population ()

This method is designed to identify which clusters in the LoD_database (from the initial population) are identify using this method.

This will place all the results from this into self.LoD_comparison_database

pop_identical_structures ()

This method will identify which clusters need to be removed from self.LoD_comparison_database based on being identical via this method, and will remove the entries of those clusters from self.LoD_database and self.LoD_comparison_database

Returns a list of the clusters to remove from the population as they violate the Predation Operator. Format is [(index in population, name fo cluster),...] * **identical_structures_report** (*dict.*): a dictionary with information on the clusters being removed and the other clusters in the population which have caused the violation to the SCM Predation Operator. This information is only used to display information so they know why there are violations to the Predation Operator when they occur. For is {removed cluster: [list of clusters that this cluster is similar to in the population.]}

Return type

- *clusters_to_remove* (*list of ints*)

remove_from_database (*cluster_names_to_remove*)

Clusters to remove from the CNA database

Parameters *cluster_names_to_remove* (*list of ints*) – A list of the names of all the clusters to remove from the CNA database

remove_similar_clusters_in_population (*clusters_to_remove*)

This method will remove the similar clusters from the population.

Parameters *cluster_names_to_remove* (*list of ints*) – A list of the names of all the clusters to remove from the CNA database

reset ()

Reset the CNA database with no inputs

update_LoD_database (collection)

This method will add the collection to the LoD_database.

Parameters collection (`Organisms.GA.Population.Population` or `Organisms.GA.Offspring_Pool.Offspring_Pool`) – This is the collection to add to the LoD_database

IDCM Predation Operator Scripts

IDCM_Methods.py

Meow

`Organisms.GA.Predation_Operators.IDCM_Predation_Operator_Scripts.IDCM_Methods.LoD_compare_t`

This method will determine if these two clusters are structurally similar or not based on the IDCM

Parameters

- **LoD_1** (*list of floats*) – This is the list of the interatomic distances in cluster 1
- **LoD_2** (*list of floats*) – This is the list of the interatomic distances in cluster 2
- **percentage_diff** (*float*) – If the differences between all the distances in LoD_1 and LoD_2 are less than this percentage difference, they are the same. If any one difference is greater than this percentage, the two clusters are different.

Returns If the differences between all the distances in LoD_1 and LoD_2 are less than percentage_diff, they are the same. If any one difference is greater than percentage_diff percent, the two clusters are different.

Return type bool.

`Organisms.GA.Predation_Operators.IDCM_Predation_Operator_Scripts.IDCM_Methods.get_cluster_c`

This method give a list of the interatomic distances between every atom in the cluster.

Parameters

- **cluster** (`Organisms.GA.Cluster.Cluster`) – This is the cluster to get all the interatomic distances between every atom.
- **neighbor_cutoff** (*float*) – If desired, this method can be programmed to not include any distances that are larger than some cutoff value. Given in Angstroms.

Returns A list of all the interatomic distances between every atom in the cluster.

Return type float

`Organisms.GA.Predation_Operators.IDCM_Predation_Operator_Scripts.IDCM_Methods.get_distance`

This gives the distance between two atoms in a cluster.

Returns The distance between two atoms in a cluster

Return type float

LoD_Comparison_Database.py

Meow

class Organisms.GA.Predation_Operators.IDCM_Predation_Operator_Scripts.LoD_Comparison_Database
This is a database that holds all the entries of CNA_Entry objects.

LoD_Similarity_Analysis (*dir_1, dir_2*)

This def will obtain the max similarity percentage from the compararison of two clusters, name_1 and name_2.

Parameters

- **name_1** (*int*) – The name of the first cluster you would like to look for an entry in the CNA_Database.
- **name_2** (*int*) – The name of the second cluster you would like to look for an entry in the CNA_Database.

Returns The result from the IDCM. This will be True if this cluster pair is in the database, False if not.

Rtypes bool

add (*dir_1, dir_2, result*)

This def will obtain the CNA Information for the comparison of name_1 and name_2, and add it to the CNA_Database

Parameters

- **name_1** (*int*) – The name of the first cluster to compare against.
- **name_2** (*int*) – The name of the second cluster to compare against.
- **result** (*bool*) – The result of the comparison of the IDCM. True if they are structurally similar by the IDCM, False if not.

are_all_entries_false (*all_collections*)

This method checks if all entries in the database are false. This method performs this after clusters have been removed from the population or offspring_pool due to violating the IDCM predatino operator. At this point, all entries should be false.

Parameters **all_collections** (*list of Organisms.GA.Collections.Collections*) – This is a list of the population and the offspring pool.

check_for_issues (*population*)

This method check to make sure there are no issue with the LoD Database after the natural selection process

Parameters **population** (*Organisms.GA.Population.Population*) – This is the population

get_cluster_names (*order=False*)

Will provide a list of all the names of all the clusters in the Collection

Parameters **order** (*bool*) – This tag will tell this method whether the user would like the list of names given in order.

Returns List of the names of all the clusters in the Population

get_entry (*dir_1*, *dir_2*)

This def will return the CNA results from the comparison of two clusters.

Parameters

- **name_1** (*int*) – the name of the first cluster you want to compare with.
- **name_2** (*int*) – the name of the second cluster you want to compare with.

Returns The CNA_Entry that contains all the CNA information about the comparison of these two clusters.

Rtypes CNA_Entry

is_cluster_pair_in_the_database (*dir_1*, *dir_2*)

Determine if a CNA entry exists for two clusters in the database

Parameters

- **name_1** (*int*) – The name of the first cluster you would like to look for an entry in the CNA_Database.
- **name_2** (*int*) – The name of the second cluster you would like to look for an entry in the CNA_Database.

Returns True if this cluster pair is in the database, False if not.

Rtypes bool

keys ()

give the names of the clusters in the database

Returns The list of the names of the clusters in the database.

Return type list of ints

make_database_table ()

This method prints a table of the database.

remove (*dir_to_remove*)

Remove all entries that exists in the database that are associated with a particular cluster.

Parameters **name_to_remove** (*int*) – The name of the cluster you would like to remove from the CNA_Database.

reset ()

Reset the LoD database with a new database

which_clusters_in_LoD_comparison_database_are_similar ()

This method will return a dict of all the similar clusters in the LoD database, where pairs of clusters have been deemed structurally similar by the IDCM.

Returns A dictionary of all the clusters that have been deemed structurally similar to each other. The format of this dictionary is {name_1: [name_2, name_3, ..., names of all the clusters that name_1 is structurally similar to by the IDCM], ... }.

Rtypes dict.

class Organisms.GA.Predation_Operators.IDCM_Predation_Operator_Scripts.LoD_Comparison_Datab
This is a Tree designed for the CNA_Database to hold references of CNA_Entry.

SCM_Predation_Operator.py

This class is designed to assess replicating features in the population and offspring as well as to provide a fitness factor which allows Natural Selection to differentiate between structures. This class is effectly an interface

```
class Organisms.GA.Predation_Operators.SCM_Predation_Operator.Cluster_Block (collection,  
col-  
lec-  
tion_type,  
in-  
dex)
```

This is used by the Remove_Cluster_Due_To_Predation_Violation definition to store information in an easy way to help the user understand what is going on in this method

Parameters

- **collection** (*Population or Offspring_Pool*) – This is the collection to record. This is either the instance of the Population or the Offspring_Pool
- **collection_type** (*str.*) – This describes if the cluster recorded is in the population (given as ‘pop’) or the offspring (given as ‘off’).
- **index** (*int*) – The position/index of the cluster in the collection.

```
class Organisms.GA.Predation_Operators.SCM_Predation_Operator.SCM_Predation_Operator (Predati  
fit-  
ness_inf  
pop-  
u-  
la-  
tion,  
no_of_c  
print_de)
```

This predation operator uses the similarities from the SCM to determine whether to exclude clusters from the population because they are too similar to each other.

Parameters

- **Predation_Information** (*dict.*) – This contains all the information needed by the Predation Operator you want to use to run.
- **fitness_information** (*dict.*) – This is all the settings needed for the SCM predation operator. This is needed if the fitness operator is the structure + energy fitness operator, where the CNA Database maybe the same database for the predation and fitness operator.
- **population** (*Organisms.GA.Population*) – This is the population that this Operator will be controlling to make sure that no two clusters in the population have the same energy.
- **no_of_cpus** (*int*) – This is the number of cpus to use.
- **print_details** (*bool*) – Print details of the predation operator, like verbose

```
add_to_database (collection)
```

Add clusters similarities to the CNA database to be stored for future generations.

Parameters **collection** (*Organisms.GA.Collection.Collection*) – update the fitnesses of clusters in the collection.

```
assess_for_violations (offspring_pool, force_replace_pop_clusters_with_offspring)
```

This definition is designed to determine which offspring (and the clusters in the population) violate the

Predation Operator. It will not remove or change any clusters in the offspring or population, but instead will record which offspring violate the Predation Operator.

It will return two tuples with nothing in them, as required by this def.

Parameters

- **offspring_pool** (`Organisms.GA.Offspring_Pool.Offspring_Pool`) – This is the collection of offspring to assess for violations to the Predation Operator.
- **force_replace_pop_clusters_with_offspring** (`bool.`) – This will tell the genetic algorithm whether to swap clusters in the population with offspring if the predation operator indicates they are the same but the predation operator has a better fitness value than the cluster in the population.

Returns A list of the names of the offspring to be removed * **force_replacement** (*tuple of (int, int)*): A list of the clusters in the population that should be replaced, and the offspring they should be replaced by.

Return type

- **offspring_to_remove** (*tuple of ints*)

assess_for_violations_force_replacement (*offspring_pool*)

This definition is designed to determine which offspring (and the clusters in the population) violate the Predation Operator. It will not remove or change any clusters in the offspring or population, but instead will record which offspring violate the Predation Operator.

It will return two tuples with nothing in them, as required by this def.

Parameters **offspring_pool** (`Organisms.GA.Offspring_Pool.Offspring_Pool`) – This is the collection of offspring to assess for violations to the Predation Operator.

Returns A list of the names of the offspring to be removed * **force_replacement** (*tuple of (int, int)*): A list of the clusters in the population that should be replaced, and the offspring they should be replaced by.

Return type

- **offspring_to_remove** (*tuple of ints*)

assess_for_violations_message (*swap_P_O, remove_O_similar_to_too_many_P, removal_O_P, removal_O_O*)

This method will tell the information that offspring are removed or replaced and why.

Parameters

- **swap_P_O** (*tuple of (Organisms.GA.SCM_Predation_Operator.Cluster_Block, Organisms.GA.SCM_Predation_Operator.Cluster_Block)*) – The clusters in the population to swap out, and the offspring to swap in its place. These will be replaced because the offspring has a higher fitness than the cluster in the population, even though they are similar.
- **remove_O_similar_to_too_many_P** (*tuple of (Organisms.GA.SCM_Predation_Operator.Cluster_Block, Organisms.GA.SCM_Predation_Operator.Cluster_Block)*) – Remove offspring because there are too many similar clusters to it in the population already.
- **removal_O_P** (*tuple of (Organisms.GA.SCM_Predation_Operator.Cluster_Block, Organisms.GA.SCM_Predation_Operator.Cluster_Block)*) – Remove offspring because they are similar to a cluster in

the population. These offspring are less fit than their similar counterparts in the population.

- **removal_O_O** (*tuple of (Organisms.GA.SCM_Predation_Operator.Cluster_Block, Organisms.GA.SCM_Predation_Operator.Cluster_Block)*) – Remove offspring because they are similar to other clusters in the Offspring_Pool. This offspring is less fit than their similar counterparts in the offspring_pool.

assess_for_violations_no_force_replacement (*offspring_pool*)

This definition is designed to determine which offspring (and the clusters in the population) violate the Predation Operator. It will not remove or change any clusters in the offspring or population, but instead will record which offspring violate the Predation Operator.

It will return two tuples with nothing in them, as required by this def.

Parameters offspring_pool (*Organisms.GA.Offspring_Pool.Offspring_Pool*) – This is the collection of offspring to assess for violations to the Predation Operator.

Returns A list of the names of the offspring to be removed * **force_replacement** (*tuple of (int, int)*): A list of the clusters in the population that should be replaced, and the offspring they should be replaced by.

Return type

- **offspring_to_remove** (*tuple of ints*)

check_initial_population (*return_report=False*)

This definition is responsible for making sure that the initialised population obeys the CNA Predation Operator.

Parameters return_report (*bool.*) – Will return a dict with all the information about what clusters are similar to what other clusters in the population.

Returns a list of the clusters to remove from the population as they violate the Predation Operator. Format is [(index in population, name fo cluster),...] * **CNA_report** (*dict.*): a dictionary with information on the clusters being removed and the other clusters in the population which have caused the violation to the SCM Predation Operator. This information is only used to display information so they know why there are violations to the Predation Operator when they occur. For is {removed cluster: [list of clusters that this cluster is similar to in the population.]}

Return type

- **clusters_to_remove** (*list of ints*)

get_similar_clusters_to_remove (*return_report*)

Will update the CNA_Database and return all the names of clusters in the population that violate the SCM predation operator

Parameters return_report (*bool.*) – Will return a dict with all the information about what clusters are similar to what other clusters in the population.

remove_from_database (*cluster_names_to_remove*)

Clusters to remove from the CNA database

Parameters cluster_names_to_remove (*list of ints*) – A list of the names of all the clusters to remove from the CNA database

reset ()

Reset the CNA database with no inputs

Fitness Operators

Fitness_Operator.py

This class is designed to assess replicating features in the population and offspring as well as to provide a fitness factor which allows Natural Selection to differentiate between structures. This class is effectively an interface

```
class Organisms.GA.Fitness_Operators.Fitness_Operator.Fitness_Operator (fitness_information,  
pop-  
ula-  
tion,  
print_details)
```

The Fitness class is designed to determine and assign the appropriate fitness value to clusters created during the genetic algorithm. This class can be thought as an extension of the predation Operator. This class contains all the appropriate steps that are needed for the fitnesses to be efficiently assigned to created clusters.

There are only two methods that need to be written for the predation Operator that is to be used. These are:

- `assign_population_fitnesses`
- `assign_all_fitnesses`

Note: This class is intended to work as an interface only. Refer to the instruction manual as to how to use this interface to build your own predation Operator.

Parameters

- **fitness_information** (*dict.*) – The information needed by the Fitness_Operator
- **population** (*Organisms.GA.Population*) – The population to assign fitnesses to
- **print_details** (*bool*) – Print the details of the energy fitness operator. True if yes, False if no

```
abstract assign_all_fitnesses_after_assess_against_predation_operator (all_offspring_pools,  
cur-  
rent_generation_no,  
off-  
spring_to_remove)
```

This method is to be used in the GA program. This will assign all the fitnesses of all clusters in the current generation (population and offspring) after the offspring are assessed to understand if they violate the predation scheme (i.e. an offspring is Class 1 similar to a cluster in the population, or another offspring).

See the description given for the “`assign_all_fitnesses`” def on what the crux of this method is.

If you write your own diversity and fitness classes, you do not need to implement this method in your fitness class.

Parameters

- **all_offspring_pools** (*list of Organisms.GA.Offspring_Pool*) – All of the offspring_pools
- **current_generation_no** (*int*) – The current generation
- **offspring_to_remove** (*list of int*) – a list of all the names of clusters in the offspring that will be removed.

```
abstract assign_all_fitnesses_after_natural_selection (current_generation_no)
```

This method is to be used in the GA program. This will assign all the fitnesses to all the clusters in the

population before the offspring are assessed to understand if they violate the predation operator (i.e. an offspring is Class 1 similar to a cluster in the population, or another offspring).

See the description given for the “assign_population_fitnesses” def on what the crux of this method is.

If you write your own diversity and fitness classes, you do not need to implement this method in your fitness class.

Parameters `current_generation_no` (*int*) – The current generation

abstract `assign_all_fitnesses_before_assess_against_predation_operator` (*all_offspring_pools*, *current_generation_no*)

This method is to be used in the GA program. This will assign all the fitnesses of all clusters in the current generation (population and offspring) before the offspring are assessed to understand if they violate the predation scheme (i.e. an offspring is Class 1 similar to a cluster in the population, or another offspring).

See the description given for the “assign_all_fitnesses” def on what the crux of this method is.

If you write your own diversity and fitness classes, you do not need to implement this method in your fitness class.

Parameters

- **all_offspring_pools** (*list of Organisms.GA.Offspring_Pool*) – All of the offspring_pools
- **current_generation_no** (*int*) – The current generation

abstract `assign_initial_population_fitnesses` ()

This method is designed to assign fitness values to the clusters of the population only at the start of the GA, when the population has been initialised

This is an abstract method. If you write your own diversity and fitness classes, you MUST write something for this method.

abstract `assign_resumed_population_fitnesses` (*resume_from_generation*)

This method is designed to assign fitness values to the clusters of the population only at the beginning of a resumed GA.

This is an abstract method. If you write your own diversity and fitness classes, you MUST write something for this method.

Parameters `resume_from_generation` (*int*) – The current generation to resume from.

check ()

This method will check that the self.fitness_switch is an available fitness operator.

Energy_Fitness.py

Meow

```
class Organisms.GA.Fitness_Operators.Energy_Fitness_Operator.Energy_Fitness_Operator (fitness_i,  
pre-  
da-  
tion_op  
pop-  
u-  
la-  
tion,  
print_de
```

The Fitness class is designed to determine and assign the appropriate fitness value to clusters created during the genetic algorithm. The fitnesses are based on the energies of the clusters in the offspring and the population.

There are only two methods that need to be written for the predation operator that is to be used. These are:

- `assign_population_fitnesses`
- `assign_all_fitnesses`

Parameters

- **`fitness_information`** (*dict.*) – This is all the information that is needed about the fitness class
- **`predation_operator`** (*Organisms.GA.Predation_Operator*) – This is the predation operator that this fitness class will take information from if needed to obtain a fitness.
- **`population`** (*Organisms.GA.Population*) – The population to assign fitnesses to
- **`print_details`** (*bool*) – Print the details of the energy fitness operator. True if yes, False if no

`add_to_database` (*collection*)

This method is required by `Organisms.GA.Fitness_Operators.Fitness_Operator`, but does not do anything

Parameters `collection` (*Organisms.GA.Collection.Collection*) – update the fitnesses of clusters in the collection.

`assign_all_fitnesses_after_assess_against_predation_operator` (*all_offspring_pools, current_generation_no, offspring_to_remove*)

This method is to be used in the GA program. This will assign all the fitnesses of all clusters in the current generation (population and offspring) after the offspring are assessed to understand if they violate the predation operator (i.e. an offspring is Class 1 similar to a cluster in the population, or another offspring).

See the description given for the “`assign_all_fitnesses`” def on what the crux of this method is.

If you write your own diversity and fitness classes, you do not need to implement this method in your fitness class.

Parameters

- **`all_offspring_pools`** (*list of Organisms.GA.Offspring_Pool*) – All of the `offspring_pools`
- **`current_generation_no`** (*int*) – The current generation
- **`offspring_to_remove`** (*list of int*) – a list of all the names of clusters in the offspring that will be removed.

`assign_all_fitnesses_after_natural_selection` (*current_generation_no*)

This method is to be used in the GA program. This will assign all the fitnesses to all the clusters in the population before the offspring are assessed to understand if they violate the predation operator (i.e. an offspring is Class 1 similar to a cluster in the population, or another offspring).

See the description given for the “`assign_population_fitnesses`” def on what the crux of this method is.

If you write your own diversity and fitness classes, you do not need to implement this method in your fitness class.

Parameters `current_generation_no` (*int*) – The current generation

assign_all_fitnesses_before_assess_against_predation_operator (*all_offspring_pools*, *current_generation_no*)

This method is to be used in the GA program. This will assign all the fitnesses of all clusters in the current generation (population and offspring) before the offspring are assessed to understand if they violate the predation operator (i.e. an offspring is Class 1 similar to a cluster in the population, or another offspring).

See the description given for the “assign_all_fitnesses” def on what the crux of this method is.

If you write your own diversity and fitness classes, you do not need to implement this method in your fitness class.

Parameters

- **all_offspring_pools** (*list of Organisms.GA.Offspring_Pool*) – list of all the offspring pools of offspring to assign fitness values to.
- **current_generation_no** (*int*) – The current generation

assign_initial_population_fitnesses ()

This method is designed to assign fitness values to the clusters of the population only at the start of the GA, when the population has been initialised

assign_resumed_population_fitnesses (*resume_from_generation*)

This method is designed to assign fitness values to the clusters of the population only at the beginning of a resumed GA.

Parameters `resume_from_generation` (*int*) – The current generation to resume from.

energy_fitness_options (*fitness_information*, *predation_operator*)

This method is designed to set the variables requires for this fitness.

Parameters

- **fitness_information** (*dict.*) – This is all the information that is needed about the fitness class
- **predation_operator** (*Organisms.GA.Predation_Operator*) – This is the predation operator that this fitness class will take information from if needed to obtain a fitness.

remove_from_database (*cluster_names_to_remove*)

This method is required by `Organisms.GA.Fitness_Operators.Fitness_Operator`, but does not do anything.

Parameters `cluster_names_to_remove` (*list of ints*) – A list of the names of all the clusters to remove from the CNA database

reset ()

This method is required by `Organisms.GA.Fitness_Operators.Fitness_Operator`, but does not do anything

Energetic_Fitness_Contribution.py

Meow

```
Organisms.GA.Fitness_Operators.Energetic_Fitness_Contribution.check_rho_i(rho_i,  
                                                                           clus-  
                                                                           ter,  
                                                                           max_energy,  
                                                                           min_energy)
```

This method will check that the rho value for the cluster is within the expected limits. If there are any errors, the program will mention what is wrong and terminate.

Parameters

- **rho_i** (*float*) – This is the rho value to check.
- **collection** (*Organisms.GA.Cluster*) – This is the cluster that contains the associated rho value to check.
- **min_energy** (*float*) – This is the lowest energy of the current lowest energetic structure across all the collections in the GA (e.g. [population, offspring]).
- **max_energy** (*float*) – This is the highest energy of the current highest energetic structure across all the collections in the GA (e.g. [population, offspring]).

```
Organisms.GA.Fitness_Operators.Energetic_Fitness_Contribution.get_energetic_fitness_contri
```

Get the fitness, based on the energy, for a cluster.

See [Theoretical study of Cu-Au nanoalloy clusters using a genetic algorithm, Darby et al., 116, 1536 \(2002\); doi: 10.1063/1.1429658](https://doi.org/10.1063/1.1429658)³⁹, page 1538 about this fitness equation, including the definition of max_energy and min_energy.

Parameters

- **cluster** (*Cluster*) – This is the cluster you want to obtain rho_i for.
- **population** (*Population*) – This is the population which will be used to get the max_energy and min_energy to obtain rho_i.

Returns The value for energetic fitness value for the cluster

Rtypes float

```
Organisms.GA.Fitness_Operators.Energetic_Fitness_Contribution.get_lowest_and_highest_energ
```

This method will return the value of the highest and lowest energy from a list of collections that are inputted into this method. The collections variable is a list of collections, for example [population,offspring] This is a private method.

Parameters **collections** (*list of collection objects*) – A list of all the collections that you want to compare..

³⁹ <http://dx.doi.org/10.1063/1.1429658>

Returns the lowest energy of clusters out of all the inputted collections, the maximum energy out of all the inputted collections.

Rtypes float, float

```
Organisms.GA.Fitness_Operators.Energetic_Fitness_Contribution.get_rho_i(cluster,  
                                                                           high-  
                                                                           est_energy,  
                                                                           low-  
                                                                           est_energy)
```

Get the rho of a cluster. Note that the max and min energy are based on that of the current population.

See Theoretical study of Cu-Au nanoalloy clusters using a genetic algorithm, Darby et al., 116, 1536 (2002); doi: 10.1063/1.1429658⁴⁰, page 1538 about this fitness equation, including the definition of max_energy and min_energy.

Parameters

- **cluster** (*Cluster*) – This is the cluster you want to obtain rho_i for.
- **highest_energy** (*float*) – This is the highest energy of the current highest energetic structure across all the collections in the GA (e.g. [population, offspring]).
- **lowest_energy** (*float*) – This is the lowest energy of the current lowest energetic structure across all the collections in the GA (e.g. [population, offspring]).

Returns The value for rho for the cluster

Rtypes float

SCM_and_Energy_Fitness_Operator.py

Meow

```
class Organisms.GA.Fitness_Operators.SCM_and_Energy_Fitness_Operator.SCM_and_Energy_Fitness
```

This class controls the how the fitness values for the Structural Comparison Method (SRM)-based predation operator are obtained.

Parameters

- **fitness_information** (*dict.*) – This is all the information that is needed about the fitness class

⁴⁰ <http://dx.doi.org/10.1063/1.1429658>

- **predation_operator** (*Organisms.GA.Predation_Operator*) – This is the predation operator that this fitness class will take information from if needed to obtain a fitness.
- **population** (*Organisms.GA.Population*) – The population to assign fitnesses to
- **generations** (*int*) – The number of generations that will be performed in the genetic algorithm.
- **no_of_cpus** (*int*) – the number of cpus to use when multiprocessing
- **print_details** (*bool*) – Print the details of the energy fitness operator. True if yes, False if no

add_to_database (*collection*)

Add clusters similarities to the CNA database to be stored for future generations.

Parameters **collection** (*Organisms.GA.Collection.Collection*) – update the fitnesses of clusters in the collection.

assign_all_fitnesses_after_assess_against_predation_operator (*all_offspring_pools*,
current_generation_no,
offspring_to_remove)

This method is to be used in the GA program. This will assign all the fitnesses of all clusters in the current generation (population and offspring) after the offspring are assessed to understand if they violate the predation operator (i.e. an offspring is Class 1 similar to a cluster in the population, or another offspring).

See the description given for the “assign_all_fitnesses” def on what the crux of this method is.

If you write your own diversity and fitness classes, you do not need to implement this method in your fitness class.

Parameters

- **all_offspring_pools** (*list of Organisms.GA.Offspring_Pool*) – All of the offspring_pools
- **current_generation_no** (*int*) – The current generation
- **offspring_to_remove** (*list of int*) – a list of all the names of clusters in the offspring that will be removed. Currently not needed, but kept in case.

assign_all_fitnesses_after_natural_selection (*current_generation_no*)

This method is to be used in the GA program. This will assign all the fitnesses to all the clusters in the population before the offspring are assessed to understand if they violate the predation operator (i.e. an offspring is Class 1 similar to a cluster in the population, or another offspring).

See the description given for the “assign_population_fitnesses” def on what the crux of this method is.

If you write your own diversity and fitness classes, you do not need to implement this method in your fitness class.

Parameters **current_generation_no** (*int*) – The current generation

assign_all_fitnesses_before_assess_against_predation_operator (*all_offspring_pools*,
current_generation_no)

This method is to be used in the GA program. This will assign all the fitnesses of all clusters in the current generation (population and offspring) before the offspring are assessed to understand if they violate the predation operator (i.e. an offspring is Class 1 similar to a cluster in the population, or another offspring).

See the description given for the “assign_all_fitnesses” def on what the crux of this method is.

If you write your own diversity and fitness classes, you do not need to implement this method in your fitness class.

Parameters

- **all_offspring_pools** (*list of Organisms.GA.Offspring_Pool*) – list of all the offspring pools of offspring to assign fitness values to.
- **current_generation_no** (*int*) – The current generation

assign_initial_population_fitnesses ()

This method is designed to assign fitness values to the clusters of the population only at the start of the GA, when the population has been initialised

assign_resumed_population_fitnesses (*resume_from_generation*)

This method is designed to assign fitness values to the clusters of the population only at the beginning of a resumed GA.

Parameters **resume_from_generation** (*int*) – The current generation to resume from.

convert_population_fitness_to_SCM_fitness_contribution (*population_fitness*)

This method will take the population fitness and use this to determine what the new coefficients (weights) for the energy and SCM coefficients are.

Parameters **population_fitness** (*float*) – The population ‘fitness value’. This value is some value between 0.0 and 1.0.

get_population_fitness (*cluster_SCM_simiarities, population_fitness_function*)

Obtain the ‘fitness’ of the population. This population fitness is based on the similarity values from all clusters in the population. This python definition will take the similarity values from all clusters in the population and get a ‘similarity value’ of the population. This definition will then take this population similarity value and convert it into a population fitness value. The population fitness value must be a read value between (and including) 0.0 and 1.0.

Parameters

- **cluster_SCM_simiarities** (*list of floats*) – The similarities of clusters of the population as obtained by the SCM. These are the sigma 1/2 similarities.
- **population_fitness_function** (*def*) – The python def that turns the population similarity value into a population fitness value.

is_there_an_similarity_range (*similarity_rounding*)

Determines if there is a range of similarities in the collection

Parameters **rounding** (*float*) – The rounding of the similarity of the cluster

returns Is there a range of similarities in the collection rtype bool

print_initial_message ()

At the start of the GA run, this message will be shown so that the user knows what their energy and fSRM (CNA)-based fitness contributions are at the start of the GA.

remove_from_database (*cluster_names_to_remove*)

Clusters to remove from the CNA database

Parameters **cluster_names_to_remove** (*list of ints*) – A list of the names of all the clusters to remove from the CNA database

reset ()

Reset the CNA database with no inputs

CNA_Fitness_Contribution.py

Meow

```
Organisms.GA.Fitness_Operators.CNA_Fitness_Contribution.get_CNA_fitness_contribution(cluster,  
max_sim  
min_min  
cna_dat  
col-  
lec-  
tion_fun  
cna_fitn
```

Get the fitness, based on the structural diversity as determined by the SCM, for a cluster.

Parameters

- **cluster** (*Organisms.GA.Cluster*) – This is the cluster you want to obtain rho_i for.
- **max_similarity** (*Not needed to be anything*) – The maximum similarity obtained from the population + offspring_pools. THIS IS NOT USED IN THIS METHOD, IS HERE FOR CONSISTANCY.
- **min_minimarity** (*Not needed to be anything*) – The minimum similarity obtained from the population + offspring_pools. THIS IS NOT USED IN THIS METHOD, IS HERE FOR CONSISTANCY.
- **cna_database** (*Organisms.GA.SCM_Scripts.CNA_Database*) – This is an in-memory database (not a ASE disk database) that records the similarities between clusters in the population.
- **cna_fitness_function** (*__func__*) – This is the function that converts the rho similarity into a fitness value.

```
Organisms.GA.Fitness_Operators.CNA_Fitness_Contribution.get_CNA_fitness_contribution_normal
```

Get the fitness, based on the structural diversity as determined by the SCM, for a cluster. This order parameter is the normalised similarity compared to the similarity of clusters in the population.

Parameters

- **cluster** (*Organisms.GA.Cluster*) – This is the cluster you want to obtain rho_i for.
- **max_similarity** (*float*) – The maximum similarity obtained from the population + offspring_pools
- **min_minimarity** (*float*) – The minimum similarity obtained from the population + offspring_pools
- **cna_database** (*Organisms.GA.SCM_Scripts.CNA_Database*) – This is an in-memory database (not a ASE disk database) that records the similarities between clusters in the population.
- **cna_fitness_function** (*__func__*) – This is the function that converts the rho similarity into a fitness value.

`Organisms.GA.Fitness_Operators.CNA_Fitness_Contribution.get_CNA_fitness_parameter` (*cluster*, *cna_database*, *collec- tion_function*)

Get the order parameter, based on the CNA, for a cluster.

Parameters

- **cluster** (*Organisms.GA.Cluster*) – This is the cluster you want to obtain rho_i for.
- **cna_database** (*Organisms.GA.SCM_Scripts.CNA_Database*) – This is an in-memory database (not a ASE disk database) that records the similarities between clusters in the population.
- **cna_fitness_function** (*__func__*) – This is the function that converts the rho similarity into a fitness value.

Returns *CNA_fitness_value*: This is the SRM fitness contribution to be used to obtain the fitness valuee

Rtypes float

`Organisms.GA.Fitness_Operators.CNA_Fitness_Contribution.get_CNA_fitness_parameter_normalise`

Get the order parameter, based on the CNA, for a cluster. This order parameter is the normalised similarity compared to the similarity of clusters in the population.

Parameters

- **cluster** (*Organisms.GA.Cluster*) – This is the cluster you want to obtain rho_i for.
- **max_similarity** (*float*) – The maximum similarity obtained from the population + offspring_pools
- **min_minimarity** (*float*) – The minimum similarity obtained from the population + offspring_pools
- **cna_database** (*Organisms.GA.SCM_Scripts.CNA_Database*) – This is an in-memory database (not a ASE disk database) that records the similarities between clusters in the population.
- **cna_fitness_function** (*__func__*) – This is the function that converts the rho similarity into a fitness value.

Returns *CNA_fitness_value*: This is the SRM fitness contribution to be used to obtain the fitness valuee

Rtypes float

Organisms.GA.Fitness_Operators.CNA_Fitness_Contribution.get_lowest_and_highest_similarities

This method will return the value of the highest and lowest similarities from a list of collections that are inputted into this method. The collections variable is a list of collections, for example [population,offspring] This is a private method.

Parameters **collections** (*list of collection objects*) – A list of all the collections that you want to compare..

Returns the lowest similarity of clusters out of all the inputted collections, the maximum similarity out of all the inputted collections.

Rtypes float, float

Fitness_Function.py

class Organisms.GA.Fitness_Operators.Fitness_Function.**Fitness_Function** (***entries*)

This class is designed to calculator the fitness value of a cluster, given a value of rho_i. This class will include all the relevant information required, such as parameters, needed for the functions to convert rho_i into a fitness value.

Parameters **entries** (*dict.*) – This is a dictionary that contains all the input variables required for the fitness function.

direct_function (*rho_i*)

This definition is designed to return the value for rho_i without any mathematical changes to it.

Equation written as required in the genetic algorithm.

Parameters **rho_i** (*float*) – a value

Returns rho_i

Rtypes float

exponential_function (*rho_i*)

This definition is designed to return the result of the exp function for this genetic algorithm.

Equation written as required in the genetic algorithm.

Parameters

• **rho_i** (*float*) – a value

• **alpha** (*float*) – a value

Returns value

Rtypes float

get_fitness (*rho_i*)

This def will give the fitness value for a given value of rho_i

Parameters **rho_i** (*float*) – a value

Returns value

Rtypes float

linear_function (*rho_i*)

This definition is designed to return the result of the linear function for this genetic algorithm.

Equation written as required in the genetic algorithm.

Parameters

- **rho_i** (*float*) – a value
- **gradient** (*float*) – a value
- **coefficient** (*float*) – a value

Returns value

Rtypes float

tanh_function (*rho_i*)

This definition is designed to return the result of the tanh function for this genetic algorithm.

Equation written as required in the genetic algorithm.

Parameters **rho_i** (*float*) – a value

Returns value

Rtypes float

SCM Scripts

SCM_initialisation.py

Meow

Organisms.GA.SCM_Scripts.SCM_initialisation.**get_SCM_methods** (*SCM_Scheme*)

This method will provide the methods needed to obtain the CNA_profile and similarity_profile, using either the T-SCM or the A-SCM.

Parameters **SCM_Scheme** (*str.*) – This determines if the SCM scheme being used is the “T-SCM” or “A-SCM”. Must be one of those two schemes.

Organisms.GA.SCM_Scripts.SCM_initialisation.**get_rCut_values** (*rCut_low*,
rCut_high,
rCut_resolution)

This method will give a list of rCut values that you want to perform the CNA across, in Angstroms

Parm rCut_low This is the lowest rCut value you want to obtain the CNA for.

Parm rCut_high This is the highest rCut value you want to obtain the CNA for.

Parm rCut_resolution This is the resolution of the rCut values you are performing all the CNA's with. The resolution can be given as one of two forms.

- If given as a float, rCut_resolution is the difference between rCut values that will be sampled. Note that your input for rCut_resolution is not divisible by (rCut_high-rCut_low), then rCut_high will not be included in rCuts.
- If given as a int, rCut_resolution is the number of rCut values in the list rCuts, evenly distributed between (and including) rCut_low and rCut_high.

Returns A list of all the rCut values you want the CNA to sample.

Return type list of floats

Organisms.GA.SCM_Scripts.SCM_initialisation.get_rCuts (*self*, *Predation_Information*)
Obtain the values for values of rCut the user wishes to investigate.

Parameters **Predation_Information** (*dict.*) – This contains all the information needed by the Predation Operator you want to use to run.

Returns A list of all the rCut values you want the CNA to sample.

Return type list of floats

Similarity_Profile.py

Meow

CNA_Entry.py, Geoffrey Weal, 29/10/2018

This class holds a CNA comparison entry between two clusters, Cluster cluster_1 and Cluster cluster_2, including the similarity_profile for different values of rCut, the average and the result of the comparison, based on the result from Geoff's Thesis

```
class Organisms.GA.SCM_Scripts.Similarity_Profile.Similarity_Profile (name_1,  
                                                                    name_2,  
                                                                    sim-  
                                                                    ilar-  
                                                                    ity_profile)
```

This class holds a CNA comparison entry between two clusters, Cluster 1 and Cluster 2, including the similarity_profile for different values of rCut, the average and the result of the comparison, based on the result from Geoff's Thesis

Parameters

- **name_1** (*int*) – Name of the first cluster.
- **name_2** (*int*) – Name of the first cluster.
- **similarity_profile** (*list of floats*) – The similarity profile between cluster 1 and cluster 2.

get_average ()

Get the average of similarity_profile

get_comparison ()

From the information from self.similarity_profile, this method will write the result of the comparison into self.similarity_category, as determined from the procedure created by Geoffrey Weal.

get_results ()

Get the result of the average of similarity_profile, and the result of the comparison.

Organisms.GA.SCM_Scripts.Similarity_Profile.get_comparison (*percentage_similarities*)

From the information from self.percentage_similarities, this method will write the result of the comparison into self.similarity_category, as determined from the procedure created by Geoffrey Weal.

Parameters **percentage_similarities** (*list of floats*) – These are a list of all the similarities from performing the SCM across many rCut values.

Returns The similarity class that the pair of clusters is assigned to based on the percentage_similarities, and the maximum similarity of percentage_similarities

Return type a str. and a float

A_SCM_Methods.py

Meow

A_SCM_Methods.py, Geoffrey Weal, 20/11/2018

This script is designed to include all the methods for performing structural recognition task in terms of the Atom-by-Atom Structural Comparison Method.

`Organisms.GA.SCM_Scripts.A_SCM_Methods.get_CNA_profile(input_data)`

This def will return the CNA profile of a cluster at a range of given values of rCut.

Parameters

- **cluster** (*Either Cluster or ase.Atoms*) – This is the cluster to obtain the CNA profile of.
- **rCuts** (*float*) – The range of values of rCuts to obtain the CNA profile of.

Returns This resutrn the name of the cluster, and the atomic_CNA_profiles.

Return type (int, Counter)

`Organisms.GA.SCM_Scripts.A_SCM_Methods.get_CNA_similarities(input_data)`

Get the full similarity profile of the two clusters.

input_data contains two parameters

Parameters

- **name_1** (*int*) – Name of the first cluster
- **name_2** (*int*) – Name of the second cluster
- **cluster_1_CNA_profile** (*[asap3.analysis.localstructure.FullCNA, ...]*) – the full CNA profile of cluster 1 for all values of rCut.
- **cluster_2_CNA_profile** (*[asap3.analysis.localstructure.FullCNA, ...]*) – the full CNA profile of cluster 2 for all values of rCut.

Returns returns the name of the two clusters, and the similarity profile.

Return type (int, int, list of float)

`Organisms.GA.SCM_Scripts.A_SCM_Methods.get_CNA_similarity(cluster_1_CNA, cluster_2_CNA, total_no_of_atoms)`

Get the similarity for the two clusters at a particular value of rCut.

Parameters

- **cluster_1_CNA** (*asap3.analysis.localstructure.FullCNA*) – the CNA profile of cluster 1 at rCut
- **cluster_2_CNA** (*asap3.analysis.localstructure.FullCNA*) – the CNA profile of cluster 2 at rCut
- **total_no_of_atoms** (*int*) – The total number of atoms in the cluster

`Organisms.GA.SCM_Scripts.A_SCM_Methods.get_atomic_CNA_profile(input_data)`

This method will obtain the Atomic CNA profile

Parameters `input_data` (*(Organisms.GA.Cluster, float)*) – Contains the cluster to perform the CNA on at the rCut value rCut.

Returns The atomic CNA profile.

Return type Counter

`Organisms.GA.SCM_Scripts.A_SCM_Methods.get_tasks(system, rCuts)`

This is a generator that allows many atomic_CNA_profiles to be obtained at many rCut values in parallel

Parameters

- **system** (*Organisms.GA.Cluster*) – This is the cluster to obtain the atomic_CNA_profiles for
- **rCuts** (*float*) – This is the rCut value to to obtain the atomic_CNA_profiles at.

Returns Yields a tuple of (the cluster, rCut)

Return type (Organisms.GA.Cluster, float)

T_SCM_Methods.py

Meow

T_SCM_Methods.py, Geoffrey Weal, 20/11/2018

This script is designed to include all the methods for performing structural recognition task in terms of the Total Structural Comparison Method.

`Organisms.GA.SCM_Scripts.T_SCM_Methods.get_CNA_profile(input_data)`

This def will return the CNA profile of a cluster at a range of given values of rCut.

Parameters

- **cluster** (*Either Cluster or ase.Atoms*) – This is the cluster to obtain the CNA profile of.
- **rCuts** (*float*) – The range of values of rCuts to obtain the CNA profile of.

Returns This resutrn the name of the cluster, and the atomic_CNA_profiles.

Return type (int, Counter)

`Organisms.GA.SCM_Scripts.T_SCM_Methods.get_CNA_similarities(input_data)`

Get the full similarity profile of the two clusters.

input_data contains two parameters

Parameters

- **name_1** (*int*) – Name of the first cluster
- **name_2** (*int*) – Name of the second cluster
- **cluster_1_CNA_profile** (*[asap3.analysis.localstructure.FullCNA, ...]*) – the full CNA profile of cluster 1 for all values of rCut.
- **cluster_2_CNA_profile** (*[asap3.analysis.localstructure.FullCNA, ...]*) – the full CNA profile of cluster 2 for all values of rCut.

Returns returns the name of the two clusters, and the similarity profile.

Return type (int, int, list of float)

`Organisms.GA.SCM_Scripts.T_SCM_Methods.get_CNA_similarity(cluster_1_CNA, cluster_2_CNA)`

Get the similarity for the two clusters at a particular value of rCut.

Parameters

- **cluster_1_CNA** (*asap3.analysis.localstructure.FullCNA*) – the CNA profile of cluster 1 at rCut
- **cluster_2_CNA** (*asap3.analysis.localstructure.FullCNA*) – the CNA profile of cluster 2 at rCut
- **total_no_of_atoms** (*int*) – The total number of atoms in the cluster

`Organisms.GA.SCM_Scripts.T_SCM_Methods.get_tasks` (*system, rCuts*)

This is a generator that allows many total_CNA_profiles to be obtained at many rCut values in parallel

Parameters

- **system** (*Organisms.GA.Cluster*) – This is the cluster to obtain the total_CNA_profiles for
- **rCuts** (*float*) – This is the rCut value to to obtain the total_CNA_profiles at.

Returns Yields a tuple of (the cluster, rCut)

Return type (*Organisms.GA.Cluster, float*)

`Organisms.GA.SCM_Scripts.T_SCM_Methods.get_total_CNA_profile` (*input_data, return_list*)

This method will obtain the Total CNA profile

Parameters **input_data** (*(Organisms.GA.Cluster, float)*) – Contains the cluster to perform the CNA on at the rCut value rCut.

Returns The atomic CNA profile.

Return type Counter

CNA_Database.py

Meow

CNA_Database.py, Geoffrey Weal, 29/10/2018

This script holds the information required to make a CNA_database.

```
class Organisms.GA.SCM_Scripts.CNA_Database.CNA_Database (rCuts, population,
                                                         cut_off_similarity,
                                                         get_cna_profile_method,
                                                         get_similarity_profile_method,
                                                         no_of_cpus, debug)
```

This is a database that holds all the entries of CNA_Entry objects.

Parameters

- **rCuts** (*float*) – These are the rCut values to scan the CNA across.
- **population** (*Organisms.GA.Population*) –
- **cut_off_similarity** (*float*) – The maximum similarity above which to be considered similar enough to exclude offspring from being accessed into future generations.
- **get_cna_profile_method** (*__func__*) – This is the method to get the CNA Profile, either from the T-SCM or the A-SCM.
- **get_similarity_profile_method** (*__func__*) – This is the method that uses the CNA profiles of two clusters in order to give the similarity profile between the two clusters, Whether this is obtained by the T-SCM or the A-SCM.

- **no_of_cpus** (*int*) – The number of cpu to use to obtain the similarity profile between two clusters
- **debug** (*bool.*) – Get data to use to debug the SCM. Default: False

add (*collection, initialise=False*)

Add the similarity data of the clusters in the collection to the CNA_Database

Parameters

- **collection** (*Organisms.GA.Collection*) – The clusters to add to the CNA_Database
- **initialise** (*bool*) – Are the clusters from a newly created population. Default: False

check_database (*population*)

Check the database to make sure there are the correct number of entries in the database.

Parameters **population** (*Organisms.GA.Population*) – The population

get_all_averages_for_a_cluster (*cluster_name*)

Get the averages SCM similarities of a cluster compared to every other cluster in the similarity profile.

Parameters **cluster_name** (*list of floats*) – Cluster to get the averages similarities of.

Returns all the average similarities between cluster_name any every other cluster in the database.

Return type list of float

get_details ()

Return the information about the CNA Database, including:

Returns The cut_off_similarity, get_cna_profile_method, get_similarity_profile_method, no_of_cpus, debug

Return type (float, __func__, __func__, int, bool)

get_max_similarity (*name_1, name_2*)

This def will obtain the max similarity percentage from the comparison of two clusters, name_1 and name_2.

Parameters

- **name_1** (*int*) – The name of the first cluster you would like to look for an entry in the CNA_Database.
- **name_2** (*int*) – The name of the second cluster you would like to look for an entry in the CNA_Database.

Returns returns the maximum similarity between cluster name_1 and name_2

Return type float

get_similar_clusters_in_database ()

Get clusters in the clusters that are deemed structurally similar in the CNA_Database

Returns a list of the names of all the clusters that are similar to each other.

Return type list of int

is_pair_in_the_database (*dir_1, dir_2*)

Determine if a CNA entry exists for two clusters in the similarity_profile_database

Parameters

- **name1** (*int*) – The dir of the first cluster you would like to look for an entry in the CNA_Database.
- **name2** (*int*) – The dir of the second cluster you would like to look for an entry in the CNA_Database.

Returns True if the similarity profile for cluster name1 and name2, False if not.

Return type bool

keys ()

Return a list of the names of all the clusters in the database

Returns A list of the names of all the clusters in the database

Return type list of int

make_simple_table (*similarity_profile_database*)

This is a simplified table that can be printed to the terminal that shows all the similarities between clusters in the population + offspring

Parameters **similarity_profile_database** (`Organisms.GA.SCM_Scripts.CNA_Database.CNA_Database`) – This is the CNA database that contains all similarity information of clusters in the population and offspring.

print_cna_database_details ()

Print information about the database.

remove (*names_to_remove*)

Remove all entries that exist in the database that are associated with a particular cluster.

Parameters **name_to_remove** (*int*) – The name of the cluster you would like to remove from the CNA_Database.

reset ()

Reset the CNA profiles of generated clusters and the similarity profile database.

class `Organisms.GA.SCM_Scripts.CNA_Database.Tree`

This is a Tree designed for the CNA_Database to hold references of CNA_Entry.

see_tree ()

This shows the clusters in the database

`Organisms.GA.SCM_Scripts.CNA_Database.cna_profile_generator` (*collection, rCuts*)

This is a generator that returns the clusters in the collection with the rCut values to scan across.

Parameters

- **collection** (`Organisms.GA.Collection`) – A collection
- **rCuts** (*list of float*) – The list of clusters to scan across with the SCM.

Returns a tuple of the cluster the rCut values to scan across with the SCM

`Organisms.GA.SCM_Scripts.CNA_Database.initial_similarity_profile_generator` (*collection, cna_database*)

This is a generator that returns the clusters in the collection with the rCut values to scan across.

Parameters

- **collection** (`Organisms.GA.Collection`) – A collection
- **cna_database** (*list of float*) – The list of clusters to scan across with the SCM.

Returns a tuple of the names of the clusters and their associated CNA profiles.

Return type (int, int, Counter, Counter)

`Organisms.GA.SCM_Scripts.CNA_Database.similarity_profile_generator` (*population, offsprings, cna_database*)

This is a generator that returns the clusters in the collection with the rCut values to scan across.

Parameters

- **population** (*Organisms.GA.Population*) – The population
- **offsprings** (*Organisms.GA.Offspring_Pool*) – The collection of offspring
- **cna_database** (*list of float*) – The list of clusters to scan across with the SCM.

Returns a tuple of the names of the clusters and their associated CNA profiles.

Return type (int, int, Counter, Counter)

MyPool.py

Meow

class `Organisms.GA.SCM_Scripts.MyPool.MyPool` (*processes=None, initializer=None, initargs=(), maxtasksperchild=None, context=None*)

Process

alias of `Organisms.GA.SCM_Scripts.MyPool.NoDaemonProcess`

class `Organisms.GA.SCM_Scripts.MyPool.NoDaemonProcess` (*group=None, target=None, name=None, args=(), kwargs={}, *, daemon=None*)

Epoch.py

meow

class `Organisms.GA.Epoch.Epoch` (*epoch_settings, path, fitness_operator, population*)

This class is responsible for the epoch method used during the genetic algorithm run

Parameters

- **epoch_settings** (*dict.*) – This is a dictionary of all the epoch settings.
- **path** (*str.*) – This is the path to store the epoch information in after each generation has completed. This information allows the genetic algorithm run to continue with information of the epoch from the last completed generation.

backup ()

Make a backup of the epoch data stored on disk

change_fitness_function (*fitness_operator*)

check_for_backup ()

Checks to see that a backup exists. If not, something has gone wrong and the algorithm needs to stop.

did_get_data_from_backup ()

Determine if the data was obtained from backup epoch file or not. True for yes, False for no. If no epoch method was used, return False, since we dont need to deal with previous epoch information if no epoch method was used.

returns Did the epoch information come from the backup epoch file or not. True for yes, False for no. If no epoch method was used, return False, since we dont need to deal with previous epoch information if no epoch method was used. rtype bool.

does_backup_exist ()

Determine if a backup of the epoch information stored on disk exists.

returns If a backup of the epoch information stored on disk exists. True for yes, False for no. rtype bool.

does_epoch_data_exist_on_disk ()

Determine if a backup of the epoch information stored on disk exists.

returns If a backup of the epoch information stored on disk exists. True for yes, False for no. rtype bool.

get_resumed_mean_energy_epoch_details (*epoch_settings*, *get_epoch_data_from*)

This contains the information required to set up the mean_energy epoch method.

Parameters

- **epoch_settings** (*dict.*) – This is a dictionary of all the epoch settings.
- **get_epoch_data_from** (*str.*) – The path of the epoch document to get information from if the genetic algorithm is being resumed.

get_resumed_same_population_epoch_details (*epoch_settings*, *get_epoch_data_from*)

This contains the information required to set up the same_population epoch method.

Parameters

- **epoch_settings** (*dict.*) – This is a dictionary of all the epoch settings.
- **get_epoch_data_from** (*str.*) – The path of the epoch document to get information from if the genetic algorithm is being resumed.

information_from_the_same_generation (*generation_number*)

Determine if the epoch file contains the relevant information for the generation to resume from.

Parameters **generation_number** (*int*) – The number of the last successful generation

returns If the epoch information from the epoch file is for the current generation. True if yes, False if no. This value is True if no epoch method is used, as we dont need to deal with any epoch information if no epoch is used. rtype bool.

initialise_mean_energy_epoch_details ()

This contains the information required to set up the mean_energy epoch method.

Parameters **epoch_settings** (*dict.*) – This is a dictionary of all the epoch settings.

initialise_same_population_epoch_details ()

This contains the information required to set up the same_population epoch method.

Parameters **epoch_settings** (*dict.*) – This is a dictionary of all the epoch settings.

no_epoch (*collection*, *generation_number*)

The procedure to determine if an epoch should occur.

Since no epoch is included in this algorithm, return False to mean do not perform an epoch

Parameters

- **collection** (*Organisms.GA.Collection*) – This is the collection to use to determine if the genetic algorithm should epoch. This Collection should be the Population.
- **generation_number** (*int*) – The number of the last successful generation

returns Should an epoch occur. False for no, since this genetic algorithm has not been told to epoch. rtype bool.

perform_epoch (*generation_number*, *population*, *energyprofile*, *population_reset_settings*, *epoch_due_to_population_energy_convergence*)
Perform an epoch by resetting all the clusters in the population.

Parameters

- **generation_number** (*int*) – The number of the last successful generation
- **population** (*Organisms.GA.Population*) – The number of the last successful generation
- **energyprofile** (*Organisms.GA.EnergyProfile*) – The energyprofile to write epoch information to.
- **population_reset_settings** (*dict.*) – This is a dict that contains all the information required to epoch the population with a set of randomly generated clusters.
- **epoch_due_to_population_energy_convergence** (*bool.*) – Did the Epoch occur because the population converged? True if yes, False if no.

returns The name of the more recently generated cluster before the epoch method was performed. rtype bool.

perform_epoch_mean_energy (*generation_number*)
This method writes the current epoch data to file for the mean_energy epoch method.

Parameters **generation_number** (*int*) – The number of the last successful generation

perform_epoch_no_epoch (*generation_number*)
This method is needed, but does not do anything.

Parameters **generation_number** (*int*) – The number of the last successful generation

perform_epoch_same_population (*generation_number*)
This method writes the current epoch data to file for the same_population epoch method.

Parameters **generation_number** (*int*) – The number of the last successful generation

remove_backup ()
Remove the backup of the epoch data stored on disk

replace_with_backup ()
Replace the current epoch data with the backup epoch data.

set_settings (*epoch_settings*, *fitness_operator*, *population*)
This sets up the epoch as desired.

Parameters **epoch_settings** (*dict.*) – This is a dictionary of all the epoch settings.

setting_up_epoch_to_resume_GA ()
This method will load the data from the epoch data on disk in order to resume.

should_epoch (*collection*, *generation_number*)
Determines if the genetic algorithm should epoch.

Parameters

- **collection** (*Organisms.GA.Collection*) – This is the collection to use to determine if the genetic algorithm should epoch. This Collection should be the Population.
- **generation_number** (*int*) – The number of the last successful generation

returns Should an epoch occur. True if yes, False if no. rtype bool.

should_epoch_using_mean_energy_epoch (*collection, generation_number*)

The procedure to determine if an epoch should occur, using the mean_energy epoch method.

Parameters

- **collection** (*Organisms.GA.Collection*) – This is the collection to use to determine if the genetic algorithm should epoch. This Collection should be the Population.
- **generation_number** (*int*) – The number of the last successful generation

returns Should an epoch occur. True if yes, False if no. rtype bool.

should_epoch_using_same_population_epoch (*collection, generation_number*)

The procedure to determine if an epoch should occur, using the same_population epoch method.

Since no epoch is included in this algorithm, return False to mean do not perform an epoch

Parameters

- **collection** (*Organisms.GA.Collection*) – This is the collection to use to determine if the genetic algorithm should epoch. This Collection should be the Population.
- **generation_number** (*int*) – The number of the last successful generation

returns Should an epoch occur. True if yes, False if no. rtype bool.

Memory_Operator.py

meow

ExternalDefinitions.py

This describe a cluster, where the system information is stored in the Genetic Algorithm

ExternalDefinitions.py, 12/04/2017, Geoffrey R Weal

This python script is designed to hold subsidiary definitions used by this Genetic Algorithm.

`Organisms.GA.ExternalDefinitions.AtomInClusterPosition` (*atom, cluster*)

This definition will return true if there is an atom already at a particular position in the cluster. Prevent two atoms being in the exact same position which local optimisation techniques can not handle.

Inputs: atom (ase.atom): The atom you want to add to the cluster, to check that atom will not be in the same position as any atom in the cluster. cluster (ase.atoms or GA.Cluster): The cluster that the atom is to be added to.

`Organisms.GA.ExternalDefinitions.Exploded` (*cluster, max_distance_between_atoms*)

This definition is designed to check to make sure a cluster has not exploded. This means that all the atoms in space are closely bound together as a nanoparticle rather than some atoms being disconnected from the majority of atoms in a cluster. This method works as follows:

- Every atom is checked for neighbours/ other atoms it is bonded to in the cluster. The information of the neighbours of each atom are placed in a neighbour list
- Write this later

Parameters

- **cluster** (*ASE.Atoms*) – the cluster to check if all clusters are attached together.
- **max_distance_between_atoms** (*float*) – defines what the maximum length of a bond is in your cluster.

Organisms.GA.ExternalDefinitions.**InclusionRadiusOfCluster** (*cluster*)

Find the radius of a sphere that could completely enclose the cluster, with radius from the centre of mass to the most outer atom from the centre of mass.

Parameters **cluster** (*ASE.Atoms*) – The cluster that the user would like to find the maximum radius from the centre of mass.

Returns max(size): The radius of the cluster from the from the centre of mass to the most outer atom from the centre of mass (in Angstroms). This radius is for a sphere that will definitely enclose the cluster within.

Return type float

Organisms.GA.ExternalDefinitions.**InclusionboxOfCluster** (*cluster*)

Get the cell length for a cube box to place your cluster in.

Parameters **cluster** (*ASE.Atoms*) – The cluster that the user would like to find the maximum radius from the centre of mass.

Returns max(size): The radius of the cluster from the from the centre of mass to the most outer atom from the centre of mass (in Angstroms). This radius is for a sphere that will definitely enclose the cluster within.

Return type float

class Organisms.GA.ExternalDefinitions.**atom_single_connection** (*atom*)

This class is designed to record all the atoms an atom in a cluster is bonded/neighboured to, where a bond is defined as a atom-atom distance less than max_distance_between_atoms. :param atom: The dir of the atom in the cluster :type atom: int

Organisms.GA.ExternalDefinitions.**get_elemental_makeup** (*cluster*)

This gives a list which indicates the types of elements, and the number of those elements, in the cluster

This method has been copied from the def get_elemental_makeup from class Cluster, in Cluster.py

Returns A list which indicates the types of elements, and the number of those elements, in the cluster

Return type {str: int, ..} (old output was [[str.,int],..])

Organisms.GA.ExternalDefinitions.**is_position_already_occupied_by_an_atom_in_Cluster** (*atom_pos*, *cluster*, *atom_ind*)

This method determines if two atoms occupy the same position.

Parameters

- **atom_position** (*(int, int, int)*) – The position of the atom, given as (x position, y position, z position).
- **cluster** (*GA.Cluster*) – The cluster to investigate.
- **atom_indices_to_exclude_from_comparison** (*list of int*) – list of the atom indices not to include in this analysis, THIS should include the atom associated with the position list atom_position

GA_Recording_System.py

This class will write the details of the genetic algorithm run.

Recording_Clusters.py, 02/10/2018, Geoffrey R Weal

```
class Organisms.GA.GA_Recording_System.GA_Recording_Database(path_of_database,  
                                                         max_no_of_recorded_structures=None,  
                                                         limit_datasize_of_database=None,  
                                                         ga_recording_scheme='None',  
                                                         limit_energy_height_of_clusters_recorded=  
                                                         lower_energy_limit=-  
                                                         inf,          up-  
                                                         per_energy_limit=inf,  
                                                         show_GA_Recording_Database_check_per
```

This is a Collection that has been designed to record clusters that have been created during the genetic algorithm.

This has been designed to give the user many ways to limit the number of clusters that are recorded. This is to prevent the size of the database from getting to big in disk size.

Parameters

- **path_of_database** (*str.*) – The path to this database
- **max_no_of_recorded_structures** (*int*) – This is the maximum number of clusters that will be recorded. If this limit is reached, the higher energy clusters will be replaced in new, lower energy clusters.
- **limit_datasize_of_database** (*str.*) – This is the maximum size the database can get. Give as a string with size + memory type (e.g. 150MB, 2.0GB)
- **ga_recording_scheme** (*str.*) – The user can indicate a specific type of recording scheme to use to limit the clusters that are recorded. See manual for more details. Default: 'None'
- **limit_energy_height_of_clusters_recorded** (*float*) – If `ga_recording_scheme == 'Limit_energy_height'`: is selected, this is the maximum energy above the LES energy. Any clusters that are lower in energy than `Energy(LES) + limit_energy_height_of_clusters_recorded` will be recorded. Any that have an energy higher than this will not be recorded. Default: `float('inf')`
- **lower_energy_limit** (*float*) – If `ga_recording_scheme == Set_energy_limits` is selected, this is the low energy limit. Any cluster with an energy lower than `lower_energy_limit` will not be recorded. Default: `-float('inf')`
- **upper_energy_limit** (*float*) – If `ga_recording_scheme == Set_energy_limits` or `ga_recording_scheme == 'Set_higher_limit'`, this is the upper energy limit. Any clusters with an energy higher than this energy limit will not be recorded. Default: `float('inf')`

add (*index, cluster*)

Adds a cluster to the Collection.

Index: *index* (int/str.): the index of the *i*th cluster in the Collection. If "End" is inputed, the cluster will be append to the end of the Collection list. *cluster* (Organisms.GA.Cluster): The cluster to add at the *i*th position in the Collection.

add_clusters_into_RAM (*cluster_dict, cluster_names*)

This method adds clusters into the RAM

Parameters

- **cluster_dict** (*{int: ASE.Cluster}*) – This is a dictionary of all the clusters from the database, given as {cluster_name: Cluster}
- **cluster_names** (*list of int*) – list of the names of the clusters that are needed for the collection

add_collection_to_database (*collection, clusters_not_to_include*)

Will record the clusters in the collection to the GA_Recording_Database.

Parameters

- **collection** (*Organisms.GA.Collection*) – The collection to be recorded
- **clusters_not_to_include** (*list of int*) – A list of names of clusters in the collection not to record in GA_Recording_Database.

check_clusters_in_database (*generation*)

Will check the clusters in the database and remove any cluster that was created during a most recent unsuccessful generation.

Parameters *generation* (*int*) – The current generation

get_cluster_names (*order=False*)

Will provide a list of all the names of all the clusters in the Collection

Inputs: *order* (bool.): This tag will tell this method whether the user would like the list of names given in order.

Returns List of the names of all the clusters in the Population

get_index (*name_to_find*)

This method will provide the index of the cluster that has the name “name_to_find” in the Collection

Inputs: *name_to_find* (int): the name of the cluster in the Collection to obtain the index for

Returns the index of the cluster in the Collection with the name “name_to_find”

Exceptions: Will break if the cluster with the name “name_to_find” can not be found in this method.

import_information_from_database (*current_generation*)

Import any data from the database if it is needed. This should only be needed for the ‘Limit_energy_height’ scheme.

Parameters *current_generation* (*int*) – The current generation that your genetic algorithm trial is being resumed from

remove_to_database (*cluster*)

Allows the user to remove a cluster in the collection from the ASE database

Inputs: *cluster* (Organisms.GA.Cluster): The cluster to remove from the database.

sort_by_energy ()

This method will sort the clusters in the list by their energy (from lowest energy to highest energy).

sort_by_name ()

This method will sort the clusters in the list by their name.

update_cluster_in_database_for_if_in_population (*clusters_in_the_population,*
ener-
gies_of_clusters_removed_from_the_population)

This method will update clusters in the database if that cluster was ever accepted into the population.

Parameters clusters_in_the_population (*list of int*) – This is a list of the names of clusters that are in the population.

class `Organisms.GA.GA_Recording_System.GA_Recording_System` (*ga_recording_information*)
This class is designed to record the clusters that are created during the Organisms program run.

Parameters ga_recording_information – This is a dictionary that contains all the information that it needs to record clusters made during the Organisms program as the user desires.

add_collection (*collection, offspring_to_remove*)
This will add the clusters to the GA_Recording_System database

Parameters

- **collection** (*Organisms.GA.Collection*) – The collection to be recorded
- **offspring_to_remove** (*list of int*) – This is a list of all the clusters not to write to the database, as they have been removed by the diversity operator, depending on if `self.exclude_recording_cluster_screened_by_diversity_scheme` is True or False.

add_metadata ()
This method is designed to assign the metadata of the ASE database, as in some versions of ASE this can not happen until at least one cluster has been added to the ASE database.

check_clusters_in_database (*generation*)
Will check the clusters in the database and remove any cluster that was created during a most recent unsuccessful generation.

Parameters generation (*int*) – The current generation

import_information_from_database (*current_generation*)
Import any data from the database if it is needed. This should only be needed for the ‘Limit_energy_height’ scheme.

Parameters current_generation (*int*) – The current generation that your genetic algorithm trial is being resumed from

record_collection (*collection, name_of_database, path_to_write_to*)
This records an identical copy of the clusters in the population at a certain generation.

Parameters

- **collection** (*Organisms.GA.Collection*) – The collection to record into GA_Recording_System.
- **name_of_database** (*str.*) – Name of the database to connect to
- **path_to_write_to** (*str.*) – Path of the database or folder of xyz files to write to.

record_initial_populations (*population*)
Will record the clusters in the initial population into GA_Recording_System.

Parameters population (*Organisms.GA.Population*) – The initial population

record_population_at_generation (*population, current_generation*)
Will record the clusters in the initial population into GA_Recording_System at the current generation.

Parameters

- **population** (*Organisms.GA.Population*) – The current population after the generation has completed
- **current_generation** (*int*) – The generation that your genetic algorithm is up to.

resume_ga_recording_system_from_current_generation (*resume_from_generation*)
Will check and restore the `ga_recording_system` is restored for a generation.

Parameters `resume_from_generations` – The generation that your genetic algorithm is up to.

update_cluster_in_database_for_if_in_population (*clusters_in_the_population,*
ener-
gies_of_clusters_removed_from_the_population)

This method will update clusters in the database if that cluster was ever accepted into the population.

Parameters

- **clusters_in_the_population** (*list of int*) – This is a list of the names of clusters that are in the population.
- **energies_of_clusters_removed_from_the_population** (*list of int*) – This is a list of the energies of clusters that are in the population.

`Organisms.GA.GA_Recording_System.convert_to_bytes` (*size*)
Will convert the size in any disk space format to bytes.

Parameters `size` (*str.*) – The size of the database in any units
returns `data_size`: The size of the database in bytes `rtype data_size`: float

`Organisms.GA.GA_Recording_System.get_size` (*size*)
will convert the most human friendly version of the disk space of the database.

Input: `size` (float): the disk space of the database in bytes

`Organisms.GA.GA_Recording_System.make_folder` (*path_to_folder*)
Will remake a folder, even if it already exists.

Input: `path_to_folder` (str.): the path to the folder to remake.

Timer.py

This class is designed

class `Organisms.GA.Timer.Timer` (*total_length_of_running_time*)

This class is designed to time the genetic algorithm, as well as to determine if the algorithm has run for longer than is desired by the `total_length_of_running_time` variable.

Parameters `total_length_of_running_time` (*float*) – This is the length of time to run the algorithm for before safely finishing the algorithm. The time is given in hours

get_time_now ()
Returns the current date and time.

get_total_length_of_running_time ()
Return the maximum amount of time to run the genetic algorithm for, in hours.

Returns `total_length_of_running_time`, the maximum amount of time to run the genetic algorithm for, in hours.

Return type float or str.

has_elapsed_time ()
This determines if the cluster has exceeded the desired running time.

print_elapsed_time ()
Returns the current running time of the algorithm

5.25 Index

5.26 Python Module Index

INDICES AND TABLES

- *Index*
- *Python Module Index*

PYTHON MODULE INDEX

O

Organisms.GA.Cluster, 91
Organisms.GA.Collection, 94
Organisms.GA.Collections_Iterator, 99
Organisms.GA.Crossover, 111
Organisms.GA.EnergyProfile, 101
Organisms.GA.Epoch, 156
Organisms.GA.ExternalDefinitions, 159
Organisms.GA.Fitness_Operators.CNA_Fitness_Contribution,
146
Organisms.GA.Fitness_Operators.Energetic_Fitness_Contribution,
142
Organisms.GA.Fitness_Operators.Energy_Fitness_Operator,
139
Organisms.GA.Fitness_Operators.Fitness_Function,
148
Organisms.GA.Fitness_Operators.Fitness_Operator,
138
Organisms.GA.Fitness_Operators.SCM_and_Energy_Fitness_Operator,
143
Organisms.GA.GA_Initiate, 105
Organisms.GA.GA_Introducing_Remarks, 106
Organisms.GA.GA_Program, 88
Organisms.GA.GA_Program_Details, 90
Organisms.GA.GA_Recording_System, 161
Organisms.GA.GA_Setup, 103
Organisms.GA.Get_Offspring, 110
Organisms.GA.Get_Predation_and_Fitness_Operators,
117
Organisms.GA.Initialise_Population, 106
Organisms.GA.Memory_Operator, 159
Organisms.GA.Mutation, 115
Organisms.GA.Offspring_Pool, 103
Organisms.GA.Population, 99
Organisms.GA.Predation_Operators.Energy_Predation_Operator,
123
Organisms.GA.Predation_Operators.Energy_Predation_Operator_Scripts.Comprehensive_Energy_Pre
125
Organisms.GA.Predation_Operators.Energy_Predation_Operator_Scripts.Comprehensive_Energy_Pre
127
Organisms.GA.Predation_Operators.Energy_Predation_Operator_Scripts.Simple_Energy_Predation
124
Organisms.GA.Predation_Operators.IDCM_Predation_Ope
128
Organisms.GA.Predation_Operators.IDCM_Predation_Ope
132
Organisms.GA.Predation_Operators.IDCM_Predation_Ope
133
Organisms.GA.Predation_Operators.No_Predation_Oper
121
Organisms.GA.Predation_Operators.Predation_Operator
119
Organisms.GA.Predation_Operators.SCM_Predation_Oper
135
Organisms.GA.SCM_Scripts.A_SCM_Methods,
151
Organisms.GA.SCM_Scripts.CNA_Database,
153
Organisms.GA.SCM_Scripts.MyPool, 156
Organisms.GA.SCM_Scripts.SCM_initialisation,
149
Organisms.GA.SCM_Scripts.Similarity_Profile,
150
Organisms.GA.SCM_Scripts.T_SCM_Methods,
152
Organisms.GA.Surface, 94
Organisms.GA.Timer, 164
Organisms.GA.Types_Of_Mutations, 116

INDEX

A

- add()* (*Organisms.GA.Collection.Collection* method), 94
add() (*Organisms.GA.GA_Recording_System.GA_Recording_Database* method), 161
add() (*Organisms.GA.Predation_Operators.IDCM_Predation_Operator_Scripts.LOD_Comparison_Database.LOD_Comparison_Database* method), 133
add() (*Organisms.GA.SCM_Scripts.CNA_Database.CNA_Database* method), 154
add_clusters_into_RAM() (*Organisms.GA.Collection.Collection* method), 94
add_clusters_into_RAM() (*Organisms.GA.GA_Recording_System.GA_Recording_Database* method), 161
add_collection() (*Organisms.GA.EnergyProfile.EnergyProfile* method), 101
add_collection() (*Organisms.GA.GA_Recording_System.GA_Recording_System* method), 163
add_collection_to_database() (*Organisms.GA.GA_Recording_System.GA_Recording_Database* method), 162
add_epoch_note() (*Organisms.GA.EnergyProfile.EnergyProfile* method), 101
add_epoch_note_due_to_population_energy_convergence() (*Organisms.GA.EnergyProfile.EnergyProfile* method), 101
add_found_LES_note() (*Organisms.GA.EnergyProfile.EnergyProfile* method), 101
add_metadata() (in module *Organisms.GA.GA_Initiate*), 105
add_metadata() (*Organisms.GA.Collection.Collection* method), 94
add_metadata() (*Organisms.GA.GA_Recording_System.GA_Recording_System* method), 163
add_to() (*Organisms.GA.EnergyProfile.EnergyProfile* method), 101
add_to_database() (*Organisms.GA.Collection.Collection* method), 95
add_to_database() (*Organisms.GA.Fitness_Operators.Energy_Fitness_Operator_Energy_Fitness_Operators.LOD_Comparison_Database.LOD_Comparison_Database* method), 140
add_to_database() (*Organisms.GA.Fitness_Operators.SCM_and_Energy_Fitness_Operators.SCM_and_Energy_Fitness_Operators* method), 144
add_to_database() (*Organisms.GA.Predation_Operators.Energy_Predation_Operator.Energy_Predation_Operators* method), 123
add_to_database() (*Organisms.GA.Predation_Operators.IDCM_Predation_Operator.IDCM_Predation_Operators* method), 129
add_to_database() (*Organisms.GA.Predation_Operators.No_Predation_Operator.No_Predation_Operators* method), 121
add_to_database() (*Organisms.GA.Predation_Operators.Predation_Operator.Predation_Operators* method), 119
add_to_database() (*Organisms.GA.Predation_Operators.SCM_Predation_Operator.SCM_Predation_Operators* method), 135
add_to_history_file() (*Organisms.GA.Collection.Collection* method), 95
are_all_entries_false() (*Organisms.GA.Predation_Operators.IDCM_Predation_Operator_Scripts.IDCM_Predation_Operators* method), 133
assess_clusters_in_database() (*Organisms.GA.Collection.Collection* method), 95
assess_for_violations() (in module *Organisms.GA.Predation_Operators.Energy_Predation_Operator_Scripts.Energy_Predation_Operators*), 125
assess_for_violations() (*Organisms.GA.Predation_Operators.Energy_Predation_Operator.Energy_Predation_Operators* method), 123
assess_for_violations() (*Organisms.GA.Predation_Operators.IDCM_Predation_Operator.IDCM_Predation_Operators* method), 129

`assess_for_violations()` (Organisms.GA.Predation_Operators.No_Predation_Operator.No_Predation_Operator method), 140
`assess_for_violations()` (Organisms.GA.Predation_Operators.No_Predation_Operator.No_Predation_Operator method), 122
`assess_for_violations()` (Organisms.GA.Predation_Operators.Predation_Operator.Predation_Operator method), 138
`assess_for_violations()` (Organisms.GA.Predation_Operators.Predation_Operator.Predation_Operator method), 119
`assess_for_violations()` (Organisms.GA.Predation_Operators.SCM_Predation_Operator.SCM_Predation_Operator method), 144
`assess_for_violations()` (Organisms.GA.Predation_Operators.SCM_Predation_Operator.SCM_Predation_Operator method), 135
`assess_for_violations_force_replacement()` (Organisms.GA.Predation_Operators.Energy_Predation_Operator.Energy_Predation_Operator method), 141
`assess_for_violations_force_replacement()` (Organisms.GA.Predation_Operators.Energy_Predation_Operator.Energy_Predation_Operator method), 126
`assess_for_violations_force_replacement()` (Organisms.GA.Predation_Operators.Energy_Predation_Operator.Energy_Predation_Operator method), 127
`assess_for_violations_force_replacement()` (Organisms.GA.Predation_Operators.IDCM_Predation_Operator.IDCM_Predation_Operator method), 129
`assess_for_violations_force_replacement()` (Organisms.GA.Predation_Operators.IDCM_Predation_Operator.IDCM_Predation_Operator method), 136
`assess_for_violations_message()` (Organisms.GA.Predation_Operators.IDCM_Predation_Operator.IDCM_Predation_Operator method), 130
`assess_for_violations_message()` (Organisms.GA.Predation_Operators.SCM_Predation_Operator.SCM_Predation_Operator method), 136
`assess_for_violations_no_force_replacement()` (Organisms.GA.Predation_Operators.Energy_Predation_Operator.Energy_Predation_Operator method), 126
`assess_for_violations_no_force_replacement()` (Organisms.GA.Predation_Operators.Energy_Predation_Operator.Energy_Predation_Operator method), 145
`assess_for_violations_no_force_replacement()` (Organisms.GA.Predation_Operators.Energy_Predation_Operator.Energy_Predation_Operator method), 127
`assess_for_violations_no_force_replacement()` (Organisms.GA.Predation_Operators.IDCM_Predation_Operator.IDCM_Predation_Operator method), 130
B
`assess_for_violations_no_force_replacement()` (Organisms.GA.Predation_Operators.SCM_Predation_Operator.SCM_Predation_Operator method), 137
`assign_all_fitnesses_after_assess_against_predation_operator()` (Organisms.GA.Fitness_Operators.Energy_Fitness_Operator.Energy_Fitness_Operator method), 140
`assign_all_fitnesses_after_assess_against_predation_operator()` (Organisms.GA.Fitness_Operators.Fitness_Operator.Fitness_Operator method), 138
`assign_all_fitnesses_after_assess_against_predation_operator()` (Organisms.GA.Fitness_Operators.SCM_and_Energy_Fitness_Operator.SCM_and_Energy_Fitness_Operator method), 144
C
`assign_all_fitnesses_after_natural_selection()` (Organisms.GA.Fitness_Operators.Energy_Fitness_Operator.Energy_Fitness_Operator method), 92
`assign_all_fitnesses_after_natural_selection()` (Organisms.GA.Fitness_Operators.Energy_Fitness_Operator.Energy_Fitness_Operator method), 92

<i>isms.GA.Predation_Operators.IDCM_Predation_Operator</i>	<i>maintain_database()</i>	(<i>Organisms.GA.Collection.Collection</i> method),
128	96	
Cluster_Block (class in <i>Organisms.GA.Predation_Operators.SCM_Predation_Operator</i>)	<i>epoch_data_exist_on_disk()</i>	(<i>Organisms.GA.Epoch.Epoch</i> method), 157
135		
CNA_Database (class in <i>Organisms.GA.SCM_Scripts.CNA_Database</i>), 153	E	
<i>cna_profile_generator()</i> (in module <i>Organisms.GA.SCM_Scripts.CNA_Database</i>), 155	<i>end_clock()</i>	(<i>Organisms.GA.GA_Program_Details.GA_Program_Details</i> method), 91
Collection (class in <i>Organisms.GA.Collection</i>), 94	<i>ending_details()</i>	(<i>Organisms.GA.GA_Program_Details.GA_Program_Details</i> method), 91
Collections_Iterator (class in <i>Organisms.GA.Collections_Iterator</i>), 99	<i>Energy_Fitness_Options</i> (class in <i>Organisms.GA.Fitness_Operators.Energy_Fitness_Operator</i>), 139	
<i>convert_population_fitness_to_SCM_fitness_contribution()</i> (<i>Organisms.GA.Fitness_Operators.SCM_and_Energy_Fitness_Operator.SCM_and_Energy_Fitness_Operator</i> method), 145	<i>energy_fitness_options()</i>	(<i>Organisms.GA.Fitness_Operators.Energy_Fitness_Operator.Energy_Fitness_Operator</i> method), 141
<i>convert_to_bytes()</i> (in module <i>Organisms.GA.GA_Recording_System</i>), 164	<i>Energy_Predation_Operator</i> (class in <i>Organisms.GA.Predation_Operators.Energy_Predation_Operator</i>), 123	
<i>create()</i> (<i>Organisms.GA.EnergyProfile.EnergyProfile</i> method), 102	<i>Energy_Predation_Operators_Options</i> (class in <i>Organisms.GA.Predation_Operators.Energy_Predation_Operator</i>), 123	
<i>create()</i> (<i>Organisms.GA.GA_Program_Details.GA_Program_Details</i> method), 90	<i>EnergyProfile</i> (class in <i>Organisms.GA.EnergyProfile</i>), 101	
<i>create_a_cluster()</i> (in module <i>Organisms.GA.Initialise_Population</i>), 110	<i>Epoch</i> (class in <i>Organisms.GA.Epoch</i>), 156	
<i>Create_An_Offspring()</i> (in module <i>Organisms.GA.Get_Offspring</i>), 110	<i>Exploded()</i> (in module <i>Organisms.GA.ExternalDefinitions</i>), 159	
<i>Create_An_Unoptimised_Offspring()</i> (in module <i>Organisms.GA.Get_Offspring</i>), 110	<i>exponential_function()</i>	(<i>Organisms.GA.Fitness_Operators.Fitness_Function.Fitness_Function</i> method), 148
<i>create_collection_history()</i> (<i>Organisms.GA.Collection.Collection</i> method), 96	F	
<i>Crossover</i> (class in <i>Organisms.GA.Crossover</i>), 111	<i>Fitness_Function</i> (class in <i>Organisms.GA.Fitness_Operators.Fitness_Function</i>), 148	
<i>current_state_file()</i> (<i>Organisms.GA.Population.Population</i> method), 100	<i>Fitness_Operator</i> (class in <i>Organisms.GA.Fitness_Operators.Fitness_Operator</i>), 138	
<i>custom_verify_cluster()</i> (<i>Organisms.GA.Cluster.Cluster</i> method), 92	G	
<i>Cut_and_Splice_Devon_and_Ho()</i> (<i>Organisms.GA.Crossover.Crossover</i> method), 112	<i>GA_Initiate()</i> (in module <i>Organisms.GA.GA_Initiate</i>), 105	
D	<i>GA_Program</i> (class in <i>Organisms.GA.GA_Program</i>), 88	
<i>deepcopy()</i> (<i>Organisms.GA.Cluster.Cluster</i> method), 92	<i>GA_Program_Details</i> (class in <i>Organisms.GA.GA_Program_Details</i>), 90	
<i>deepcopy_skeleton()</i> (<i>Organisms.GA.Cluster.Cluster</i> method), 92	<i>GA_Logo</i> (in module <i>Organisms.GA.GA_Introducing_Remarks</i>), 106	
<i>delete_collection_database()</i> (<i>Organisms.GA.Collection.Collection</i> method), 96	<i>GA_Recording_Database</i> (class in <i>Organisms.GA.GA_Recording_System</i>), 161	
<i>did_get_data_from_backup()</i> (<i>Organisms.GA.Epoch.Epoch</i> method), 156		
<i>direct_function()</i> (<i>Organisms.GA.Fitness_Operators.Fitness_Function.Fitness_Function</i> method), 148		
<i>does_backup_exist()</i> (<i>Organisms.GA.Epoch.Epoch</i> method), 157		

GA_Recording_System (class in Organisms.GA.GA_Recording_System), 163

GA_Setup() (in module Organisms.GA.GA_Setup), 103

GA_Starts() (Organisms.GA.EnergyProfile.EnergyProfile method), 101

get_all_averages_for_a_cluster() (Organisms.GA.SCM_Scripts.CNA_Database.CNA_Database method), 154

get_atomic_CNA_profile() (in module Organisms.GA.SCM_Scripts.A_SCM_Methods), 151

get_average() (Organisms.GA.SCM_Scripts.Similarity_Profile.Similarity_Profile method), 150

get_cluster_distance_list() (in module Organisms.GA.Predation_Operators.IDCM_Predation_Operator_Scripts.IDCM_Methods), 132

get_cluster_energies() (Organisms.GA.Collection.Collection method), 96

get_cluster_from_name() (Organisms.GA.Collection.Collection method), 96

get_cluster_names() (Organisms.GA.Collection.Collection method), 96

get_cluster_names() (Organisms.GA.GA_Recording_System.GA_Recording_System method), 162

get_cluster_names() (Organisms.GA.Predation_Operators.IDCM_Predation_Operator_Scripts.IDCM_Methods method), 133

get_clusters() (Organisms.GA.Collection.Collection method), 96

get_CNA_fitness_contribution() (in module Organisms.GA.Fitness_Operators.CNA_Fitness_Contribution), 146

get_CNA_fitness_contribution_normalised() (in module Organisms.GA.Fitness_Operators.CNA_Fitness_Contribution), 146

get_CNA_fitness_parameter() (in module Organisms.GA.Fitness_Operators.CNA_Fitness_Contribution), 146

get_CNA_fitness_parameter_normalised() (in module Organisms.GA.Fitness_Operators.CNA_Fitness_Contribution), 147

get_CNA_profile() (in module Organisms.GA.SCM_Scripts.A_SCM_Methods), 151

get_CNA_profile() (in module Organisms.GA.SCM_Scripts.T_SCM_Methods), 152

get_CNA_similarities() (in module Organisms.GA.SCM_Scripts.A_SCM_Methods), 151

get_CNA_similarities() (in module Organisms.GA.SCM_Scripts.T_SCM_Methods), 152

get_CNA_similarity() (in module Organisms.GA.SCM_Scripts.A_SCM_Methods), 151

get_CNA_similarity() (in module Organisms.GA.SCM_Scripts.T_SCM_Methods), 152

get_CNA_similarity() (in module Organisms.GA.SCM_Scripts.IDCM_Methods), module Organisms.GA.SCM_Scripts.Similarity_Profile), 150

get_comparison() (Organisms.GA.SCM_Scripts.Similarity_Profile.Similarity_Profile method), 150

get_current_generation_and_last_cluster_generated() (Organisms.GA.EnergyProfile.EnergyProfile method), 102

get_current_generation_from_state_file() (Organisms.GA.Population.Population method), 100

get_data_from_current_state_file() (Organisms.GA.Population.Population method), 100

get_DrCNA() (Organisms.GA.Population.Population method), 100

get_elemental_makeup() (in module Organisms.GA.ExternalDefinitions), 160

get_elemental_makeup() (Organisms.GA.Cluster.Cluster method), 93

get_energetic_fitness_contribution() (in module Organisms.GA.Fitness_Operators.Energetic_Fitness_Contribution), 142

get_energy_predation_methods() (Organisms.GA.Predation_Operators.Energy_Predation_Operator_Energy_Predation method), 124

get_entry() (Organisms.GA.Predation_Operators.IDCM_Predation_Operator_Scripts.IDCM_Methods method), 132

method), 133

get_fitness() (Organisms.GA.Fitness_Operators.Fitness_Function.Fitness_Function method), 148

get_fitness_operator() (in module Organisms.GA.Get_Predation_and_Fitness_Operators), 117

get_history_path() (Organisms.GA.Collection.Collection method), 96

get_identical_structures_initial_population() (Organisms.GA.Predation_Operators.IDCM_Predation_Operator.IDCM_Predation_Operator method), 131

get_index() (Organisms.GA.Collection.Collection method), 96

get_index() (Organisms.GA.GA_Recording_System.GA_Recording_Database method), 162

get_lowest_and_highest_energies_from_collection() (in module Organisms.GA.Fitness_Operators.Energetic_Fitness_Contribution), 142

get_lowest_and_highest_similarities_from_collection() (in module Organisms.GA.Fitness_Operators.CNA_Fitness_Contribution), 147

get_max_mean_min_energies() (Organisms.GA.Collection.Collection method), 97

get_max_similarity() (Organisms.GA.SCM_Scripts.CNA_Database.CNA_Database method), 154

get_mutation_type() (Organisms.GA.Mutation.Mutation method), 115

get_pool_folder_size() (Organisms.GA.Population.Population method), 100

get_population_fitness() (Organisms.GA.Fitness_Operators.SCM_and_Energy_Fitness_Operators.SCM_and_Energy_Fitness_Operators method), 145

get_predation_and_fitness_operators() (in module Organisms.GA.Get_Predation_and_Fitness_Operators), 118

get_predation_operator() (in module Organisms.GA.Get_Predation_and_Fitness_Operators), 118

get_rCut_values() (in module Organisms.GA.SCM_Scripts.SCM_initialisation), 149

get_rCuts() (in module Organisms.GA.SCM_Scripts.SCM_initialisation), 150

get_results() (Organisms.GA.SCM_Scripts.Similarity_Profile.Similarity_Profile method), 150

get_time_from_generation() (in module Organisms.GA.GA_Initiate), 105

get_resumed_mean_energy_epoch_details() (Organisms.GA.Epoch.Epoch method), 157

get_resumed_same_population_epoch_details() (Organisms.GA.Epoch.Epoch method), 157

get_rho_i() (in module Organisms.GA.Fitness_Operators.Energetic_Fitness_Contribution), 143

get_similar_clusters_in_database() (Organisms.GA.SCM_Scripts.CNA_Database.CNA_Database method), 154

get_similar_clusters_to_remove() (Organisms.GA.Predation_Operators.SCM_Predation_Operator.SCM_Predation_Operator method), 137

get_tasks() (in module Organisms.GA.GA_Recording_System), 164

get_tasks() (in module Organisms.GA.Initialise_Population), 110

get_tasks() (in module Organisms.GA.SCM_Scripts.A_SCM_Methods), 152

get_tasks() (in module Organisms.GA.SCM_Scripts.T_SCM_Methods), 153

get_tasks() (Organisms.GA.GA_Program.GA_Program method), 90

get_time_now() (Organisms.GA.Timer.Timer method), 164

get_total_cluster_energy() (Organisms.GA.Cluster.Cluster method), 93

get_total_CNA_profile() (in module Organisms.GA.SCM_Scripts.SCM_initialisation), 153

get_total_length_of_running_time() (Organisms.GA.Timer.Timer method), 164

H

half_index_custom_method() (Organisms.GA.Crossover.Crossover method), 112

half_index_half_method() (Organisms.GA.Crossover.Crossover method), 113

half_index_random_method() (Organisms.GA.Crossover.Crossover method), 113

half_index_weighted_method() (Organisms.GA.Crossover.Crossover method), 113

has_elapsed_time() (Organisms.GA.Timer.Timer method), 164

history_file_name() (*Organisms.GA.Collection.Collection* method), 97

homotopMutate() (*in module Organisms.GA.Types_Of_Mutations*), 116

I

IDCM_Predation_Operator (*class in Organisms.GA.Predation_Operators.IDCM_Predation_Operator*), 128

import_clusters_from_database_to_memory() (*Organisms.GA.Collection.Collection* method), 97

import_information_from_database() (*Organisms.GA.GA_Recording_System.GA_Recording_System* method), 162

import_information_from_database() (*Organisms.GA.GA_Recording_System.GA_Recording_System* method), 163

import_surface() (*in module Organisms.GA.Cluster*), 93

InclusionboxOfCluster() (*in module Organisms.GA.ExternalDefinitions*), 160

InclusionRadiusOfCluster() (*in module Organisms.GA.ExternalDefinitions*), 160

information_from_the_same_generation() (*Organisms.GA.Epoch.Epoch* method), 157

Initate_New_GAProgram() (*in module Organisms.GA.GA_Initiate*), 105

Initial_ProgramChecking() (*in module Organisms.GA.GA_Initiate*), 105

initial_similarity_profile_generator() (*in module Organisms.GA.SCM_Scripts.CNA_Database*), 155

initialise_mean_energy_epoch_details() (*Organisms.GA.Epoch.Epoch* method), 157

Initialise_Population() (*in module Organisms.GA.Initialise_Population*), 106

Initialise_Population_with_Randomly_Generated_Clusters() (*in module Organisms.GA.Initialise_Population*), 107

initialise_same_population_epoch_details() (*Organisms.GA.Epoch.Epoch* method), 157

Introducing_Remarks() (*in module Organisms.GA.GA_Introducing_Remarks*), 106

is_cluster_pair_in_the_database() (*Organisms.GA.Predation_Operators.IDCM_Predation_Operator_Scripts.LoD_Comparison_Database.LoD_Comparison_Database* method), 134

is_LES_note_in_EnergyProfile() (*Organisms.GA.EnergyProfile.EnergyProfile* method), 102

is_pair_in_the_database() (*Organisms.GA.SCM_Scripts.CNA_Database.CNA_Database* method), 154

is_position_already_occupied_by_an_atom_in_Cluster() (*in module Organisms.GA.ExternalDefinitions*), 160

is_there_an_energy_range() (*Organisms.GA.Collection.Collection* method), 97

is_there_an_similarity_range() (*Organisms.GA.Fitness_Operators.SCM_and_Energy_Fitness_Operators* method), 145

isfloat() (*in module Organisms.GA.Mutation*), 116

K

keys() (*Organisms.GA.Predation_Operators.IDCM_Predation_Operator* method), 134

keys() (*Organisms.GA.SCM_Scripts.CNA_Database.CNA_Database* method), 155

L

linear_function() (*Organisms.GA.Fitness_Operators.Fitness_Function.Fitness_Function* method), 149

LoD_compare_two_structures() (*in module Organisms.GA.Predation_Operators.IDCM_Predation_Operator_Scripts.LoD_Comparison_Database*), 132

LoD_Comparison_Database (*class in Organisms.GA.Predation_Operators.IDCM_Predation_Operator_Scripts.LoD_Comparison_Database*), 133

LoD_Similarity_Analysis() (*Organisms.GA.Predation_Operators.IDCM_Predation_Operator_Scripts.LoD_Similarity_Analysis* method), 133

M

make_collection_folder() (*Organisms.GA.Collection.Collection* method), 97

make_database_table() (*Organisms.GA.Predation_Operators.IDCM_Predation_Operator_Scripts.LoD_Comparison_Database* method), 134

make_folder() (*in module Organisms.GA.GA_Recording_System*), 164

make_simple_table() (*Organisms.GA.SCM_Scripts.CNA_Database.CNA_Database* method), 155

mate_Cut_and_Splice() (*Organisms.GA.Crossover.Crossover* method), 113

mate_Cut_and_Splice_error_checking() (*Organisms.GA.Crossover.Crossover* method), 113

mate_Cut_and_Splice_error_checking_2() (*Organisms.GA.Crossover.Crossover* method), 113

mate() (*Organisms.GA.Crossover.Crossover* method), 114

max_energy() (*Organisms.GA.Collection.Collection*
method), 97
 mean_energy() (*Organisms.GA.Collection.Collection*
method), 97
 min_energy() (*Organisms.GA.Collection.Collection*
method), 98
 module
 Organisms.GA.Cluster, 91
 Organisms.GA.Collection, 94
 Organisms.GA.Collections_Iterator,
 99
 Organisms.GA.Crossover, 111
 Organisms.GA.EnergyProfile, 101
 Organisms.GA.Epoch, 156
 Organisms.GA.ExternalDefinitions,
 159
 Organisms.GA.Fitness_Operators.CNA_Fitness_Operators, 146
 Organisms.GA.Fitness_Operators.Energetic_Fitness_Operators, 142
 Organisms.GA.Fitness_Operators.Energy_Fitness_Operators, 139
 Organisms.GA.Fitness_Operators.Fitness_Functions, 148
 Organisms.GA.Fitness_Operators.Fitness_Operators, 138
 Organisms.GA.Fitness_Operators.SCM_and_Energy_Fitness_Operators, 143
 Organisms.GA.GA_Initiate, 105
 Organisms.GA.GA_Introducing_Remarks,
 106
 Organisms.GA.GA_Program, 88
 Organisms.GA.GA_Program_Details, 90
 Organisms.GA.GA_Recording_System,
 161
 Organisms.GA.GA_Setup, 103
 Organisms.GA.Get_Offspring, 110
 Organisms.GA.Get_Predation_and_Fitness_Operators, 117
 Organisms.GA.Initialise_Population,
 106
 Organisms.GA.Memory_Operator, 159
 Organisms.GA.Mutation, 115
 Organisms.GA.Offspring_Pool, 103
 Organisms.GA.Population, 99
 Organisms.GA.Predation_Operators.Energy_Predation_Operator,
 123
 Organisms.GA.Predation_Operators.Energy_Predation_Operators, 125
 Organisms.GA.Predation_Operators.Energy_Predation_Operators.Comprehensive_Energy_Predation_Operators, 127
 Organisms.GA.Predation_Operators.Energy_Predation_Operator_Scripts.Simple_Energy_Predation_Operators, 124
 Organisms.GA.Predation_Operators.IDCM_Predation_Operators, 128
 Organisms.GA.Predation_Operators.IDCM_Predation_Operators, 132
 Organisms.GA.Predation_Operators.IDCM_Predation_Operators, 133
 Organisms.GA.Predation_Operators.No_Predation_Operators, 121
 Organisms.GA.Predation_Operators.Predation_Operators, 119
 Organisms.GA.Predation_Operators.SCM_Predation_Operators, 135
 Organisms.GA.SCM_Scripts.A_SCM_Methods,
 151
 Organisms.GA.SCM_Scripts.CNA_Database,
 153
 Organisms.GA.SCM_Scripts.MyPool, 156
 Organisms.GA.SCM_Scripts.SCM_initialisation,
 149
 Organisms.GA.SCM_Scripts.SCM_Scripts.Similarity_Profile,
 150
 Organisms.GA.SCM_Scripts.T_SCM_Methods,
 152
 Organisms.GA.Surface, 94
 Organisms.GA.Timer, 164
 Organisms.GA.Types_Of_Mutations, 116
 move_backup_database_to_normal_backup()
 (*Organisms.GA.Collection.Collection*
 method), 98
 move_backup_to_current_files() (*Organisms.GA.Population.Population*
 method), 100
 moveMutate() (*in module Organisms.GA.Types_Of_Mutations*), 116
 Mutation (*class in Organisms.GA.Mutation*), 115
 mutation() (*Organisms.GA.Mutation.Mutation*
 method), 115
 MyConstraint (*class in Organisms.GA.Surface*), 94
 MyPool (*class in Organisms.GA.SCM_Scripts.MyPool*), 156
N
 natural_selection() (*Organisms.GA.GA_Program.GA_Program*
 method), 90
 no_epoch() (*Organisms.GA.Epoch.Epoch*
 method), 157
 No_Predation_Operator (*class in Organisms.GA.Predation_Operators.No_Predation_Operators*), 121
 No_Predation_Operators (*class in Organisms.GA.Predation_Operators.No_Predation_Operators*), 121
 My_Predation_Operators (*class in Organisms.GA.SCM_Scripts.MyPool*), 156

O

- Offspring_Pool (class in *Organisms.GA.Offspring_Pool*), 103
 - open() (*Organisms.GA.Collection.Collection* method), 98
 - open() (*Organisms.GA.EnergyProfile.EnergyProfile* method), 102
 - open() (*Organisms.GA.GA_Program_Details.GA_Program_Details* method), 91
 - Organisms.GA.Cluster module, 91
 - Organisms.GA.Collection module, 94
 - Organisms.GA.Collections_Iterator module, 99
 - Organisms.GA.Crossover module, 111
 - Organisms.GA.EnergyProfile module, 101
 - Organisms.GA.Epoch module, 156
 - Organisms.GA.ExternalDefinitions module, 159
 - Organisms.GA.Fitness_Operators.CNA_Fitness_Contribution module, 146
 - Organisms.GA.Fitness_Operators.Energetic_Fitness_Contribution module, 142
 - Organisms.GA.Fitness_Operators.Energy_Fitness_Operator module, 139
 - Organisms.GA.Fitness_Operators.Fitness_Function module, 148
 - Organisms.GA.Fitness_Operators.Fitness_Operator module, 138
 - Organisms.GA.Fitness_Operators.SCM_and_Energy_Fitness_Operator module, 143
 - Organisms.GA.GA_Initiate module, 105
 - Organisms.GA.GA_Introducing_Remarks module, 106
 - Organisms.GA.GA_Program module, 88
 - Organisms.GA.GA_Program_Details module, 90
 - Organisms.GA.GA_Recording_System module, 161
 - Organisms.GA.GA_Setup module, 103
 - Organisms.GA.Get_Offspring module, 110
 - Organisms.GA.Get_Predation_and_Fitness_Operators module, 117
 - Organisms.GA.Initialise_Population module, 106
 - Organisms.GA.Memory_Operator module, 159
 - Organisms.GA.Mutation module, 115
 - Organisms.GA.Offspring_Pool module, 103
 - Organisms.GA.Population module, 99
 - Organisms.GA.Predation_Operators.Energy_Predation_Operator module, 123
 - Organisms.GA.Predation_Operators.Energy_Predation_Operator module, 125
 - Organisms.GA.Predation_Operators.Energy_Predation_Operator module, 127
 - Organisms.GA.Predation_Operators.Energy_Predation_Operator module, 124
 - Organisms.GA.Predation_Operators.IDCM_Predation_Operator module, 128
 - Organisms.GA.Predation_Operators.IDCM_Predation_Operator module, 132
 - Organisms.GA.Predation_Operators.IDCM_Predation_Operator module, 133
 - Organisms.GA.Predation_Operators.No_Predation_Operator module, 121
 - Organisms.GA.Predation_Operators.Predation_Operator module, 119
 - Organisms.GA.Predation_Operators.SCM_Predation_Operator module, 135
 - Organisms.GA.SCM_Scripts.A_SCM_Methods module, 151
 - Organisms.GA.SCM_Scripts.CNA_Database module, 153
 - Organisms.GA.SCM_Scripts.MyPool module, 156
 - Organisms.GA.SCM_Scripts.SCM_initialisation module, 149
 - Organisms.GA.SCM_Scripts.Similarity_Profile module, 150
 - Organisms.GA.SCM_Scripts.T_SCM_Methods module, 152
 - Organisms.GA.Surface module, 94
 - Organisms.GA.Timer module, 164
 - Organisms.GA.Types_Of_Mutations module, 116
- P**
- perform_epoch() (*Organisms.GA.Epoch.Epoch* method), 158
 - perform_epoch_mean_energy() (*Organisms.GA.Epoch.Epoch* method), 158
 - perform_epoch_no_epoch() (*Organisms.GA.Epoch.Epoch* method), 158

perform_epoch_same_population() (*Organisms.GA.Epoch.Epoch method*), 158

pickClusterFromThePopulation() (*Organisms.GA.Mutation.Mutation method*), 115

pickParentsFromThePopulation() (*Organisms.GA.Crossover.Crossover method*), 114

Place_Already_Created_Clusters_In_Population() (*in module Organisms.GA.Initialise_Population*), 109

pop() (*Organisms.GA.Collection.Collection method*), 98

pop_identical_structures() (*Organisms.GA.Predation_Operators.IDCM_Predation_Operator.IDCM_Predation_Operator method*), 131

Population (*class in Organisms.GA.Population*), 99

Predation_Operator (*class in Organisms.GA.Predation_Operators.Predation_Operator*), 119

print_clusters() (*Organisms.GA.Population.Population method*), 100

print_cna_database_details() (*Organisms.GA.SCM_Scripts.CNA_Database.CNA_Database method*), 155

print_elapsed_time() (*Organisms.GA.Timer.Timer method*), 164

print_initial_message() (*Organisms.GA.Fitness_Operators.SCM_and_Energy_Fitness_Operators.SCM_and_Energy_Fitness_Operators method*), 145

Process (*Organisms.GA.SCM_Scripts.MyPool.MyPool attribute*), 156

R

randomMutate() (*in module Organisms.GA.Types_Of_Mutations*), 116

read_collection_database() (*Organisms.GA.Collection.Collection method*), 98

record_collection() (*Organisms.GA.GA_Recording_System.GA_Recording_System method*), 163

record_initial_populations() (*Organisms.GA.GA_Recording_System.GA_Recording_System method*), 163

record_population_at_generation() (*Organisms.GA.GA_Recording_System.GA_Recording_System method*), 163

remove() (*Organisms.GA.Collection.Collection method*), 98

remove() (*Organisms.GA.Predation_Operators.IDCM_Predation_Operator.IDCM_Predation_Operator method*), 134

remove() (*Organisms.GA.SCM_Scripts.CNA_Database.CNA_Database method*), 155

remove_backup() (*Organisms.GA.Epoch.Epoch method*), 158

remove_backup_database() (*Organisms.GA.Collection.Collection method*), 98

remove_backup_database_if_exists() (*Organisms.GA.Collection.Collection method*), 98

remove_backup_files() (*Organisms.GA.Population.Population method*), 100

remove_backup_state_file() (*Organisms.GA.Population.Population method*), 101

remove_backup_state_file_if_exists() (*Organisms.GA.Population.Population method*), 101

remove_calculator() (*Organisms.GA.Cluster.Cluster method*), 93

remove_clusters_from_database_that_are_from_unsuccessful() (*Organisms.GA.Collection.Collection method*), 98

remove_end_lines_from_text() (*in module Organisms.GA.EnergyProfile*), 102

remove_from_database() (*Organisms.GA.Fitness_Operators.Energy_Fitness_Operator.Energy_Fitness_Operator method*), 141

remove_from_database() (*Organisms.GA.Fitness_Operators.SCM_and_Energy_Fitness_Operators.SCM_and_Energy_Fitness_Operators method*), 145

remove_from_database() (*Organisms.GA.Predation_Operators.Energy_Predation_Operator.Energy_Predation_Operator method*), 124

remove_from_database() (*Organisms.GA.Predation_Operators.IDCM_Predation_Operator.IDCM_Predation_Operator method*), 131

remove_from_database() (*Organisms.GA.Predation_Operators.No_Predation_Operator.No_Predation_Operator method*), 122

remove_from_database() (*Organisms.GA.Predation_Operators.Predation_Operator.Predation_Operator method*), 120

remove_from_database() (*Organisms.GA.Predation_Operators.SCM_Predation_Operator.SCM_Predation_Operator method*), 137

remove_offspring_and_replace_with_population_that_violate_the_predation_operator() (*Organisms.GA.Predation_Operators.Predation_Operator.Predation_Operator method*), 120

remove_offspring_that_violate_the_predation_operator() (*Organisms.GA.Predation_Operators.Predation_Operator.Predation_Operator method*), 120

remove_similar_clusters_in_population() (*Organisms.GA.SCM_Scripts.LoD_Comparison_Database.LoD_Comparison_Database method*), 130

remove_to_database() (*Organisms.GA.Predation_Operators.IDCM_Predation_Operator.IDCM_Predation_Operator method*), 131

remove_to_database() (*Organisms.GA.Predation_Operators.No_Predation_Operator.No_Predation_Operator method*), 122

remove_to_database() (*Organisms.GA.Predation_Operators.Predation_Operator.Predation_Operator method*), 120

remove_to_database() (*Organisms.GA.Predation_Operators.SCM_Predation_Operator.SCM_Predation_Operator method*), 137

Tree (class in Organisms.GA.SCM_Scripts.CNA_Database), 155

U

update_cluster_in_database_for_if_in_population() (Organisms.GA.GA_Recording_System.GA_Recording_Database method), 162

update_cluster_in_database_for_if_in_population() (Organisms.GA.GA_Recording_System.GA_Recording_System method), 164

update_LoD_database() (Organisms.GA.Predation_Operators.IDCM_Predation_Operator.IDCM_Predation_Operator method), 132

V

verify_cluster() (Organisms.GA.Cluster.Cluster method), 93

version_no() (in module Organisms.GA.GA_Introducing_Remarks), 106

view() (Organisms.GA.Cluster.Cluster method), 93

view_cluster() (Organisms.GA.Collection.Collection method), 99

W

which_clusters_in_LoD_comparison_database_are_similar() (Organisms.GA.Predation_Operators.IDCM_Predation_Operator_Scripts.LoD_Comparison_Database.LoD_Comparison_Database method), 134

Will_Mutation_Occur() (in module Organisms.GA.Get_Offspring), 111