

navicatGA

Release 0.0.0

rlaplaza

Oct 28, 2021

1	Contents	1
2	About	3
3	Install	5
4	Documentation	7
5	Examples	9
6	navicatGA.base_solver module	11
7	navicatGA.cache module	13
8	navicatGA.chemistry_selfies module	15
9	navicatGA.chemistry_smiles module	17
10	navicatGA.chemistry_xyz module	19
11	navicatGA.exception_messages module	21
12	navicatGA.exceptions module	23
13	navicatGA.fitness_functions_float module	25
14	navicatGA.fitness_functions_selfies module	27
15	navicatGA.fitness_functions_xyz module	29
16	navicatGA.float_solver module	31
17	navicatGA.helpers module	33
18	navicatGA.logger module	35
19	navicatGA.progressBars module	37

20	navicatGA.quantum_wrappers_selfies module	39
21	navicatGA.quantum_wrappers_xyz module	41
22	navicatGA.queue_wrappers_xyz module	43
23	navicatGA.score_modifiers module	45
24	navicatGA.selfies_solver module	47
25	navicatGA.smiles_solver module	49
26	navicatGA.timeout module	51
27	navicatGA.wrappers_selfies module	53
28	navicatGA.wrappers_smiles module	55
29	navicatGA.wrappers_xyz module	57
30	navicatGA.xyz_solver module	59
	Python Module Index	61
	Index	63

Contents

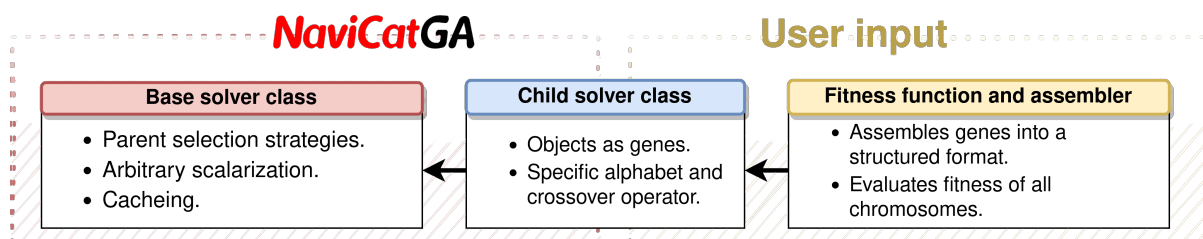
- Chapter 2
- Chapter 3
- Chapter 4
- Chapter 5

About

The code runs on pure python and is thus not extremely optimized. However, it is easy to adapt for particular applications. Dependencies are minimal for the base class:

- `numpy`
- `matplotlib`

The library is projected as base class containing the core methods, which are inherited by child classes to define the problem.



Several child solver classes are provided as fairly complete examples, but might need to be adapted or monkeypatched for particular applications.

The child classes and some functionalities have additional dependencies:

- The `selfies_solver` class implementation requires `selfies` (<https://github.com/aspuru-guzik-group/selfies>) and `rdkit` (<https://www.rdkit.org/>). `rdkit` may be replaced manually by `openbabel` python bindings or other chemoinformatic modules.
- The `smiles_solver` class implementation requires `rdkit` (<https://www.rdkit.org/>). `rdkit` may be replaced manually by `openbabel` python bindings or other chemoinformatic modules.
- The `xyz_solver` class implementation requires `AaronTools` (<https://github.com/QChASM/AaronTools.py>).
- Wrappers and chemistry modules contain functions that depend on `pyscf` to solve the electronic structure problem. However, these are provided for exemplary purposes and not a core functionality.
- `matter-chimera` (<https://github.com/aspuru-guzik-group/chimera>) is recommended for scalarization. Alternatively, a scalarizer object with a `scalarize` method can be passed to the solver.

Additional features require `alive-progress` (for progress bars, very useful for CLI usage). However, these are implemented by monkeypatching the base class, and thus no functionality is lost without them.

Install

Installation is as simple as:

```
python setup.py install --record files.txt
```

This ensures easy uninstall. Just remove all files listed in files.txt using:

```
rm $(cat files.txt)
```

Documentation

The documentation is available [here](#).

Examples

The tests subdirectory contains a copious amount of tests which double as examples.

navicatGA.base_solver module

class navicatGA.base_solver.**GenAlgSolver** (*n_genes: int, fitness_function=None, assembler=None, max_gen: int = 1000, max_conv: int = 100, pop_size: int = 100, mutation_rate: float = 0.15, selection_rate: float = 0.5, selection_strategy: str = 'roulette_wheel', excluded_genes: Optional[Sequence] = None, n_crossover_points: int = 1, random_state: Optional[int] = None, lru_cache: bool = False, scalarizer=None, prune_duplicates=False, verbose: bool = True, show_stats: bool = False, plot_results: bool = False, to_stdout: bool = True, to_file: bool = True, logger_file: str = 'output.log', logger_level: str = 'INFO', progressBars: bool = False, problem_type: str = 'base'*)

Bases: object

calculate_fitness (*population*)

Calculates the fitness of the population using the defined fitness function.

Parameters: :param population: population (array of chromosomes)

Returns: :return fitness: scalarized fitness of the current population, will be used :return pfitness: not-scalarized fitness of the current population, for printing

check_input_base (*fitness_function, selection_strategy, pop_size: int, excluded_genes*)

Function to check that the main arguments have been passed to the GenAlgSolver instance.

Parameters: :param fitness_function: a fitness function that takes a chromosome and returns a fitness :param selection_strategy: a selection strategy string that can be recognized by this class :param pop_size: the number of chromosomes :param excluded_genes: a sequence of genes that should not change or mutate

close_solve_logger ()

Closes the logger of this solver. This avoid multiple loggers stacking when another solver is created.

static create_offspring (*first_parent, sec_parent, crossover_pt, offspring_number*)

Creates an offspring from 2 parent chromosomes. It uses the crossover point(s) to determine how to perform the crossover. To be implemented on each child class. Must return the resulting offspring chromosome.

get_boltzmann_probabilities (*fitness*)

Calculates selection probabilities according to a fitness Boltzmann distribution with an increasing temperature.

get_crossover_points ()

Retrieves random crossover points.

get_number_mutations ()

Returns the number of mutations that need to be performed.

get_selection_probabilities ()

Calculates selection probabilities either randomly or scaled by position.

abstract initialize_population ()

Initializes the population of the problem. To be implemented in each child class.

interval_selection (value)

Select based on self.prob_intervals, which are given by the selection strategy.

Parameters: :param value: random value defining which individual is selected from the probability intervals

Returns: :return: the selected individual from the population

mutate_population (population, n_mutations)

Mutates the population according to a given user defined rule. To be defined further in each child class. Each direct child class can call this super method to retrieve the mutation rows and mutations columns in population.

static plot_fitness_results (mean_fitness, max_fitness, iterations: int)

Plots the evolution of the mean and max fitness of the population using matplotlib.

Parameters: :param mean_fitness: mean fitness array for each generation :param max_fitness: max fitness array for each generation :param iterations: total number of generations

print_stats (time_str)

Prints the statistics of the optimization run.

select_parents (fitness)

Selects the parents according to a given selection strategy. Options are: roulette_wheel: Selects individuals from mating pool giving higher probabilities to fitter individuals. two_by_two: Pairs fittest individuals two by two random: Selects individuals from mating pool randomly. tournament: Selects individuals by choosing groups of 3 candidate individuals and then selecting the fittest one from the 3.

Parameters: :param fitness: the fitness values of the whole population at a given iteration

Returns: :return (ma, pa): a tuple containing the selected 2 parents for each mating

solve (niter=None)

Performs the genetic algorithm optimization according to the parameters loaded in __init__. Will run for max_gen or until it converges for max_conv iterations, or for min(niter,max_gen) iterations if niter is an integer. Will start using previous state if available.

Parameters: :param niter: the number of generations to run

static sort_by_fitness (fitness, population, printable_fitness)

Sorts fitness, population and printable fitness according to fitness.

tournament_selection (fitness, range_max)

Performs tournament selection.

Parameters: :param fitness: the fitness values of the population at a given iteration :param range_max: range of individuals that can be selected for the tournament

Returns: :return: the selected individuals

static tournament_selection_helper (selected_individuals, fitness)

Helper for tournament selection method. Selects the fittest individual from a pool of candidate individuals.

navicatGA.cache module

`navicatGA.cache.calculate_fitness_cache (self, population)`

Calculates the fitness of the population using a hashable fitness function.

Parameters: :param population: population state at a given iteration :return: the fitness of the current population

`navicatGA.cache.calculate_one_fitness_cache (hashable, fitness_function)`

`navicatGA.cache.set_lru_cache (self)`

Monkeypatches the `calculate_fitness` method of the base solver class in order to use a lru cache. If a specific wrapper exists for a given solver, it will try to use the unique expression of genes given by that wrapper to generate a hashable fitness function. If not, it will require a hashable fitness function given by the user AND expect the given `fitness_function` to generate a unique hash from a gene.

navicatGA.chemistry_selfies module

`navicatGA.chemistry_selfies.check_selfie_chars (chromosome)`

Check if a list of selfies characters leads to a valid smiles string. Uses `sanitize_smiles` to check the smiles string from `selfies.decoder`.

Parameters: :param chromosome: list of selfie characters

Returns: :return: True if the smiles string is deemed valid by `sanitize_smiles`, False otherwise

`navicatGA.chemistry_selfies.count_selfie_chars (selfie)`

Count the number of selfie characters in a selfie string. Returns the number.

`navicatGA.chemistry_selfies.decode_selfies_list (selfies_list)`

Decode a list of smiles to a list of selfies using `timed_decoder`.

`navicatGA.chemistry_selfies.diagnose_mol (mol)`

Tries to identify and print to logger whatever was or is wrong with the `chemistry_selfies` of an `rdkit.mol` object.

`navicatGA.chemistry_selfies.draw_selfies (selfies, root_name)`

`navicatGA.chemistry_selfies.encode_smiles_list (smiles_list)`

Encode a list of smiles to a list of selfies using `selfies.encoder`.

`navicatGA.chemistry_selfies.get_ECFP4 (mol)`

`navicatGA.chemistry_selfies.get_conformer_rmsd (mol)`

`navicatGA.chemistry_selfies.get_confs_ff (mol, maxiters=250)`

`navicatGA.chemistry_selfies.get_interatomic_distances (conf)`

`navicatGA.chemistry_selfies.get_selfie_chars (selfie, maxchars=25)`

Obtain an ordered list of all selfie characters in string `selfie` padded to `maxchars` with `[nop]`s.

Parameters: :param selfie: a selfie string representing a molecule :param maxchars: maximum number of elements in the list

Returns: :return chars_selfie: list of selfie characters present in molecule selfie

`navicatGA.chemistry_selfies.get_structure_ff (mol, n_confs=5)`

Generates a reasonable set of 3D structures using forcefields for a given `rdkit.mol` object. It will try several 3D generation approaches in `rdkit`. It will try to sample several conformations and get the minima.

Parameters: :type mol: a `rdkit.mol` object :type n_confs: the number of conformations to sample

Returns: :return mol_structure: the same `rdkit mol` with 3D coordinates

`navicatGA.chemistry_selfies.has_transition_metals (mol)`

Returns True if the `rdkit.mol` object passed as argument has a (transition)-metal atom, False if

else.

`navicatGA.chemistry_selfies.is_transition_metal (at)`

Returns True if the rdkit.Atom object passed as argument is a transition metal, False if else.

`navicatGA.chemistry_selfies.prune_mol_conformers (mol, energies_list)`

`navicatGA.chemistry_selfies.randomize_selfies (selfies, num_random=1)`

`navicatGA.chemistry_selfies.randomize_smiles (mol)`

`navicatGA.chemistry_selfies.sanitize_multiple_smiles (smiles_list)`

Calls `sanitize_smiles` for every item in a list.

Parameters: :param smiles_list list of smiles strings to be sanitized

Returns: :return sanitized_smiles: list of sanitized smile strings with None in errors

`navicatGA.chemistry_selfies.sanitize_smiles (smiles)`

Return a canonical smile representation of smi. If there are metals, it will try to fix the bonds as dative.

Parameters: :param smi: smiles string to be sanitized :type smi: smiles string

Returns: :return mol: corresponding rdkit.mol object, None if exception caught :return smi_canon: canonicalized smile representation of smi, None if exception caught :return conversion_successful: True if no exception caught, False if exception caught

`navicatGA.chemistry_selfies.set_dative_bonds (mol, fromAtoms=(7, 8, 15, 16))`

Tries to replace bonds with metal atoms by dative bonds, while keeping valence rules enforced. Adapted from G. Landrum.

`navicatGA.chemistry_selfies.timed_decoder (selfie)`

Decode a selfies string to smiles using `selfies.decoder`, call exception and return None if decoder takes more than 10 seconds to run.

`navicatGA.chemistry_selfies.timed_sanitizer (smiles)`

Convert smiles string to rdkit.mol, call exception and return None if it takes more than 10 seconds to run.

navicatGA.chemistry_smiles module

`navicatGA.chemistry_smiles.diagnose_mol (mol)`

Tries to identify and print to logger whatever was or is wrong with the chemistry_selfies of an `rdkit.mol` object.

`navicatGA.chemistry_smiles.draw_smiles (smiles, root_name)`

`navicatGA.chemistry_smiles.get_ECFP4 (mol)`

`navicatGA.chemistry_smiles.get_conformer_rmsd (mol)`

`navicatGA.chemistry_smiles.get_confs_ff (mol, maxiters=250)`

`navicatGA.chemistry_smiles.get_interatomic_distances (conf)`

`navicatGA.chemistry_smiles.get_structure_ff (mol, n_confs=5)`

Generates a reasonable set of 3D structures using forcefields for a given `rdkit.mol` object. It will try several 3D generation approaches in `rdkit`. It will try to sample several conformations and get the minima.

Parameters: :param mol: an `rdkit mol` object :type mol: `rdkit.mol` :param n_confs: number of conformations to sample :type n_confs: `int`

Returns: :return mol_structure: mol with 3D coordinate information set

`navicatGA.chemistry_smiles.has_transition_metals (mol)`

Returns `True` if the `rdkit.mol` object passed as argument has a (transition)-metal atom, `False` if else.

`navicatGA.chemistry_smiles.is_transition_metal (at)`

Returns `True` if the `rdkit.Atom` object passed as argument is a transition metal, `False` if else.

`navicatGA.chemistry_smiles.prune_mol_conformers (mol, energies_list)`

`navicatGA.chemistry_smiles.randomize_smiles (mol)`

`navicatGA.chemistry_smiles.sanitize_multiple_smiles (smiles_list)`

Calls `sanitize_smiles` for every item in a list.

Parameters: :param smiles_list: list of smile strings to be sanitized.

Returns: :return sanitized_smiles: list of sanitized smile strings or `None` in case a smiles led to error

`navicatGA.chemistry_smiles.sanitize_smiles (smiles)`

Return a canonical smile representation of smi. If there are metals, it will try to fix the bonds as dative.

Parameters: :param smi: smile string to be canonicalized :type smi: `str`

Returns: :return mol: `rdkit mol` object, `None` if exception caught :return smi_canon: canonicalized

smile representation of smi, None if exception caught. :return conversion_successful: True if no exception caught, False if exception caught.

`navicatGA.chemistry_smiles.set_dative_bonds (mol, fromAtoms=(7, 8, 15, 16))`

Tries to replace bonds with metal atoms by dative bonds, while keeping valence rules enforced. Adapted from G. Landrum.

`navicatGA.chemistry_smiles.timed_sanitizer (smiles)`

Convert smiles string to rdkit.mol, call exception and return None if it takes more than 10 seconds to run.

navicatGA.chemistry_xyz module

class navicatGA.chemistry_xyz.**Hashable_Geometry** (**args, **kw*)

Bases: AaronTools.geometry.Geometry

This is a modified version of the Geometry class in AaronTools.py, which is identical but hashable.

navicatGA.chemistry_xyz.**check_xyz** (*chromosome*)

Check if a list of Geometries can lead to a valid structure.

navicatGA.chemistry_xyz.**draw_xyz** (*outname, geom*)

navicatGA.chemistry_xyz.**get_alphabet_from_path** (*path='alphabet/'*)

navicatGA.chemistry_xyz.**get_default_alphabet** ()

navicatGA.chemistry_xyz.**get_starting_xyz_from_file** (*file_list*)

navicatGA.chemistry_xyz.**get_starting_xyz_from_path** (*path='scaffolds'*)

navicatGA.chemistry_xyz.**get_starting_xyz_from_smi** (*smiles_list*)

navicatGA.chemistry_xyz.**pad_xyz_list** (*xyz, maxchars*)

Pads chromosome with empty elements.

navicatGA.chemistry_xyz.**random_merge_xyz** ()

Default chromosome manipulator: randomly generates an XYZ structure from fragments.

navicatGA.exception_messages module

navicatGA.exceptions module

exception navicatGA.exceptions.**InvalidInput** (**args, **kwargs*)
Bases: Exception

exception navicatGA.exceptions.**NoAssembler** (**args, **kwargs*)
Bases: Exception

exception navicatGA.exceptions.**NoFitnessFunction** (**args, **kwargs*)
Bases: Exception

navicatGA.fitness_functions_float module

navicatGA.fitness_functions_float.**fitness_function_float** (*function_number*)

navicatGA.fitness_functions_selfies module

navicatGA.fitness_functions_xyz module

navicatGA.float_solver module

navicatGA.helpers module

`navicatGA.helpers.check_error (func, *args, **kw)`

`navicatGA.helpers.concatenate_list ()`

Default chromosome manipulator: concatenates all elements of a list.

`navicatGA.helpers.get_elapsed_time (start_time, end_time)`

`navicatGA.helpers.get_input_dimensions (lst, n_dim=0)`

`navicatGA.helpers.make_array ()`

Default chromosome manipulator: turns list into array.

navicatGA.logger module

`navicatGA.logger.addLoggingLevel (levelName, levelNum, methodName=None)`

Adds a new logging level to the logging module and the currently configured logging class.

`navicatGA.logger.close_logger (logger)`

`navicatGA.logger.configure_logger (logger_file='output.log', logger_level='INFO',
to_stdout=True, to_file=True, libraries_level=None)`

navicatGA.progressBars module

`navicatGA.progressBars.set_progressBars (self)`

`navicatGA.progressBars.solve_progress (self, niter=None)`

Performs the genetic algorithm optimization according to the parameters loaded in `__init__`. Will run for `max_gen` or until it converges for `max_conv` iterations, or for `min(niter,max_gen)` iterations if `niter` is an integer. Will start using previous state if available.

Parameters: :param niter: the number of generations to run

navicatGA.quantum_wrappers_selfies module

navicatGA.quantum_wrappers_xyz module

navicatGA.queue_wrappers_xyz module

navicatGA.score_modifiers module

`navicatGA.score_modifiers.AbsoluteModifier (score, target, parameter=1) → float`

`navicatGA.score_modifiers.GaussianModifier (score, target, parameter=1) → float`

`navicatGA.score_modifiers.SquaredModifier (score, target, parameter=1) → float`

`navicatGA.score_modifiers.score_modifier (score, target, score_modifier_number=1, parameter=1) → float`

navicatGA.selfies_solver module

```
class navicatGA.selfies_solver.SelfiesGenAlgSolver ( starting_selfies: list = ['[nop]', start-
ing_random: bool = False, starting_stoned: bool = False, alphabet_list: list = ['[N+1expl]', '[O]', '[S+1ex-
pl]', '[P]', '[#P+1expl]', '[=P+1expl]', '[N-1expl]', '[Expl=Ring2]', '[P]', '[P+1expl]', '[Expl=Ring1]',
'[S-1expl]', '[C]', '[=P-1expl]', '[#C-1expl]', '[O-1expl]', '[Ring2]', '[Branch1_1]', '[Branch1_3]', '[N+1ex-
pl]', '[Branch2_1]', '[N]', '[=S]', '[Branch3_2]', '[I]', '[C+1expl]', '[#P-1expl]', '[Ring3]', '[P-1expl]',
'[Branch1_2]', '[Hexpl]', '[Cl]', '[#N]', '[#C+1expl]', '[S+1expl]', '[#S-1expl]', '[#O+1expl]', '[=N]', '[#C]',
'[=S-1expl]', '[#N+1expl]', '[Expl=Ring3]', '[Branch2_2]', '[#S]', '[O]', '[Branch2_3]', '[F]', '[=N-1expl]',
'[Branch3_1]', '[O+1expl]', '[=S+1expl]', '[=C]', '[=C+1expl]', '[Ring1]', '[Branch3_3]', '[Br]', '[=C-1expl]',
'[S]', '[C-1expl]', '[#P]', '[=O+1expl]'], chromosome_to_selfies=<function concatenate_list.<locals>.sc2str>,
multi_alphabet: bool = False, equivalences: Optional[Sequence] = None, branching: bool = False, variables_lim-
its: bool = False, max_counter: int = 10, n_genes: int = 1, fitness_function=None, max_gen: int = 500, max_-
conv: int = 100, pop_size: int = 100, mutation_rate: float = 0.05, selection_rate: float = 0.25, selection_strategy:
str = 'tournament', excluded_genes: Optional[Sequence] = None, n_crossover_points: int = 1, random_state:
Optional[int] = None, lru_cache: bool = False, scalarizer=None, prune_duplicates=False, verbose: bool = True,
show_stats: bool = False, plot_results: bool = False, to_stdout: bool = True, to_file: bool = True, logger_file: str =
'output.log', logger_level: str = 'INFO', progressBars: bool = False, problem_type='selfies' )
```

Bases: `navicatGA.base_solver.GenAlgSolver`

chromosomize (str_list)

Pad or truncate starting_population chromosome to build a population chromosome.

create_offspring (first_parent, sec_parent, crossover_pt, offspring_number)

Creates an offspring from 2 parents.

get_crossover_points ()

Retrieves random crossover points :return: a numpy array with the crossover points

initialize_population ()

Initializes the population of the problem according to the population size and number of genes and according to the problem type (either integers or floats).

Returns: :return: a numpy array with initialized population

mutate_population (population, n_mutations)

Mutates the population by randomizing specific positions of the population individuals.
:param population: the population at a given iteration :param n_mutations: number of muta-
tions to be performed. :return: the mutated population

refill_population (nrefill=0)

write_population (basename='chromosome')

Print xyz for all the population at the current state.

`navicatGA.selfies_solver.test_benzene_selfies` ()

navicatGA.smiles_solver module

```
class navicatGA.smiles_solver.SmilesGenAlgSolver ( starting_population: list = [['']], starting_random: bool = False, alphabet_list: list = ['C', 'N', 'P', 'O', '[Si]', 'F', '[H]'], chromosome_to_smiles=<function concatenate_list.<locals>.sc2str>, multi_alphabet: bool = False, equivalences: Optional[Sequence] = None, max_counter: int = 10, n_genes: int = 1, fitness_function=None, max_gen: int = 500, max_conv: int = 100, pop_size: int = 100, mutation_rate: float = 0.05, selection_rate: float = 0.25, selection_strategy: str = 'tournament', excluded_genes: Optional[Sequence] = None, n_crossover_points: int = 1, random_state: Optional[int] = None, lru_cache: bool = False, scalarizer=None, prune_duplicates=False, verbose: bool = True, show_stats: bool = False, plot_results: bool = False, to_stdout: bool = True, to_file: bool = True, logger_file: str = 'output.log', logger_level: str = 'INFO', progressBars: bool = False, problem_type='smiles' )
```

```
    Bases: navicatGA.base_solver.GenAlgSolver
```

```
    chromosomize ( str_list )
```

Pad or truncate starting_population chromosome to build a population chromosome.

```
    create_offspring ( first_parent, sec_parent, crossover_pt, offspring_number )
```

Creates an offspring from 2 parents.

```
    get_crossover_points ( )
```

Retrieves random crossover points :return: a numpy array with the crossover points

```
    initialize_population ( )
```

Initializes the population of the problem according to the population size and number of genes and according to the problem type (either integers or floats).

Returns: :return: a numpy array with initialized population

```
    mutate_population ( population, n_mutations )
```

Mutates the population by randomizing specific positions of the population individuals.
:param population: the population at a given iteration :param n_mutations: number of mutations to be performed. :return: the mutated population

```
    refill_population ( nrefill=0 )
```

```
    write_population ( basename='chromosome' )
```

Print xyz for all the population at the current state.

navicatGA.timeout module

```
navicatGA.timeout.handler ( signum, frame )
```

```
class navicatGA.timeout.timeout ( seconds=10, error_message='Function call exceeded the time limit.' )
```

```
    Bases: object
```

```
        handle_timeout ( signum, frame )
```

```
navicatGA.timeout.timer_alarm ( n_sec, handler )
```

navicatGA.wrappers_selfies module

```
navicatGA.wrappers_selfies.mol_structure2cm ( mol )

navicatGA.wrappers_selfies.mol_structure2depictions ( mol_structure,
root_name='output' )
    Generate 2D and 3D depictions from an rdkit.mol object with 3D coordinates.

navicatGA.wrappers_selfies.sc2cm ( chromosome, order='C' )

navicatGA.wrappers_selfies.sc2depictions ( chromosome, root_name='output', lot=0 )
    Generate 2D and 3D depictions from a list of selfies characters.

navicatGA.wrappers_selfies.sc2ilogp ( chromosome )

navicatGA.wrappers_selfies.sc2krr ( chromosome )

navicatGA.wrappers_selfies.sc2levenshtein_to_target ( chromosome, target_selfie )

navicatGA.wrappers_selfies.sc2logp ( chromosome )

navicatGA.wrappers_selfies.sc2mol_structure ( chromosome, lot=0 )
    Generates a rdkit.mol object with 3D coordinates from a list of selfies characters.

navicatGA.wrappers_selfies.sc2mv ( chromosome )

navicatGA.wrappers_selfies.sc2mw ( chromosome )

navicatGA.wrappers_selfies.sc2mwilogp ( chromosome )

navicatGA.wrappers_selfies.sc2nmw ( chromosome )

navicatGA.wrappers_selfies.sc2selfies ( chromosome )
    Generate a selfies string from a list of selfies characters.

navicatGA.wrappers_selfies.sc2smiles ( chromosome )
    Generate a canonical smiles string from a list of selfies characters.

navicatGA.wrappers_selfies.sc2tanimoto_to_target ( chromosome, target_selfie )
```

navicatGA.wrappers_smiles module

`navicatGA.wrappers_smiles.check_smiles_chars (chromosome)`

Checks if a chromosome corresponds to a proper SMILES.

`navicatGA.wrappers_smiles.chromosome_to_smiles ()`

Wrapper function for simplicity.

`navicatGA.wrappers_smiles.mol_structure2cm (mol)`

`navicatGA.wrappers_smiles.mol_structure2depictions (mol_structure, root_name='output')`

Generate 2D and 3D depictions from an rdkit.mol object with 3D coordinates.

`navicatGA.wrappers_smiles.sc2cm (chromosome, order='C')`

`navicatGA.wrappers_smiles.sc2depictions (chromosome, root_name='output', lot=0)`

Generate 2D and 3D depictions from a chromosome.

`navicatGA.wrappers_smiles.sc2ilogp (chromosome)`

`navicatGA.wrappers_smiles.sc2logp (chromosome)`

`navicatGA.wrappers_smiles.sc2mol_structure (chromosome, lot=0)`

Generates a rdkit.mol object with 3D coordinates from a list of selfies characters.

`navicatGA.wrappers_smiles.sc2mv (chromosome)`

`navicatGA.wrappers_smiles.sc2mw (chromosome)`

`navicatGA.wrappers_smiles.sc2mwilogp (chromosome)`

`navicatGA.wrappers_smiles.sc2nmw (chromosome)`

`navicatGA.wrappers_smiles.sc2smiles (chromosome)`

Generate a SMILES string from a list of SMILES characters. To be customized.

`navicatGA.wrappers_smiles.smiles2mol_structure (smiles, lot=0)`

Generates a rdkit.mol object with 3D coordinates from a SMILES.

navicatGA.wrappers_xyz module

class navicatGA.wrappers_xyz.**Hashable_Geometry** (**args, **kw*)

Bases: AaronTools.geometry.Geometry

This is a modified version of the Geometry class in AaronTools.py, which is identical but hashable.

navicatGA.wrappers_xyz.**chromosome_to_xyz** ()

Wrapper function for simplicity.

navicatGA.wrappers_xyz.**geom2angle** (*geom, a1, a2, a3*)

navicatGA.wrappers_xyz.**geom2bond** (*geom, a1, a2*)

navicatGA.wrappers_xyz.**geom2dihedral** (*geom, a1, a2, a3, a4*)

navicatGA.wrappers_xyz.**geom2sub_sterimol** (*geom, pos, parameter*)

navicatGA.wrappers_xyz.**g12geom** (*chromosome, h_positions='19-20'*)

Check if a chromosome (list of geometries) can lead to a valid structure.

navicatGA.xyz_solver module

```
class navicatGA.xyz_solver.XYZGenAlgSolver ( starting_population: list = [[None]], start-
ing_random: bool = False, alphabet_list: list = [], chromosome_to_xyz=<function random_merge_xyz.<local-
s>.merge>, multi_alphabet: bool = False, equivalences: Optional[Sequence] = None, max_counter: int = 10,
n_genes: int = 1, fitness_function=None, max_gen: int = 15, max_conv: int = 100, pop_size: int = 5, muta-
tion_rate: float = 0.1, selection_rate: float = 0.25, selection_strategy: str = 'tournament', excluded_genes:
Optional[Sequence] = None, n_crossover_points: int = 1, random_state: Optional[int] = None, lru_cache: bool
= False, scalarizer=None, prune_duplicates=False, verbose: bool = True, show_stats: bool = False, plot_results:
bool = False, to_stdout: bool = True, to_file: bool = True, logger_file: str = 'output.log', logger_level: str =
'INFO', progressBars: bool = False, problem_type: str = 'xyz' )
```

Bases: `navicatGA.base_solver.GenAlgSolver`

chromosomize (str_list)

Pad or truncate starting_population chromosome to build a population chromosome.

create_offspring (first_parent, sec_parent, crossover_pt, offspring_number)

Creates an offspring from 2 parents.

get_crossover_points ()

Retrieves random crossover points :return: a numpy array with the crossover points

initialize_population ()

Initializes the population of the problem according to the population size and number of genes and according to the problem type (XYZ fragments here). :return: a numpy array with a sanitized initialized population

mutate_population (population, n_mutations)

Mutates the population by randomizing specific positions of the population individuals.
:param population: the population at a given iteration :param n_mutations: number of muta-
tions to be performed. :return: the mutated population

refill_population (nrefill=0)

write_population (basename='chromosome')

Print xyz for all the population at the current state.

-
- genindex
 - modindex
 - search
-

n

navicatGA, 9

- navicatGA.base_solver, 11
- navicatGA.cache, 13
- navicatGA.chemistry_selfies, 15
- navicatGA.chemistry_smiles, 17
- navicatGA.chemistry_xyz, 19
- navicatGA.exception_messages, 21
- navicatGA.exceptions, 23
- navicatGA.fitness_functions_float, 25
- navicatGA.helpers, 33
- navicatGA.logger, 35
- navicatGA.progressBars, 37
- navicatGA.score_modifiers, 45
- navicatGA.selfies_solver, 47
- navicatGA.smiles_solver, 49
- navicatGA.timeout, 51
- navicatGA.wrappers_selfies, 53
- navicatGA.wrappers_smiles, 55
- navicatGA.wrappers_xyz, 57
- navicatGA.xyz_solver, 59

A

AbsoluteModifier() (in module navicatGA.s-
core_modifiers), 45
addLoggingLevel() (in module navicatGA.log-
ger), 35

C

calculate_fitness() (navicatGA.base_solver.-
GenAlgSolver method), 11
calculate_fitness_cache() (in module navicat-
GA.cache), 13
calculate_one_fitness_cache() (in module navi-
catGA.cache), 13
check_error() (in module navicatGA.helpers),
33
check_input_base() (navicatGA.base_solver.-
GenAlgSolver method), 11
check_selfie_chars() (in module navicatGA.-
chemistry_selfies), 15
check_smiles_chars() (in module navicat-
GA.wrappers_smiles), 55
check_xyz() (in module navicatGA.chem-
istry_xyz), 19
chromosome_to_smiles() (in module navicat-
GA.wrappers_smiles), 55
chromosome_to_xyz() (in module navicat-
GA.wrappers_xyz), 57
chromosomize() (navicatGA.selfies_solver.Self-
iesGenAlgSolver method), 47
chromosomize() (navicatGA.smiles_solver.S-
milesGenAlgSolver method), 49
chromosomize() (navicatGA.xyz_solver.XYZ-
GenAlgSolver method), 59
close_logger() (in module navicatGA.logger), 35
close_solve_logger() (navicatGA.base_solver.-
GenAlgSolver method), 11
concatenate_list() (in module navicat-
GA.helpers), 33
configure_logger() (in module navicatGA.log-
ger), 35
count_selfie_chars() (in module navicatGA.-

chemistry_selfies), 15

create_offspring() (navicatGA.base_solver.-
GenAlgSolver static method), 11
create_offspring() (navicatGA.selfies_solver.-
SelfiesGenAlgSolver method), 47
create_offspring() (navicatGA.smiles_solver.S-
milesGenAlgSolver method), 49
create_offspring() (navicatGA.xyz_solver.XYZ-
GenAlgSolver method), 59

D

decode_selfies_list() (in module navicatGA.-
chemistry_selfies), 15
diagnose_mol() (in module navicatGA.chem-
istry_selfies), 15
diagnose_mol() (in module navicatGA.chem-
istry_smiles), 17
draw_selfies() (in module navicatGA.chem-
istry_selfies), 15
draw_smiles() (in module navicatGA.chem-
istry_smiles), 17
draw_xyz() (in module navicatGA.chem-
istry_xyz), 19

E

encode_smiles_list() (in module navicatGA.-
chemistry_selfies), 15

F

fitness_function_float() (in module navicatGA.-
fitness_functions_float), 25

G

GaussianModifier() (in module navicatGA.s-
core_modifiers), 45
GenAlgSolver (class in navicatGA.base_solver),
11
geom2angle() (in module navicatGA.wrapper-
s_xyz), 57
geom2bond() (in module navicatGA.wrapper-
s_xyz), 57

geom2dihedral() (in module navicatGA.wrappers_xyz), 57
 geom2sub_sterimol() (in module navicatGA.wrappers_xyz), 57
 get_alphabet_from_path() (in module navicatGA.chemistry_xyz), 19
 get_boltzmann_probabilities() (navicatGA.base_solver.GenAlgSolver method), 11
 get_conformer_rmsd() (in module navicatGA.chemistry_selfies), 15
 get_conformer_rmsd() (in module navicatGA.chemistry_smiles), 17
 get_confs_ff() (in module navicatGA.chemistry_selfies), 15
 get_confs_ff() (in module navicatGA.chemistry_smiles), 17
 get_crossover_points() (navicatGA.base_solver.GenAlgSolver method), 11
 get_crossover_points() (navicatGA.selfies_solver.SelfiesGenAlgSolver method), 47
 get_crossover_points() (navicatGA.smiles_solver.SmilesGenAlgSolver method), 49
 get_crossover_points() (navicatGA.xyz_solver.XYZGenAlgSolver method), 59
 get_default_alphabet() (in module navicatGA.chemistry_xyz), 19
 get_ECFP4() (in module navicatGA.chemistry_selfies), 15
 get_ECFP4() (in module navicatGA.chemistry_smiles), 17
 get_elapsed_time() (in module navicatGA.helpers), 33
 get_input_dimensions() (in module navicatGA.helpers), 33
 get_interatomic_distances() (in module navicatGA.chemistry_selfies), 15
 get_interatomic_distances() (in module navicatGA.chemistry_smiles), 17
 get_number_mutations() (navicatGA.base_solver.GenAlgSolver method), 11
 get_selection_probabilities() (navicatGA.base_solver.GenAlgSolver method), 11
 get_selfie_chars() (in module navicatGA.chemistry_selfies), 15
 get_starting_xyz_from_file() (in module navicatGA.chemistry_xyz), 19
 get_starting_xyz_from_path() (in module navicatGA.chemistry_xyz), 19
 get_starting_xyz_from_smi() (in module navicatGA.chemistry_xyz), 19
 get_structure_ff() (in module navicatGA.chemistry_selfies), 15
 get_structure_ff() (in module navicatGA.chemistry_smiles), 17
 gl2geom() (in module navicatGA.wrapper-

s_xyz), 57

H

handle_timeout() (navicatGA.timeout.timeout method), 51
 handler() (in module navicatGA.timeout), 51
 has_transition_metals() (in module navicatGA.chemistry_selfies), 15
 has_transition_metals() (in module navicatGA.chemistry_smiles), 17
 Hashable_Geometry (class in navicatGA.chemistry_xyz), 19
 Hashable_Geometry (class in navicatGA.wrappers_xyz), 57

I

initialize_population() (navicatGA.base_solver.GenAlgSolver method), 12
 initialize_population() (navicatGA.selfies_solver.SelfiesGenAlgSolver method), 47
 initialize_population() (navicatGA.smiles_solver.SmilesGenAlgSolver method), 49
 initialize_population() (navicatGA.xyz_solver.XYZGenAlgSolver method), 59
 interval_selection() (navicatGA.base_solver.GenAlgSolver method), 12
 InvalidInput, 23
 is_transition_metal() (in module navicatGA.chemistry_selfies), 16
 is_transition_metal() (in module navicatGA.chemistry_smiles), 17

M

make_array() (in module navicatGA.helpers), 33
 module
 navicatGA, 9
 navicatGA.base_solver, 11
 navicatGA.cache, 13
 navicatGA.chemistry_selfies, 15
 navicatGA.chemistry_smiles, 17
 navicatGA.chemistry_xyz, 19
 navicatGA.exception_messages, 21
 navicatGA.exceptions, 23
 navicatGA.fitness_functions_float, 25
 navicatGA.helpers, 33
 navicatGA.logger, 35
 navicatGA.progressBars, 37
 navicatGA.score_modifiers, 45
 navicatGA.selfies_solver, 47
 navicatGA.smiles_solver, 49
 navicatGA.timeout, 51
 navicatGA.wrappers_selfies, 53
 navicatGA.wrappers_smiles, 55
 navicatGA.wrappers_xyz, 57

navicatGA.xyz_solver, 59
 mol_structure2cm() (in module navicatGA.wrappers_selfies), 53
 mol_structure2cm() (in module navicatGA.wrappers_smiles), 55
 mol_structure2depictions() (in module navicatGA.wrappers_selfies), 53
 mol_structure2depictions() (in module navicatGA.wrappers_smiles), 55
 mutate_population() (navicatGA.base_solver.GenAlgSolver method), 12
 mutate_population() (navicatGA.selfies_solver.SelfiesGenAlgSolver method), 47
 mutate_population() (navicatGA.smiles_solver.SmilesGenAlgSolver method), 49
 mutate_population() (navicatGA.xyz_solver.XYZGenAlgSolver method), 59

N

navicatGA
 module, 9
 navicatGA.base_solver
 module, 11
 navicatGA.cache
 module, 13
 navicatGA.chemistry_selfies
 module, 15
 navicatGA.chemistry_smiles
 module, 17
 navicatGA.chemistry_xyz
 module, 19
 navicatGA.exception_messages
 module, 21
 navicatGA.exceptions
 module, 23
 navicatGA.fitness_functions_float
 module, 25
 navicatGA.helpers
 module, 33
 navicatGA.logger
 module, 35
 navicatGA.progressBars
 module, 37
 navicatGA.score_modifiers
 module, 45
 navicatGA.selfies_solver
 module, 47
 navicatGA.smiles_solver
 module, 49
 navicatGA.timeout
 module, 51
 navicatGA.wrappers_selfies
 module, 53
 navicatGA.wrappers_smiles
 module, 55
 navicatGA.wrappers_xyz

 module, 57
 navicatGA.xyz_solver
 module, 59
 NoAssembler, 23
 NoFitnessFunction, 23

P

pad_xyz_list() (in module navicatGA.chemistry_xyz), 19
 plot_fitness_results() (navicatGA.base_solver.GenAlgSolver static method), 12
 print_stats() (navicatGA.base_solver.GenAlgSolver method), 12
 prune_mol_conformers() (in module navicatGA.chemistry_selfies), 16
 prune_mol_conformers() (in module navicatGA.chemistry_smiles), 17

R

random_merge_xyz() (in module navicatGA.chemistry_xyz), 19
 randomize_selfies() (in module navicatGA.chemistry_selfies), 16
 randomize_smiles() (in module navicatGA.chemistry_selfies), 16
 randomize_smiles() (in module navicatGA.chemistry_smiles), 17
 refill_population() (navicatGA.selfies_solver.SelfiesGenAlgSolver method), 47
 refill_population() (navicatGA.smiles_solver.SmilesGenAlgSolver method), 49
 refill_population() (navicatGA.xyz_solver.XYZGenAlgSolver method), 59

S

sanitize_multiple_smiles() (in module navicatGA.chemistry_selfies), 16
 sanitize_multiple_smiles() (in module navicatGA.chemistry_smiles), 17
 sanitize_smiles() (in module navicatGA.chemistry_selfies), 16
 sanitize_smiles() (in module navicatGA.chemistry_smiles), 17
 sc2cm() (in module navicatGA.wrappers_selfies), 53
 sc2cm() (in module navicatGA.wrappers_smiles), 55
 sc2depictions() (in module navicatGA.wrappers_selfies), 53
 sc2depictions() (in module navicatGA.wrappers_smiles), 55
 sc2ilogp() (in module navicatGA.wrappers_selfies), 53
 sc2ilogp() (in module navicatGA.wrappers_smiles), 55
 sc2krr() (in module navicatGA.wrappers_selfies), 53

[sc2levenshtein_to_target\(\)](#) (in module `navicatGA.wrappers_selfies`), [53](#)
[sc2logp\(\)](#) (in module `navicatGA.wrappers_selfies`), [53](#)
[sc2logp\(\)](#) (in module `navicatGA.wrappers_smiles`), [55](#)
[sc2mol_structure\(\)](#) (in module `navicatGA.wrappers_selfies`), [53](#)
[sc2mol_structure\(\)](#) (in module `navicatGA.wrappers_smiles`), [55](#)
[sc2mv\(\)](#) (in module `navicatGA.wrappers_selfies`), [53](#)
[sc2mv\(\)](#) (in module `navicatGA.wrappers_smiles`), [55](#)
[sc2mw\(\)](#) (in module `navicatGA.wrappers_selfies`), [53](#)
[sc2mw\(\)](#) (in module `navicatGA.wrappers_smiles`), [55](#)
[sc2mwilogp\(\)](#) (in module `navicatGA.wrappers_selfies`), [53](#)
[sc2mwilogp\(\)](#) (in module `navicatGA.wrappers_smiles`), [55](#)
[sc2nmw\(\)](#) (in module `navicatGA.wrappers_selfies`), [53](#)
[sc2nmw\(\)](#) (in module `navicatGA.wrappers_smiles`), [55](#)
[sc2selfies\(\)](#) (in module `navicatGA.wrappers_selfies`), [53](#)
[sc2smiles\(\)](#) (in module `navicatGA.wrappers_selfies`), [53](#)
[sc2smiles\(\)](#) (in module `navicatGA.wrappers_smiles`), [55](#)
[sc2tanimoto_to_target\(\)](#) (in module `navicatGA.wrappers_selfies`), [53](#)
[score_modifier\(\)](#) (in module `navicatGA.score_modifiers`), [45](#)
[select_parents\(\)](#) (`navicatGA.base_solver.GenAlgSolver` method), [12](#)
[SelfiesGenAlgSolver](#) (class in `navicatGA.selfies_solver`), [47](#)
[set_dative_bonds\(\)](#) (in module `navicatGA.chemistry_selfies`), [16](#)
[set_dative_bonds\(\)](#) (in module `navicatGA.chemistry_smiles`), [18](#)
[set_lru_cache\(\)](#) (in module `navicatGA.cache`), [13](#)
[set_progressBars\(\)](#) (in module `navicatGA.progressBars`), [37](#)
[smiles2mol_structure\(\)](#) (in module `navicatGA.wrappers_smiles`), [55](#)
[SmilesGenAlgSolver](#) (class in `navicatGA.smiles_solver`), [49](#)
[solve\(\)](#) (`navicatGA.base_solver.GenAlgSolver` method), [12](#)
[solve_progress\(\)](#) (in module `navicatGA.progressBars`), [37](#)
[sort_by_fitness\(\)](#) (`navicatGA.base_solver.GenAlgSolver` static method), [12](#)

[SquaredModifier\(\)](#) (in module `navicatGA.score_modifiers`), [45](#)

T

[test_benzene_selfies\(\)](#) (in module `navicatGA.selfies_solver`), [47](#)
[timed_decoder\(\)](#) (in module `navicatGA.chemistry_selfies`), [16](#)
[timed_sanitizer\(\)](#) (in module `navicatGA.chemistry_selfies`), [16](#)
[timed_sanitizer\(\)](#) (in module `navicatGA.chemistry_smiles`), [18](#)
[timeout](#) (class in `navicatGA.timeout`), [51](#)
[timer_alarm\(\)](#) (in module `navicatGA.timeout`), [51](#)
[tournament_selection\(\)](#) (`navicatGA.base_solver.GenAlgSolver` method), [12](#)
[tournament_selection_helper\(\)](#) (`navicatGA.base_solver.GenAlgSolver` static method), [12](#)

W

[write_population\(\)](#) (`navicatGA.selfies_solver.SelfiesGenAlgSolver` method), [47](#)
[write_population\(\)](#) (`navicatGA.smiles_solver.SmilesGenAlgSolver` method), [49](#)
[write_population\(\)](#) (`navicatGA.xyz_solver.XYZGenAlgSolver` method), [59](#)

X

[XYZGenAlgSolver](#) (class in `navicatGA.xyz_solver`), [59](#)