GASTOp Documentation

Rory Conlin, Paul Kaneelil, Cristian Lacey, Susan Redmond, Dan Shaw, Amlan Sinha

Created: Tuesday 15th January, 2019

Table of contents

1 Quickstart **2**
- 1.1 Installation . . . 2
- 1.2 Usage . . . 2
- 1.3 Contribute . . . 2
- 1.4 License . . . 2

2 Installation **3**
- 2.1 From PyPI . . . 3
- 2.2 From GitHub . . . 3

3 Usage **4**
- 3.1 Command Line . . . 4
- 3.2 Python Package . . . 4

4 Config File Formatting and Options **6**
- 4.1 Required Parameters . . . 6
  - 4.1.1 General Parameters . . . 6
  - 4.1.2 Fitness Function Parameters . . . 7
  - 4.1.3 Evaluator Parameters . . . 7
  - 4.1.4 Genetic Algorithm Parameters . . . 7
  - 4.1.5 Progress Monitor Parameters . . . 8
- 4.2 Optional Parameters . . . 8
  - 4.2.1 Random Generation Parameters . . . 8
  - 4.2.2 Crossover Parameters . . . 8
  - 4.2.3 Mutator Parameters . . . 8
  - 4.2.4 Selector Parameters . . . 9
- 4.3 Properties Parsing . . . 9

5 API Documentation **10**
- 5.1 Crossover . . . 10
- 5.2 Evaluator . . . 11
- 5.3 FitnessFunction . . . 13
- 5.4 GenAlg . . . 15
- 5.5 Mutator . . . 16
- 5.6 Progress Monitor . . . 18
- 5.7 Selector . . . 19
- 5.8 Truss . . . 20
- 5.9 encoders . . . 22
- 5.10 utilities . . . 23

6 Examples **25**
6.1 Pyramid Example Configuration File 25
6.2 Pyramid Example Results 26
6.3 Cantilever Example Results 27

Chapter 1

Quickstart

GASTOp is a **G**enetic **A**lgorithm for **S**tructural design and **T**opological **Op**timization. Given a set of boundary conditions such as applied loads and fixtures, it will design a structure to support those loads while minimizing weight and deflections and maximize factor of safety.

1.1 Installation

Install gastop by running:

```
$ pip install gastop
```

1.2 Usage

Look how easy it is to use:

```
import gastop
config_file_path = "./path_to_config_file.txt"
ga = gastop.GenAlg(config_file_path)
ga.initialize_population(pop_size=1e4)
best_truss, history = ga.run(num_generations=100, progress_display=1, num_threads=4)
```

1.3 Contribute

- Issue Tracker: https://github.com/f0uriest/GASTOp/issues
- Source Code: https://github.com/f0uriest/GASTOp/
- Documentation: https://gastop.readthedocs.io/

1.4 License

The project is licensed under the GNU GPLv3 license.

Chapter 2

Installation

gastop can either be install from PyPI, or directly by cloning the git repository and installing manually.

2.1 From PyPI

The easiest way to install gastop is to use pip to install from PyPI:

```
$ pip install gastop
```

This will install the base package, and shortcuts to use gastop from the command line. However, this will not install additional components such as the test suite and sample config files.

2.2 From GitHub

gastop can also be built from source by cloning the git repository.

```
$ git clone https://github.com/f0uriest/GASTOp.git
```

Once cloned, it can be installed by running

```
$ python setup.py install
```

from within the repository home folder. This will install the base package and command line shortcut. The git repository also contains the test suite and sample config files. The test suite can be run from the main folder with

```
$ python -m pytest
```

Please note that the tests may take several minutes to run.

Chapter 3

Usage

gastop can either be run from the command line, or used in a python script.

3.1 Command Line

If it was installed from PyPI using pip, it can be run from the command line as:

```
$ gastop <config_file_path>
```

If the code was cloned from github instead, the normal commandline shortcut will not be installed. It can still be run from the repository main directory as:

```
$ python -m gastop <config_file_path>
```

In either case, additional arguments can be passed via the command line:

```
usage: gastop [-h] [-p] [-g] [-t] [-q | -d] config_path

positional arguments:
config_path          file path to gastop config file

optional arguments:
-h, --help           show this help message and exit
-p , --pop_size      population size. If not specified, defaults to what is
                     in config.
-g , --num_gens      number of generations. If not specified, defaults to
                     what is in config.
-t , --num_threads   number of threads to use. If not specified, defaults to
                     what is in config.
-q, --quiet          hide progress display window
-d, --display        show progress display window
```

3.2 Python Package

gastop can also be used from within python, either in a script or interactively.

```
import gastop
config_file_path = "./path_to_config_file.txt"
ga = gastop.GenAlg(config_file_path)
ga.initialize_population(pop_size=1e4)
best_truss, history = ga.run(num_generations=100, progress_display=1, num_threads=4)
```

For a full description of available commands and options, see the *API Documentation*

Chapter 4

Config File Formatting and Options

The config file includes all the input parameters used to instantiate a GenAlg() object. Certain parameters must be specified by the user, while other more advanced parameters can be left blank for simplicity and will default to reasonable values.

The config file is parsed as a nested dictionary. Each dictionary is indicated by `[dict]`, and nested dictionaries are indicated by nested squared brackets, `[[nested dict]]`. Each dictionary contains multiple arguments indicated by `key: value`. If the value is an integer, float, or string, simply input the value without quotation marks. For instance, `key: 3`, `key: 3.14`, or `key: pi`. If the value is a numpy array, input the value as an array in list format, within single quotes, like `key: '[[3.14 3.14],[3.14 3.14]]'`.

For instance, the config file:

```
[dict1]
key1: 3
key2: 3.14
[[sub_dict1]]
sub_key1: apples
[[sub_dict2]]
sub_key2: None
[dict2]
key3: '[[3.14 12.8],[6.7 88.9999]]'
```

would be parsed, forming the dictionary:

```
{'dict1':
       {'key1':3,
       'key2':3.14,
       'sub_dict1':{'sub_key1':'apples'},
       'sub_dict2':{'sub_key2':None}},
 'dict2':
       {'key3':array([[3.14 12.8],[6.7 88.9999]])}}
```

4.1 Required Parameters

4.1.1 General Parameters

`[general]` contains the following parameters:

user_spec_nodes (nx3 numpy array of floats) User-specified nodes (nodes with provided loads and displacement boundary conditions) in the format `'[[x1 y1 z1],[x2 y2 z2],...,[xn yn zn]]'`.

loads (nx6 numpy array of floats) The forces and moments acting on each user-specified node in the format `'[[Fx1,Fy1,Fz1,Mx1,My1,Mz1][Fx2,Fy2,Fz2,Mx2,My2,Mz2],...,[Fxn,Fyn,Fzn,Mxn,Myn,Mzn]]'`.

fixtures (nx6 numpy array of ints) The translational and rotational displacements for each user-specified node in the format `'[[tx1,ty1,tz1,rx1,ry1,rz1],[tx2,ty2,tz2,rx2,ry2,rz2],...,[txn,tyn,tzn,rxn,ryn,rzn]]'`. Here `tx1` is the translational degree of freedom in the x direction of the first user-specified node, and `rx1` is the rotational degree of freedom about the x-axis of the first user-specified node. A `1` indicates fixed, while a `0` indicates the node is free to move along or about the corresponding degree of freedom.

num_rand_nodes (int) Maximum number of random nodes.

num_rand_edges (int) Maximum number of random edges.

properties_path (str) Path to the properties CSV, relative to the location of the config file. For example, `../properties.csv`.

domain (3x2 numpy array of floats) Allowable domain in the format `'[[xmin xmax],[ymin ymax],[zmin zmax]]'`.

4.1.2 Fitness Function Parameters

`[fitness_params]` contains the following parameters (see fitness_function):

equation (str) Method for calculating fitness. *Options: weighted_sum, sphere, rosenbrock, rastrigin.*

parameters (dict) Additional fitness function parameters.

parameters['goal_fos'] (int) Desired factor of safety.

parameters['critical_nodes'] (1xn numpy array of ints) Array of nodes numbers for which deflection should be minimized. If empty, defaults to all.

parameters['w_fos'] (float) Penalty weight for low fos. Only applied if truss.fos $<$ *goal_fos*.

parameters['w_mass'] (float) Penalty applied to mass. Relative magnitude of *w_mass* and *w_fos* determines importance of minimizing mass vs maximizing fos.

parameters['w_deflection'] (float) Penalty applied to deflections. If scalar, applies the same penalty to all critical nodes. Can also be an array the same size as *critical_nodes* in which case different penalties will be applied to each node.

4.1.3 Evaluator Parameters

`[evaluator_params]` contains the following parameters (see evaluator):

struct_solver (str) Method for solving truss. *Options: mat_struct_analysis_DSM Default: mat_struct_analysis_DSM*

mass_solver (str) Method of calculating the mass of a truss. *Options: mass_basic Default: mass_basic*

interferences_solver (str) Method of determining interferences. *Options: blank_test, interference_ray_tracing Default: blank_test*

cost_solver (str) Method of calculating the cost of a truss. *Options: cost_calc Default: cost_calc*

4.1.4 Genetic Algorithm Parameters

`[ga_params]` contains the following parameters (see gen_alg):

num_threads (int) Number of threads. If equal to one, the GenAlg.run() method will execute in serial. If greater than one, it will run in parallel.

pop_size (int) Number of trusses in each generation.

num_generations (int) Number of generations to run.

num_elite (int) Number of fittest trusses to carry over to the next generation without modification.
percent_mutation (float) Percent of trusses in the next generation (after subtracting elites) to be derived from mutation of current trusses.
percent_crossover (float) Percent of trusses in the next generation (after subtracting elites) to be derived from crossover of current trusses.
save_frequency (int) Number of generations after which the population and config are saved to .json files.
save_filename_prefix (str) Prefix for the save filenames. For example, `save_`.

4.1.5 Progress Monitor Parameters

`[monitor_params]` contains the following parameters (see progress_monitor):

progress_fitness (bool) Progress monitor display mode, if true displays best fitness score of the population each generation.
progress_truss (bool) Progress monitor display mode, if true displays the truss with the best fitness score each generation.

4.2 Optional Parameters

4.2.1 Random Generation Parameters

`[random_params]` contains the following parameters:

4.2.2 Crossover Parameters

`[crossover_params]` contains the following parameters (see crossover):

node_crossover_method (str) Method for performing node crossover. *Options: uniform_crossover, single_point_split, two_points_split Default: uniform_crossover*
edge_crossover_method (str) Method for performing edge crossover. *Options: uniform_crossover, single_point_split, two_points_split Default: uniform_crossover*
property_crossover_method (str) Method for performing edge crossover. *Options: uniform_crossover, single_point_split, two_points_split Default: uniform_crossover*
node_crossover_params (dict) Additional node crossover parameters.
edge_crossover_params (dict) Additional edge crossover parameters.
property_crossover_params (dict) Additional property crossover parameters.

4.2.3 Mutator Parameters

`[mutator_params]` contains the following parameters (see mutator):

node_mutator_method (str) Method for performing node mutation. *Options: gaussian, pseudo_bit_flip, shuffle_index Default: gaussian*
edge_mutator_method (str) Method for performing edge mutation. *Options: gaussian, pseudo_bit_flip, shuffle_index Default: pseudo_bit_flip*
property_mutator_method (str) Method for performing property mutation. *Options: gaussian, pseudo_bit_flip, shuffle_index Default: pseudo_bit_flip*
node_mutator_params (dict) Additional node mutator parameters.
node_mutator_params['std'] (float) Standard deviation for mutation. If array-like, std[i] is used as the standard deviation for array[:,i].

edge_mutator_params (dict) Additional edge mutator parameters.
edge_mutator_params['proportions'] (float) Probability of a given entry being mutated.
property_mutator_params (dict) Additional property mutator parameters.
property_mutator_params['proportions'] (float) Probability of a given entry being mutated.

4.2.4 Selector Parameters

`[selector_params]` contains the following parameters (see selector):

method (str) Method for performing selection. *Options: inverse_square_rank_probability, tournament Default: inverse_square_rank_probability*
tourn_size (int) The number of truss indices in each tournament. Must be less than 32.
tourn_prob (float) The probability of the fittest truss in a tournament to be selected.

4.3 Properties Parsing

While parsing the config file, GASTOp will read the path to a file that contains the user-specified property information from a CSV file. The file exists by default as `properties.csv` with a few available material options:

beam	material	OD (m)	ID (m)	elastic_modulus (Pa)	yield_strength (Pa)	dens (kg/m^3)	poisson_ratio	cost ($/m)
0	steel	0.025	0.02	200000000000	250000000	8050	0.3	1
1	steel	0.012	0.01	200000000000	250000000	8050	0.3	0.75
2	aluminum	0.025	0.02	69000000000	95000000	2700	0.32	2
3	aluminum	0.012	0.01	69000000000	95000000	2700	0.32	1.5
4	2024 aluminum	0.042	0.032	69000000000	276000000	2700	0.32	3

Adding additional materials is as simple as adding a row to the default file, with all values separated by commas. One could also alternatively create a new properties file, duplicating the format of the default, replacing all material data, and specifying the path to the new properties file in the config file.

Chapter 5

API Documentation

5.1 Crossover

class gastop.crossover.Crossover(*crossover_params*)

Mixes attributes belonging to two different parents to produce two children with specific characteristics from both parents.

When creating a new Crossover() object, it must be initialized with dictionary crossover_params (containing crossover method). The Crossover() object can then be used as a function that produces children according to the specified crossover method, such as uniform_crossover, single_point_split or two_points_split.

__call__(*truss_1*, *truss_2*)

Calls a crossover object on two trusses to combine them.

Crossover object must have been instantiated specifying which methods to use.

Parameters
- truss_1 (*Truss object*) – First truss to be combined.
- truss_2 (*Truss object*) – Second truss to be combined.

Returns *child_1, child_2 (Truss objects)* – Children trusses produced by crossover.

__init__(*crossover_params*)

Creates a Crossover object.

Once instantiated, the Crossover object can be called as a function to combine two trusses using the specified methods and parameters.

Parameters crossover_params (*dict*) – Dictionary containing:
- 'node_crossover_method' *(str)*: Name of method to use for node crossover.
- 'edge_crossover_method' *(str)*: Name of method to use for edge crossover.
- 'property_crossover_method' *(str)*: Name of method to use for property crossover.
- 'user_spec_nodes' *(ndarray)*: Array of user specified nodes that should be passed on unaltered.

Returns Crossover callable object.

static single_point_split(*array_1*, *array_2*)

Performs a single point split crossover between two parents

The single split crossover method takes specific information from two parents and returns two children containing characteristics from both parents. In order to achieve this, it chooses a random point and splits the two parents into two different parts. Then it merges the first half of the first parent with the second half of the second parent and vice versa.

Parameters
- `array_1` (*ndarray*) – Numpy array containing information for parent 1.
- `array_2` (*ndarray*) – Numpy array containing information for parent 2.

Returns *child1, child2 (ndarrays)* – Numpy arrays containing information for children.

`static two_points_split(`*array_1*, *array_2*`)`

Takes specific values of two parents and return two children containing characteristics from both parents.

The two points split method chooses two random points and splits the two parents into three different parts. Then, it replaces the central part of the first parent with the central part of the second parent.

Parameters
- `array_1` (*ndarray*) – Numpy array containing information for parent 1.
- `array_2` (*ndarray*) – Numpy array containing information for parent 2.

Returns *child_1, child_2 (ndarrays)* – Numpy arrays containing information for children.

`static uniform_crossover(`*parent_1*, *parent_2*`)`

Performs a uniform crossover on the two parents

The uniform crossover method creates two child arrays by randomly mixing together information taken from two parent arrays. To do this, the uniform crossover method creates two arrays of ones and zeros -one being the complement of the other- with the same shape as the parent arrays. The first array is multiplied with parent1 and the complementary array is multiplied with parent2 before adding the results together to make child1. The exact opposite multiplication is done to make child2.

Parameters
- `parent_1` (*ndarray*) – Numpy array containing information for parent 1.
- `parent_2` (*ndarray*) – Numpy array containing information for parent 2.

Returns *child1, child2 (ndarrays)* – Numpy arrays containing information for children.

5.2 Evaluator

`class gastop.evaluator.Evaluator(`*struct_solver*, *mass_solver*, *interferences_solver*, *cost_solver*, *boundary_conditions*, *properties_dict*`)`

Implements various methods for scoring the truss in different areas.

Methods include calculations of mass, factor of safety, deflections, and interference with user specified areas.

The class is designed to be instantiated as an Evaluator object which will fully evaluate a Truss object using specified methods and parameters.

`__call__(`*truss*`)`

Computes mass, deflections, etc, and stores it in truss object.

Used when an Evaluator object has been created with the methods to be used and any necessary parameters.

Parameters `truss` (*Truss object*) – truss to be evaluated.
Returns None

`__init__(`*struct_solver*, *mass_solver*, *interferences_solver*, *cost_solver*, *boundary_conditions*, *properties_dict*`)`

Creates an Evaluator callable object.

Once created, the Evaluator can be called on a Truss object to calculate and assign mass, factor of safety, deflections, etc to the truss.

Parameters
- `struct_solver` (*str*) – Name of the method to be used for structural analysis and calculating fos and deflections, as a string. e.g. `'mat_struct_analysis_DSM'`.

- `mass_solver` (*str*) – Name of the method to be used to calculate mass. e.g. `'mass_basic'`.
- `interferences_solver` (*str*) – Name of method to be used to determine interferences. e.g. `'interferences_ray_tracing'`.
- `boundary_conditions` (*dict*) – Dictionary containing:
  - `'loads'` *(ndarray)*: Array of loads applied to the structure. First index corresponds to the node where the load is applied, second index is the force in x,y,z and moment about x,y,z, third index is for multiple loading scenarios.
  - `'fixtures'` *(ndarray)*: Array of flags denoting whether a node is fixed or free. First index corresponds to the node, the second index corresponds to fixing displacements in x,y,z and rotations about x,y,z. The third index corresponds to multiple loading scenarios with different fixtures for each. Values of the array are 0 (free) or 1 (fixed).
- `properties_dict` (*dict*) – Dictionary containing beam properties.
- `should be 1D arrays, with length equal to the number of` (*Entries*) –
- `options. Each entry in the array is the value of the key` (*beam*) –
- `for the specified beam type. Properties include` (*property*) –
  - `'OD'`: Outer diameter of the beam, in meters.
  - `'ID'`: Inner diameter of the beam, in meters.
  - `'elastic_modulus'`: Elastic or Young's modulus of the material, in Pascals.
  - `'yield_strength'`: Yield or failure strength of the material, in Pascals.
  - `'shear_modulus'`: Shear modulus of the material, in Pascals.
  - `'poisson_ratio'`: Poisson ratio of the material, dimensionless.
  - `'x_section_area'`: Cross sectional area of the beam, in square meters.
  - `'moment_inertia_y'`: Area moment of inertia about beams y axis, in meters^4.
  - `'moment_inertia_z'`: Area moment of inertia about beams z axis, in meters^4.
  - `'polar_moment_inertia'`: Area moment of inertia about beams polar axis, in meters^4.
  - `'dens'`: Density of the material, in kilograms per cubic meter.
- `cost_solver` (*str*) – Name of the method to be used to calculate cost. e.g. `'cost_calc'`.

Returns callable Evaluator object.

`static blank_test(`*truss*, **args*, ***kwargs*`)`

Blank function used for testing GA when no evaluation needed

Parameters `truss` (*Truss object*) – Dummy Truss object, no attributes required.
Returns 2-element tuple of (None, None)

`static cost_calc(`*truss*, *properties_ dict*`)`

Calculates cost of structure

Considers only members, does not account for additional cost due to welds or connection hardware.

Parameters `truss` – Truss to be evaluated. Must have nodes, edges, and properties defined.
Returns *cost (float)* – Cost of the structure in $.

`static interference_ray_tracing(`*truss*`)`

Not implemented yet.

TODO: method to determine if truss members are crossing into user specified areas. Used when a structure must be designed around something, such as a passenger compartment or other design components.

`static mass_basic(`*truss*, *properties_ dict*`)`

Calculates mass of structure

Considers only members, does not account for additional mass due to welds or connection hardware.

Parameters `truss` – Truss to be evaluated. Must have nodes, edges, and properties defined.
Returns *mass (float)* – Mass of the structure in kilograms.

`static mat_struct_analysis_DSM(`*truss*, *boundary_ conditions*, *properties_ dict*`)`

Calculates deflections and stresses using direct stiffness method.

Constructs global stiffness matrix from nodes and connections, and computes deflections under each loading scenario. From deflections, calculates internal forces, stresses, and factor of safety in each member under each loading scenario

Parameters

- `truss` (*`Truss object`*) – Truss to be evaluated. Must have nodes, edges, and properties defined.
- `boundary_conditions` (*`dict`*) – Dictionary containing:
  - `'loads'` *(ndarray)*: Array of loads applied to the structure. First index corresponds to the node where the load is applied, second index is the force in x,y,z and moment about x,y,z, third index is for multiple loading scenarios.
  - `'fixtures'` *(ndarray)*: Array of flags denoting whether a node is fixed or free. First index corresponds to the node, the second index corresponds to fixing displacements in x,y,z and rotations about x,y,z. The third index corresponds to multiple loading scenarios with different fixtures for each. Values of the array are 0 (free) or 1 (fixed).
- `properties_dict` (*`dict`*) – Dictionary containing beam properties.
- `should be 1D arrays, with length equal to the number of` (*`Entries`*) –
- `options. Each entry in the array is the value of the key` (*`beam`*) –
- `for the specified beam type. Properties include` (*`property`*) –
  - `'OD'`: Outer diameter of the beam, in meters.
  - `'ID'`: Inner diameter of the beam, in meters.
  - `'elastic_modulus'`: Elastic or Young's modulus of the material, in Pascals.
  - `'yield_strength'`: Yield or failure strength of the material, in Pascals.
  - `'shear_modulus'`: Shear modulus of the material, in Pascals.
  - `'poisson_ratio'`: Poisson ratio of the material, dimensionless.
  - `'x_section_area'`: Cross sectional area of the beam, in square meters.
  - `'moment_inertia_y'`: Area moment of inertia about beams y axis, in meters^4.
  - `'moment_inertia_z'`: Area moment of inertia about beams z axis, in meters^4.
  - `'polar_moment_inertia'`: Area moment of inertia about beams polar axis, in meters^4.
  - `'dens'`: Density of the material, in kilograms per cubic meter.

Returns

2-element tuple containing –

- **fos** *(ndarray)*: 2D array of factor of safety values. First index corresponds to members, second index corresponds to different loading scenarios. Factor of safety is defined as the materials yield strength divided by the von Mises stress in the member. If structure is statically indeterminate under a given loading scenario, fos will be zero.
  Factor of safety in member i under loading j is fos[i, j]
- **deflections** *(ndarray)*: 3D array of node deflections. Distances in meters, angles in radians. First index corresponds to node number, second index is deflections in global x,y,z coordinates, and rotations about global x,y,z axes. The third axis corresponds to different loading scenarios.
  Deflection at node i under loading j is deflections[i, :, j] = [dx, dy, dz, d_theta_x, d_theta_y, d_theta_z]

5.3 FitnessFunction

`class gastop.fitness.FitnessFunction(`*equation*, *parameters*`)`

Implements fitness functions for computing fitness scores.

The fitness function assigns a single value to each truss based on various parameters, so that comparisons between trusses can be made.

The class is designed to be instantiated as a FitnessFunction object which operates on Truss objects to assign a fitness score, though methods from the class may also be called directly.

`__call__`(*truss*)
Computes fitness score and stores it in truss object.

Used when a FitnessFunction object has been created with the method to be used and any necessary parameters.

Parameters `truss` (*Truss object*) – truss to be scored.
Returns None

`__init__`(*equation*, *parameters*)
Creates a FitnessFunction object

Once created, the object acts like a function and can be called on a Truss object to assign it a fitness score.

Parameters
- `equation` (*string*) – The name of the method to be used to compute fitness, as a string. eg, `'weighted_sum'` or `'rosenbrock'`.
- `parameters` (*dict*) – Dictionary of keyword parameter values for the method specified in *equation*.

Returns FitnessFunction callable object

`static rastrigin`(*truss*)
n-dimensional Restrigin function.

Global min at x=0 where f=0. This method is primarily supplied for testing of the genetic algorithm, and should not be used for structural design.

Parameters `truss` (*Truss object*) – only uses truss object as container for nodes. No other attributes needed.
Returns
float – fitness score. Computed as $f(x) = 10n + \Sigma_{i=1}^{n}(x_i^2 - 10\cos(2\pi x_i))$
n is determined from size of nodes array, x_i are entries of node array.

`static rosenbrock`(*truss*)
n-dimensional Rosenbrock function.

Sum of n/2 2D Rosenbrock functions. Global min at x=1 where f=0. This method is primarily supplied for testing of the genetic algorithm, and should not be used for structural design.

Parameters `truss` (*Truss object*) – only uses truss object as container for nodes. No other attributes needed.
Returns
float – Fitness score. Computed as $f(x) = \Sigma_{i=1}^{n/2}(100 * (x_{2i-1}^2 - x_{2i}^2)^2 + (x_{2i-1} - 1)^2)$
n is determined from size of nodes array, x_i are entries of node array.

`static sphere`(*truss*)
Sum of squares of node array elements. aka, sphere function.

Global min at x = 0 where f = 0. This method is primarily supplied for testing of the genetic algorithm, and should not be used for structural design.

Parameters `truss` (*Truss object*) – only uses truss object as container for nodes. No other attributes needed.
Returns
float – Fitness score. Computed as $f(x) = \Sigma_{i=1}^{n} x_i^2$
n is determined from size of nodes array, x_i are entries of node array.

`static weighted_sum`(*truss*, *goal_fos*, *critical_nodes*, *w_fos*, *w_mass*, *w_deflection*)
Computes fitness score using a weighted sum of parameters.

Parameters

- `truss` (*Truss object*) – truss to be scored. Must have *mass*, *fos*, and *deflections* attributes defined (for example, by using evaluator).
- `goal_fos` (*float >= 0*) – Desired factor of safety. Trusses with a smaller fos will be penalized according to *w_fos*.
- `critical_nodes` (*int, array*) – Array of nodes #s for which deflection should be minimized. If empty, defaults to all.
- `w_fos` (*float >= 0*) – Penalty weight for low fos. Only applied if truss.fos $<$ *goal_fos*.
- `w_mass` (*float >= 0*) – Penalty applied to mass. Relative magnitude of *w_mass* and *w_fos* determines importance of
  minimizing mass vs maximizing fos.
- `w_deflection` (*float >=0, array*) – Penalty applied to deflections. If scalar, applies the same penalty to all critical nodes. Can also be an array the same size as *critical_nodes* in which case different penalties will be applied to each node.

Returns

float – Fitness score. Computed as: $f = w_m m + w_{fos} \max(\text{fos}_{goal} - \text{fos}_{min}, 0) + w_{def}||\text{deflections}||_2$
m is the mass of the stucture, fos_{min} is the lowest fos for the structure under all load conditions. If $\text{fos}_{min} > \text{fos}_{goal}$, no fos penalty is applied, so f depends only on mass and deflections.

5.4 GenAlg

`class gastop.genalg.GenAlg(`*config*`)`

Creates, updates, tracks, loads, and saves populations.

The GenAlg Class orchestrates all of the other functions that perform functions to change the population and its elements. In this case, such classes are crossover, evaluator, encoders, fitness, mutator, selector, and truss.

In brief, GenAlg calls many other functions in order to create a generation which is then analyzed to fully determine its relavant properties. These properties are then used to create a new generation and the process repeats until a final solution is reached.

`__init__(`*config*`)`

Creates a GenAlg object

Once created, the object will store all of the relavant information about a population. The object also contains the necessary functions to modify itself, evaluate its 'goodness', and then create new members for the next generation.

Parameters

- `Either` –
- `config` (*str*) – Configuration dictionary with parameters, such as one created by *gastop.utilities.init_file_parser()*
- `config` – File path to config file to be parsed. Used instead of passing config dictionary directly.

Returns GenAlg callable object

`generate_random()`

Generates and returns new truss objects with random properties

The random method first determines the desired ranges of all values that will be calculated. Then, random numbers for the node locations, connections, and properties are all determined with the numpy.random methods.

Parameters `None` –
Returns *(Truss object)* – Truss object with the newly determined values

`initialize_population(pop_size=None)`
Initializes population with randomly creates Truss objects.

Population is stored in instance of GenAlg object as population attribute.

Parameters `pop_size` (*int*) – size of the population. If not specified, it defaults to what is in the config dict.
Returns None

`static load_state(dest_config='config.json', dest_pop='population.json')`
Loads the current population and config settings from JSON files.

Parameters
- `dest_config` (*string*) – Path to config data file.
- `dest_pop` (*string*) – Path to population data file.

Returns ga (GenAlg object)

`run(num_generations=None, progress_fitness=None, progress_truss=None, num_threads=None)`
Runs the genetic algorithm over all populations and generations

Parameters
- `num_generations` (*int*) – number of generations to be performed.
- `progress_fitness` (*bool*) – Whether to show display window showing fitness score vs generation.
- `progress_truss` (*bool*) – Whether to show display window showing truss evolution.
- `num_threads` (*int*) – number of threads the multiprocessing should employ. If zero or None, it will use the number returned by `os.cpu_count()`.

Returns
2-element tuple containing –
- **best** *(Truss)*: The Truss with the best fitness score after elapsed generations.
- **pop_progress** (dict): Dictionary of dictionaries containing:
  - `'Generation 1'` *(dict)*: Dictionary of info about generation 1.
  - `'Generation 2'` *(dict)*: Dictionary of info about generation 2, etc.

`save_state(dest_config='config.json', dest_pop='population.json')`
Saves the current population and config settings to JSON files.

Parameters
- `dest_config` (*string*) – Path to save config data file. If file doesn't exist, creates it.
- `dest_pop` (*string*) – Path to save population data file. If file doesn't exist, creates it.

Returns None

`update_population()`
Creates new population by performing crossover and mutation, as well as taking elites and randomly generating trusses.

First sorts the population by fitness score, from most fit to least fit. Creates selector object from population and method. Calls selector to get list of parents for crossover and mutation. Performs crossover and mutation.

Parameters `None` –
Returns None

5.5 Mutator

`class gastop.mutator.Mutator(mutator_params)`
Randomly mutates the whole/specific attributes belonging to the parents.

When creating a new Mutator() obejct, it must be initialized with dictionary mutator_params (containing mutation method). The Mutator() Object can then be used as a function that mutates parents according to the specified method, such as gaussian, pseudo_bit_flip and shuffle_index.

`__call__`(*truss*)
Calls a mutator object on a truss to change it.

Mutator object must have been instantiated specifying which methods to use.

Parameters `truss` (*Truss object*) – Truss to be mutated.
Returns *child (Truss object)* – Child truss produced by mutation.

`__init__`(*mutator_params*)
Creates a Mutator object.

Once instantiated, the Mutator object can be called as a function to alter the parent array using the specified methods and parameters

Parameters `mutator_params` (*dict*) – Dictionary containing:
- `'node_mutator_method'` *(str)*: Name of method to use for node mutation.
- `'edge_mutator_method'` *(str)*: Name of method to use for edge mutation.
- `'property_mutator_method'` *(str)*: Name of method to use for property mutation.
- `'node_mutator_params'` *(dict)*: Dictionary of parameters for node method.
- `'edge_mutator_params'` *(dict)*: Dictionary of parameters for edge method.
- `'property_mutator_params'` *(dict)*: Dictionary of parameters for property method.
- `'user_spec_nodes'` *(ndarray)*: Array of user specified nodes that should be passed on unaltered.

Returns Mutator callable object.

`static gaussian`(*array*, *std*, *boundaries*, *int_flag*)
Performs a gaussian mutation on the given parent array

The gaussian mutator method creates a child array by mutating the given parent array. The mutation is done by adding a random value from the gaussian distribution with a user specified standard deviation to each of the elements in the parent array. Since values need to be within a specified boundary, any elements that are mutated out of bounds on one side are looped inside the other side by the same amount, assuming a periodic boundary.

Parameters
- `array` (*ndarray*) – Numpy array containing the information for the parent array that is being mutated.
- `std` (*float or array-like*) – Standard deviation for mutation. If array-like, std[i] is used as the standard deviation for array[:,i].
- `boundaries` (*array-like*) – Domain of allowable values. If a value is mutated outside this region, it is looped back around to the other side.
- `int_flag` (*bool*) – flag specifying whether output should be ints.

Returns *new_array (ndarray)* – Numpy array containing information for the mutated child.

`static pseudo_bit_flip`(*parent*, *boundaries*, *proportions*, *int_flag*)
Mutate specific values of the parent and return the mutant child.

The pseudo_bit_flip method creates a random binary matrix with a fixed ratio of 1s and 0s as specified by the user. It also creates another random matrix with elements within the domain specified by the user. It then replaces the elements from the original matrix with the corresponding elements in the new matrix only if the corresponding element in the binary matrix is 1.

Parameters
- `parent` (*ndarray*) – Numpy array containing the information for the parent array that is being mutated.
- `boundaries` (*array-like*) – Domain of allowable values.

- `proportions` (*float*) – Ratio of 1s and 0s in the binary matrix used in the pseudo bit flip algorithm
- `int_flat` (*bool*) – flag specifying whether output should be ints.

Returns *child (numpy array)* – Numpy array containing information for the mutated child.

`static shuffle_index(`*parent*`)`

Mutate the parent by swapping an index with another within the same array.

First, the shuffle_index method creates two random matrices. It then compares the two matrices. If the entry in the first matrix is greater than the entry in the second matrix, then it permutes the corresponding elements in the original matrix.

Parameters `parent` (*numpy array*) – Numpy array containing the information for the parent array that is being mutated.

Returns *child (numpy array)* – Numpy array containing information for the mutated child.

5.6 Progress Monitor

`class gastop.progmon.ProgMon(`*progress_fitness*, *progress_truss*, *num_generations*, *domain=None*, *loads=None*, *fixtures=None*`)`

Plots fitness score or truss evolution and stores population statistics.

This class takes in the current sorted population and displays information based on the user requests. If truss monitoring is requested it calls the plot method. The population stats are returned via GenAlg and written to a json file, allowing the user to plot the evolution after the optimization is complete.

`__init__(`*progress_fitness*, *progress_truss*, *num_generations*, *domain=None*, *loads=None*, *fixtures=None*`)`

Creates a ProgMon object

Once created, the object will store all of the relavant information about a progess monitor. The figures are also initialized upon object instatiation.

Parameters

- `progress_fitness` (*boolean*) – if true the minimum fitness score of the population is plotted each iteration
- `progress_truss` (*boolean*) – if true the truss corresponding to the minimum fitness score is displayed each iteration
- `num_generations` (*integer*) – indicates the number of generations, used when initializing the fitness figure
- `domain` (*numpy array*) – indicates bounds of design area, used when *progress_truss* is true
- `loads` (*numpy array*) – indicates magnitude and direction of loads applied to *user_spec_nodes*, used when *progress_truss* is true
- `fixtures` (*numpy array*) – indicates fixed DOFs of *user_spec_nodes*, used when *progress_truss* is true

Returns Nothing

`progress_monitor(`*current_gen*, *population*`)`

Updates progress monitor plots

Function is passed the sorted population and plots either the current generation's best fitness score, best truss, or both. If the truss is displayed, the plot method of the truss object is called but passed the figure instantiated in the init method.

Parameters

- `progress_fitness` (*boolean*) – if true the minimum fitness score of the population is plotted each iteration

- `progress_truss` (*boolean*) – if true the truss corresponding to the minimum fitness score is displayed each iteration
- `num_generations` (*integer*) – indicates the number of generations, used when initializing the *progress_fitness* figure
- `domain` (*numpy array*) – indicates bounds of design area, used when *progress_truss* is true
- `loads` (*numpy array*) – indicates magnitude and direction of loads applied to *user_spec_nodes*, used when *progress_truss* is true
- `fixtures` (*numpy array*) – indicates fixed DOFs of *user_spec_nodes*, used when *progress_truss* is true

Returns Nothing

5.7 Selector

`class gastop.selector.Selector(`*sel_params*`)`

Selects parents to be used for crossover and mutation.

When creating a new Selector() obejct, must be initialized with dictionary sel_params (containing selection method). Object can then be used as a function that selects parents according to the specified method.

`__call__(`*num_parents*, *population*`)`

Calls selector object on a population to get parent indices.

Parameters
- `num_parents` (*int*) – Number of parents to select.
- `population` (*list*) – Population of trusses to select from, must be sorted by fitness score in ascending order.

Returns *parents (ndarray)* – Array of indices of parents in population list.

`__init__(`*sel_params*`)`

Creates a Selector object.

Parameters `sel_params` (*dict*) – Dictionary containing:
- `'method'` *(str)*: Name of chosen selection method.
- `'method_params'` *(dict)*: Dictionary of parameters required by chosen method.

Returns selector (Selector object)

`static inverse_square_rank_probability(`*num_parents*, *population*`)`

Selects parents according to inverse square rank method.

Creates a cdf, with each entry the cumulative sum of 1/sqrt(N) for N = 1, ... Random values are then produced between the largest and smallest elements of the list. Each parent is chosen as the index in the cdf that the corresponding random value falls. In this way, the most probable parents are those with the highest fitness scores.

Parameters
- `num_parents` (*int*) – The number of parents to select.
- `population` (*list*) – List of Truss objects that constitutes the current generation.

Returns *parents (ndarray)* – Numpy array of indices in population corresponding to selected parents.

`static tournament(`*num_parents*, *population*, *tourn_size*, *tourn_prob*`)`

Selects parents according to tournament method.

Randomly selects truss indices from population in groups called tournaments according to "tourn_size." Each tournament is then sorted by index (lower means more fit) in ascending order and a single index from each tournament is selected. The selection from each tournament is chosen probabilistically, assigning the

first, most fit, index with probability p = tourn_prob, and then subsequent indices by p*(1-p)^n. The winners of each tournament are then returned as the parents array.

Parameters

- `num_parents` (*int*) – The number of parents to select.
- `population` (*list*) – List of Truss objects that constitutes the current generation.
- `tourn_size` (*int*) – Number of trusses to include in a given tournament. Must be <= 31.
- `tourn_prob` (*float*) – Probability of selecting first index in each tournament. Must be between 0 and 1.

Returns *parents (ndarray)* – Numpy array of indices in population corresponding to selected parents.

5.8 Truss

`class gastop.truss.Truss(`*user_spec_nodes*, *rand_nodes*, *edges*, *properties*, *fos=None*, *deflection=None*, *mass=None*, *interference=None*, *cost=None*, *num_joints=None*, *fitness_score=None*`)`

Implements the Truss object, which is the fundamental object/data type in GASTOp.

Each truss is defined by a collection of nodes (points in x,y,z space), edges (connections between nodes), and properties (material and geometric properties of the connections between nodes).

A truss can also have assigned attributes such as factor of safety, deflections, mass, cost, or fitness score. These attributes are calculated based on the nodes, edges, and properties.

`__init__(`*user_spec_nodes*, *rand_nodes*, *edges*, *properties*, *fos=None*, *deflection=None*, *mass=None*, *interference=None*, *cost=None*, *num_joints=None*, *fitness_score=None*`)`

Creates a Truss object

Parameters

- `user_spec_nodes` (*ndarray*) – Array of user specified nodes, such as where loads are applied or where the structure is supported. Array shape should be nx3, where n is the number of specified nodes. Each row should contain the x,y,z coordinates of a node.
- `rand_nodes` (*ndarray*) – Randomly generated nodes. No loads or supports should be assigned to random nodes, as their position may change. Array shape should be mx3 where m is the number of random nodes. Each row should contain the x,y,z coordinates of a node.
- `edges` (*ndarray*) – Array of connections between nodes. Array shape should be kx2, where k is the number of connections or beams in the structure. Each row should be 2 integers, the first being number of the starting node and the second being the ending node. A value of -1 indicates no connection, and will be ignored.
- `properties` (*ndarray*) – Array of indices for beam properties. Array shape should be a 1d array of length k, where k is the number of connections or beams in the structure. Each entry should be an integer index into the properties dictionary, with values between [0, number of beam types].
- `fos` (*ndarray*) – Array of factor of safety values. Default None.
- `deflection` (*ndarray*) – Array of node deflections under load, in meters. Default None.
- `mass` (*float*) – Mass of the structure, in kilograms. Default None.
- `interference` (*float*) – Total length of members passing through user specified areas. Default None.
- `cost` (*float*) – Cost of the structure in dollars. Default None.
- `num_joints` (*int*) – Number of connections between members. Default None.
- `fitness_score` (*float*) – Fitness score of the truss. Default None.

Returns Truss object.

`__str__()`
Prints the truss to the terminal as a formatted array.

Prints node numbers and locations, edge numbers and connections, and beam material property ID's

If deflections, mass, fos, or cost are defined, they will be printed as well.

Parameters `None` –
Returns None

`cleaned_params()`
Returns cleaned copies of node, edge, and property arrays.

Parameters `None` –
Returns
3-element tuple containing –
- **nodes** *(ndarray)*: Concatenation of user_spec_nodes and rand_nodes.
- **edges** *(ndarray)*: Edges array after removing rows with -1 values.
- **properties** *(ndarray)*: Properties corresponding to remaining edges.

`mark_duplicates()`
Checks truss for duplicate edges or self connected nodes and marks them.

Any edge that connects a node to itself, or any duplicate edges are changed to -1.

Parameters `None` –
Returns None

`plot(`*domain=None*, *loads=None*, *fixtures=None*, *deflection=False*, *load_scale=None*, *def_scale=100*, *ax=None*, *fig=None*, *setup_only=False*`)`
Plots a truss object as a 3D wireframe

Parameters
- `self` (*`Truss object`*) – truss to be plotted. Must have user_spec_nodes, rand_nodes, edges defined.
- `domain` (*`ndarray`*) – (optional) axis limits in x,y,z, specified as a 3x2 array: [[xmin, xmax],[ymin,ymax],[zmin,zmax]].
- `loads` (*`ndarray`*) – (optional) Array of loads to be plotted as arrows. Specified as nx6 array, each row corresponding to the load at the node matching the row #. Load format: [Fx,Fy,Fz,Mx,My,Mz]
- `fixtures` (*`ndarray`*) – (optional) Array of fixtures to be plotted as blobs. Specified as an nx6 array, each row corresponding to fixtures at the node matching the row #. Format: [Dx,Dy,Dz,Rx,Ry,Rz] value of 1 means fixed in that direction, value of zero is free.
- `deflection` (*`bool`*) – If True, deflections will be plotted superposed on the undeformed structure. Relative size of deflections is governed by *def_scale*.
- `load_scale` (*`float`*) – Size load vector arrows should be scaled by.
- `def_scale` (*`float`*) – Scaling for deflections. *def_scale*=1 means actual size, larger than 1 magnifies.
- `ax` (*`axis`*) – Axis to plot truss on, if an axis is passed to the function, the function is being called by ProgMon and *prog* is set to 1. If axis is none, a new one is created.
- `fig` (*`fig`*) – Figure belonging to the axis.
- `setup_only` (*`boolean`*) – If true, only the loads and fixtures are plotted.

Returns None

5.9 encoders

class gastop.encoders.ConfigEncoder(*skipkeys=False, ensure_ascii=True, check_circular=True, allow_nan=True, sort_keys=False, indent=None, separators=None, default=None*)

Encodes config file in JSON format.

If the object is a numpy array, converts it to a list and appends '__numpy__' metadata for decoding.

default(*obj*)

Implement this method in a subclass such that it returns a serializable object for `o`, or calls the base implementation (to raise a `TypeError`).

For example, to support arbitrary iterators, you could implement default like this:

```
def default(self, o):
    try:
        iterable = iter(o)
    except TypeError:
        pass
    else:
        return list(iterable)
    # Let the base class default method raise the TypeError
    return JSONEncoder.default(self, o)
```

class gastop.encoders.PopulationEncoder(*skipkeys=False, ensure_ascii=True, check_circular=True, allow_nan=True, sort_keys=False, indent=None, separators=None, default=None*)

Encodes population file in JSON format.

If the object is a numpy array, converts it to a list and appends '__numpy__' metadata for decoding. Handles that population is composed of truss objects.

default(*obj*)

Implement this method in a subclass such that it returns a serializable object for `o`, or calls the base implementation (to raise a `TypeError`).

For example, to support arbitrary iterators, you could implement default like this:

```
def default(self, o):
    try:
        iterable = iter(o)
    except TypeError:
        pass
    else:
        return list(iterable)
    # Let the base class default method raise the TypeError
    return JSONEncoder.default(self, o)
```

gastop.encoders.numpy_decoder(*dct*)

Decodes JSON files to config and population.

If the object is has '__numpy__' metadata, converts it to a numpy array.

Parameters `dct` (*dict*) – Dictionary in JSON file.
Returns *dct (dict)* – Dictionary with numpy arrays decoded.

5.10 utilities

gastop.utilities.beam_file_parser(*properties_path*)
Parses csv file of beam material properties

Each line of the properties file denotes one type of beam, with a specified cross section and material properties.

Property entries should be formatted as: beam #, material name, OD (m), ID (m), elastic_modulus (Pa), yield_strength (Pa), density (kg/m^3), poisson_ratio, cost ($)

Parameters `properties_path` (*str*) – Path to the properties csv file, relative to the directory GASTOp is being executed from.

Returns *properties_dict (dict)* – Dictionary of property values. Each entry is an ndarray of the keyed property of each beam. For example, properties_dict['dens'] is an ndarray of the density of each beam type.

gastop.utilities.init_file_parser(*init_file_path*)
Parse init file for input parameters.

Creates ConfigObj object, which reads input parameters as a nested dictionary of strings. The string are then converted to their correct types using the ConfigObj walk method and a transform function. Defaults are then set with if statements.

Parameters `init_file_path` (*string*) – Path to the init file, relative to the directory GASTOp is being executed from.

Returns *config (ConfigObj object)* – Nested dicitonary of input parameters.

gastop.utilities.load_progress_history(*path_progress_history='progress_history.json'*)
Loads the population history (progress_history) from a JSON file.

Parameters `path_progress_history` (*string*) – Path to progress_history data file.

Returns *progress_history (dict)* – History of each generation, including generation number, fittest truss, etc.

gastop.utilities.save_gif(*progress_history*, *progress_fitness*, *progress_truss*, *animation_path*, *num_gens*, *config*, *gif_pause=0.5*)
Saves progress history to gif

Clears contents of folder specified then creates png of each generation of the evolution and then combines the png's into a gif. Accomplishes this by creating progress monitor instance and passing it the truss object stored in the progress history.

Parameters

- `progress_history` (*dictionary of dictionaries*) – population statistics and best truss from each generation.
- `progress_fitness` (*boolean*) – indicates whether to plot the fitness score.
- `progress_truss` (*boolean*) – indicates whether to plot the current truss.
- `animation_path` (*string*) – path to the file where the gif should be created.
- `num_gens` (*integer*) – total number of generations
- `config` (*dictionary of dictionaries*) – stores domain, loads, and fixtures
- `gif_pause` (*float*) – pause between images in the gif

Returns Nothing

gastop.utilities.save_progress_history(*progress_history*, *path_progress_history='progress_history.json'*)
Saves the population history (progress_history) to a JSON file.

Parameters

- `progress_history` (*dict*) – History of each generation, including generation number, fittest truss, etc.

- `path_progress_history` (*string*) – Path to save progress_history data file. If file doesn't exist, creates it.

Returns None

Chapter 6

Examples

6.1 Pyramid Example Configuration File

```
[general]
user_spec_nodes = '[[0,-.5,0],[0,.5,0],[0,0,1],[2,0,0]]'
loads = '[[0,0,0,0,0,0],[0,0,0,0,0,0],[0,0,0,0,0,0],[0,0,-10000,0,0,0]]'
fixtures = '[[1,1,1,1,1,1],[1,1,1,1,1,1],[1,1,1,1,1,1],[0,0,0,0,0,0]]'
num_rand_nodes = 10 # int
num_rand_edges = 10 # int
properties_path = 'gastop-config/properties.csv'
domain = '[[-1, -1, -1], [5, 1, 2]]'
```

```
[fitness_params]
equation = weighted_sum
       [[parameters]]
       goal_fos = 4
       critical_nodes = '[3]'
       w_fos = 10000
       w_mass = 1
       w_deflection = 100
```

```
[evaluator_params]
struct_solver = mat_struct_analysis_DSM
mass_solver = mass_basic
interferences_solver = blank_test
cost_solver = cost_calc
```

```
[ga_params]
num_threads = 1
pop_size = 1500
num_generations = 50
num_elite = 15
percent_mutation =
percent_crossover =
save_frequency = 5
save_filename_prefix = Recorded_States_
```

```
[monitor_params]
progress_fitness = True
progress_truss = True
```

```
# optional stuff
```

```
[random_params]
rng_seed =
```

```
[crossover_params]
node_crossover_method =
edge_crossover_method =
property_crossover_method =
      [[node_crossover_params]]
      [[edge_crossover_params]]
      [[property_crossover_params]]
```

```
[mutator_params]
node_mutator_method =
edge_mutator_method =
property_mutator_method =
      [[node_mutator_params]]
      std =
      [[edge_mutator_params]]
      proportions =
      [[property_mutator_params]]
      proportions =
```

```
[selector_params]
method =
       [[method_params]]
       tourn_size =
       tourn_prob =
```

6.2 Pyramid Example Results

For this example there are three nodes that are fixed in all six degrees of freedom at XYZ locations [0,-.5,0] , [0,.5,0] , [0,0,1]. A load is applied at [2,0,0] in the negative Z direction with a magnitude of 10,000 N. The genetic algorithm uses a population size of 1500, runs for 50 generations, and is the structure is required to have a safety factor of four. The displacement is minimized for node 3 which is the loaded node at [2,0,0]. To run this example the command is as shown below:

```
$ gastop gastop-config/struct_making_test_init2.txt
```

The results of this simulation are:

```
(py37) nat-oitwireless-inside-vapornet100-c-19252:APC524_FinalProject Dan$ python -m GASTOp
Overall: 100%|████████████████████████████████████████████████████████████| 50/50 [00:40<00:00,  1.29it/s]
Updating: 100%|███████████████████████████████████████████████████████| 1486/1486 [00:00<00:00, 12337.26it/s]
```

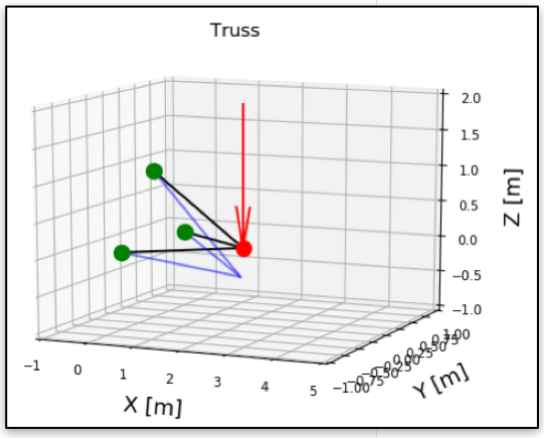
Nodes

#	x	y	z	Deflections dx	dy	dz
0	0.00	−0.50	0.00	0.000e+00	0.000e+00	0.000e+00
1	0.00	0.50	0.00	0.000e+00	0.000e+00	0.000e+00
2	0.00	0.00	1.00	0.000e+00	0.000e+00	0.000e+00
3	2.00	0.00	0.00	−6.198e−04	−1.323e−23	−4.027e−03
4	4.03	−0.47	1.99	0.000e+00	0.000e+00	0.000e+00
5	−0.68	0.89	0.33	0.000e+00	0.000e+00	0.000e+00
6	3.11	−0.13	0.64	0.000e+00	0.000e+00	0.000e+00
7	1.83	−0.97	1.06	0.000e+00	0.000e+00	0.000e+00
8	0.24	0.79	1.35	0.000e+00	0.000e+00	0.000e+00
9	−0.12	−0.70	−0.68	0.000e+00	0.000e+00	0.000e+00
10	2.12	−0.62	−0.85	0.000e+00	0.000e+00	0.000e+00
11	3.90	−0.30	0.47	0.000e+00	0.000e+00	0.000e+00
12	2.22	0.54	−0.83	0.000e+00	0.000e+00	0.000e+00
13	−0.45	−0.38	1.21	0.000e+00	0.000e+00	0.000e+00

Edges

#	Start Node	End Node	Property Type	FoS
0	0	3	0	4.29
1	1	3	0	4.29
2	2	3	4	7.17

```
Mass: 9.374 kg
Cost: $ 10.83
```



Here the fixed nodes are shown in green, the loads are shown in red, and the truss is shown in black. The blue lines show how the truss deforms under the load magnified by a factor of 50 to be visible. The evolution of the optimization algorithm is shown here:

6.3 Cantilever Example Results

For this example there are six nodes that are fixed in all six degrees of freedom at XYZ locations [0,0,0], [0,0.001,1], [0,1.5,0], [0,1.5001,1], [0,3,0], [0,3.0001,1]. Three loads are applied at [3,0,1], [3,1.5,1], [3,3,1] in the negative Z direction, each with a magnitude of 1000 N. The genetic algorithm uses a population size of 1000, runs for 250 generations, and is the structure is required to have a safety factor of four. The displacement is minimized for node 7 which is the middle loaded node at [3,1.5,1]. To run this example the command is as shown below:

```
$ gastop gastop-config/init_cantilevered_test.txt
```

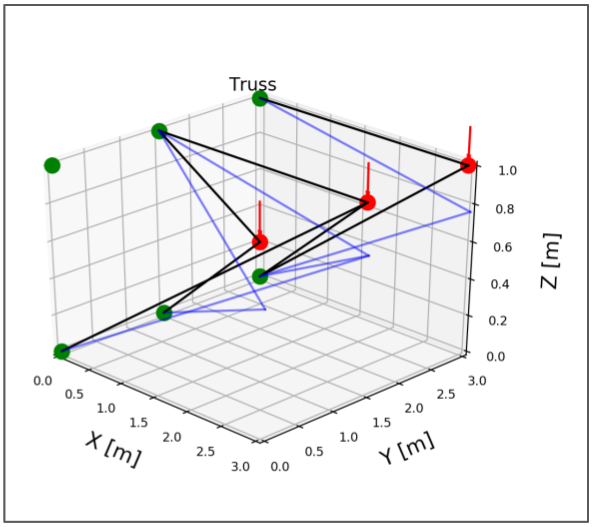
The results of this simulation are:

```
Overall: 100%|████████████████████████████| 250/250 [09:37<00:00,  2.39s/it]
Updating: 100%|████████████████████████████| 990/990 [00:00<00:00, 9050.62it/s]
```

	Nodes			Deflections		
#	x	y	z	dx	dy	dz
0	0.00	0.00	0.00	0.000e+00	0.000e+00	0.000e+00
1	0.00	0.00	1.00	0.000e+00	0.000e+00	0.000e+00
2	0.00	1.50	0.00	0.000e+00	0.000e+00	0.000e+00
3	0.00	1.50	1.00	0.000e+00	0.000e+00	0.000e+00
4	0.00	3.00	0.00	0.000e+00	0.000e+00	0.000e+00
5	0.00	3.00	1.00	0.000e+00	0.000e+00	0.000e+00
6	3.00	0.00	1.00	1.162e-03	2.634e-04	-6.606e-03
7	3.00	1.50	1.00	7.377e-04	1.378e-07	-5.313e-03
8	3.00	3.00	1.00	7.375e-04	6.109e-04	-4.804e-03
9	0.42	2.60	0.22	0.000e+00	0.000e+00	0.000e+00
10	0.43	2.65	0.53	0.000e+00	0.000e+00	0.000e+00
11	2.64	1.43	0.32	0.000e+00	0.000e+00	0.000e+00
12	2.45	2.16	0.07	0.000e+00	0.000e+00	0.000e+00
13	0.24	2.31	0.76	0.000e+00	0.000e+00	0.000e+00
14	0.18	0.55	0.40	0.000e+00	0.000e+00	0.000e+00
15	0.61	2.44	0.44	0.000e+00	0.000e+00	0.000e+00
16	2.24	1.84	0.62	0.000e+00	0.000e+00	0.000e+00
17	2.23	1.40	0.91	0.000e+00	0.000e+00	0.000e+00
18	1.77	1.24	0.96	0.000e+00	0.000e+00	0.000e+00

	Edges			
#	Start Node	End Node	Property Type	FoS
0	4	8	2	4.98
1	5	8	2	5.17
2	3	7	2	5.13
3	4	7	1	4.80
4	0	7	1	4.80
5	2	6	2	4.49
6	3	6	2	4.63

```
Mass: 9.589 kg
Cost: $ 37.28
```



Here the fixed nodes are shown in green, the loads are shown in red, and the truss is shown in black. The blue lines show how the truss deforms under the load magnified by a factor of 50 to be visible.

Index

Symbols

__call__() (gastop.crossover.Crossover method), 10
__call__() (gastop.evaluator.Evaluator method), 11
__call__() (gastop.fitness.FitnessFunction method), 13
__call__() (gastop.mutator.Mutator method), 17
__call__() (gastop.selector.Selector method), 19
__init__() (gastop.crossover.Crossover method), 10
__init__() (gastop.evaluator.Evaluator method), 11
__init__() (gastop.fitness.FitnessFunction method), 14
__init__() (gastop.genalg.GenAlg method), 15
__init__() (gastop.mutator.Mutator method), 17
__init__() (gastop.progmon.ProgMon method), 18
__init__() (gastop.selector.Selector method), 19
__init__() (gastop.truss.Truss method), 20
__str__() (gastop.truss.Truss method), 20

B

beam_file_parser() (in module gastop.utilities), 23
blank_test() (gastop.evaluator.Evaluator static method), 12

C

cleaned_params() (gastop.truss.Truss method), 21
ConfigEncoder (class in gastop.encoders), 22
cost_calc() (gastop.evaluator.Evaluator static method), 12
Crossover (class in gastop.crossover), 10

D

default() (gastop.encoders.ConfigEncoder method), 22
default() (gastop.encoders.PopulationEncoder method), 22

E

Evaluator (class in gastop.evaluator), 11

F

FitnessFunction (class in gastop.fitness), 13

G

gaussian() (gastop.mutator.Mutator static method), 17
GenAlg (class in gastop.genalg), 15
generate_random() (gastop.genalg.GenAlg method), 15

I

init_file_parser() (in module gastop.utilities), 23
initialize_population() (gastop.genalg.GenAlg method), 15
interference_ray_tracing() (gastop.evaluator.Evaluator static method), 12
inverse_square_rank_probability() (gastop.selector.Selector static method), 19

L

load_progress_history() (in module gastop.utilities), 23
load_state() (gastop.genalg.GenAlg static method), 16

M

mark_duplicates() (gastop.truss.Truss method), 21
mass_basic() (gastop.evaluator.Evaluator static method), 12
mat_struct_analysis_DSM() (gastop.evaluator.Evaluator static method), 12
Mutator (class in gastop.mutator), 16

N

numpy_decoder() (in module gastop.encoders), 22

P

plot() (gastop.truss.Truss method), 21
PopulationEncoder (class in gastop.encoders), 22
ProgMon (class in gastop.progmon), 18
progress_monitor() (gastop.progmon.ProgMon method), 18
pseudo_bit_flip() (gastop.mutator.Mutator static method), 17

R

rastrigin() (gastop.fitness.FitnessFunction static method), 14

rosenbrock() (gastop.fitness.FitnessFunction static method), 14
run() (gastop.genalg.GenAlg method), 16

S

save_gif() (in module gastop.utilities), 23
save_progress_history() (in module gastop.utilities), 23
save_state() (gastop.genalg.GenAlg method), 16
Selector (class in gastop.selector), 19
shuffle_index() (gastop.mutator.Mutator static method), 18
single_point_split() (gastop.crossover.Crossover static method), 10
sphere() (gastop.fitness.FitnessFunction static method), 14

T

tournament() (gastop.selector.Selector static method), 19
Truss (class in gastop.truss), 20
two_points_split() (gastop.crossover.Crossover static method), 11

U

uniform_crossover() (gastop.crossover.Crossover static method), 11
update_population() (gastop.genalg.GenAlg method), 16

W

weighted_sum() (gastop.fitness.FitnessFunction static method), 14