Welcome to the PonyGE2 wiki!

Below is an outline of the main topics detailed herein. If you feel anything in particular is missing or incorrectly described, please raise a new issue on GitHub.

Grammatical Evolution (GE) is a population-based evolutionary algorithm, where a BNF-style [[grammar|Grammars]] is used in the [[genotype-to-phenotype mapping process|Representation#genotype-phenotype-mapping-process]].

PonyGE2 is an implementation of GE in Python. It's intended as an advertisement and a starting-point for those new to GE, a reference for students and researchers, a rapid-prototyping medium for our own experiments, and as a Python workout.

The original version of PonyGE (https://github.com/jmmcd/ponyge) was originally designed to be a small single-file implementation of GE. However, over time this has grown to a stage where a more formal structured approach was needed. This has led to the development of PonyGE2 (https://github.com/PonyGE/PonyGE2), presented here.

A technical paper describing PonyGE2 has been published and been made freely available on arXiv at:

https://arxiv.org/abs/1703.08535

PonyGE2 can be referenced using the following citation:

```
Fenton, M., McDermott, J., Fagan, D., Forstenlechner, S., Hemberg, E., and
O'Neill, M. PonyGE2: Grammatical Evolution in Python. arXiv preprint,
arXiv:1703.08535, 2017.
```

The PonyGE2 development team can be contacted via GitHub.

PonyGE2 was reviewed in GPEM Volume 22:383-385 by Tuong Manh Vu. Thanks for the review!

## Requirements

---

PonyGE2 requires Python 3.5 or higher. Using matplotlib, numpy, scipy, scikit-learn (sklearn), pandas.

All requirements can be satisfied with Anaconda.

## Running PonyGE2

---

We don't provide any setup script. You can run an example problem (the default is [[regression|Example-Problems#regression]]) just by typing:

```
$ cd src
$ python ponyge.py
```

This will run an example problem and generate a results folder. The folder contains several files showing the run's stats, producing graphs and documenting the parameters used, as well as a file detailing the best individual. For a more verbose command line experience run the following:

```
$ cd src
$ python ponyge.py --verbose
```

Each line of the verbose output corresponds to a generation in the evolution, and prints out all statistics on the current run (only if `--verbose` is specified). Upon completion of a run, the best individual is printed to the command line, along with summary statistics.

There are a number of arguments that can be used for passing values via the command-line. To see a full list of these just run the following:

```
$ python ponyge.py --help
```

## About PonyGE2

---

Grammatical Evolution (GE) [[[O'Neill & Ryan, 2003|References]]] is a grammar-based form of Genetic Programming [[[Koza, 1992|References]]]. It marries principles from molecular biology to the representational power of formal grammars. GE’s rich modularity gives a unique flexibility, making it possible to use alternative search strategies, whether evolutionary, deterministic or some other approach, and to radically change its behaviour by merely changing the grammar supplied. As a grammar is used to describe the structures that are generated by GE, it is trivial to modify the output structures by simply editing the plain text grammar. This is one of the main advantages that makes the GE approach so attractive. The genotype-phenotype mapping also means that instead of operating exclusively on solution trees, as in standard GP, GE allows search operators to be performed on the genotype (e.g., integer or binary chromosomes), in addition to partially derived phenotypes, and the fully formed phenotypic derivation trees themselves.

PonyGE2 is primarily a Python implementation of canonical Grammatical Evolution, but it also includes a number of other popular techniques and EC aspects.

One of the central components of PonyGE is the `algorithm.parameters.params` dictionary. This dictionary is referenced throughout the entire program and is used to streamline the whole process by keeping all optional parameters in the one place. This also means that there is little to no need for arguments to the various functions in PonyGE, as these arguments can often be read directly from the parameters dictionary. Furthermore, the parameters dictionary is used to specify and store optional functions such as `initialisation`, `crossover`, `mutation`, and `replacement`.

## Setting Parameters

There are three different ways to specify operational parameters with PonyGE: by editing the dictionary itself, by setting a parameters file, and by specifying parameters from the command line.

### The Parameters Dictionary

The most basic way to set specific parameters values in PonyGE2 is to directly modify the `algorithm.parameters.params` dictionary in the code. This is not encouraged, as the `algorithm.parameters.params` dictionary contains the default values for many parameters. As such, PonyGE2 may perform in an unpredictable manner in some cases.

### Parameters Files

The most versatile and robust method for specifying desired parameters is to list your desired parameters in a specialised parameters text file. Specific parameters files can then be read in by PonyGE2, and will set any parameters specified therein.

Parameters files are simple text files (with no commenting or other documentation) that simply list all of the desired parameters to be set in the central `algorithm.parameters.params` dictionary. Parameters can be specified in a parameters file using the following formatting:

```
PARAMETER_NAME: PARAMETER_VALUE
```

where `PARAMETER_NAME` is the name of the desired parameter from the central `algorithm.parameters.params` dictionary to be set, and `PARAMETER_VALUE` is the desired value to be stored for that parameter.

Entire lines can be commented out of parameters files simply by prefacing the line with a `#` symbol, as with normal Python code. The use of comments allows parameters files to be neatly and clearly organised into grouped sections if so desired. Note that blank lines in parameters files are ignored.

***NOTE*** *that parameter names and values must be colon-separated, i.e. each line of the parameters file will be read in as a string, and the string will be parsed based on the location of the first instance of a colon.* ***NOTE*** *that this* ***does*** *preclude the use of colons in parameter* ***names*** *(i.e. the keys of the params dictionary), but that this does* ***not*** *preclude the use of colons in the parameter* ***values****.*

Example parameters files are located in the `parameters` folder. When using parameters files, it is necessary to specify the desired parameter file from the command line. This is done with the argument:

```
--parameters [FULL FILE NAME INCLUDING EXTENSION]
```

## Command-Line Arguments

The third and final method for setting parameters is to list desired parameters directly from the command line. PonyGE2 has an extensive command line parser, located in `utilities.algorithm.command_line_parser`. All stock parameters listed in the original `algorithm.parameters.params` dictionary can be set or changed via command line arguments. Command line arguments for all parameters are the same as the dictionary keys, except all in lower case. For example, the dictionary key for the size of the evolutionary population is `POPULATION_SIZE`. This value can be set from the command line with the argument:

```
--population_size [SIZE]
```

where `[SIZE]` is an integer specifying the desired population size.

To see a list of all currently available command-line arguments implemented in the parser, type

```
$ python ponyge.py --help
```

## Order of Setting Parameters

Parameters are set in the `algorithm.parameters.params` dictionary in strict order based on the above three methods. Initial values are set in directly by the [[dictionary|Evolutionary-Parameters#the-parameters-dictionary]] itself. Any specified [[parameters file|Evolutionary-Parameters#parameters-files]] is then read in, and any values specified therein will over-write the values contained so far in the dictionary. Finally, any specified [[command-line arguments|Evolutionary-Parameters#command-line-arguments]] will over-write all previously set parameters.

## Path Specification

The modular structure of PonyGE2 has specialised modules (folders) for each aspect of the overall system:

1. [[Datasets|Evaluation#datasets]] *must* be stored in the `datasets` folder,
2. [[Grammar files|Grammars]] *must* be stored in the `grammars` folder,
3. [[Parameters files|Evolutionary-Parameters#parameters-files]] *must* be stored in the `parameters` folder,
4. [[Population seeds|Seeding#2-seeding-runs-with-one-or-more-target-solutions]] *must* be stored in the `seeds` folder,
5. [[Fitness functions|Evaluation#fitness-functions]] *must* be stored in `src.fitness`,
6. [[Error metrics|Evaluation#error-metrics]] *must* be stored in `utilities.fitness.error_metric`,
7. Operators are typically stored in the operators folder:
    1. [[Initialisation]] functions are stored in `src.operators.initialisation`,
    2. [[Selection]] functions are stored in `src.operators.selection`,

    3. [[Crossover|Variation#crossover]] functions are stored in `src.operators.crossover`,
    4. [[Mutation|Variation#mutation]] functions are stored in `src.operators.mutation`,
    5. [[Replacement]] functions are stored in `src.operators.replacement`.
8. [[Search framework functions|Search-Options]] are typically stored in the algorithm folder:
    1. [[Search Loop|Search-Options#search-loop]] functions are typically stored in `src.algorithm.search_loop`,
    2. [[Step|Search-Options#step]] functions are typically stored in `src.algorithm.step`.

Numbers 1-6 above are hard-coded into the PonyGE2 framework, i.e. these various components of PonyGE2 *must* be stored in their respective folders. However, optional operators have more freedom in the PonyGE2 framework (this will be explained presently).

For all specified arguments listed above, PonyGE2 automatically parses the correct path. Since numbers 1-6 above are hard-coded, only the direct names of the desired modules or functions within the respective folders/modules need to be specified. For example, when specifying a grammar file, one simply needs to specify the path to the file itself *from within the `grammars` folder*, i.e. `letter.bnf` or `supervised_learning/Dow.bnf`. When specifying a fitness function, one simply needs to specify the name of the function class itself *from within the `fitness` module*, i.e. `string_match` or `supervised_learning.regression`. There is no need to specify the containing folder or module (e.g. `grammars` or `fitness` in the above two cases, respectively). Doing so will produce an error.

However, operators and search framework functions (as listed in points 7 and 8 above) have more freedom in where they can be saved. These functions can either be located in their respective default `operators` or `algorithm` modules, or in any desired location within the overall PonyGE2 `src` directory.

If you are importing operators that are saved in their respective default locations, there is no need to specify the full module path; you can simply just specify the name of the desired operator function. For example, if you are seeking to use PonyGE2's built-in [[subtree crossover|Variation#subtree]] function, then you simply need to specify the function name itself, `subtree`, for the relevant parameter input (e.g. `CROSSOVER: subtree` in a parameters file, or `--crossover subtree` from the command line). To summarize, if a single function name is specified, PonyGE2 will attempt to load that function from its default location.

On the other hand, you can specify full module paths (from within the `src` directory) to operator or search functions should you so desire. This enables if you to import these methods from any location within the `src` directory. For example, if you are seeking to use PonyGE2's built-in [[Late-Acceptance Hill Climbing|Search-Options#hill-climbing]] function, you can specify `algorithm.hill_clilmbing.LAHC_search_loop` for the relevant parameter input (as above). PonyGE2 will attempt to import the module directly from the full specified path, and if this fails, it will then try to import the module from the `algorithm` module location (taking the specified path into account).

If PonyGE2 cannot find the specified functions or modules, error reports will be printed out specifying exactly where PonyGE2 is trying to search. Should they appear, these reports should help you to diagnose what has gone awry.

# Adding New Parameters

---

It is possible to add new parameters to the `algorithm.parameters.params` dictionary either by editing the dictionary directly (this is not recommended), or simply by specifying any desired parameters in a parameters file. Since parameters files are used to set new or over-write existing key:value pairs in the parameters dictionary, it is possible to specify any number of new/unseen parameters in a parameters file (even if they do not already appear in the default `algorithm.parameters.params` dictionary). Such new or custom parameters can then be accessed throughout the entire PonyGE2 code base. However, it is not currently possible to specify new parameters directly from the command line. This is because any existing parameters that were merely mis-spelled from the command line would then create new parameters, and the evolutionary system would not perform as expected. Thus, any parameters entered from the command line that do not exist in the command line parser in `utilities.algorithm.command_line_parser` will produce an error. Of course, it is entirely possible to modify the command line parser to accept new arguments.

***NOTE*** *that parameter names should follow the naming convention demonstrated with the existing parameters. All names should contain only uppercase letters and underscores. Avoid the use of spaces where possible and do not use a colon (`:`) in the name of a parameter.*

Along with the [[fitness function|Evaluation#fitness-functions]], grammars are one of the most problem-specific components of the PonyGE2 algorithm. The performance of PonyGE2 can be greatly affected by the quality of the grammar used. When tackling a problem with GE, a suitable BNF (Backus-Naur Form) grammar must initially be defined. The BNF can be either the specification of an entire language or, perhaps more usefully, a subset of a language geared towards the problem at hand.

In GE, a BNF grammar is used to describe the output language to be produced by the system. BNF is a notation for expressing the grammar of a language in the form of production rules.

Each production rule (or "rewrite rule") is composed of a left-hand side (a single non-terminal), followed by the "goes-to" symbol `::=`, followed by a list of production choices separated by the "or" symbol `|`. Production choices can be composed of any combination of terminals or non-terminals. Non-terminals are enclosed by angle brackets `<>`. For example, consider the following production rule:

```
<a> ::= <b>c | d
```

In this rule, the non-terminal `<a>` maps to either the choice `<b>c` (a combination of a new non-terminal `<b>` and a terminal `c`), or a single terminal `d`.

A full grammar is built up of any combinations of such rules.

***NOTE** that all non-terminals must be fully defined in the grammar, i.e. every non-terminal that appears in the grammar must have a full production rule with defined production choices.*

## Recursion

One of the most powerful aspects of GE is that the representation can be variable in length. Notably, rules can be recursive (i.e. a non-terminal production rule can contain itself as a production choice), which can allow GE to generate solutions of arbitrary size, e.g.:

```
<a> ::= <a> + b | b
```

The code produced by a grammar will consist of elements of the terminal set `T`. The grammar is used in a developmental approach whereby the evolutionary process evolves the production rules to be applied at each stage of a mapping process, starting from the start symbol, until a complete program is formed. A complete program is one that is comprised solely from elements of `T`.

In PonyGE2, the BNF grammar is comprised entirely of the set of production rules, with the definition of terminals and non-terminals implicit in these rules. The first non-terminal symbol is by default the start symbol. As the BNF grammar is a plug-in component of the system, it means that GE can produce code in any language thereby giving the system a unique flexibility. However, in PonyGE, the most common scenario is for the solutions to be executable Python code.

## Writing Grammars

---

Grammars can have any of the following features:

- Production separators in multiple lines. A single production rule can span multiple lines. For example, the following code is valid:

  ```
  <a> ::= x | y | z
  ```

  The following code is also equally valid:

  ```
  <a> ::= x |
          y |
          z
  ```

- Entire lines of commented text
  - Note that multi-line comments need each line to be commented out.
- Comments at the end of any line,
- Single quotation within double quotation and vice versa, and
- Any characters can be used in quotation, even separators (`"|"`) or angle brackets (`"<"` / `">"`).

These options can make the code more readable as well as maintainable. Importantly, this method of writing BNF grammars is not as error prone as previous techniques used with GE. Example grammars are provided in the `grammars` folder.

Grammars are parsed using regular expressions. Examples on parsing some of grammars can be found here:

- Whole grammar
- Rule
- Production

# Special Grammars: expressions vs code

Grammars are inextricably linked to [[fitness functions|Evaluation#fitness-functions]] in GE. The following sections will refer to relevant fitness evaluation documentation.

## Expressions

Many users of GP systems will be familiar with regression and symbolic expressions. These expressions are essentially single-line equations that can be evaluated in order to capture their output. The key term here is "evaluate". With many GE applications, phenotypic solutions can be evaluated using Python's built-in `eval()` statement. This statement will take a string input and will attempt to evaluate the output of the string by parsing it into a Python expression, evaluate the expression, and return the output. For example, let us define a local variable `x = 3`. Now, consider the following expression `s`, which is saved as a string:

```
x = 3
s = "5 + x"
```

Note that `s` is a string. If `"5 + x"` were not a string, `s` would merely be stored as the sum of the two numbers, since x is a local variable stored as an `int`.

```
print(x, type(x))

>>> 3 <class 'int'>

print(s, type(s))

>>> 5 + x <class 'str'>
```

Now, we can evaluate this expression in Python in order to capture the output. Using the built-in `eval()` function, we can evaluate the string expression `s` in order to find the output of the proposed expression:

```
output = eval(s)
print(output, type(output))

>>> 8 <class 'int'>
```

Note that the expression evaluated correctly, as `x` was defined as a local variable. When attempting to evaluate the string `s`, Python encountered the variable call `x` within `s`, and since `x` is defined as a local variable, `s` is capable of being evaluated. Knowing this, it is possible to write a grammar which contains references to local and global variables that are either defined or imported into your fitness function. When evaluating your phenotype string, Python will then use these variables if they appear in the string to be evaluated.

The `eval()` statement is used by PonyGE2 to evaluate symbolic expressions in a number of fitness functions, such as those in supervised learning. See the supervised learning fitness functions and grammars for more examples of this functionality.

***NOTE** that locally or globally defined variables or functions can be referenced and accessed directly by expressions at the time of evaluation.*

## Code

Arguably the most powerful feature of GE is its ability to generate arbitrary phenotype strings in an arbitrary language. In particular, this allows GE to generate *programs* (or pieces of code) in an arbitrary language.

Programs differ from most traditional grammar outputs in a number of ways:

1. Phenotype outputs in the form of code typically span multiple lines,
2. Many languages (e.g. Python) use indentation as a necessary part of the code structure. Others use indentation simply for readability.
3. Whereas expressions are `eval`-able, code must be *executed* in order to exert some phenotypic effects (e.g. performing actions, calling external functions, etc.) or to provide some output returns. (Note that in this documentation we discuss the execution of ***Python*** code specifically. Other languages may require specialised execution environments.)

While the execution/evaluation of the code phenotype (i.e. point 3 above) is handled in the [[fitness function|Evaluation#fitness-functions]], provision has been made in PonyGE2 for these first two points through the use of specialised flags. Saving a grammar file with the extension `".pybnf"` rather than the traditional `".bnf"` will automatically activate additional functionality for code-generating grammars through the use of PonyGE2's Python filter (the Python filter can be found at `utilities.representation.python_filter`). This Python filter is automatically called during the mapping process and converts a single-line phenotype string into a multi-line, correctly indented code string. This code string can then be executed using Python's built-in `exec()` statement (rather than the `eval()` statement discussed in [[expressions|Grammars#expressions]] above).

The most important aspect of Python code grammars is the ability to both generate new lines and to reliably indent and dedent code blocks. Both of these are done through the use of curly brackets and colons.

### Indentation

To create a new line and open an indented code block, open a new indentation block with an opening curly bracket followed by a colon: `{:`. To close the previous code block and then de-dent back to the previous indentation level, simply close the previous indentation block with a colon followed by a closing curly bracket: `:}`. An example phenotype code string (generated by a grammar with a `.pybnf` extension) is shown below:

```
s = "def print_arg(arg):{:print(arg):}print_arg("Hello")"
```

Since the grammar file used the `.pybnf` extension rather than the traditional `.bnf` extension, PonyGE2's Python filter is automatically used to parse the phenotype *before* evaluation. The parsed phenotype string then becomes:

```
s = "def print_arg(arg):
         print(arg)
     print_arg("Hello")"
```

This string is a complete program, and can be executed using Python's built-in `exec()` module:

```
exec(s)

>>> Hello
```

***NOTE** that just like normal code, code phenotype strings may need to contain calls to any functions that are defined therein.*

### New Lines

Since opening a new indentation block with `{:` creates a new line, one simply needs to create a complete empty indentation block `{::}` in order to create a new line. This can be seen in the following example:

```
s = "arg = 'Hello World!'{::}def print_arg(arg):{:print(arg:}print_arg(arg)"
```

When automatically parsed through PonyGE2's Python filter, this phenotype string becomes:

```
s = "arg = 'Hello World!'
     def print_arg(arg):
         print(arg)
     print_arg(arg)"
```

This string is a complete program, which can then be executed:

```
exec(s)

>>> Hello World!
```

### Capturing Output, and Local & Global Variables

While capturing the output of `eval`-able expressions is a simple matter of assigning a new variable to the `eval()` call (as seen [[above|Grammars#expressions]]), capturing the output (if any) of a code string execution is not so simple. Python's `exec()` function can take a dictionary as an optional extra input variable. Unlike with evaluable expression strings, executable code strings cannot directly access local and global variables or functions beyond the core built-in Python library. However, if a dictionary is specified in the `exec()` call, then the code string is executed with full read and write access to the specified dictionary. This has two effects:

1. Any keys defined in the given dictionary will be accessible as local variables to the code string during execution, and
2. Any outputs from executing the code string (including defined function objects and generated variables) will be saved as new key:value pairs in the given dictionary.

For example, let us define a dictionary `d`, with a key `"x"`:

```
d = {"x": 5}
```

Let us now define a code string (for convenience, let's say this string has already been parsed by the Python filter):

```
s = "def square(var):
         return var*var
     ans = square(x)"
```

Note that this code string `s` references a local variable `x`, which is contained in the dictionary `d`. Now, if we execute the code string `s` on its own (i.e. with no dictionary), we get no output or returns:

```
out = exec(s)
print(out)

>>>
```

However, if we execute the code string `s` along with the dictionary `d`, then `s` has access to any local variables defined in `d` at the time of execution (i.e. `x`). Furthermore, any output or variables defined during the execution of `s` will be captured and stored in `d`:

```
exec(s, d)
for key in sorted(d.keys()):
    print(key, type(d[key]), d[key])

>>> ans <class 'int> 25
>>> square <class 'function'> <function square at 0x10ad80048>
>>> x <class 'int'> 5
```

Here, we see that the dictionary `d` has been updated with both the newly defined variable `ans`, and the actual `square()` function object itself. This data can then be accessed and used directly by the main body of code in the rest of the fitness function.

***NOTE*** *that the dictionary `d` above also contains a dictionary `__builtins__` of built-in core Python modules. We have omitted this from the above example for clarity's sake.*

***NOTE*** *that while it is possible to pass in either the `locals()` or `globals()` dictionaries (or both) during execution of a code string, this can potentially have adverse effects such as the over-writing of existing data. It is safer to retain full control over the execution environment of phenotype code strings by explicitly defining the desired usable variables in a specific execution dictionary.*

## Calling predefined functions in grammars

It is possible to call predefined functions in grammars. The requirement is that the function must be available, eg in the `local` namespace, when the solution string is `eval`-ed or `exec`-ed. Built-in functions such as `len` are already available. Other well-known GP functions such as `plog` are defined in `src/fitness/supervised_learning/supervised_learning.py` and are imported in the base fitness function class. Other functions such as `np.sin`, or `cv2` functions, or user-written functions, can be imported by the user in a custom fitness function module.

## Variable ranges in grammars

A useful special case is available when writing grammars: ranges of numbers can be specified dynamically in the grammar. This allows for more compact and readable grammars to be written. For example, if a non-terminal `<n>` were to specify all numbers from 0 to 9, the traditional way would be to write:

```
 <n> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

A cleaner method has been implemented in PonyGE2, allowing for such ranges to be directly specified using `GE_RANGE:N`, where `N` is an integer. Wherever the PonyGE2 grammar parser finds an instance of `GE_RANGE`, it substitutes it with a list of the appropriate terminal choices of the given range. The example `<n>` above would therefore be the result of using `GE_RANGE:9` in a grammar.

This functionality can be quite powerful and useful, as it allows for grammars to create dynamic ranges of numbers. Furthermore, fitness-function specific optional arguments can be accessed by the grammar in order to generate such dynamic lists. One such example is the use of an `n_vars` attribute within the fitness function. If the grammar parser encounters `GE_RANGE:dataset_n_vars`, and if the fitness function has an `n_vars` attribute (note that `n_vars` must be an integer), the grammar parser will automatically substitute a range of productions equal to the range of the n_vars attribute found in the fitness function. An example use case for this can be seen in the `supervised_learning/supervised_learning.bnf` grammar file and the `supervised_learning.supervised_learning` fitness function, where the `n_vars` attribute is set by the number of columns in the dataset file. Using grammar productions like the following, we can avoid hard-coding the number of independent variables in the grammar:

```
<var> ::= x[<varidx>]
<varidx> ::= GE_RANGE:dataset_n_vars
```

This is further explained in the [[evaluation|Evaluation#dynamic-ranging-of-variables-within-the-grammar]] section.

## Grammar Files

All grammars are stored in the grammars folder. Grammars can be set with the argument:

```
--grammar_file [FILE_NAME]
```

or by setting the parameter `GRAMMAR_FILE` to `[FILE_NAME]` in either a parameters file or in the params dictionary, where `[FILE_NAME]` is a the full file name of the desired grammar file including the file extension.

***NOTE*** *that the full file extension (e.g. ".bnf")* ***must*** *be specified, but the full file path (e.g.* `grammars/example_grammar.bnf`*)* ***does not*** *need to be specified.*

## A note on unit productions.

Traditionally GE would not consume a codon for unit productions. This was a design decision taken by O'Neill et al [[[O'Neill and Ryan, 2003|References]]]. However, in PonyGE2 unit productions consume codons, the logic being that it helps to do linear tree-style operations. Furthermore, the checks needed for unit productions during the running of the algorithm can add up to millions of checks that aren't needed if we just consume codons for unit productions.

The original design decision on unit productions was also taken before the introduction of evolvable grammars whereby the arity of a unit production could change over time. In this case consuming codons will help to limit the ripple effect from that change in arity. This also replicates non coding regions of genome as seen in nature.

In summary, the merits for not consuming a codon for unit productions are not clearly defined in the literature. The benefits in consuming codons are a reduction in computation and improved speed with linear tree style operations. Other benefits are an increase in non-coding regions in the chromosome (more in line with nature) that through evolution of the grammar may then express useful information.

# Linear Genome Representation

---

Canonical Grammatical Evolution uses linear genomes (also called chromosomes) to encode genetic information [[[O'Neill & Ryan, 2003|References]]]. These linear genomes are then mapped via the use of a formal BNF-style grammar to produce a phenotypic output. All individuals in PonyGE2 have an associated linear genome which can be used to exactly reproduce that individual.

PonyGE2 contains a number of operators that manage linear genomes. These are discussed in later sections.

***NOTE** that in general the use of a linear genome does not allow for [[context-aware (or "intelligent") operations|Representation#context-aware-intelligent-operations]]. Although intelligent linear genome operators exist, e.g.* [[[Byrne et al., 2009|References]]], *they are not implemented here as similar functions can be performed in a more intuitive manner using derivation-tree based operations.*

## Codon Size

Each codon in a genome is an integer value that maps to a specific production choice when passed through the grammar. When generating a codon to represent a production choice, a random integer value is chosen that represents that correct production choice. The maximum value a codon can take is set by default at 10000. This value can be changed with the argument:

```
--codon_size [INT]
```

or by setting the parameter `CODON_SIZE` to `[INT]` in either a parameters file or in the params dictionary, where `[INT]` is an integer which specifies the maximum value a codon can take.

## Genotype-Phenotype Mapping Process

The genotype is used to map the start symbol as defined in the grammar onto terminals by reading codons to generate a corresponding integer value, from which an appropriate production rule is selected by using the following mapping function:

```
Rule = c % r
```

where `c` is the codon integer value, `r` is the number of rule choices for the current non-terminal symbol, and `%` is the modulus rule (also called the mod rule).

Consider the following rule from a hypothetical grammar. Given the non-terminal `<op>`, which describes the set of mathematical operators that can be used elsewhere in the grammar, there are four production rules to select from. As can be seen, the choices are effectively labelled with integers counting from zero.

```
    <op> ::= +    (0)
           | -    (1)
           | *    (2)
           | /    (3)
```

If we assume the codon being read produces the integer 6, then:

```
6 % 4 = 2
```

would select rule (2) `*`. Therefore, the non-terminal `<op>` is replaced with the terminal `*` in the derivation string. Each time a production rule has to be selected to transform a non-terminal, another codon is read. In this way the system traverses the genome.

## Many-to-one Mapping

Since the genotype-to-phenotype mapping process in GE uses the modulus rule to map a gene to a production choice through a given production rule, it is possible for multiple genes/codons to select the same production choice from the same production rule. For example, consider the following production rule with three production choices:

```
<var> ::= x | y | z
```

For this production rule, *any* codon `c` that has a remainder of 0 (i.e. `c % 3 = 0`) will result in the first choice (`x`) being returned. Similarly, any codon with a remainder of 1 will return `y`, and any codon with a remainder of 2 will return `z`. This is called a "many-to-one" mapping, and is a key cornerstone of the flexibility of the genetic mapping process in GE [[[O'Neill & Ryan, 2003|References]]]. Many-to-one mapping allows for neutral mutations, degeneracy, and introns, among other things [[[O'Neill & Ryan, 2003|References]]].

## Wrapping

During the genotype-to-phenotype mapping process, it is possible for the genome to run out of codons before the mapping process has terminated. In this case, a *wrapping* operator can applied which results in the mapping process re-reading the genome again from the start (i.e. wrapping past the end of the genome back to the beginning). As such, codons are reused when wrapping occurs. This is quite an unusual approach in evolutionary algorithms as it is entirely possible for certain codons to be used two or more times depending on the number of wraps specified. This technique of wrapping the individual draws inspiration from the gene-overlapping phenomenon that has been observed in many organisms [[[Lewin, 2000|References]]]. GE works with or without wrapping, and wrapping has been shown to be useful on some problems [[[O'Neill & Ryan, 2003|References]]], however, it does come at the cost of introducing functional dependencies between codons that would not otherwise arise.

By default, wrapping in PonyGE2 is not used (i.e. the `MAX_WRAPS` parameter is set to 0). This can be changed with the argument:

```
--max_wraps [INT]
```

or by setting the parameter `MAX_WRAPS` to `[INT]` in either a parameters file or in the params dictionary, where `[INT]` is an integer which specifies the desired maximum number of times the mapping process is permitted to wrap past the end of the genome back to the beginning again.

***NOTE** that **permitting** the mapping process to wrap on genomes does not necessarily mean it **will** wrap across genomes. The provision is merely allowed.*

## Invalid Individuals

In GE each time the same codon is expressed it will always generate the same integer value, but depending on the current non-terminal to which it is being applied, it may result in the selection of a different production rule. This feature is referred to as *intrinsic polymorphism*. What is crucial however, is that each time a particular individual is mapped from its genotype to its phenotype, the same output is generated. This is the case because the same choices are made each time. It is possible that an incomplete mapping could occur, even after several wrapping events, and typically in this case the mapping process is aborted and the individual in question is given the lowest possible fitness value. The selection and replacement mechanisms then operate accordingly to increase the likelihood that this individual is removed from the population.

An incomplete mapping could arise if the integer values expressed by the genotype were applying the same production rules repeatedly. For example, consider an individual whose full genome consists of three codons `[3, 21, 9]`, all three of which map to production choice 0 from the following rule:

```
<e> ::= (<e><op><e>) (0)
      | <e>          (1)
      | <op>         (2)
```

Even after wrapping, the mapping process would be incomplete and would carry on indefinitely unless terminated. This occurs because the non-terminal `<e>` is being mapped recursively by production rule 0, i.e., `<e>` becomes `(<e><op><e>)`. Therefore, the leftmost `<e>` after each application of a production would itself be mapped to a `(<e><op><e>)`, resulting in an expression continually growing as follows:

```
((<e><op><e>)<op><e>)
```

followed by

```
(((<e><op><e>)<op><e>)<op><e>)
```

and so on.

Since the genome has been completely traversed (even after wrapping), and the derivation string (i.e. the derived expression) still contains non-terminals, such an individual is dubbed *invalid* as it will never undergo a complete mapping to a set of terminals. Invalid individuals are given a `NaN` fitness in PonyGE2, and are counted in the stats. All population statistics in PonyGE2 are compiled ignoring these `NaN` fitness individuals.

### Reducing Invalids

An upper limit on the number of wrapping events that can occur is imposed (as specified by the parameter `MAX_WRAPS`, detailed above), otherwise mapping could continue indefinitely in the invalid case. During the mapping process therefore, beginning from the left hand side of the genome codon integer values are generated and used to select rules from the BNF grammar, until one of the following situations arise:

1. A complete program is generated. This occurs when all the non-terminals in the expression being mapped are transformed into elements from the terminal set of the BNF grammar.
2. The end of the genome is reached, in which case the wrapping operator is invoked. This results in the return of the genome reading frame to the left hand side of the genome once again. The reading of codons will then continue, unless an upper threshold representing the maximum number of wrapping events has occurred during this individual's mapping process.
3. In the event that a threshold on the number of wrapping events has occurred and the individual is still incompletely mapped, the mapping process is halted, and the individual is assigned a `NaN` fitness value.

To reduce the number of invalid individuals being passed from generation to generation various strategies can be employed. Strong selection pressure could be applied, for example, through a steady state replacement. One consequence of the use of a steady state method is its tendency to maintain fit individuals at the expense of less fit, and in particular, invalid individuals. Alternatively, a repair strategy can be adopted, which ensures that every individual results in a valid program. For example, in the case that there are non-terminals remaining after using all the genetic material of an individual (with or without the use of wrapping) default rules for each non-terminal can be pre-specified that are used to complete the mapping in a deterministic fashion. Another strategy is to remove the recursive production rules that cause an individual's phenotype to grow, and then to reuse the genotype to select from the remaining non-recursive rules. Finally, the use of genetic operators which manipulate the derivation tree rather than the linear genome can be used to ensure the generation of completely mapped phenotype strings.

# Derivation Tree Representation

---

During the [[genotype-to-phenotype mapping process|Representation#genotype-phenotype-mapping-process]], a derivation tree is implicitly generated. Since each production choice in standard GE requires a codon (note that [[unit productions consume codons in PonyGE2|Grammars#a-note-on-unit-productions]]), each chosen production choice can be viewed as a node in an overall derivation tree. The parent rule/non-terminal node that generated that choice is viewed as the parent node, and any production choices resultant from non-terminals in the current production choice are viewed as child nodes. The depth of a particular node is defined as how many parents exist in the tree directly above it, with the root node of the entire tree (the start symbol of the grammar) being at depth 1. Finally, the root of each individual node in the derivation tree is the non-terminal production rule that generated the node choice itself.

While linear genome mapping means that each individual codon specifies the production choice to be selected from the given production rule, it is possible to do the opposite. Deriving an individual solution purely using the derivation tree (i.e. *not* using the [[genotype-to-phenotype mapping process|Representation#genotype-phenotype-mapping-process]]) is entirely possible, and indeed provides a lot more flexibility towards the generation of individuals than a linear mapping.

In a derivation tree based mapping process, each individual begins with the start rule of the grammar (as with the linear mapping). However, instead of a codon from the genome defining the production to be chosen from the given rule, a random production is chosen. Once a production is chosen, it is then possible to retroactively *create* a codon that would result in that same production being chosen if a linear mapping were to be used. In order to generate a viable codon, first the index of the chosen production is taken from the overall list of production choices for that rule. Then, a random integer from within the range `[number of choices : number of choices : CODON_SIZE]` (i.e. a number from `number of choices` to `CODON_SIZE` with a step size of `number of choices`). Finally, the index of the chosen production is added to this random integer. This results in a codon which will re-produce the production choice. For example, consider the following rule:

```
<e> ::= a | b | c
```

Now, let us randomly select the production choice `b`. The index of production choice `b` is 1. Next, we randomly select an integer from within the range `[3, CODON_SIZE]`, giving us a random number of 768. Finally, we add the index of production choice `b`, to give a codon of 769. In this manner it is possible to build a derivation tree, where each node will have an associated codon. Simply combining all codons into a list gives the full genome for the individual.

Importantly, since the genome does not define the mapping process, it is not possible for "[[invalid|Representation#invalid-individuals]]" solutions to be generated by derivation tree based methods.

## Context-Aware "Intelligent" Operations

Since production choices are not set with the use of a derivation tree representation (i.e. the production choice defines the codon, rather than the codon defining the production choice), it is possible to build derivation trees in an "intelligent" manner by restricting certain production choices and using context-aware operators. For example, it is possible to force derivation trees to a certain depth by only allowing recursive production choices to be made until the tree is deep enough that branches can be terminated at the desired depth. This is the basis of "intelligent" derivation methods such as [[Ramped Half-Half (or Sensible) initialisation|Initialisation#ramped-half-half]].

It is also possible to perform intelligent variation operations using derivation tree methods. For example, subtree [[crossover|Variation#subtree]] and [[mutation|Variation#subtree-1]] can be controlled by only selecting desired types of sub-trees for variation. Such operators are included in PonyGE2, and are described in the [[Variation]] section.

Bloat occurs in evolutionary algorithms when large increases in genetic material are observed without an observed increase in fitness. There are currently three methods implemented to control genetic bloat in PonyGE2:

1. Limiting the maximum derivation tree depth
2. Limiting the number of nodes in a derivation tree
3. Limiting the maximum length of the genome.

Any combination of these three methods can be used with PonyGE2. Furthermore, these bloat control measures are **not** limited to specific representation types; it is possible to use genome length limitation with derivation tree based operators, and vice versa.

***NOTE*** *that these techniques may not be necessary with all setups. These should be implemented if and when genetic bloat is observed.*

## Max Tree Depth

By default the maximum depth limit for a derivation tree is set to 90. This limit is due to Python's `eval()` stack limit which is set from 92-99 (depending on the Python version used), but grammar design also plays a large part in stack and memory overflow errors. Such a high depth limit can lead to genetic bloat, dramatically slowing down the overall evolutionary process. One way to prevent this is to specify a global maximum tree depth with the argument:

```
--max_tree_depth [INT]
```

or by setting the parameter `MAX_TREE_DEPTH` to `[INT]` in either a parameters file or in the params dictionary, where `[INT]` is an integer which specifies the desired maximum depth limit for derivation trees.

***NOTE*** *that setting the parameter* `MAX_TREE_DEPTH` *or argument* `--max_tree_depth` *to 0 is the same as setting no maximum tree depth, i.e. trees will be allowed to grow in an un-controlled manner. This may cause memory errors and cause PonyGE2 to crash.*

***NOTE*** *that the parameter* `MAX_TREE_DEPTH` *is distinct from the parameter* `MAX_INIT_TREE_DEPTH` *The former sets the global limit across the entire evolutionary process, while the latter is used solely to control derivation tree depth during derivation tree-based initialisation.* ***NOTE*** *also that* `MAX_TREE_DEPTH` *>=* `MAX_INIT_TREE_DEPTH`.

## Max Tree Nodes

By default there are no limits to the maximum number of nodes a derivation tree can have. This can lead to genetic bloat, dramatically slowing down the overall evolutionary process. One way to prevent this is to specify a global maximum number of derivation tree nodes with the argument:

```
--max_tree_nodes [INT]
```

or by setting the parameter `MAX_TREE_NODES` to `[INT]` in either a parameters file or in the params dictionary, where `[INT]` is an integer which specifies the desired maximum number of nodes for derivation trees.

***NOTE*** *that setting the parameter* `MAX_TREE_NODES` *or argument* `--max_tree_nodes` *to 0 is the same as setting no limit on the maximum number of nodes a derivation tree can have, i.e. trees will be allowed to grow in an un-controlled manner.*

## Max Genome Length

By default there are no limits to the maximum length a genome can take. This can lead to genetic bloat, dramatically slowing down the overall evolutionary process. One way to prevent this is to specify a global maximum genome length with the argument:

```
--max_genome_length [INT]
```

or by setting the parameter `MAX_GENOME_LENGTH` to `[INT]` in either a parameters file or in the params dictionary, where `[INT]` is an integer which specifies the desired maximum global genome length.

***NOTE*** *that setting the parameter* `MAX_GENOME_LENGTH` *or argument* `--max_genome_length` *to 0 is the same as setting no limit to the lengths of genomes, i.e. genomes will be allowed to grow in an un-controlled manner.*

***NOTE*** *that the parameter* `MAX_GENOME_LENGTH` *is distinct from the parameter* `MAX_INIT_GENOME_LENGTH`*, which is used solely to control genome size during genome-based initialisation.*

# Search Loop and Step

## Search Loop

The search loop in PonyGE2 controls the overall duration of the search process, including the initialisation of the initial population, along with the main generations loop. In canonical GE, the search process loops over the total number of specified generations.

***NOTE*** *that in PonyGE2 the total number of generations refers to the number of generations over which the search process loops (i.e. over which evolution occurs),* ***NOT*** *including initialisation. Thus, specifying 50 generations will mean an initial population will be generated and evaluated, and then the evolutionary process will loop for 50 generations. Since the initialised generation will be Generation 0, the total number of individuals evaluated across an entire evolutionary run will by* ***population x (generations + 1)***.

The default search loop function in PonyGE2 is stored in `algorithm.search_loop`. However, users are free to implement their own search loops. These search loops can be stored either in the default `algorithm.search_loop` module, or wherever users desire. The search loop is a parameterisable function that can be [[set in the parameters dictionary|Evolutionary-Parameters#setting-parameters]].

While PonyGE2 is currently set up to only use the main search loop function (save for special cases such as re-loading an evolutionary run from state), it is possible for users to write their own search loop functions. It is possible to specify the desired search loop function directly through the parameters dictionary with the argument:

```
--search_loop [SEARCH_LOOP]
```

or by setting the parameter `SEARCH_LOOP` to `[SEARCH_LOOP]` in either a parameters file or in the params dictionary, where `[SEARCH_LOOP]` is the name of the desired search loop function.

Along with the default search loop, a number of [[Hill Climbing|Search-Options#hill-climbing]] search loops are provided in PonyGE2.

## Step

At each generation in the main search loop, the `step` function is called, which by default is `algorithm.step.step()`. The `step` function executes one full iteration of the canonical evolutionary process, typically:

1. [[Selection]]
2. [[Variation]]
   - [[Crossover|Variation#crossover]]
   - [[Mutation|Variation#mutation]]
3. [[Evaluation]]
4. [[Replacement]]

Step functions are stored in `algorithm.step`. While PonyGE2 is currently set up to only use the main step function, it is possible for users to write their own step functions. It is possible to specify the desired step function directly through the parameters dictionary with the argument:

```
--step [STEP]
```

or by or by setting the parameter `STEP` to `[STEP]` in either a parameters file or in the params dictionary, where `[STEP]` is the name of the desired step function.

***NOTE** that if a step function is saved in the main `algorithm.step` module, there is no need to specify the full path to the function. However, if the desired step function is located elsewhere, it **is** necessary to specify the full module path to the function.*

# Population Options

There are a number of parameters within PonyGE2 for controlling overall populations.

## Population Size

The population size controls the total number of individuals to be generated at each generation. The default value is 500. This value can be changed with the argument:

```
--population_size [INT]
```

or by setting the parameter `POPULATION_SIZE` to `[INT]` in either a parameters file or in the params dictionary, where `[INT]` is an integer which specifies the population size.

Higher population sizes can improve performance on difficult problems, but require more computational effort and may lead to premature convergence.

## Generations

The number of generations the evolutionary algorithm will run for. The default value is 50. This value can be changed with the argument:

```
--generations [INT]
```

or by setting the parameter `GENERATIONS` to `[INT]` in either a parameters file or in the params dictionary, where `[INT]` is an integer which specifies the number of generations.

Higher numbers of generations can improve performance, but will lead to longer run-times.

# Hill Climbing

A simple hill-climbing approach, and two modern variants, are provided in `algorithm.hill_climbing`. In simple HC, there is a single current individual (i.e. no population). At each step of the search loop, a mutation operator creates a new candidate individual. If this individual is better than the original, the move is "accepted" -- the candidate replaces the current individual. In Late-Acceptance Hill-Climbing (LAHC), due to Bykov, and Step-Counting Hill-Climbing (SCHC), due to Bykov, the acceptance decision depends on whether the candidate is better than one considered some time previously, dependent on a "history length" parameter. This can allow search to escape local optima. The following arguments will select one of these methods:

```
--search_loop algorithm.hill_climbing.LAHC_search_loop

--search_loop algorithm.hill_climbing.SCHC_search_loop
```

or by setting the parameter `SEARCH_LOOP` to either `algorithm.hill_climbing.LAHC_search_loop` or `algorithm.hill_climbing.SCHC_search_loop` in either a parameters file or the parameters dictionary.

The "history length" parameter is set as follows:

```
--hill_climbing_history [INT]
```

or by setting the parameter `HILL_CLIMBING_HISTORY` to `[INT]` in either a parameters file or the parameters dictionary, where `[INT]` is an integer value specifying the desired hill climbing history length.

Step-Counting Hill Climbing has an additional `SCHC_COUNT_METHOD` parameter that can be specified. This parameter sets the counting method to be used with the SCHC algorithm. This parameter can be set with the argument:

```
--schc_count_method [METHOD]
```

or by setting the parameter `SCHC_COUNT_METHOD` to `[METHOD]` in either a parameters file or the parameters dictionary, where `[METHOD]` is a string specifying the desired step counting method. Currently implemented acceptable strings are:

1. "count_all"

   This is the default option, and counts all moves.

2. "acp"

   This option counts accepted moves only (i.e. where the candidate solution is better than the cost bound and better than or equal to the current best).

3. "imp"

> This option counts improving moves only (i.e. where the candidate solution is better than the current best).

***NOTE** that both LAHC and SCHC reduce to a simple hill-climbing algorithm when used with a history length of 1.*

***NOTE** that crossover is **not** used in hill-climbing algorithms.*

# Distributed Search

In the standard implementation of PonyGE2, the evolutionary search process runs on a single machine (or agent) which has access to all the individuals in the population. It can then choose among the highest performing individuals from the population to perform the various genetic operations. In a multi-agent implementation, each agent has its own instance of genetic information. An agent can only have access to its own genotype. The agent moves in an environment and communicates to nearby agents to share the genetic information. After collecting genetic information from a handful of agents, it can apply the genetic operations by itself for the information that it has gathered. This multi-agent approach is particularly useful for robotics or distributed implementations across many nodes, as each agent can act individually with only local knowledge.

## Distributed Search Loop

When the `MULTIAGENT` parameter is set to `True`, search_loop and step methods from `algorithm.distributed_algorithm` automatically get activated. The first step in the distributed search loop is to initialize the various agents objects. Based on the parameter set for `AGENT_SIZE` that the number of agents is created, where the `INTERACTION_PROBABILITY` parameter is passed while initializing agents. This parameter defines the probability of interaction with other agents. Higher this value, there are higher changes interacting with other agents and higher chances of finding solutions faster. Then after initializing agents, for each generation, step module is called which takes the list of agents as a function parameter and returns a list of agents.

## Distributed Step

In this module, for each agent in the environment, the agent will perform: sense, act and update action sequentially. These three action or modules are built in for the agents. These modules can be extended or overridden based on the type of agent required to solve a particular problem. After performing these actions, a list of agents is returned.

## Agents

It is the class that defines the behavior of the agents. Right now we have only defined a very primitive agent but based on the requirements the method of this class can be overridden. The Sense method describes the way the agent gets information from the environment. The Act method provides a way from the agent to process the information obtained from Sense method. The update method is agent's way to update its state. In our case, the Sense method is used to find other nearby agents, the Act method is used to apply genetic operations to the stack of genetic information collected from nearby agents and the update method is used to update the agent's genetic information.

As detailed previously, there are two main ways to initialise a GE individual: by generating a [[genome|Representation#linear-genome-representation]], or by generating a [[derivation tree|Representation#derivation-tree-representation]]. Generation of a genome does not take tree size/shape into account. Population initialisation via derivation tree generation on the other hand allows for fine control over many aspects of the initial population, e.g. the distribution of depth limits or derivation tree shape. Unlike with genome initialisation, there are a number of different ways to initialise a population using derivation trees. Currently implemented methods are detailed below.

## Genome

### Random

The simplest and worst-performing initialisation approach is to generate individuals from uniformly generated genomes. Use command-line argument:

```
--initialisation uniform_genome
```

or set the parameter `INITIALISATION` to `uniform_genome` in either a parameters file or in the params dictionary.

***NOTE** that random genome initialisation in Grammatical Evolution should be used with caution as poor grammar design can have a negative impact on the quality of randomly initialised solutions due to the inherent bias capabilities of GE* [[[Fagan et al., 2016|References]]; [[Nicolau & Fenton, 2016|References]]].

An improved approach, *random with valids only and no duplicates* (named RVD and described by Nicolau) just discards invalid individuals and those with duplicate phenotypes. According to Nicolau it performs well, while still being easy to implement:

```
--initialisation rvd
```

or set the parameter `INITIALISATION` to `rvd` in either a parameters file or in the params dictionary.

By default in PonyGE2, genomes of length 200 codons are generated when using random genome initialisation. However, this parameter can be changed using the argument:

```
--init_genome_length [INT]
```

or by setting the parameter `INIT_GENOME_LENGTH` to `[INT]` in either a parameters file or in the params dictionary, where `[INT]` is an integer which specifies the length of genomes to be initialised.

## Derivation Tree

There are currently three options provided in PonyGE2 for initialising a population of individuals using derivation tree methods. You can either initialise a population of random derivation trees, or you can use various "smart" initialisation methods implemented here.

## Random

Random derivation tree initialisation generates individuals by randomly building derivation trees up to the specified maximum initialisation depth limit.

Activate with:

```
--initialisation uniform_tree
```

or by setting the parameter `INITIALISATION` to `uniform_tree` in either a parameters file or in the params dictionary.

***NOTE** that there is no obligation that randomly generated derivation trees will extend to the depth limit; they will be of random size* [[[Fagan et al., 2016|References]]].

***NOTE** that randomly generated derivation trees will have a tendency towards smaller tree sizes with the use of a grammar-based mapping* [[[Fagan et al., 2016|References]]].

## Ramped Half-Half

Ramped Half-Half initialisation in Grammatical Evolution is often called "Sensible Initialisation" [[[Ryan and Azad, 2003|References]]]. Sensible Initialisation follows traditional GP Ramped Half-Half initialisation by initialising a population of individuals using two separate methods: `Full` and `Grow`.

`Full` initialisation generates a derivation tree where all branches extend to the specified depth limit. This tends to generate very bushy, evenly balanced trees [[[Fagan et al., 2016|References]]].

`Grow` initialisation generates a randomly built derivation tree where no branch extends *past* the depth limit.

***NOTE** that `Grow` is analogous to random derivation tree initialisation, i.e. no branch in the tree is **forced** to reach the specified depth. Depending on how the grammar is written, this can result in a very high probability of small trees being generated, regardless of the specified depth limit.*

Activate with:

```
--initialisation rhh
```

or by setting the parameter `INITIALISATION` to `rhh` in either a parameters file or in the params dictionary.

RHH initialisation generates pairs of solutions using both `full` and `grow` methods for a ramped range of depths. The maximum initialisation depth is set with the argument:

```
--max_init_tree_depth [INT]
```

or by setting the parameter `MAX_INIT_TREE_DEPTH` to `[INT]` in either a parameters file or in the params dictionary, where `[INT]` is an integer which specifies the maximum depth to which derivation trees are to be initialised. The default value is set at 10.

By default in PonyGE, initialisation ramping *begins* at a depth where sufficient unique solutions can be generated for the number of required solutions at that depth [[[Nicolau & Fenton, 2016|References]]]. However, this value can be over-written in favor of a user-defined minimum ramping depth. This can be set with the argument:

```
--min_init_tree_depth [INT]
```

or by setting the parameter `MIN_INIT_TREE_DEPTH` to `[INT]` in either a parameters file or in the params dictionary, where `[INT]` is an integer which specifies the minimum depth from which derivation trees are to be initialised.

***NOTE** that RHH initialisation with the use of a grammar-based mapping process such as GE can potentially result in a high number of duplicate individuals in the initial generation, resulting from a potentially high number of very small solutions* [[[Nicolau & Fenton, 2016|References]]; [[Fagan et al., 2016|References]]]. *As such, caution is advised when using RHH initialisation in grammar-based systems, as particular care needs to be given to grammar design in order to minimise this effect* [[[Fagan et al., 2016|References]]].

## Position Independent Grow (PI Grow)

Position Independent Grow (PI Grow) initialisation in Grammatical Evolution mirrors Sensible/Ramped Half-Half initialisation by initialising a population of individuals over a ramped range of depths. However, while RHH uses two separate methods `Full` and `Grow` to generate pairs of individuals at each depth, PI Grow eschews the `Full` component and only uses the `Grow` aspect. There are two further differences between traditional GP `Grow` and PI Grow [[[Fagan et al., 2016|References]]]:

1. At least one branch of the derivation tree is forced to the specified maximum depth in PI Grow, and
2. Non-terminals are expanded in random (i.e. position independent) order rather than the left-first derivation of traditional mappers.

Activate with:

```
--initialisation PI_grow
```

or by setting the parameter `INITIALISATION` to to `PI_grow` in either a parameters file or in the params dictionary.

As with RHH initialisation, PI Grow initialisation generates individuals for a ramped range of depths. The maximum initialisation depth is set with the argument:

```
--max_init_tree_depth [INT]
```

or by setting the parameter `MAX_INIT_TREE_DEPTH` to `[INT]` in either a parameters file or in the params dictionary, where `[INT]` is an integer which specifies the maximum depth to which derivation trees are to be initialised. The default value is set at 10.

By default in PonyGE, initialisation ramping *begins* at a depth where sufficient unique solutions can be generated for the number of required solutions at that depth [[[Nicolau & Fenton, 2016|References]]]. However, this value can be over-written in favor of a user-defined minimum ramping depth. This can be set with the argument:

```
--min_init_tree_depth [INT]
```

or by setting the parameter `MIN_INIT_TREE_DEPTH` to `[INT]` in either a parameters file or in the params dictionary, where `[INT]` is an integer which specifies the minimum depth from which derivation trees are to be initialised.

The selection process is a key step in Evolutionary Algorithms. Selection drives the search process towards specific areas of the search space. The selection process operates on a population of individuals, and produces a population of "parents". These parents are then traditionally used by [[variation operators|Variation]].

The linear genome mapping process in Grammatical Evolution can generate [["invalid"|Representation#invalid-individuals]] individuals. Only valid individuals are selected by default in PonyGE2, however this can be changed with the argument:

```
--invalid_selection
```

or by setting the parameter `INVALID_SELECTION` to `True` in either a parameters file or in the params dictionary.

## Tournament

Tournament selection selects `TOURNAMENT_SIZE` individuals from the overall population, sorts them, and then returns the single individual with the best fitness. Since no individuals are removed from the original population, it is possible that the same individuals may be selected multiple times to appear in multiple tournaments, although the same individual may not appear multiple times *in the same tournament*.

Activate with:

```
--selection tournament
```

or by setting the parameter `SELECTION` to `tournament` in either a parameters file or in the params dictionary.

Tournament size is set by default at 2. This value can be changed with the argument:

```
--tournament_size [INT]
```

or by setting the parameter `TOURNAMENT_SIZE` to `[INT]` in either a parameters file or in the params dictionary, where `[INT]` is an integer which specifies the tournament size.

## Truncation

Truncation selection takes an entire population, sorts it, and returns the best `SELECTION_PROPORTION` of that population.

Activate with:

```
--selection truncation
```

or by setting the parameter `SELECTION` to `truncation` in either a parameters file or in the params dictionary.

Selection proportion is set by default at 0.5 (i.e. return the top 50% of the population). This value can be changed with the argument:

```
--selection_proportion [NUM]
```

or by setting the parameter `SELECTION_PROPORTION` to `[NUM]` in either a parameters file or in the params dictionary, where `[NUM]` is a float between 0 and 1.

***NOTE** that unless the specified `SELECTION_PROPORTION` is 1.0 (i.e. 100%), truncation selection necessarily returns a selected parent population that is smaller in size than the original population.*

## NSGA2

NSGA2 selection performs a pareto tournament selection on `TOURNAMENT_SIZE` individuals. Pareto selection takes into account both the pareto front of the individuals (to check for dominance) and the crowding comparison of the individuals (to ensure dispersion on the fronts). For more details on the NSGA-II operator see [[[Deb et al., 2002|References]]].

Activate with:

```
--selection nsga2_selection
```

or by setting the parameter `SELECTION` to `nsga2_selection` in either a parameters file or in the params dictionary.

***NOTE** that NSGA2 selection can only be used in conjunction with [[multiple objective optimisation|Evaluation#multiple-objective-optimisation]] problems.*

***NOTE** that multi-objective optimisation compatible operators require a `multi_objective = True` attribute.*

Variation operators in evolutionary algorithms explore the search space by varying genetic material of individuals in order to explore new areas of the search space. As detailed in the [[Representation|Representation]] section, PonyGE2 actively maintains two forms of genetic material for each individual throughout the evolutionary process: the [[linear genome|Representation#linear-genome-representation]], and the [[derivation tree|Representation#derivation-tree-representation]]. Variation operators in PonyGE2 operate exclusively on *one* of these two representations, i.e. any given variation operator will operate on either the derivation tree or the linear genome. To monitor and manage this, each variation operator in PonyGE2 has a `representation` attribute which is set to either `"linear"` or `"subtree"` as appropriate.

While variations are carried out exclusively on one representation at a time, both representations are maintained prior to variation. This means that a change to an individual's linear genome will see a corresponding change to their derivation tree, and vice-versa. This is done by re-mapping each individual using the new genetic material (e.g. the new genome or new derivation tree) after a complete change is made. While not strictly necessary, this allows for mixing and matching of varying types of variation operators, and allows for variation operators to be used as a toolbox (i.e. any number of variation operators can be chained in any desired order any number of times).

There are two main types of variation operator:

1. [[Crossover|Variation#crossover]]
2. [[Mutation|Variation#mutation]]

## Crossover

Given a parent population of individuals picked using the given [[selection process|Selection]], crossover randomly selects two parents and directly swaps genetic material between them. Parents are selected from the parent population in a non-exclusive manner, i.e. it is possible to select the same parent multiple times for multiple crossover events.

Given these two parents, the crossover probability defines the probability that a given crossover operator will perform crossover on their genetic material. The probability of crossover occurring is set with the argument:

```
--crossover_probability [NUM]
```

or by setting the parameter `CROSSOVER_PROBABILITY` to `[NUM]` in either a parameters file or in the params dictionary, where `[NUM]` is a float between 0 and 1. The default value for crossover is 0.75 (i.e. two selected parent individuals have a 75% chance of having genetic material crossed over between them).

Unlike canonical Genetic Programming [[[Koza, 1992|References]]], crossover in Grammatical Evolution always produces **two children** given two parents [[[O'Neill et al., 2003|References]]]. However, this is not a requirement for PonyGE2; the user is free to add new crossover operators producing as many children as desired.

There are currently four linear genome crossover operators implemented in PonyGE2:

1. [[Fixed Onepoint|Variation#fixed-onepoint]]

2. [[Fixed Twopoint|Variation#fixed-twopoint]]
3. [[Variable Onepoint|Variation#variable-onepoint]]
4. [[Variable Twopoint|Variation#variable-twopoint]]

There is also currently one derivation tree based crossover method implemented in PonyGE2:

5. [[Subtree|Variation#subtree]]

Since linear genome crossover operators are not [["intelligent"|Representation#context-aware-intelligent-operations]], i.e. crossover is applied randomly, it is therefore possible for linear crossover operators to generate [[invalid individuals|Representation#invalid-individuals]] (i.e. individuals who do not terminate mapping). In order to mitigate this issue, provision has been made in PonyGE2 to prevent crossover from generating invalid solutions. While this option is set to `False` by default, it can be selected with the argument:

```
--no_crossover_invalids
```

or by setting the parameter `NO_CROSSOVER_INVALIDS` to `True` in either a parameters file or in the params dictionary.

If the `NO_CROSSOVER_INVALIDS` parameter is used, crossover will select two new parents and perform crossover again in order to generate two valid children. This process loops until valid children are created.

***NOTE** that since the `NO_CROSSOVER_INVALIDS` parameter uses a `while` loop to force crossover to generate valid solutions, it is possible for crossover to get stuck in an infinite loop if this option is selected. As such, caution is advised when using this option.*

***NOTE** that since crossover operators modify the parents in some fashion, copies of the parents must first be made before crossover is applied. If copies are not made, then the original parents in the selected parent population would be modified in-place, and subsequent modification of these parents would change any children so produced.*

### Fixed Onepoint

Given two individuals, fixed onepoint crossover creates two children by selecting the same point on both genomes for crossover to occur. The head of genome 0 is then combined with the tail of genome 1, and the head of genome 1 is combined with the tail of genome 0. This means that genomes will always remain the same length after crossover. Fixed onepoint crossover can be activated with the argument:

```
--crossover fixed_onepoint
```

or by setting the parameter `CROSSOVER` to `fixed_onepoint` in either a parameters file or in the params dictionary.

Crossover points are selected within the used portion of the genome by default (i.e. crossover does not occur in the unused tail of the individual). This parameter can be selected with the argument:

```
--within_used
```

or by setting the parameter `WITHIN_USED` to either `True` or `False` in either a parameters file or in the params dictionary.

***NOTE** that by default `WITHIN_USED` is set to `True`.*

***NOTE** that selecting the argument `--within_used` will also set the `WITHIN_USED` parameter to `True` As such, the only way to change the `WITHIN_USED` parameter to `False` is to set so in either a parameters file or in the params dictionary.*

### Fixed Twopoint

Given two individuals, fixed twopoint crossover creates two children by selecting the same points on both genomes for crossover to occur. The head and tail of genome 0 are then combined with the mid-section of genome 1, and the head and tail of genome 1 are combined with the mid-section of genome 0. This means that genomes will always remain the same length after crossover. Fixed twopoint crossover can be activated with the argument:

```
--crossover fixed_twopoint
```

or by setting the parameter `CROSSOVER` to `fixed_twopoint` in either a parameters file or in the params dictionary.

As with all linear genome crossovers, crossover points are selected within the used portion of the genome by default (i.e. crossover does not occur in the unused tail of the individual).

### Variable Onepoint

Given two individuals, variable onepoint crossover creates two children by selecting a different point on each genome for crossover to occur. The head of genome 0 is then combined with the tail of genome 1, and the head of genome 1 is combined with the tail of genome 0. This allows genomes to grow or shrink in length. Variable onepoint crossover can be activated with the argument:

```
--crossover variable_onepoint
```

or by setting the parameter `CROSSOVER` to `variable_onepoint` in either a parameters file or in the params dictionary.

As with all linear genome crossovers, crossover points are selected within the used portion of the genome by default (i.e. crossover does not occur in the unused tail of the individual).

***NOTE** that variable linear crossovers can cause individuals to grow in size, leading to bloat.*

### Variable Twopoint

Given two individuals, variable twopoint crossover creates two children by selecting two different points on each genome for crossover to occur. The head and tail of genome 0 are then combined with the mid-section of genome 1, and the head and tail of genome 1 are combined with the mid-section of genome 0. This allows genomes to grow or shrink in length. Variable twopoint crossover can be activated with the argument:

```
--crossover variable_twopoint
```

or by setting the parameter `CROSSOVER` to `variable_twopoint` in either a parameters file or in the params dictionary.

As with all linear genome crossovers, crossover points are selected within the used portion of the genome by default (i.e. crossover does not occur in the unused tail of the individual).

***NOTE** that variable linear crossovers can cause individuals to grow in size, leading to bloat.*

### Subtree

Given two individuals, subtree crossover creates two children by selecting candidate subtrees from both parents based on matching non-terminal nodes in their derivation trees. The chosen subtrees are then swapped between parents, creating new children. Subtree crossover can be activated with the argument:

```
--crossover subtree
```

or by setting the parameter `CROSSOVER` to `subtree` in either a parameters file or in the params dictionary.

***NOTE** that subtree crossover can cause individuals to grow in size, leading to bloat.*

***NOTE** that subtree crossover will **not** produce invalid individuals, i.e. given two valid parents, both children are guaranteed to be valid.*

## Mutation

While crossover operates on pairs of selected parents to produce new children, mutation in Grammatical Evolution operates on every individual in the child population *after* crossover has been applied. Note that this is different in implementation so canonical GP crossover and mutation, whereby a certain percentage of the population would be selected for crossover with the remaining members of the population subjected to mutation [[Koza, 1992|References]].

There are currently two linear mutation operators and one subtree mutation operator implemented in PonyGE2:

1. [[Int Flip Per Codon|Variation#int-flip-per-codon]]
2. [[Int Flip Per Ind|Variation#int-flip-per-ind]]
3. [[Subtree|Variation#subtree-1]]

***NOTE** that linear genome mutation operators are not intelligent, i.e. mutation is applied randomly. It is therefore possible for linear mutation operators to generate invalid individuals (i.e. individuals who do not terminate* *mapping).*

Since linear genome mutation operators are not [["intelligent"|Representation#context-aware-intelligent-operations]], i.e. mutation is applied randomly, it is therefore possible for linear mutation operators to generate [[invalid individuals|Representation#invalid-individuals]] (i.e. individuals who do not terminate mapping). In order to mitigate this issue, provision has been made in PonyGE2 to prevent mutation from generating invalid solutions. While this option is set to `False` by default, it can be selected with the argument:

```
--no_mutation_invalids
```

or by setting the parameter `NO_MUTATION_INVALIDS` to `True` in either a parameters file or in the params dictionary.

If the `NO_MUTATION_INVALIDS` parameter is used, mutation will be performed on the individual indefinitely until a valid solution is created.

***NOTE** that even though the `NO_MUTATION_INVALIDS` parameter uses a `while` loop to force mutation to generate valid solutions, unlike with crossover it is not possible for mutation to get stuck in an infinite loop if this option is selected.*

### Int Flip Per Codon

Int Flip Per Codon mutation operates on linear genomes and randomly mutates every individual codon in the genome with a probability `[MUTATION_PROBABILITY]`. Int Flip Per Codon mutation can be activated with the argument:

```
--mutation int_flip_per_codon
```

or by setting the parameter `MUTATION` to `int_flip_per_codon` in either a parameters file or in the params dictionary. The default mutation probability is for every codon 1 over the entire length of the genome. This can be changed with the argument:

```
--mutation_probability [NUM]
```

or by setting the parameter `MUTATION_PROBABILITY` to `[NUM]` in either a parameters file or in the params dictionary, where `[NUM]` is a float between 0 and 1. This will change the mutation probability for each codon to the probability specified. Mutation is performed over the entire genome by default, but the argument `within_used` is provided to limit mutation to only the effective length of the genome.

***NOTE** that specifying the `within_used` argument for `int_flip_per_codon` mutation will alter the probability of per-codon mutation accordingly as the used portion of the genome may be shorter than the overall length of the genome.*

### Int Flip Per Ind

Int Flip Per Ind mutation operates on linear genomes and mutates `MUTATION_EVENTS` randomly selected codons in the genome. Int Flip Per Ind mutation can be activated with the argument:

```
--mutation int_flip_per_ind
```

or by setting the parameter `MUTATION` to `int_flip_per_ind` in either a parameters file or in the params dictionary. The default mutation events is set to 1. This can be changed with the argument:

```
--mutation_events [INT]
```

or by setting the parameter `MUTATION_EVENTS` to `[INT]` in either a parameters file or in the params dictionary, where `[INT]` is an integer specifying the number of desired mutation events across the entire genome. Mutation is performed over the entire genome by default, but the argument `within_used` is provided to limit mutation to only the effective length of the genome.

***NOTE** that the parameter* `MUTATION_PROBABILITY` *does not apply to* `int_flip_per_ind` *mutation.*

### Subtree

Subtree mutation randomly selects a subtree from the overall derivation tree of an individual and mutates that subtree by building a new random subtree from the root node. Subtree mutation uses the same random derivation function as the `Grow` component of Ramped Half-Half initialisation. Subtree mutation can be activated with the argument:

```
--mutation subtree
```

or by setting the parameter `MUTATION` to `subtree` in either a parameters file or in the params dictionary.

***NOTE** that subtree mutation will **not** produce invalid individuals, i.e. given a valid individual, the mutated individual is guaranteed to be valid.*

***NOTE** that the parameter* `MUTATION_PROBABILITY` *does not apply to* `subtree` *mutation, i.e. each individual is guaranteed* `MUTATION_EVENTS` *mutation events.*

### Mutation Events

The ability to specify the number of mutation events per individual is provided in PonyGE2. This works for all mutation operators currently implemented, but works slightly differently in each case. The default number of mutation events is 1 per individual. This value can be changed with the argument:

```
--mutation_events [INT]
```

or by setting the parameter `MUTATION_EVENTS` to `[INT]` in either a parameters file or in the params dictionary, where `[INT]` is an integer which specifies the number of mutation events per individual.

For subtree mutation, exactly `MUTATION_EVENTS` number of mutation events will occur. This is accomplished by calling the subtree mutation operator `MUTATION_EVENTS` times for each individual. ***NOTE*** *that this means that the same subtree can be mutated multiple times.*

For linear genome mutation operators, the `MUTATION_EVENTS` parameter operates slightly differently to subtree mutation. As detailed previously, with the linear mutation operator `int_flip_per_ind`, exactly `MUTATION_EVENTS` mutations will occur on the genome (i.e. there is no `MUTATION_PROBABILITY` used). However, with `int_flip_per_codon` mutation the `MUTATION_EVENTS` parameter will only affect the *probability* of per-codon mutation events occurring. This is done by changing the probability of mutation to `MUTATION_EVENTS` divided by the length of the genome.

***NOTE*** *that the default value for* `MUTATION_EVENTS` *is 1, meaning to the default mutation probability for* `int_flip_per_codon` *mutation is 1 divided by the length of the genome unless either* `MUTATION_EVENTS` *or* `MUTATION_PROBABILITY` *are explicitly specified.*

***NOTE*** *that the parameters* `MUTATION_EVENTS` *and* `MUTATION_PROBABILITY` *cannot both be specified for* `int_flip_per_codon` *mutation as these are mutually exclusive parameters in this case.*

# Fitness Functions

Evaluation of individuals in PonyGE2 is carried out by the specified fitness function. All fitness functions are located in `src/fitness`. Fitness functions in PonyGE2 must be a class instance contained in its own separate file.

***NOTE** that fitness function classes in PonyGE2 **must** have the same name as their containing file. For example, a fitness function class titled "my_ff" **must** be contained in a file called "my_ff.py". Doing otherwise will produce an error.*

## The Base Fitness Function Class

All fitness functions in PonyGE2 inherit from a main base fitness function class, contained in `fitness.base_ff_classes.base_ff`. Using object inheritance allows for various checks and balances to be implemented in the base class, which makes writing new fitness functions a more streamlined and easier process.

***NOTE** that while all example fitness functions in PonyGE2 inherit from the `fitness.base_ff_classes.base_ff` fitness function class, it is not strictly necessary that any new fitness functions do so. However, it is strongly recommended that new fitness functions follow this convention.*

Inheriting from the `fitness.base_ff_classes.base_ff` fitness function class has a number of benefits:

1. The base fitness function class defines the [[default fitness|Evaluation#default-fitness]] correctly as `NaN`. Any fitness function that inherits from the base class automatically inherits the correct default fitness value, and as such does not have to implement/define the default fitness.
2. The base fitness function class defines the [[fitness maximisation/minimisation|Evaluation#fitness-maximisationminimisation]] attribute as `False`. Any fitness function that inherits from the base class is automatically set up to minimise fitness values, and as such does not have to implement/define the `maximise` attribute if this is the aim of the new fitness function.

The base fitness function class defines the framework for any new fitness function which seeks to inherit from the base class. There are three methods defined in the base class:

1. `__init__(self)`

   The `__init__()` method contains all code to be executed during the initialisation of the fitness function. This typically includes the definitions of class attributes or instance variables such as [[default fitness|Evaluation#default-fitness]] or [[fitness maximisation/minimisation|Evaluation#fitness-maximisationminimisation]].

2. `__call__(self, ind, **kwargs)`

   The `__call__()` method of a fitness function class is called when an individual is evaluated. Fitness functions in PonyGE2 are called automatically from `representation.individual.Individual.evaluate`. In the base fitness function class, the `__call__()` method contains a number of exception handling clauses designed to catch a number of pre-determined errors (see the [[default fitness|Evaluation#default-fitness]] section for more information). The `__call__()` method of the base fitness function class directly calls the `evaluate()` method of the class.

3. `evaluate(self, ind, **kwargs)`

   The `evaluate()` method of a fitness function contains the code that is used to directly evaluate the phenotype string of an individual and return an appropriate fitness value.

### Inheritance from the Base Class

Through the magic of object inheritance, any new fitness function which inherits from the base fitness function class only needs to define ***two*** of the above three methods:

1. A new `__init__(self)` method. As with the base class, this method can be used to define class attributes or instance variables, and can also be used to execute potentially computationally expensive code such as the importing/reading of [[dataset|Evaluation#datasets]] files.

   ***NOTE*** *that any new fitness function which inherits from the base fitness function class* ***must*** *initialise the* `super()` *class (i.e. the* `fitness.base_ff_classes.base_ff` *from whence inheritance comes) in the* `__init__()` *of the new fitness function with the following line of code:*

   ```
   super().__init__()
   ```

   *See all PonyGE2 fitness functions for examples of this.*

2. A new `evaluate(self, ind, **kwargs)` method. A skeleton `evaluate()` method has been made available in the base fitness function class, for the sole explicit function of being over-written by any class which inherits from the base class. Defining a new `evaluate()` method in a new fitness function will over-write the existing `evaluate()` method in the base class, but will still retain the `__call__()` method of the base class. Thus, when the call to evaluate an individual is made (from `representation.individual.Individual.evaluate`), the original `__call__()` method of the base fitness function class is first called, and the new `evaluate()` method of the new fitness function is then called from within the `__call__()` method.

   ***NOTE*** *that any new fitness function which inherits from the base fitness function class and which over-writes the base* `evaluate()` *method* ***must*** *use the same method signature as the original base* `evaluate()` *method, i.e. the definition of the* `evaluate()` *method in any fitness function which inherits from the base class* ***must*** *be:*

   ```
   def evalaute(self, ind, **kwargs):
   ```

   *Doing otherwise will produce an error.*

   ***NOTE*** *that any new fitness function which* ***does not*** *inherit from the base fitness function class* ***does not*** *require an* `evaluate()` *method to be defined.*

Any new fitness function which inherits from the base class ***does not*** need to define a new `__call__()` method. Doing so will over-write the original `__call__()` method of the base class, and will essentially render inheritance from the base class useless, as all exception handling checks from the base class will be over-written. This is why a dedicated skeleton `evaluate()` method has been written for the base class.

## The Fitness Function Template

To make the process of writing new fitness functions as easy as possible, a blank fitness function template for new implementations has been included with PonyGE2. This template can be found at `fitness.base_ff_classes.ff_template`. The fitness function template is set up with all necessary code, including an optional overwrite for [[fitness maximisation/minimisation|Evaluation#fitness-maximisationminimisation]], necessary initialisation of the `super()` base fitness function class (from whence the template inherits), and a basic definition of the necessary `evaluate()` function. All that is required to write a new fitness function using this template is to create a copy of the template, name it appropriately (remember to give the file and class the same name), and insert the necessary code for evaluating the phenotype of an individual into the `evaluate()` function. See all PonyGE2 fitness functions for examples of this.

For more information on *how* to evaluate/execute an individual phenotype, see the Grammars section on [[evaluating -vs- executing phenotypes|Grammars#special-grammars-expressions-vs-code]]

***NOTE** that the fitness function template does **NOT** include a definition of a `__call__()` method, as doing so would over-write the same method from the base class.*

It is strongly encouraged for users to base new fitness functions from this template.

## Default Fitness

The use of a "default fitness" is used in PonyGE2 for a number of reasons:

1. With traditional GE, [[invalid individuals|Representation#invalid-individuals]] are individuals which cannot produce a valid phenotype string from their given genome. Since they do not have a valid phenotype, they cannot be executed/evaluated correctly, and as such can not be assigned a fitness value in the regular way. Note that this does not occur with the use of [[context-aware "intelligent" operators|Representation#context-aware-intelligent-operations]].
2. Grammars can often produce large individuals, or individuals with complex behaviour. While valid phenotype strings may be generated, in some cases (such as nested exponential terms, or division by zero), these phenotypes may not evaluate correctly and may raise Exceptions or Errors. Examples of such errors include `FloatingPointError`, `ZeroDivisionError`, `OverflowError`, or `MemoryError`. These individuals are also counted as "runtime_error" individuals in the stats.

Since all individuals require *some* fitness value to be assigned to them, this is accomplished in the above cases through the use of a "default" fitness value. The default "default fitness" value in the base PonyGE2 fitness function class is `NaN`. This allows for easy filtering during the statistical gathering process, to ensure that invalid individuals are not so included.

Since a default fitness value is defined in the [[base fitness function class|Evaluation#the-base-fitness-function-class]], any fitness function which inherits from the base class does not need to define a new default fitness value. However, any newly implemented fitness function which ***does not*** inherit from the base class ***must*** define a `default_fitness` attribute.

***NOTE*** *that the use of "default fitness" in PonyGE2 is distinct from the concept of a fitness "penalty", as might be found in e.g. constrained optimisation problems.*

## Fitness Maximisation/Minimisation

A key concept in any optimisation technique is the notion of a fitness gradient, which allows solutions to be ranked in binary terms of "better" or "worse" on any single given objective. The definition of how one single objective fitness value is considered "better" or "worse" than another fitness value is done through maximisation or minimisation.

Put simply, if the objective of a single given fitness function is to maximise fitness, then given two different individuals with valid (i.e. not [[default fitness|Evaluation#default-fitness]]) fitness values, the individual with the greater fitness value will be deemed the "better" solution. Conversely, if *minimisation* is the goal of the fitness function, then the individual with the lowest fitness value will be deemed the "better" of the two.

Fitness maximisation or minimisation is handled in PonyGE2 by the `maximise` attribute of the fitness function. If the `maximise` attribute is set to `True`, then the fitness function will seek to *maximise* fitness. Conversely, if the `maximise` attribute is set to `False`, then the fitness function will seek to *minimise* fitness. Fitness comparison between two individuals is done through the definition of a customised `__lt__(self, other)` function in the `representation.individual.Individual` class.

Since the `maximise` attribute is set to `False` in the [[base fitness function class|Evaluation#the-base-fitness-function-class]], any fitness function which inherits from the base class is automatically set up to ***minimise*** fitness by default. However, if you are either seeking to ***maximise*** fitness, or if your newly implemented fitness function ***does not*** inherit from the base class, you ***must*** define a `maximise` attribute in the new fitness function. The [[fitness function template|Evaluation#the-fitness-function-template]] contains the code to over-write the base maximisation/minimisation attribute should you so desire to use it.

## Additional Functionality

### Dynamic ranging of variables within the grammar

As detailed in the section on [[variable ranges in grammars|Grammars#variable-ranges-in-grammars]], it is possible to set a specific `n_vars` attribute in the fitness function, which can then be accessed and used in the grammar through the use of `GE_RANGE:dataset_n_vars`. This can be especially useful for cases where a dataset has a large number of input variables. Traditional grammar parsers would require each variable to be explicitly mentioned in the grammar, e.g.:

```
<vars> ::= x[0] | x[1] | x[2] | x[3] | x[4] | x[5] | ...
```

For datasets with a high number of variables, this can get quite tedious, and can be difficult to read. The use of the `n_vars` attribute in the fitness function allows you to set the number of variables in the dataset. In our fitness function, we would define:

```
self.n_vars = [THE DESIRED NUMBER OF VARIABLES]
```

The `<vars>` line from the grammar above could then be replaced with:

```
<vars>   ::= x[<varidx>]
<varidx> ::= GE_RANGE:dataset_n_vars
```

See `fitness.supervised_learning.supervised_learning` and the `supervised_learning/supervised_learning.bnf` grammar for a working usage example.

### Evaluating on Training and Test Data

For fitness functions which evaluate solutions on either training or test [[datasets|Evaluation#datasets]] (e.g. supervised learning problems), PonyGE2 can automatically evaluate the best solution at the end of the evolutionary run on unseen "test" data. This is done through the use of a `training_test` attribute in the fitness function, which PonyGE2 checks for during the stats collecting process in `stats.stats.get_soo_stats` or `stats.stats.get_moo_stats`.

PonyGE2 switches between evaluation on training or test data with the use of an optional input argument of `"dist"`. By default, the `dist` argument is set to `"training"`. However, for test evaluation, the input argument is changed to `"test"`. Optional input arguments for PonyGE2 fitness functions are handled through the `**kwargs` input. In order to set a default input flag for `"dist"` to `"training"`, the following line of code is required in the `evaluate()` method of the fitness function:

```
dist = kwargs.get('dist', 'training')
```

This line defines a `dist` attribute for the fitness function, which can then be used in the `evaluate()` method to automatically set the training or test dataset upon which evaluation is to take place. An example of how this can be achieved is provided below:

```
if dist == "training":
    # Set training datasets to be used.
    x = self.training_in
    y = self.training_exp

elif dist == "test":
    # Set test datasets to be used.
    x = self.test_in
    y = self.test_exp
```

Note that the above example is contingent on `self.training_in`, `self.training_exp`, `self.test_in`, and `self.test_exp` variables being defined and set in the `__init__()` method of the fitness function. See `fitness.supervised_learning.supervised_learning` for a working usage example.

## Error Metrics

Some supervised learning fitness functions require an error metric (e.g. mean-squared error, or `mse`) to be specified. While the default `regression` and `classification` fitness functions provided in PonyGE2 have their error metrics set to `rnse` and `f1_score` respectively by default, it is possible to specify new error metrics with the argument:

```
--error_metric [ERROR_METRIC_NAME]
```

or by setting the parameter `ERROR_METRIC` to `[ERROR_METRIC_NAME]` in either a parameters file or in the params dictionary, where `[ERROR_METRIC_NAME]` is a string specifying the name of the desired error metric. A list of currently implemented error metrics is available in `utilities.fitness.error_metric`.

***NOTE*** *that for some supervised learning problems that use specified error metrics (e.g. mean-squared error), these error metrics have their own* `maximise` *attributes. This* `maximise` *attribute is automatically used by the specified supervised learning fitness function in place of the standard [[maximise|Evaluation#fitness-maximisationminimisation]] attribute.*

## Datasets

Some supervised learning fitness functions may require a dataset across which to be evaluated. Datasets for PonyGE2 are saved in the `datasets` folder. Most supervised learning problems require two datasets: training and test data. These are specified independently in order to allow for problems to be run only on training data (i.e. if no test dataset is specified, the best individual at the end of a run will *not* be evaluated on unseen test data).

Training datasets can be specified with the argument:

```
--dataset_train [DATASET_NAME]
```

or by setting the parameter `DATASET_TRAIN` to `[DATASET_NAME]` in either a parameters file or in the params dictionary, where `[DATASET_NAME]` is a string specifying the full file name including file extension of the desired training dataset.

Testing datasets can be specified with the argument:

```
--dataset_test [DATASET_NAME]
```

or by setting the parameter `DATASET_TEST` to `[DATASET_NAME]` in either a parameters file or in the params dictionary, where `[DATASET_NAME]` is a string specifying the full file name including file extension of the desired testing dataset.

If using any of the tree example supervised learning fitness functions (`supervised_learning.supervised_learning`, `supervised_learning.regression`, or `supervised_learning.classification`), datasets are automatically loaded by the `supervised_learning.supervised_learning` fitness function class using the `utilities.fitness.get_data.get_data` function. If implementing a new fitness function, then this `get_data` function should be called during the `__init__()` method of the fitness function:

```
self.training_in, self.training_exp, self.test_in, self.test_exp = \
    get_data(params['DATASET_TRAIN'], params['DATASET_TEST'])
```

***NOTE*** *that when using training and test datasets for evaluation, you* ***must*** *include a* `self.training_test = True` *attribute in the fitness function, as detailed [[above|Evaluation#evaluating-on-training-and-test-data]].*

***NOTE*** *that you* ***must*** *specify the file extension when specifying the dataset names.*

While it is recommended that supervised learning problems implement training *and* unseen testing data, it is not necessary to use testing data with these problems. If you wish to run PonyGE2 with no test dataset, you can simply use the argument:

```
--dataset_test None
```

***NOTE*** *that if no testing dataset is being used, there is no need to set a* `training_test` *attribute in the fitness function. This can be done automatically in the* `__init__()` *method of the fitness function with the following check:*

```
if params['DATASET_TEST']:
    self.training_test = True
```

### Dataset Delimiters

By default, PonyGE2 will try to automatically parse specified datasets using the following set of data delimiters in the following order:

1. "\t" [tab]
2. "," [comma]
3. ";" [semi-colon]
4. ":" [colon]

If the data cannot be parsed using the above separators, then PonyGE2 will default to using whitespace as the delimiter for separating data (a warning will be printed in this case). However, it is possible to directly specify the desired dataset delimiter with the argument:

```
--dataset_delimiter [DELIMITER]
```

or by setting the parameter `DATASET_DELIMITER` to `[DELIMITER]` in either a parameters file or in the params dictionary, where `[DELIMITER]` is a string specifying the desired dataset delimiter.

***NOTE** that you **do not** need to escape special characters such as "\t" with "\\t" when passing in the `--dataset_delimiter` argument from the command line. Simply specify the raw desired string.*

### Targets

Some fitness functions may require a target value. For example, the `string_match` fitness function requires a target string to match. Target strings can be specified with the argument:

```
--target [TARGET_STRING]
```

or by setting the parameter `TARGET` to `[TARGET_STRING]` in either a parameters file or in the params dictionary, where `[TARGET_STRING]` is a string specifying the desired target string. Once set, the `TARGET` string can be accessed throughout PonyGE2 via the `params` dictionary.

***NOTE** that all target values will be stored in PonyGE2 as a string by default. If a fitness function requires any data structure other than a string, the target string itself must be parsed to the desired data structure within the fitness function itself.*

***NOTE** that if this parsing of the target string is done in the `__init__()` call of the fitness function, it only needs to be done once rather than at every fitness evaluation.*

### Specifying Fitness Functions

Fitness functions can be specified with the argument:

```
--fitness_function [FITNESS_FUNCTION_NAME]
```

or by setting the parameter `FITNESS_FUNCTION` to `[FITNESS_FUNCTION_NAME]` in either a parameters file or in the params dictionary, where `[FITNESS_FUNCTION_NAME]` is a string specifying the name of the desired fitness function.

# Multiple Objective Optimisation

Multiple objective optimisation (MOO) in the form of the Non-Dominated Sorting Genetic Algorithm 2 (NSGA2) [[[Deb et al., 2002|References]]] has been included with PonyGE2. The developers of PonyGE2 have sought to implement multiple objective optimisation in the simplest and most intuitive way possible.

The core concept of how MOO is implemented in PonyGE2 requires a single fitness function for each desired objective fitness. Each individual fitness function is then a standalone fitness function (i.e., PonyGE2 should be capable of being used with these fitness functions in a normal single-objective manner) with all of the structure and attributes of a standard fitness function (as detailed [[above|Evaluation#the-base-fitness-function-class]]). The user can simply then specify all desired fitness functions to be used, and PonyGE2 will automatically use NSGA2.

## Example Problems

Two example problems are provided:

1. Regression with bloat control

   This example runs the standard Vladislavleva-4 [[regression|Example-Problems#regression]] example, but with a second fitness function that aims to minimise the number of nodes in the derivation tree. To try this problem, specify the following command-line argument:

   ```
   --parameters moo/regression_bloat_control.txt
   ```

2. An example benchmark multi-objective optimisation problem from [[[Zitzler, et al.|References]]] is provided. To try this problem, specify the following command-line argument:

   ```
   --parameters moo/zdt1.txt
   ```

## Specifying Multiple Fitness Functions

Multiple fitness functions are specified in PonyGE2 in the same manner that single fitness functions are specified.

### Command Line

If using the command line, fitness functions can be specified with the argument:

```
--fitness_function [FITNESS_FUNCTION_1_NAME] [FITNESS_FUNCTION_2_NAME] ...
```

where `[FITNESS_FUNCTION_1_NAME]`, `[FITNESS_FUNCTION_1_NAME]`, etc are the names of the desired individual fitness functions.

***NOTE** that when specifying multiple fitness functions from the command line it is **not** necessary to use commas or any separators between the various fitness functions. Equally, it is **not** necessary to use any form of bracketing when specifying multiple fitness functions from the command line. The command line parser of PonyGE2 handles this automatically.*

### Parameters File / Params Dictionary

Unlike with the command line, when specifying multiple fitness functions from either the `algorithm.parameters.params` dictionary or from within a parameters file, it *is* necessary to place the names of each desired fitness function into a list and to use a comma to delineate between individual fitness functions. An example of this would be:

```
FITNESS_FUNCTION: [[FITNESS_FUNCTION_1_NAME], [FITNESS_FUNCTION_2_NAME]]
```

where `[FITNESS_FUNCTION_1_NAME]`, `[FITNESS_FUNCTION_1_NAME]`, etc are the names of the desired individual fitness functions.

***NOTE** that when specifying multiple fitness functions from either the params dictionary or in a parameters file that it **is** necessary to place the desired fitness functions in a list, i.e. with bracketing and commas.*

Example parameters files for multi-objective optimisation problems can be seen in the `parameters/moo` folder.

## Multi-Objective Fitness Function Implementation

PonyGE2 handles the use of multiple fitness functions through a dedicated base MOO fitness function, contained in `fitness.base_ff_classes.moo_ff`. When [[multiple fitness functions are specified|Evaluation#specifying-multiple-fitness-functions]], this fitness function class is automatically used as a holding class for each individual fitness function. Fitness functions are handled as a list of individual fitness function class instances. When evaluation of an individual is called, each fitness function within this list is called in turn in order to produce an array of fitness values for the given individual. Since each individual fitness function acts as a standalone fitness function, this means that ***any*** fitness function in PonyGE2 can be used as a fitness component for multiple objective optimisation.

***NOTE** that the base MOO fitness function class automatically initialises each of the specified fitness functions during its own initialisation.*

Since the base MOO fitness function class generates a list of individual (single objective) fitness functions, the base MOO fitness function does not need to inherit from the [[base fitness function class|Evaluation#the-base-fitness-function-class]] as each individual (single objective) fitness function so inherits. Since each individual fitness function inherits from the base class, the only checks that are required in the MOO fitness class pertain to its multiple objective nature and the handling of default fitness values.

### Default Fitness

Since the actual implementation of multiple objective optimisation in PonyGE2 is defined as a list of individual fitness functions, the [[default fitness|Evaluation#default-fitness]] must also be handled as a list. The base MOO fitness function class defines its own `default_fitness` attribute as a list of all of the individual `default_fitness` values of the individual specified fitness functions.

With regards to the setting of default fitness values, if any single fitness value for any given fitness function returns a `NaN` value (i.e. the default fitness of the base class), then all fitness values for that particular individual are set to their default fitness values (i.e. all fitness are given the `default_fitness` value of their respective fitness functions). Furthermore, if any single fitness function encounters a runtime error, the entire individual is counted by the stats as a "runtime_error" individual (Note that a check is in place to ensure that violations across multiple fitness functions are not recorded more than once for a single individual).

## Multi-Objective Operators

Since multiple objective optimisation creates a pareto front of individuals, standard single-objective operators which implement comparisons between individuals (e.g. selection and replacement) are not compatible with MOO problems. As such, NSGA2 has its own implementation of [[selection|Selection#nsga2]] and [[replacement|Replacement#nsga2]] operators. However, these are not set by default if multiple fitness functions are used. As with normal selection and replacement operators, MOO-compatible operators must be directly specified.

***NOTE*** *that MOO-compatible operators require a* `multi_objective = True` *attribute. Failure to do so will result in an Error.*

## Multi-Objective Statistics

Aside from [[selection|Selection#nsga2]] and [[replacement|Replacement#nsga2]] operators, one of the major distinctions between single objective optimisation and multiple-objective optimisation is how statistics are collected and reported on the population. As such, the use of multiple objective optimisation requires a dedicated statistics handler. Such a function is provided in `stats.stats.get_moo_stats`. This MOO stats parser is automatically used by PonyGE2 in the event of multiple fitness functions being used.

The main differences between single objective and multiple objective statistics are:

1. A single "best ever" individual is not tracked by `utilities.stats.trackers.best_ever`. Instead, the first (non-dominated) pareto front of individuals is tracked therein. This is also true for the "best_fitness" statistic, which is replaced with the "first_fronts" statistic. This new statistic tracks the number of individuals on the first (non-dominated) pareto front.

2. Since the single objective "best_fitness" is no longer tracked, normal fitness plotting can not be done with multiple objectives. As such, fitness plotting is replaced with the plotting of all fitness values on the first (non-dominated) pareto front across each generation. Fronts are color-coordinated for visual distinction. Axes are automatically labelled with the respective names of the individual fitness functions.

   ***NOTE*** *that at present fitness plotting is only enabled in PonyGe2 for problems with either one or two objectives. Problems with three or more objectives do not yet have fitness plotting enabled.*

3. Since the single "best ever" individual is replaced with a list of all individuals on the first (non-dominated) front, saving of the single best individual with multi-objective optimisation saves a folder containing all individuals on the first front.

4. Fitness metrics across the entire population (such as average fitness) cannot be tracked. The "ave_fitness" statistic is replaced with the "pareto_fronts" statistic. This new statistic tracks the total number of pareto fronts.

# Multicore evaluation

Evaluation of a population of individuals can be done in series (single core evaluation) or in parallel (multi core evaluation). Multicore evaluation can be activated with the argument:

```
--multicore
```

or by setting the parameter `MULTICORE` to `True` in either a parameters file or in the params dictionary.

Additionally, the number of processor cores used for multicore evaluation can be controlled with the argument:

```
--cores [INT]
```

or by setting the parameter `CORES` to `[INT]` in either a parameters file or in the params dictionary, where `[INT]` is an integer which specifies the number of cores used for fitness evaluations. The default value is to use all available cores.

***NOTE** that multicore evaluation may require a slightly longer setup time on Windows operating systems. For more information, see the documentation string of the function* `utilities.algorithm.initialise_run.pool_init()`.

***NOTE** that multicore evaluations may not necessarily improve computational runtime for small problems as a certain overhead is necessary to run the multicore evaluation process.*

***NOTE** that for smaller problems fitness evaluations may not necessarily present a bottleneck in terms of computational run-time. It is advised to use a python profiler to ascertain whether or not fitness evaluations present such a bottleneck. If this is the case, multicore evaluation may improve the run-time of a single evolutionary run.*

***NOTE** that when running batches of multiple experiments, it will **always** be faster to run multiple single-core experiments in parallel, rather than multiple multi-core experiments in series.*

# Caching

Caching is provided in PonyGE2 to save on fitness evaluations and to track the number of unique solutions encountered during an evolutionary run. Cached individuals have their fitness stored in the `utilities.trackers.cache` dictionary. Dictionary keys are the string of the phenotype. Using the phenotype string as the key for the cache means that duplicate phenotypes will not be evaluated, even if their genetic makeup is distinct (see the [[many-to-one mapping process|Representation#many-to-one-mapping]] section).

Caching can be activated with the argument:

```
--cache
```

or by setting the parameter `CACHE` to `True` in either a parameters file or in the params dictionary.

There are currently three optional extras for use with the cache, all of which are activated in `fitness.evaluation.evaluate_fitness`:

## Fitness Lookup

This is the default case when caching is activated. Individuals whose phenotypes match one which has already been evaluated have the previous fitness read directly from the cache, thus saving fitness evaluations. Fitness lookup can be *de*-activated with:

```
--dont_lookup_fitness
```

or by setting the parameter `LOOKUP_FITNESS` to `False` in either a parameters file or in the params dictionary.

## Fitness Penalty

Individuals which have already been evaluated (i.e. duplicate individuals) are given a default bad fitness. The fitness to be assigned is the default fitness value specified in the fitness function. This parameter can be activated with the argument:

```
--lookup_bad_fitness
```

or by setting the parameter `LOOKUP_BAD_FITNESS` to `True` in either a parameters file or in the params dictionary.

## Mutate Duplicates

Individuals which have already been evaluated (i.e. duplicate individuals) are mutated to produce new unique individuals which have not been encountered yet by the search process. This parameter can be activated with the argument:

```
--mutate_duplicates
```

or by setting the parameter `MUTATE_DUPLICATES` to `True` in either a parameters file or in the params dictionary.

***NOTE** that the various caching options are **mutually exclusive**. For example, you cannot specify* `--mutate_duplicates` *with* `--lookup_bad_fitness`.

***NOTE** that if a linear mutation operator is used, it is possible for PonyGE2 to get stuck in an infinite loop if* `--mutate_duplicates` *is specified.*

The replacement strategy for an Evolutionary Algorithm defines which parents and children survive into the next generation.

## Generational

Generational replacement replaces the entire parent population with the newly generated child population at every generation. Generational replacement can be activated with the argument:

```
--replacement generational
```

or by setting the parameter `REPLACEMENT` to `generational` in either a parameters file or in the params dictionary.

### Elitism

Generational replacement is most commonly used in conjunction with elitism. With elitism, the best `[ELITE_SIZE]` individuals in the parent population are copied over unchanged to the next generation. Elitism ensures continuity of the best ever solution at all stages through the evolutionary process, and allows for the best solution to be updated at each generation.

The number of children created at each generation is known as the `GENERATION_SIZE`, and is equal to `[POPULATION_SIZE]` - `[ELITE_SIZE]`. This parameter is set automatically by PonyGE2.

The default number of elites is 1 percent of the population size. This value can be changed with the argument:

```
--elite_size [INT]
```

or by setting the parameter `ELITE_SIZE` to `[INT]` in either a parameters file or in the params dictionary, where `[INT]` is an integer which specifies the number of elites to be saved between generations.

## Steady State

With steady state replacement, only 2 children are created by the evolutionary process at each evolutionary step (i.e. the `GENERATION_SIZE` is automatically set to 2). Steady state replacement first selects two parents, performs crossover on them to produce two children, mutates and evaluates these children, and then replaces the two worst individuals in the original population with the new children, regardless of whether or not these children are fitter than the individuals they replace. As such, steady state replacement implements its own specialised `step` loop.

At each generation, steady state replacement continues until `POPULATION_SIZE` children have been created and inserted into the original population. Adopting a steady state replacement strategy ensures that successive populations overlap to a significant degree (i.e. parents and their children can co-exist). This requires less memory as only one population of individuals needs to be maintained at any given point in time. This strategy also allows the evolutionary process to exploit good solutions as soon as they appear.

Steady state replacement can be activated with the argument:

```
--replacement steady_state
```

or by setting the parameter `REPLACEMENT` to `steady_state` in either a parameters file or in the params dictionary.

## NSGA2

NSGA2 replacement replaces the old population with the new population based on crowding distance per pareto front. The crowding distance comparison of the individuals ensures dispersion on the pareto fronts. Elitism is implicitly maintained by the existance of the first pareto front. For more details on the NSGA-II operator see [[Deb et al., 2002|References]].

Activate with:

```
--replacement nsga2_replacement
```

or by setting the parameter `REPLACEMENT` to `nsga2_replacement` in either a parameters file or in the params dictionary.

***NOTE** that NSGA2 replacement can only be used in conjunction with [[multiple objective optimisation|Evaluation#multiple-objective-optimisation]] problems.*

***NOTE** that multi-objective optimisation compatible operators require a* `multi_objective = True` *attribute.*

A number of example problems are currently provided with PonyGE2:

1. [[String-match|Example-Problems#string-match]]
2. [[Regression|Example-Problems#regression]]
3. [[Classification|Example-Problems#classification]]
4. [[Pymax|Example-Problems#pymax]]
5. [[Program synthesis|Example-Problems#program-synthesis]]
6. [[Genetic Improvement of Regex Runtime Performance|Example-Problems#genetic-improvement-of-regex-runtime-performance]]
7. [[Multiple Objective Optimisation|Example-Problems#multiple-objective-optimisation]]

A brief description is given below of each problem, along with the command-line arguments necessary to call each problem. The developers of PonyGE2 encourage users to test out the various different operators and options available within PonyGE2 using these example problems in order to gain an appreciation of how they work.

## String-match

The grammar specifies words as lists of vowels and consonants along with special characters. The aim is to match a target string.

To use it, specify the following command-line argument:

```
--parameters string_match.txt
```

The default string match target is `Hello world!`, but this can be changed with the `--target` argument.

## Regression

The grammar generates a symbolic function composed of standard mathematical operations and a set of variables. This function is then evaluated using a pre-defined set of inputs, given in the datasets folder. Each problem suite has a unique set of inputs. The aim is to minimise some error between the expected output of the function and the desired output specified in the datasets. This is the default problem for PonyGE.

To try this problem, specify the following command-line argument:

```
--parameters regression.txt
```

The default regression problem is `Vladislavleva4`, but this can be changed with the `--grammar_file`, `--dataset_train` and `--dataset_test` arguments.

### Optimisation of constants

When running supervised learning problems like regression and classification, PonyGE2 can attempt to optimise constants. These constants will exist in the individual's phenotype string as `c[0]`, `c[1]`, etc., so the grammar should create such terms. Then, if we specify the following argument:

```
--optimize_constants
```

or set the parameter `OPTIMIZE_CONSTANTS` to `True` in either a parameters file or the params dictionary, then PonyGE2 will perform a gradient descent on the values of the constants (given the data) to minimise the error (as measured by the chosen `--error_metric`). It will use the L-BFGS-B method. By default, optimisation of constants is switched off.

## Classification

Classification can be considered a special case of symbolic regression but with a different error metric. Like with regression, the grammar generates a symbolic function composed of standard mathematical operations and a set of variables. This function is then evaluated using a pre-defined set of inputs, given in the datasets folder. Each problem suite has a unique set of inputs. The aim is to minimise some classification error between the expected output of the function and the desired output specified in the datasets.

To try this problem, specify the following command-line argument:

```
--parameters classification.txt
```

The default classification problem is `Banknote`, but this can be changed with the `--grammar_file`, `--dataset_train` and `--dataset_test` arguments.

## Pymax

One of the strongest aspects of a grammatical mapping approach such as PonyGE2 is the ability to generate executable computer programs in an arbitrary language [[[O'Neill & Ryan, 2003|References]]]. In order to demonstrate this in the simplest way possible, we have included an example Python programming problem.

The `Pymax` problem is a traditional maximisation problem, where the goal is to produce as large a number as possible. However, instead of encoding the grammar in a symbolic manner and evaluating the result, we have encoded the grammar for the `Pymax` problem as a basic Python programming example. The phenotypes generated by this grammar are executable python functions, whose outputs represent the fitness value of the individual. Users are encouraged to examine the `pymax.pybnf` grammar and the resultant individual phenotypes to gain an understanding of how grammars can be used to generate such arbitrary programs.

To try this problem, specify the following command-line argument:

```
--parameters pymax.txt
```

## Program synthesis

The General Program Synthesis Benchmark Suite is available in PonyGE2. Grammars and datasets have been provided by HeuristicLab.CFGGP. The individuals produce executable Python code.

To try this problem, specify the following command-line argument:

```
--parameters progsys.txt
```

***NOTE** that [[multicore evaluation|Evaluation#multicore-evaluation]] is **not** currently supported for this type of problem.*

## Genetic Improvement of Regex Runtime Performance

An example of Genetic Improvement for software engineering is given for program improvement of Regular Expressions (regexs). The given examples improve existing regexes by seeding them into the population. The fitness function measures runtime and functionality of regexs.

The fitness function for the regex performance improvement problem has a number of sub-modules and programs which are used for generating test cases and for timing the execution of candidate programs. This is an exmaple of how a fitness function can be extended in PonyGE2 with customised modules and classes to perform complex tasks.

To try this problem, specify the following command-line argument:

```
--parameters regex_improvement.txt
```

***NOTE** that [[multicore evaluation|Evaluation#multicore-evaluation]] is **not** currently supported for this type of problem.*

## Multiple Objective Optimisation

As detailed in the section on [[multiple objective optimisation|Evaluation#multiple-objective-optimisation]], PonyGE2 comes as standard with an implementation of NSGA2 to optimise multiple fitness functions simultaneously.

Two example problems are provided:

1. Regression with bloat control

   This example runs the standard Vladislavleva-4 [[regression|Example-Problems#regression]] example, but with a second fitness function that aims to minimise the number of nodes in the derivation tree. To try this problem, specify the following command-line argument:

```
--parameters moo/regression_bloat_control.txt
```

***NOTE** that it is advisable to use [[subtree mutation|Variation#subtree-1]] in combination with the [[mutate duplicates|Evaluation#mutate-duplicates]] option with this example problem.*

2. An example benchmark multi-objective optimisation problem from [[[Zitzler, et al.|References]]] is provided. To try this problem, specify the following command-line argument:

```
--parameters moo/zdt1.txt
```

## Adding New Problems

---

It has been made as simple as possible to add new problems to PonyGE. To add in a new problem, you will need:

1. a new [[grammar file|Grammars]] specific to your problem,
2. a new [[fitness function|Evaluation#the-fitness-function-template]] (if you don't want to use a previously existing one), and
3. if you are doing supervised learning then you may also need to add some new [[datasets|Evaluation#datasets]].

A [[template|Evaluation#the-fitness-function-template]] for new fitness functions is provided in `fitness.base_ff_classes.ff_template`. This template allows for new fitness functions to be easily and quickly created in an easily compatible and extensible manner.

***NOTE** that it may be beneficial to create a **new parameters file** for any new problem.*

# Seeding GE Runs with target solutions

---

Combining the [[GE LR Parser|Scripts#ge-lr-parser]] with the full PonyGE2 library, it is possible to parse a target string into a GE individual and then to seed an evolutionary run of PonyGE2 with that individual. Provision is made in PonyGE2 to allow for the seeding of as many target individuals as desired into an evolutionary run.

There are two ways to seed individuals into a PonyGE2 run:

## 1. Seeding runs with a single target solution

If a single target phenotype string is to be included into the initial population, users can specify the argument:

```
--reverse_mapping_target [TARGET_STRING]
```

or set the parameter `REVERSE_MAPPING_TARGET` to `[TARGET_STRING]` in either a parameters file or in the params dictionary, where `[TARGET_STRING]` is the phenoytpe string specifying the target string to be parsed by the GE LR Parser into a GE individual.

***NOTE*** *that as with the GE LR Parser described above, a compatible grammar file needs to be specified along with the target string. If the target string cannot be parsed using the specified grammar, an error will occur.*

## 2. Seeding runs with one or more target solutions

Alternatively, if one or more target individuals are to be seeded into a GE population, a folder has been made available for saving populations of desired individuals for seeding. The root directory contains a `seeds` folder. Any number of desired target individuals for seeding can be saved in ***separate text files*** within a unique folder in the scripts directory. This target seed folder can then be specified with the argument:

```
--target_seed_folder [TARGET_SEED_FOLDER]
```

or by setting the parameter `TARGET_SEED_FOLDER` to `[TARGET_SEED_FOLDER]` in either a parameters file or in the params dictionary, where `[TARGET_SEED_FOLDER]` is the name of the target folder within the `scripts` directory which contains target seed individuals.

PonyGE2 currently supports four formats for saving and re-loading of such individuals (examples of each are given in the `seeds/example_pop` folder):

1. (`example_1.txt` in `seeds/example_pop`) PonyGE2 can re-load "best.txt" outputs from previous PonyGE2 runs. These files contain the saved genotypes and phenotypes of the best solution evolved over the course of an evolutionary run. Re-using these output files greatly improves the seeding process, as the genotypes can be quickly used to re-map the exact identical individual evolved by

PonyGE2. If possible, this is the preferred option for seeding populations as the use of genomes to re-build previous individuals guarantees the same genetic information will be retained.
2. (`example_2.txt` in `seeds/example_pop`) Target phenotypes can be saved as a simple text file with a single header of "`Phenotype:`", followed by the phenotype string itself on the following line. The phenotype will then be parsed into a PonyGE2 individual using the GE LR Parser.
3. (`example_3.txt` in `seeds/example_pop`) Target genotypes can be saved as a simple text file with a single header of "`Genotype:`", followed by the genotype itself on the following line. The genotype will then be mapped into a PonyGE2 individual using the normal GE mapping process. As with option 1 above, this will result in an identical PonyGE2 individual being re-created from the specified genome.
4. (`example_4.txt` in `seeds/example_pop`) Target phenotypes can be saved as a simple text file where the ***only*** content of the file is the phenotype string itself (i.e. no descriptive text, headers, comments, etc). The content of these files will then be parsed into PonyGE2 individuals using the GE LR Parser.

***NOTE*** *that the names of individual files contained in a specified target population folder in the `seeds` directory **do not matter**. These files can be named however so desired.*

***NOTE*** *that as with the GE LR Parser described above, a compatible grammar file needs to be specified along with the target `seeds` folder. If the target string cannot be parsed using the specified grammar, an error will occur. If the target genotype results in a different phenotype to that specified, an error will occur.*

***NOTE*** *that at present, phenotypes spanning multiple lines can only be parsed correctly using file format 4 above, i.e. the phenotype string constitutes the sole information in the file. If a genotype exists for such phenotypes, best practice is to use the genotype to seed the solution using file format 3 above, i.e. discard the phenotype string and allow the genotype to re-produce it.*

## Initialisation

All standard [[initialisation|Initialisation]] techniques in PonyGE2 are compatible with seeding evolutionary runs with target individuals. When seeding a run with target solutions in conjunction with a known initialisation technique (e.g. [[Ramped Half-Half|Initialisation#ramped-half-half]]), the initial population size generated by the specified initialisation technique is reduced from the original specified `POPULATION_SIZE` parameter by the total number of seed individuals. This ensures that the total size of the initial generation matches the specified `POPULATION_SIZE` parameter.

An additional initialisation option is also included in PonyGE2 which may be of some use in the case of Genetic Improvement. An option is available to initialise the entire population with only identical copies of the specified seed individual (or individuals). If only one target seed is specified, the initial population will consist of `POPULATION_SIZE` copies of that individual. If multiple target seeds are specified, the initial population will consist of equal amounts of copies of each specified seed. This option can be specified with the argument:

```
--initialisation seed_individuals
```

or by setting the parameter `INITIALISATION` to `seed_individuals` in either a parameters file or in the params dictionary.

## Examples

An example parameters file for seeding runs with a number of individuals has been included in the parameters folder under `seed_run_target.txt`. An example folder named `example_pop` with a range of compatible formatting types for seeding target solutions is included in the `seeds` directory.

## Re-creating PonyGE2 runs

It is possible to set the random seeds for the various random number generators (RNGs) used by PonyGE2 in order to exactly re-create any given evolutionary run. All PonyGE2 runs by default save their random seeds. By simply specifying the argument:

```
--random_seed [RANDOM_SEED]
```

or by setting the parameter `RANDOM_SEED` to `[RANDOM_SEED]` in either a parameters file or in the params dictionary, where `[RANDOM_SEED]` is an integer which specifies the desired random seed.

At present, the main branch of PonyGE2 only uses two RNGs:

1. The core Python `random` module, and
2. The numpy `np.random` module.

Both of these RNGs are set using the same seed. When the `RANDOM_SEED` parameter is set, and provided the grammar, fitness function, and all parameters remain unchanged, then PonyGE2 will produce identical results to any previous run executed using this random seed.

Besides the main `PonyGE.py` file that can be found in the source directory, a number of extra scripts are provided with PonyGE2. These are located in the `scripts` folder. These extra scripts have been designed to work either as standalone files, or to work in tandem with PonyGE2. Various functions from within these scripts can provide extra functionality to PonyGE2.

## Basic Experiment Manager

A basic experiment manager is provided in the `scripts` folder. This experiment manager allows users to execute multiple evolutionary runs across multiple cores using python's `multiprocessing` library. Experiments are saved in `results/[EXPERIMENT_NAME]` where `[EXPERIMENT_NAME]` is a parameter which specifies the name of the experiment. This can be set with the argument:

```
--experiment_name [EXPERIMENT_NAME]
```

or by setting the parameter `EXPERIMENT_NAME` to `[EXPERIMENT_NAME]` in either a parameters file or in the params dictionary, where `[EXPERIMENT_NAME]` is a string which specifies the desired name of the experiment.

***NOTE** that the `EXPERIMENT_NAME` parameter **must** be set when using the experiment manager.*

The number of evolutionary runs to be executed can be set with the arguemt:

```
--runs [INT]
```

or by setting the parameter `RUNS` to `[INT]` in either a parameters file or in the params dictionary, where `[INT]` is an integer which specifies the number of evolutionary runs to be completed. The experiment manager initialises each evolutionary run with a different unique random seed. The random seeds for a batch of evolutionary experiments are the indexes of the individual experiments (i.e. the first run will have seed 0, the second will have seed 1, and so on up to seed `[INT] - 1`.)

***NOTE** that the experiment manager uses pythons `multiprocessing` library to launch multiple runs simultaneously. As such, it is not possible to use the experiment manager with the `MULTICORE` parameter set to `True`. If the `MULTICORE` parameter is already set to `True`, it will be turned off automatically.*

Since python uses the central `algorithm.parameters.params` and `stats.stats.stats` dictionaries to manage various aspects of individual runs, it is not currently possible to launch multiple simultaneous runs of PonyGE2 from within a Python environment as the central dictionaries would be overwritten by the concurrent processes. As such, the experiment manager calls individual PonyGE2 runs from the command line using Python's `subprocess.call()` function.

***NOTE** that all functionality available to the main PonyGE file is available to the experiment manager, i.e. all command line arguments can be used including the specification of parameters files.*

To run the experiment manager, type:

```
$ python scripts/experiment_manager.py --experiment_name [EXPERIMENT_NAME] --runs
[INT]
```

where `[EXPERIMENT_NAME]` is a string which specifies the desired name of the experiment and where `[INT]` is an integer which specifies the number of evolutionary runs to be completed.

***NOTE** that since the `[MULTICORE]` parameter in PonyGE2 does not work with Windows operating systems, at present the experiment manager will not work with Windows operating systems.*

### Post-run Analysis

A basic statistics parser is also included in the `scripts` folder. This parser is called automatically by the experiment manager upon successful completion of all specified runs, but can also be called on its own if so desired. The statistics parser generates summary `.csv` files and `.pdf` graphs for all stats generated by all runs saved in an `[EXPERIMENT_NAME]` folder.

The statistics parser extracts the `stats.tsv` files from all runs contained in the specified `[EXPERIMENT_NAME]` folder. For each stat, a unique `.csv` file is generated containing that statistic across all stats files. Average and standard deviations for each stat are calculated, and graphs displaying the average values (with standard deviations) across all generations are generated. Finally, the statistics parser saves a main `full_stats.csv` file containing all statistics across all runs in a single file. All `.csv` summary files can be used with any numerical statistics package, such as R.

While the experiment manager calls the statistics parser after all experiments have been completed, it is possible to call the statistics parser as a standalone program to generate these files for any given `[EXPERIMENT_NAME]` folder. This can be done from the command line by typing:

```
$ python scripts/parse_stats.py --experiment_name [EXPERIMENT_NAME]
```

where `[EXPERIMENT_NAME]` is a string which specifies the desired name of the experiment contained in the `results` folder.

## GE LR Parser

A powerful script that has been included with PonyGE2 is the deterministic GE LR Parser. This script will parse a given target string using a specified `.bnf` grammar and will return a PonyGE2 individual that can be used in PonyGE2. Provided the target string can be fully and correctly represented by the specified grammar, the LR parser uncovers a derivation tree which matches the target string by building the overall tree from the terminals used in the solution. A repository of phenotypically correct sub-trees whose outputs match portions of the target string (termed 'snippets') is compiled. Deterministic concatenation operators are employed to build the desired solution. Provided the grammar remains unchanged, these reverse-engineered solutions can be saved and used in an evolutionary setting.

Since the GE LR Parser is fully deterministic, the same GE individual will be returned every time it is executed.

To run the GE LR Parser, only two parameters need to be specified:

```
--grammar_file [FILE_NAME.bnf]

--reverse_mapping_target [TARGET_STRING]
```

where `[TARGET_STRING]` is a string specifying the target string to be parsed by the GE LR Parser.

***NOTE** that the full file extension for the grammar file (e.g. ".bnf")* ***must*** *be specified, but the full file path for the grammar file (e.g.* `grammars/example_grammar.bnf`*)* ***does not*** *need to be specified.*

Alternatively, both the `GRAMMAR_FILE` and `REVERSE_MAPPING_TARGET` can be specified in either the `algorithm.parameters.params` dictionary or in a separate parameters file. An example parameters file can be seen in the parameters folder. To run this example, type:

```
$ python scripts/GE_LR_parser.py --parameters GE_parse.txt
```

## Grammar Analyser

A simple grammar analyser has been provided with PonyGE2. Given a specified grammar file, this analyser will simply print out the number of unique combinations and permutations of phenotypes that are possible at a number of derivation tree depths.

Only one parameter needs to be specified with the grammar analyser:

```
--grammar_file [FILE_NAME.bnf]
```

where `[FILE_NAME.bnf]` is the name of the desired grammar file (including file extension).

The number of derivation tree depths across which permutations are displayed can be set with the argument:

```
--permutation_ramps [INT]
```

or by setting the parameter `PERMUTATION_RAMPS` to `[INT]` in the default params dictionary, where `[INT]` specifies the number of desired derivation tree depths.

***NOTE** that the* `PERMUTATION_RAMPS` *parameter specifies the number of derivation tree depths across which permutations will be shown, starting from the minimum path of the grammar. It does not necessarily specify the maximum derivation tree depth that permutations are to be calculated to.*

To run the Grammar Analyser, simply type:

```
$ python scripts/grammar_analyser.py --grammar_file [FILE_NAME.bnf]
```

***NOTE*** *that parameters files cannot be specified with the grammar analyser.*

Michael O'Neill and Conor Ryan, "Grammatical Evolution: Evolutionary Automatic Programming in an Arbitrary Language", Kluwer Academic Publishers, 2003.

Michael O'Neill, Erik Hemberg, Conor Gilligan, Elliott Bartley, and James McDermott, "GEVA: Grammatical Evolution in Java", ACM SIGEVOlution, 2008. http://portal.acm.org/citation.cfm?id=1527066. Get GEVA: http://ncra.ucd.ie/Site/GEVA.html

O'Neill, M., Ryan, C., Keijzer, M. and Cattolico, M., 2003. "Crossover in grammatical evolution." Genetic programming and evolvable machines, 4(1), pp.67-93. DOI: 10.1023/A:1021877127167

Ryan, C. and Azad, R.M.A., 2003. "Sensible initialisation in grammatical evolution." In Proceedings of the Bird of a Feather Workshops, Genetic and Evolutionary Computation Conference (GECCO 2003), pp. 142-145.

Fagan, D., Fenton, M. and O'Neill, M., 2016. "Exploring Position Independent Initialisation in Grammatical Evolution." IEEE Congress on Evolutionary Computation, Vancouver, Canada. IEEE Press.

Nicolau, M. and Fenton, M., 2016. "Managing Repetition in Grammar-based Genetic Programming." ACM GECCO 2016 Proceedings of the Genetic and Evolutionary Computation Conference, Denver, Colorado, USA.

Byrne, J., O'Neill, M. and Brabazon, A., 2009, "Structural and nodal mutation in grammatical evolution." In Proceedings of the 11th Annual conference on Genetic and evolutionary computation (pp. 1881-1882). ACM.

Koza, J.R., 1992. "Genetic programming: on the programming of computers by means of natural selection (Vol. 1)". MIT press.

Lewin B., 2000. "Genes VII". Oxford University Press.

Deb, K., Pratap, A., Agarwal, S. and Meyarivan, T.A.M.T., 2002. "A fast and elitist multiobjective genetic algorithm: NSGA-II". IEEE Transactions on Evolutionary Computation, 6(2), pp.182-197.

Zitzler, E., Deb, K., and Thiele, L., 2000. Comparison of multiobjective evolutionary algorithms: Empirical results. Evolutionary computation 8.2 (2000): 173-195.