

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.*