esec Quick Reference Guide

Steve Dower

Overview

esec enables research of both simple and complex ecosystem models of evolutionary computation. It supports highly customisable evolutionary systems through the use of Evolutionary System Definition Language (ESDL).

A wide range of standard **models** and benchmark **problems** are included, such as real valued continuous optimisation, binary problem landscapes, tree-based genetic programming and Grammatical Evolution.

esec is written in the Python programming language and is compatible with CPython 2.6, **CPython 2.7**, IronPython 2.6 and **IronPython 2.7**. For full functionality, the **Numpy** and **Psyco** packages are also required.

The `run.py` script provides an efficient command line interface to run single or multiple experiments using esec. All configuration information can be provided using **command line arguments**, however for research experiments the recommended method is to use a **configuration or batch file**.

Configuration files are based on Python dictionaries and contain all the settings necessary to conduct a single experiment. Batch files provide a sequence of configurations, allowing a multitude of experiments to be conducted automatically.

This document outlines:

- The `run.py` script's **command line options** (page 2).
- The available **configuration names** (page 3) and **landscapes** (page 4).
- How to extend esec using **plug-ins** (page 14).
- How to automate esec using **batch and configuration files** (page 16).
- The **ESDL syntax** (page 18) used for system definitions.
- The available **selectors**, **filters**, **joiners**, **initialisers**, and **systems** (pages 19–28).

Command Line Options

General Options

Option	Description
`--optimise -o`	Uses **Psyco** optimisation if available. (Note that using Psyco does not guarantee that processing speed will improve. Python's `-OO` option may improve performance by omitting many safety checks.)
`--profile -p`	Uses **cProfile** during a single simulation run. Profiling is not supported under IronPython.
`--verbose -v`	Sets the verbosity level. Valid levels are 0 through 5 (inclusive).

Single-Run Options

Option	Description
`--config -c`	Specifies a set of configuration names, configuration files or plug-ins joined by plus symbols ('+'). Items are applied in the order that they appear. See **Configuration Names** (page 3) for a list of built-in configuration names in esec, **Plug-ins** (page 14) for an overview of creating new plug-ins and **Batch and Configuration Files** (page 15) for an overview of creating configuration files. Examples: `-c RVP.Sphere+n2+GA`, `-c KozaSymbolicRegression+noseed`
`--settings -s`	Manually override any configuration setting. The parameter must be a quoted string of parameter-value pairs, separated by semicolons. Values are evaluated using Python's `eval` method and assigned in the order that they appear. These overrides are applied after any settings loaded with `--config`. Example: `-s "system.size=200; monitor.limits.iterations=100"`

Batch Options

Option	Description
`--batch -b`	Specifies the name of a batch file and any tags to include or exclude. This batch file must exist in the `cfgs` directory with an identical name (including case) and a `.py` extension. See **Batch and Configuration Files** (page 15) for an overview of creating batch files. Example: `-b GEMultiplexer3+pop100`

Usage Examples

```
python.exe -OO run.py -o --config RVP.Sphere+GA --settings "system.size=150"

ipy.exe run.py -c GEMultiplexer3 -s "monitor.limits.unique=1"
```

Configuration Names

Configuration names identify collections of settings to apply to the current experiment. Names are used with the `--config` command line option to simplify single experiments, or with **batch files** to make use of standard settings or plug-ins.

Plug-ins may add new configuration names, though these are only usable when the plug-in is specified.

Name	Description
`noseed`	Uses a time-dependent value to seed the breeding system's random number generator (the default seed is 12345). (Note: Landscapes use a separate seed.)
`landscape_noseed`	Uses a time-dependent value to seed the landscape's random number generator (the default landscape seed is 12345). (Note: Breeding systems use a separate seed.)
`n2`	Sets the `parameters` setting of the landscape to 2. (Note: This setting is not used by all landscapes.)
`n3`	Sets the `parameters` setting of the landscape to 3.
`n10`	Sets the `parameters` setting of the landscape to 10.
`n100`	Sets the `parameters` setting of the landscape to 100.
`i`	Uses inverted fitness comparisons, that is, use minimisation on problems that are normally maximisation problems and vice-versa.
`short`	Restricts the experiment to ten generations.
`long`	Restricts the experiment to one hundred generations.
`infinite`	Removes any generation limit on the experiment.
`debug`	Sets the verbosity level to its highest setting (5).
`csv`	Selects the CSV monitor and directs output to automatically named files in the `results` directory.

Landscapes

Each landscape is categorised into a type based on the species that is expected. A landscape may be specified on the command line by the names listed here, including the category prefix. Landscape parameters can only be specified on the command line using the `--settings` option.

Parameters marked with an asterisk (`*`) are also available by the name `parameters`, which may be set on the command line using configuration names `n2`, `n3`, `n10` and `n100`.

Binary Landscapes (BVP)

`BVP.OneMax`	Simple binary maximisation problem. The fitness is determined by the number of bits with the value 1. The fitness range is zero to N, inclusive.	
	`N*`	Desired genome size.
`BVP.RoyalRoad`	A discrete non-deceptive unimodal problem space. The genome is divided into Q blocks of C bits, with each block contributing C to the fitness when all its bits have the value 1. The fitness range is zero to $Q \times C$, inclusive.	
	`Q`	Number of blocks
	`C`	Number of bits per block
`BVP.GoldbergD3B`	Goldberg's Deceptive 3-bit function. A mapping array is used for each group of 3 bits to determine its fitness contribution. The fitness range is zero to $8 \times N$, inclusive.	
	`N*`	Number of 3-bit segments
`BVP.WhitleyD4B`	Whitley's Deceptive 4-bit function. A mapping array is used for each group of 4 bits to determine its fitness contribution. The fitness range is zero to $30 \times N$, inclusive.	
	`N*`	Number of 4-bit segments
`BVP.Multimodal`	N-dimensional random binary multimodal landscape.	
	`N*`	Number of parameters.
	`P`	Number of peaks.

BVP.CNF_SAT	N-dimensional random CNF epistasis generator. The fitness range is 0.0 to 1.0, assuming a valid landscape. The size.exact parameter is set by the landscape and must be used when initialising individuals.	
	L	Number of clauses
	K	Number of literals per clause
	N	Number of variables per literal
	SAW	True to enable adaptive weights
BVP.NK	NK landscape generator. The size.exact parameter is set by the landscape and must be used when initialising individuals.	
	N	Number of genes
	K	Number of interactions
BVP.NKC	NKC landscape generator. The size.exact parameter is set by the landscape and must be used when initialising individuals.	
	N	Number of genes
	K	Number of self-interactions
	C	Number of external interactions
	group	Size of evaluation group
BVP.MMDP6	Massively Multimodal Deceptive Problem (6-bit) The size.exact parameter is set by the landscape and must be used when initialising individuals.	
	subs*	Number of sub-strings
BVP.ECC	Error Correcting Code Design Problem The size.exact parameter is set by the landscape and must be used when initialising individuals.	
	n	Code length
	M	Number of code words
	d	Minimum distance

BVP.SUS	Subset Sum Problem Generator The size.exact parameter is set by the landscape and must be used when initialising individuals.	
	N*	Number of integers in full set
	maxN	Largest integer in full set
	even	True to require set to contain only even integers.
BVP.MAXCUT	Maximum Cut of a Graph The size.exact parameter is set by the landscape and must be used when initialising individuals.	
	N*	Number of vertices
	P	Probability of an edge between two vertices
BVP.MTTP	Minimum Tardy Task Problem The size.exact parameter is set by the landscape and must be used when initialising individuals.	
	tasks*	Number of tasks
BVP.Graph2c	N×N connectivity matrix with odd numbered column constraints The size.exact parameter is set by the landscape and must be used when initialising individuals.	
	parameters	Number of dimensions
BVP.Graph2r	N×N connectivity matrix with odd numbered row constraints The size.exact parameter is set by the landscape and must be used when initialising individuals.	
	parameters	Number of dimensions

Integer Landscapes (IVP)

The bounds parameter applies to all integer valued landscapes unless otherwise specified. Values may be set as bounds.lower and bounds.upper. All integer valued landscapes have suitable defaults set for these parameters.

Unless otherwise specified, the size of the individuals does not need to be provided to the landscape. In some cases, the landscape will require an individual of a particular size and will provide this value through its size property (including min and max limits, and, if stated, exact).

`IVP.Nsum`	Integer maximisation problem. The fitness is the sum of the gene values.
`IVP.Nmax`	Uses the maximum value for each gene as the target. The fitness is: $$f(x) = -\sqrt{\sum (u_i - x_i)^2}$$ Where u_i is the upper bound of gene x_i.
`IVP.Nmatch`	Sets the target as the centre of the value range for each gene. The fitness is: $$f(x) = -\sqrt{\sum (t_i - x_i)^2}$$ Where t_i is the target value for gene x_i.
`IVP.Robbins`	Robbins landscape. The fitness is: $$f(x) = \sum_{i=0}^{i_{\max}} x_i 2^{i_{\max}-i-1}$$ If the genome consists solely of the values 0 and 1, this is a regular binary to integer conversion.

Real Landscapes (RVP)

The parameters size and bounds apply to all real valued landscapes unless otherwise specified. Values may be set as size.min and size.max (for variable length genomes) or size.exact (for fixed length genomes) and bounds.lower and bounds.upper (or lower_bounds and upper_bounds).

All real valued landscapes have suitable defaults set for these parameters and some have strict requirements that they are not changed.

RVP.Linear	Numeric maximisation problem. $$f(x) = \sum x_i$$	
RVP.Neutral	A flat landscape with a constant fitness of *mean*. $$f(x) = \text{mean}$$	
	mean	Fixed fitness value
RVP.Stabilising	A Cauchy-Lorentz distribution with the global maximum $f_{max} = n(\text{amplitude})$ at $x_* = \text{mean}$. $$f(x) = \sum \text{amplitude}\left(\frac{\text{gamma}^2}{(x_i - \text{mean})^2 + \text{gamma}^2}\right)$$	
	mean	Distribution centre
	amplitude	Peak amplitude
	gamma	Distribution spread
RVP.Disruptive	A negative form of the Stabilising landscape with the global maximum $f_{max} = 0$ at $x_* = \text{mean}$. $$f(x) = \sum \text{amplitude}\left(1 - \frac{\text{gamma}^2}{(x_i - \text{mean})^2 + \text{gamma}^2}\right)$$	
	mean	Distribution centre
	amplitude	Peak amplitude
	gamma	Distribution spread
RVP.Sphere	N-dimensional spherical (parabola, parabolic or basin) landscape with the global minimum $f_{min} = 0$ at $x_* = 0$. $$f(x) = \sum x_i^2$$	
RVP.Ellipsoid	A simple unimodal surface, also known as the "axis parallel ellipsoid function" with the global minimum $f_{min} = 0$ at $x_* = 0$. $$f(x) = \sum i x_i^2$$	

`RVP.HyperEllipsoid`	A simple convex unimodal surface, also known as the “axis parallel hyper-ellipsoid function” with the global minimum $f_{min} = 0$ at $x_* = 0$. $$f(x) = \sum i^2 x_i^2$$
`RVP.Quadric` `RVP.RotatedHyper-Ellipsoid`	Quadric landscape. Also known as Schwefel’s function 1.2, Schwefel’s Double Sum and the rotated hyper-ellipsoid function. The global minimum $f_{min} = 0$ is at $x_* = 0$. $$f(x) = \sum_{i=1}^{n} \left(\sum_{j=1}^{i} x_j \right)^2$$
`RVP.NoisyQuartic`	N-dimensional quartic function with Gaussian noise. $$f(x) = \sum (ix_i^4 + G)$$ Where G is selected from a Gaussian distribution with range 0 to 1. The global minimum $f_{min} \sim 0$ is at $x_* = 0$.
`RVP.Easom`	A two dimensional landscape with few gradient “hints”. $$f(x_1, x_2) = \cos(x_1)\cos(x_2)\,\mathrm{e}^{-((x_1-\pi)^2+(x_2-\pi)^2)}$$ `size.exact` is fixed to 2 for this landscape. The global maximum $f_{max} = 1$ is at $x_* = \pi$.
`RVP.Rosenbrock`	Also known as De Jong Function 2, Rosenbrock’s Saddle and the Banana function. The global minimum $f_{min} = 0$ is at $x_* = 1$. See **http://mathworld.wolfram.com/RosenbrockFunction.html** for details on the two-dimensional version.
`RVP.Rastrigin`	A parabolic landscape with additional local minima. $$f(x) = 10.0n + \sum (x_i^2 - 10.0 \cos 2\pi x_i)$$ Variable length genomes are not supported for this landscape. The global minimum $f_{min} = 0$ is at $x_* = 0$.
`RVP.Griewangk`	A parabolic landscape with additional local minima. $$f(x) = \frac{1}{4000}\left(\sum x_i - 100\right)^2 - \prod \left(\frac{x_i - 100}{\sqrt{i}}\right) + 1$$ Variable length genomes are not supported for this landscape. The global minimum $f_{min} = 0$ is at $x_* = 0$.
`RVP.Ackley`	An exponential well modulated by a cosine term. $$f(x) = 20 + e - 20e^{-0.2\left(\sqrt{\frac{\sum x_i^2}{n}}\right)} - e^{\frac{1}{n}\sum(\cos(2\pi x_i))}$$ Variable length genomes are not supported for this landscape. The global minimum $f_{min} = 0$ is at $x_* = 0$.

`RVP.Schwefel`	Schwefel problem landscape. $$f(x) = 418.9829n + \sum x_i \sin\sqrt{\lvert x_i \rvert}$$ Variable length genomes are not supported for this landscape. The global minimum $f_{min} = 0$ is at $x_* = -420.9687$.
`RVP.Michalewicz`	A multimodal domain with $n!$ local optima. $$f(x) = -\sum \sin x_i \sin^{2m} \frac{ix_i^2}{\pi}$$ Variable length genomes are not supported for this landscape. The global minimum varies depending on m.
`RVP.MultiPeak1`	Sinusoidal multiple peaks landscape. $$f(x) = sin^6(5\pi x)$$ `size.exact` is fixed to 1 for this landscape. Global maxima of $f_{max} = 1$ are at $x \in \{0.1, 0.3, 0.5, 0.7, 0.9\}$.
`RVP.MultiPeak2`	Sinusoidal multiple peaks landscape. $$f(x) = \sin^6\left(5\pi\left(x^{\frac{3}{4}} - 0.05\right)\right)$$ `size.exact` is fixed to 1 for this landscape.
`RVP.MultiPeak3`	Sinusoidal multiple peaks landscape. $$f(x) = e^{-2(\log 2)\left(\frac{x-0.08}{0.854}\right)^2} \sin^6(5\pi x)$$ `size.exact` is fixed to 1 for this landscape.
`RVP.MultiPeak4`	Sinusoidal multiple peaks landscape. $$f(x) = e^{-2(\log 2)\left(\frac{x-0.08}{0.854}\right)^2} \sin^6\left(5\pi\left(x^{\frac{3}{4}} - 0.05\right)\right)$$ `size.exact` is fixed to 1 for this landscape.
`RVP.Booth`	A constrained two-dimensional landscape with several local minima and one global minimum of $f_{min} = 0$ at $x = (1, 3)$. $$f(x_1, x_2) = (x_1 + 2x_2 - 7)^2 + (2x_1 + x_2 - 5)^2$$ `size.exact` is fixed to 2 for this landscape.
`RVP.Himmelblau`	A two-dimensional multimodal minimisation problem landscape with four near equal optimum (although there are differences at higher value resolution). $$f(x_1, x_2) = (x_1^2 + x_2 - 11)^2 + (x_1 + x_2^2 - 7)^2$$ `size.exact` is fixed to 2 for this landscape.

`RVP.SixHumpCamel-Back`	A two-dimensional non-separable multimodal and multi-solution minimisation problem with six minimum features within an asymmetric bounded domain. $$f(x_1,x_2) = 4x_1^2 - 2.1x_1^4 + \frac{1}{3}x_1^6 + x_1x_2 - 4x_2^2 + 4x_2^4$$ `size.exact` and `bounds` are fixed for this landscape. Two global minima of $f_{min} = -1.0316$ are at $x = (\pm 0.08983, \mp 0.7126)$.
`RVP.FMS`	The Frequency Modulation Sounds landscape. A highly complex, multimodal function with strong epistasis where the objective is to determine six real value parameters used in a FM sound model. See **http://tracer.lcc.uma.es/problems/fms/fms.html** for information on this landscape model. `size.exact` and `bounds` are fixed for this landscape.
`RVP.MSG`	Maximum set of Gaussians landscape generator. Uses parameters to generate a wide range of landscapes. (This landscape requires **Numpy**. It is implemented as an example of providing a plugin landscape. It can be found in the `/plugins` folder.)
	`ngauss` — Dimensionality of the landscape
	`gvalue` — Value of the global optimum
	`ratio` — Ratio of the global optimum to local optima

Tree-based Genetic Programming Landscapes (TGP)

Apart from landscape parameters, TGP solutions are very strongly affected by the selection of an instruction set. The available instructions are specified when initialising groups of individuals, rather than being set by the landscape.

Any TGP landscape may specify the `instruction_set` parameter, which identifies the name of the instruction set to expect. All landscapes expose a `terminals` parameter that is determined by the landscape and should not be modified: it is intended to be referenced from the ESDL system definition through the `config.landscape.terminals` variable.

The fitness values returned by TGP landscapes contain two values: the score and the cost. Individuals are more fit if their score is better (whether that means higher or lower depends on the landscape). If two scores are identical, the one with the lower cost (which is based on the number of instructions in the program) is more fit.

`TGP.Multiplexer`	Boolean multiplexer design. The returned value is the value of the terminal selected by a set of address bits. The fitness range is $[0, 2^t]$, where $t = \text{parameters} + 2^{\text{parameters}}$. The optimum fitness is 2^t.	
	`parameters`	The number of address bits to use.
`TGP.Symbolic-Regression`	Attempts to generate an equation that produces output matching `expr`. $$f(g(x)) = -\sum \lvert g_i(x) - \text{expr}(x) \rvert$$ `parameters` is fixed to 1 for this landscape.	
	`expr`	An expression (in a string using Python syntax) to be used to generate test data. This is the target equation over the variable `x`.

Grammatical Evolution Landscapes (GE)

GE solutions are very strongly affected by the grammar used for genotype-phenotype mapping. The grammar is specified when initialising individuals. Most landscapes provide a `rules` member which contains a suggested grammar.

The `size_penalty_square_factor` and `size_penalty_linear_factor` parameters may be used on any landscape to adjust the rate at which genome length affects the fitness. All landscapes expose the `terminals` parameter, which is set by the landscape and should not be modified.

`GE.Multiplexer`	Boolean multiplexer design. The returned value is the value of the terminal selected by a set of address bits. The fitness range is $[0,2^t]$, where $t = \text{parameters} + 2^{\text{parameters}}$. The optimum fitness is 2^t.	
	`parameters`	The number of address bits to use.
`GE.Symbolic-Regression`	Attempts to generate an equation that produces output matching `expr`. The optimum fitness is zero. $$f(g(x)) = \sum (g_i(x) - \text{expr}(x))^2$$ Invalid individuals and those with a fitness worse (greater) than 10^5 are assigned $f = \infty$. `parameters` is fixed to 1 for this landscape.	
	`expr`	An expression (in a string using Python syntax) to be used to generate test data. This is the target equation over the variable `x`.

Plug-ins

Plug-ins allow esec to easily import added functionality in the form of scripts or packages. They are primarily intended for use with the `run.py` script but also simplify the programmatic initialisation of experiments. Plug-ins may include new or extended species, landscapes, dialects and configuration names, or modify the behaviour of existing settings.

Only one plug-in is intended to be loaded at any time, since there is only one set of configuration names. Loading multiple plug-ins may cause conflicting definitions and is not supported; however, a plug-in may import and explicitly expose other plug-ins.

The main plug-in file is a Python script or module stored in the `plugins/` directory. The (case-sensitive) name of the script or module is used as a **configuration name** that loads the plug-in. (If developing a plug-in as a module, “plug-in script” refers to the `__init__.py` file of the module).

Configuration Names

The plug-in script may provide a dictionary containing configuration names in a variable `configs`. The key of each element is the name and the value is a dictionary to overlay onto the current configuration. These elements are added to the set of known configuration names, replacing any previous elements with matching names.

A dictionary of default values may be included in a variable `defaults`. These values are overlaid onto the active configuration immediately and allow settings such as monitor formats and the system definition to be set to sensible defaults for the plug-in.

Landscapes

New landscapes are exposed as configuration names. By convention, the name used is an abbreviation of the relevant species (such as BVP for binary valued problems or TGP for tree-based genetic programming), a period and the name of the landscape.

The value associated with the configuration name typically sets the `landscape.class` setting to the landscape type object, along with any other required settings.

```python
import landscape.real
configs = {
    'RVP.Linear': {
        'landscape': {
            'class': landscape.real.Linear,
            'N': 5,
        },
    },
}
```

Alternatively, the landscape setting may be directly set with an instance of the landscape. This, however, prevents further customisation of the landscape from the command line, and is not suggested for plug-ins.

Dialects

New dialects are exposed as configuration names. By convention, the name used is an appropriate acronym of the dialect's name (such as SSGA for Steady-State Genetic Algorithm or EP for Evolutionary Programming).

The value associated with the configuration name typically sets the `system.definition` setting to the ESDL code for the dialect, along with any other names or variables required by the system.

```
configs = {
    'GA': {
        'system': {

            'definition': r'''
FROM random_int(length=10) SELECT (size) population
YIELD population

BEGIN generation
    FROM population SELECT (size) offspring \
         USING binary_tournament

    FROM offspring SELECT population \
         USING crossover_one(per_indiv_rate=0.8), \
               mutate_random(per_indiv_rate=(1.0/size))

    YIELD population
END generation
            ''',

            'size': 10,
        },
    },
}
```

Values specified as variables (such as `size`, above) can be overridden by other configuration names or the `--settings` command line option; values specified directly in the definition cannot.

Species

Species classes must be advertised to ensure that the breeding system is aware of any names exposed through its `public_context` variable (such as generators or mutation operators). Species are advertised by importing the main species package and calling `include`.

```
from species import RealSpecies, BinarySpecies

import esec.species
esec.species.include(RealSpecies, BinarySpecies)
```

Batch and Configuration Files

Configuration files provide reproducible automation of esec over both single and multiple experiments. Files that specify a single run are called configuration files, while those that specify multiple runs are batch files.

Batch and configuration files are Python scripts (with extension `.py`) stored in the `cfgs/` directory. Despite the distinguishing terminology, a single file can be both a configuration and a batch file.

Configuration files are functionally identical to plug-ins, though they typically provide only configuration details. This configuration is provided as a dictionary in the variable `config` (which behaves identically to `defaults` in a plug-in) that is overlaid onto the active configuration when the configuration file is loaded.

Batch files provide a method `batch()`, which takes no parameters and returns a list or sequence of dictionaries[1] that specify each experiment to run.

The content of each dictionary is:

Key	Value
`tags`	A sequence of tag strings identifying the experiment category.
`names`	A set of configuration names as for `--config`.
`config`	A pre-initialised dictionary of configuration settings.
`settings`	A settings string as for `--settings`.
`format`	A format string to display in the tag summary file.

Elements not provided are assumed to be `None` (the Python keyword representing no value; `null` in some other languages). `None` is a valid value for any element.

Tag strings are used to simplify the process of running a limited part of a batch file. For example, a batch file that uses a number of sets of parameters may identify each set with a tag, providing a simple mechanism for running one set at a time. Each configuration may have multiple tags: if any of these are specified in `include_tags` and none of them is specified in `exclude_tags`, the configuration will be included in the run.

Tags may also be specified on the command line as part of the parameter to the `--batch` switch. After the name of the batch file (which must always appear first), tags may be appended separated by plus (`+`) characters. Tag names prefixed with an exclamation mark (`!`) are excluded from the run, while others are included.

[1] In fact, any object with a `get(key, default)` method may be returned here.

Batch files support an extra set of settings that control which experiments to run. When provided on the command line with the `--settings` option, the setting name must be prefixed with `batch.`, for example, `batch.dry_run=True`. When provided in the `settings` variable in the batch file, `dry_run=True` has the same effect.

Setting	Description
`dry_run`	If `True`, creates all experiments but does not run any of them.
`start_at`	The index of the first experiment to run.
`stop_at`	The index of the last experiment to run.
`include_tags`	A list of tags specifying which experiments to run. These may also be specified on the command line by appending `+tag` to the `--batch` switch.
`exclude_tags`	A list of tags specifying which experiments to ignore. These may also be specified on the command line by appending `+!tag` to the `--batch` switch.
`pathbase`	A directory path relative to `run.py` to store results in.
`csv`	If `True`, writes CSV formatted files.
`summary`	If `True`, creates a summary of the entire batch results.
`low_priority`	If `True`, runs the Python process at low CPU priority.
`quiet`	If `True`, only summaries of each experiment are displayed to the console.

Evolutionary System Definition Language

Basic Syntax Examples

```
#  Comment style 1

;  Comment style 2

// Comment style 3

FROM generator SELECT (size) population

EVAL group USING evaluator

YIELD group

variable = 1.0

result = (function(parameter=variable) + 2) * 4 - 3

BEGIN block

    FROM group SELECT new_group USING filter, filter, ...

    FROM group SELECT (size1) subgroup1, (size2) subgroup2 USING filter

    FROM group SELECT (size) subgroup, the_rest USING bounded_selector

    REPEAT count

        FROM group1, group2 SELECT merged_group

        JOIN group1, group2 INTO joined_group USING joiner

        FROM joined_group SELECT group USING filter

    END repeat

    YIELD group1, group2, ...

END block
```

Filters

Filters constitute any operation that is applied to a group of individuals to produce a new group using a FROM-SELECT statement. The new group may differ in the order of individuals, some individuals may be excluded from the source group(s) or new individuals may have been added. Filters are roughly grouped into **Selection Filters**, which do not modify individuals, and **Variation Filters**, which may.

Selection Filters

Filters for selection (sometimes *selectors*) are not specific to species and may be applied to any group.

Unbounded selectors do not terminate and require all destination groups to have a size limit specified. *Bounded* selectors are guaranteed to terminate and may be used with a destination group that does not have a size specified *unless* one of the sources is a **generator**.

`repeat`	Returns every group repeatedly. This selector is unbounded.	
`best_only`	Returns the individual with the best fitness in the group(s) repeatedly. This selector is unbounded.	
`worst_only`	Returns the individual with the worst fitness in the group(s) repeatedly. This selector is unbounded.	
`best` `truncate_best`	Returns the complete group(s) in order of descending fitness (best first). This selector is bounded when `only` is `False`.	
	`only`	If `True`, returns the best repeatedly (default `False`).
`worst` `truncate_worst`	Returns the complete group(s) in order of ascending fitness (worst first). This selector is bounded when `only` is `False`.	
	`only`	If `True`, returns the worst repeatedly (default `False`).
`unique`	Returns the set of unique individuals from the group(s). Uniqueness is based on the textual representation of the phenotype. This selector is bounded.	
`best_of_tuple`	Returns the individual with the highest fitness from each tuple. Requires joined individuals in the source group(s). This selector is bounded.	
`tournament`	Returns the best individual from a randomly selected sample of k individuals. This selector is unbounded when `replacement` is `True`; otherwise, it is bounded.	
	`k`	The tournament size ($k \geq 2$, default is 2)
	`replacement`	Select with replacement (default is `True`).

binary_tournament	Returns the best individual from a randomly selected sample of two individuals. This selector is unbounded when replacement is True; otherwise, it is bounded. replacement Select with replacement (default is True).
uniform_random	Returns individuals selected randomly. This selector is unbounded when replacement is True; otherwise, it is bounded. replacement Select with replacement (default is True).
uniform_shuffle	Returns individuals selected randomly without replacement. Behaves identically to uniform_random when replacement is False. This selector is bounded.
fitness_proportional	Returns individuals selected proportionally to their fitness. This selector is bounded when replacement and sus are False; otherwise, it is unbounded. The value of mu does not limit the total number of selections, but may affect the sample distribution. replacement Select with replacement (default is True). sus Use stochastic universal sampling (default is False). mu Number of selections in SUS (default is the group size).
rank_proportional	Returns individuals selected proportionally to their rank (when sorted by fitness). This selector is bounded when replacement and sus are False; otherwise, it is unbounded. The value of mu does not limit the total number of selections, but may affect the sample distribution. replacement Select with replacement (default is True). sus Use stochastic universal sampling (default is False). mu Number of selections in SUS (default is the group size).
fitness_sus	Returns individuals selected using stochastic universal sampling proportional to fitness. The value of mu does not limit the total number of selections, but may affect the sample distribution. mu Number of selections (default is the group size).
rank_sus	Returns individuals selected using stochastic universal sampling proportional to rank (when sorted by fitness). The value of mu does not limit the total number of selections, but may affect the sample distribution. mu Number of selections (default is the group size).

Variation Filters

All variation filters should accept the parameters `per_indiv_rate` and (where applicable) `per_gene_rate`. The parameter `per_indiv_rate` represents the probability (as a value between 0.0 and 1.0, inclusive) of a particular individual being mutated, with `per_gene_rate` representing the probability for each gene of that individual. If `per_indiv_rate` is zero, `per_gene_rate` is always irrelevant.

Crossover operations generally prefer the use of `per_pair_rate` rather than `per_indiv_rate`, though both are supported.

`crossover_uniform`	Per-gene crossover, selecting individual genes randomly from each member of the pair.
`crossover_one`	Single-point crossover, splitting at the same location.
`crossover_one_-different`	Single-point crossover, splitting at different locations in each individual. `longest_result` The maximum length permitted in the resulting genome.
`crossover_tuple`	Merges the individuals from a joined individual into a single individual by selecting each gene from a random individual. `per_gene_rate` The probability of the next gene *not* being selected from the first individual. The distribution for the remaining individuals is uniform. If this is not specified, an equal probability is applied to all individuals. If the joined individuals contain two individuals each, this is equivalent to `crossover_uniform`.
`mutate_random`	Performs random replacement mutation on individual genes.
`mutate_bitflip`	Performs single gene inversion mutation on binary individuals.
`mutate_inversion`	Performs full genome inversion mutation on binary individuals.

mutate_gap_inversion	Performs segment inversion mutation on binary individuals.	
	length	The number of consecutive bits to flip.
	shortest	The least number of bits to flip (default 1).
	longest	The most number of bits to flip (default 10).
	(Specifying length overrides shortest and longest.)	
mutate_delta	Performs step mutation on integer and real-valued individuals.	
	step_size	The amount to modify each gene (default 1 for integer, 0.1 for real).
	positive_rate	The probability of increasing the value versus decreasing the value (default 0.5).
mutate_gaussian	Performs Gaussian step mutation on integer and real-valued individuals. The actual step size is a random value from a Gaussian distribution with mean zero and standard deviation sigma or step_size × 1.253 (if sigma is omitted).	
	step_size	The "mutation step" (default 1 for integer, 0.1 for real).
	sigma	The standard deviation.
mutate_insert	Inserts a randomly generated segment into an existing genome.	
	length	The number of genes to insert.
	shortest	The least number of genes to insert.
	longest	The most number of genes to insert.
	longest_result	The maximum length genome to create.
	(Specifying length overrides shortest and longest.)	
mutate_delete	Deletes a randomly selected segment from a genome.	
	length	The number of genes to delete.
	shortest	The least number of genes to delete.
	longest	The most number of genes to delete.
	shortest_result	The minimum length genome to create.
	(Specifying length overrides shortest and longest.)	

Joiners

Joiners can either combine individuals from separate populations for the purposes of evaluation, or they can select multiple individuals from the same population for specialised mutation or crossover operations (such as in Differential Evolution).

If a `JOIN` statement is specified without a joiner, `tuples` is assumed.

`tuples`	Matches individuals based on their index within a group. (An individual's index within a group is arbitrary but consistent for a particular group.)	
`full_combine`	Matches every individual with every other individual, resulting in $m \times n$ individuals from two groups of size m and n.	
`best_with_rest`	Matches the individual with the highest fitness from one group to every individual from every other group. This is equivalent to using `full_combine` followed by selecting only those joined individuals with a particular individual.	
	`best_from`	The zero-based group index to select the best from.
`random_tuples`	Matches each individual from the first group with a randomly selected individual from each other group. (The same group may be specified multiple times.)	
	`distinct`	Avoid (if possible) matching an individual with itself (default `False`).
`distinct_random_-tuples`	Matches (if possible) each individual from the first group with a different individual from each other group. (The same group may be specified multiple times.) This is equivalent to using `random_tuples` with `distinct` set to `True`.	

Generators

All generators (unless otherwise specified) include length, shortest and longest settings that determine the size of the generated individuals. By default, the length is ten. Specifying length takes precedence over shortest and longest and produces individuals that are all the same size. Otherwise, a random length between shortest and longest (inclusive) is selected for each individual.

For these generators, providing a dictionary to length containing min, max and exact members (for example, the config.landscape.size variable) will be interpreted correctly, allowing simpler system definitions.

random_binary	Creates randomly initialised binary individuals.
binary_zero binary_one binary_toggle	Creates binary individuals with all bits initialised to zero, one or with each subsequent individual alternating between all ones and all zeros.
random_int random_integer	Creates randomly initialised integer individuals lowest — The lowest value a gene may take (default 0). highest — The highest value a gene may take (default 255). bounds — The actual limits for each gene: (lower, higher). Defaults to (lowest, highest).
integer_low integer_high integer_toggle	Creates integer individuals with all values initialised to lowest, highest or with each subsequent individual alternating between highest and lowest. (Parameters as for random_int.)
integer_increment	Creates integer individuals with genes incrementing from lowest to highest, restarting from lowest when highest is exceeded. (Parameters as for random_int.)
integer_count	Creates integer individuals with all genes incrementing from lowest to highest, restarting from lowest when highest is exceeded. The first individual is identical to that returned by integer_low. (Parameters as for random_int.)

random_int_binary	Creates integer individuals with binary representations. All binary variation operators (such as mutate_bitflip) may be used, but those designed for integer representations (such as mutate_delta) will not work. counted is the default encoding.	
	length	The number of integer values in the individual. (shortest and longest are not available.)
	bits_per_value	The number of bits for each integer value. This may be a list with length elements or a single number to apply to all values.
	counted	True to convert binary to integer by counting high bits. The resulting range is $[0, n_{\text{bits}}]$
	ones_complement	True to convert directly to an integer representation. The resulting range is $[0, 2^{n_{\text{bits}}})$
	twos_complement	True to convert directly to an integer representation. The resulting range is $\left[-2^{(n_{\text{bits}}-1)}, 2^{(n_{\text{bits}}-1)}\right)$
	gray_code	True to convert directly using a Gray code mapping. The resulting range is $[0, 2^{n_{\text{bits}}})$
binary_zero_int binary_one_int	Creates integer individuals with binary representations with all bits set to zero or one.	
	(Parameters as for random_int_binary.)	
random_real	Creates randomly initialised real-valued individuals	
	lowest	The lowest value a gene may take (default 0.0).
	highest	The highest value a gene may take (default 1.0).
	bounds	The actual limits for each gene: (lower, higher). Defaults to (lowest, highest).
real_low real_high real_toggle	Creates real-valued individuals with all values initialised to lowest, highest or with each subsequent individual alternating between highest and lowest.	
	(Parameters as for random_real.)	

random_real_binary	Creates real-valued individuals with binary representations. All binary variation operators (such as mutate_bitflip) may be used, but those designed for real-values representations (such as mutate_gaussian) will not work. counted is the default encoding. Real values are determined by finding the integer value according to the encoding scheme and scaling it to the range given by lowest and highest.	
	length	The number of real values in the individual. (shortest and longest are not available.)
	bits_per_value	The number of bits for each real value. This may be a list with length elements or a single number to apply to all values.
	lowest, highest	The inclusive range to map the binary values to. These may be lists to allow different ranges for each value.
	counted	True to convert by counting high bits.
	ones_complement	True to convert directly to an integer representation.
	twos_complement	True to convert directly to an integer representation.
	gray_code	True to convert directly using a Gray code mapping.
binary_zero_real binary_one_real	Creates real-valued individuals with binary representations with all bits set to zero or one.	
	(Parameters as for random_real_binary.)	
boolean_tgp integer_tgp real_tgp random_tgp	Creates TGP individuals using either a boolean, integer arithmetic, real-valued arithmetic or a custom instruction set. (length, shortest and longest are not available.)	
	terminals	The number of terminals to include (default 2).
	deepest	The deepest tree to create (default 10).
	adfs	The number of functions to include (default 0).
	terminal_prob	The probability of a branch terminating before deepest (default 0.5).
	transcendentals	Include instructions sin, cos, log and exp (real_tgp only).
	instructions	A list of species.tgp.Instruction objects that may be used in programs (random_tgp only).

random_ge	Creates GE individuals initialised with random values.	
	grammar	The grammar used for genotype-phenotype mapping.
	defines	A set of definitions to include with every evaluation.
	lowest	The lowest value a gene may take (default 0).
	highest	The highest value a gene may take (default 255).
	wrap_count	The maximum number of times a genome may be re-used during genotype-phenotype mapping (default 0).
	(shortest and longest default to 1 and 100, respectively. length is not available.)	

Evaluators

An evaluator is an object instance providing a method `eval(self, indiv)` that returns the fitness of the provided individual. The `esdl_eval` decorator (`from esec import esdl_eval`) produces a suitable object from an unbound function (`eval(indiv)`) and makes it available for using with ESDL's `EVALUATE` statement.

Every **landscape** instance is a valid evaluator.

Complete Systems

The following systems are available by the configuration names shown. These are specially designed to automatically detect the required species type and will not work with other species. All are specified in the `dialects.py` file and may be used as a reference for creating custom system definitions.

Specifying `system.size` using the `--settings` option affects the number of individuals in the primary population for most of these systems.

UN	A simple, unspecified system. Does not modify the initial population in any way.
GA	Genetic Algorithm. Performs single-point crossover on 80% of individuals and random mutation on 10% of the genes of one individual.
ES	$(1 + \lambda)$ Evolution Strategy. Uses a population of 1, produces `system.size` (λ) offspring using adaptive Gaussian mutation and retains the best.
EP	Evolutionary Programming. Mutates the entire population using Gaussian mutation and retains the best of the offspring and parents.
SSGA	Steady State Genetic Algorithm. Performs single-point crossover on two individuals with 90% probability and random mutation on 1% of the genes of both. The better of the two is retained in place of a randomly selected individual from the primary population.
NKC_GA	Genetic Algorithms specifically configured for the `BVP.NKC` landscape. Performs single-point crossover on 80% of individuals and random mutation on 10% of the genes of one individual. For evaluation, each individual is paired with a randomly selected individual. The first individual is assigned the full fitness value awarded to the pair.