Specifying Differential Evolution in ESDL

Steve Dower* Clinton Woodward

Swinburne University of Technology
Melbourne, Australia
November 10, 2010

Abstract

Evolutionary Systems Definition Language (ESDL) is a domain-specific language for search algorithms based on iterative improvements to a solution population. Differential Evolution (DE) is an evolutionary algorithm that uses real-valued vectors for individuals and a "weighted differential" vector variation scheme. This report describes DE using ESDL and validates the performance against earlier work.

1 Introduction

1.1 ESDL

Evolutionary System Definition Language (ESDL) is a domain-specific language for describing the flow of evolutionary algorithms (EAs) [1]. It defines a system as *groups* of *individuals* and the process followed to create new groups from existing ones. Groups are created by selecting and modifying individuals from existing groups or from *generators*.

ESDL does not specify the implementation or behaviour of specific operations: it requires an underlying framework to provide the functionality. For this report, `esec`[1] is the supporting framework.

1.2 Differential Evolution

Differential Evolution (DE) is an EA introduced by Storn and Price [2–4]. The canonical approach uses a population of real-valued vectors and follows a basic EA generational flow. Reproduction is performed using a "weighted differential" vector for mutation followed by uniform crossover, rather than a traditional crossover and mutation scheme.

In DE, a target vector and a base vector are selected from the current population. The base vector and a weighted difference vector, created from two additional sample vectors, are added to create a mutant vector. The target vector and the mutant vector are then combined, using crossover, to create a trial vector. If the trial vector is better than the target vector the target vector is replaced. Figure 1 shows this process as a flowchart, reproduced from the authors' original description in [4].

2 System Definition

ESDL can be used to define DE applied to a two-dimensional function as shown in Listing 1. The only element provided separately from the ESDL definition is `mutate_DE`, which performs the weighted difference-vector variation. All other elements are provided by `esec`; descriptions of their behaviour

*Contact via http://stevedower.id.au/

[1] Available online at http://code.google.com/p/esec. The plugin and configuration files used in this report are also available from here.

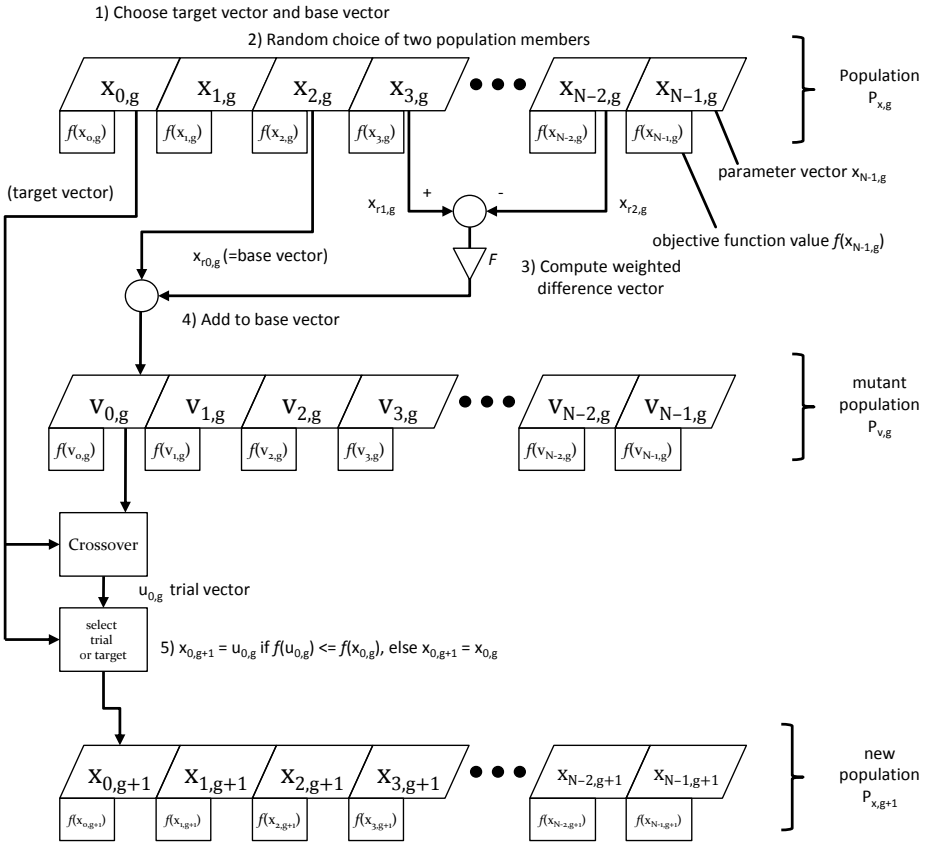

Figure 1: The DE generate and test loop from [4]

are included below for completeness. The problem landscape is specified externally. Changing the problem landscape may require modification of the numeric values in line 1.

Listing 1: ESDL definition of the DE algorithm

```
1 FROM random_real(length=2,lowest=-2.0,highest=2.0) SELECT (size) population
2 YIELD population
3
4 BEGIN GENERATION
5     # SELECT without a USING makes a copy of the group
6     FROM population SELECT (size) targets
7
8     # Stochastic Universal Sampling for bases
9     FROM population SELECT (size) bases USING fitness_sus(mu=size)
10
11    # Ensure r0 != r1 != r2, but any may equal i
12    JOIN bases, population, population INTO mutators \
13                                       USING random_tuples(distinct=True)
14
15    FROM mutators SELECT mutants USING mutate_DE(scale=0.8)
16
17    JOIN targets, mutants INTO target_mutant_pairs USING tuples
18    FROM target_mutant_pairs SELECT trials \
19                             USING crossover_tuple(per_gene_rate=0.8)
20
21    JOIN targets, trials INTO targets_trial_pairs USING tuples
22    FROM targets_trial_pairs SELECT population USING best_of_tuple
23
24    YIELD population
25 END GENERATION
```

Behaviour for the `mutate_DE` operator is specified in Listing 2. It is applied to a tuple containing three real-valued vectors, created by the `JOIN`–`INTO` instruction on lines 12 and 13 of Listing 1.

Listing 2: Pseudocode definition for mutate_DE

```
function mutate_DE(source, scale):
    for each vector b, p1 and p2 in source:
        yield new vector b + scale × (p1 − p2)
```

Tuples created on lines 12 and 13 of Listing 1 consist of each vector from the first source group `bases` and one vector randomly selected from each remaining group. Specifying `distinct` ensures that each selected vector is different from those already selected. Lines 17 and 21 create tuples by matching pairs of individuals from the two groups, which is preferable over merging the groups with `FROM`–`SELECT`. Joining creates a strong association between joined elements while merging does not ensure that the trial vectors in `targets_trial_pairs` are matched with their original vector from `targets`.

As used on lines 18 and 19 of Listing 1, the `crossover_tuple` operator creates new individuals by selecting single genes from each member of the tuple; `per_gene_rate` specifies the probability of a gene being selected from an individual other than the first. Filtering with `best_of_tuple` returns the individual in the tuple with the highest fitness.

3 Validation

While the ESDL description has well-defined behaviour, an empirical analysis is required to verify that this behaviour is actually DE.

As a benchmark, we generated results using the MATLAB® code from [4].[2] A modification was made to this code to return the iteration count. Strategy "DE/rand/1/bin" was used, which matches the system defined in Listing 1 in all but two ways. Stochastic uniform sampling is used in the ESDL system to select base vectors, which may result in some being used multiple times, and mutation vectors may be re-used for separate bases. The benchmark code uses a random permutation of the population to select base vectors and mutation vectors, resulting in each vector being used once as a base vector and each of the two mutation vectors. According to the analysis in [4], these differences should result in slightly better performance by the ESDL system.

A two-dimensional Rosenbrock function (detailed in Appendix A) was selected to be used as a benchmark. Each experiment was run until a fitness of less than 1.0×10^{-6} was achieved or 1000 generations had been evaluated; the result shown is the mean of the number of generations taken over 100 experiments. `esec` was used as the underlying framework; Listing 2 was implemented in Python.

Two parameters were varied: `scale` and the per-gene rate of crossover. The full set of parameters, benchmark results and new results are shown in Table 1.

Table 1: Comparison of varying parameter sets

Scale (F)	**Crossover Rate** (CR)	**Benchmark Result**	**New Result**
0.2	0.8	789	673
0.8	0.8	78.1	69.0
1.2	0.8	122	116
1.8	0.8	255	249
0.8	**0.0**	1000	991
0.8	**0.2**	753	665
0.8	**0.5**	161	153
0.8	**0.8**	78.0	68.3
0.8	**1.0**	55.1	50.4

Figure 2 shows that the ESDL-based implementation produces similar results to the software made available by DE's original authors. The slight performance improvement observed in the ESDL implementation results is most likely a result of allowing base vectors to be used more than once in each generation.

[2] Also available online at `http://www.icsi.berkeley.edu/~storn/code.html`.

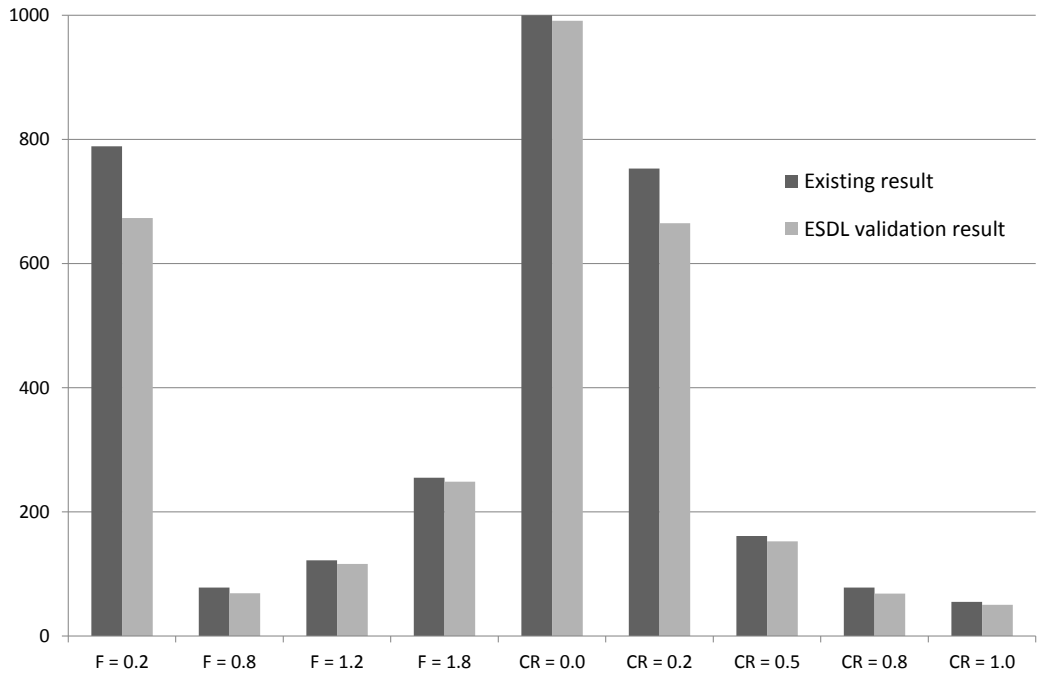

Figure 2: Average generation count for each parameter set

4 Summary

This report provides a description of the Differential Evolution algorithm using ESDL. The classic algorithm is shown concisely in ESDL and the weighted difference-vector mutation operator is specified using pseudocode.

The ESDL description was used with `esec` to produce a working implementation, which was validated against a TSP benchmark problem. Results show that the ESDL implementation is consistent with existing published software.

References

[1] S. Dower and C. Woodward, "Evolutionary System Definition Language," Swinburne University of Technology, Tech. Rep. TR/CIS/2010/1, 2010. 1

[2] R. M. Storn and K. V. Price, "Differential Evolution - a simple and efficient adaptive scheme for global optimization over continuous spaces," Berkeley, Tech. Rep. TR-95-012, 1995. 1

[3] ——, "Differential Evolution: A simple and efficient adaptive scheme for global optimization over continuous spaces," *Journal of Global Optimization*, vol. 11, 1997. 1

[4] K. V. Price, R. M. Storn, and J. A. Lampinen, *Differential Evolution*. Springer, 2005. 1, 2, 4

[5] H. H. Rosenbrock, "An automatic method for finding the greatest or least value of a function," *The Computer Journal*, vol. 3, 1960. 6

A Rosenbrock

Rosenbrock's valley is a well-known classic optimisation problem [5] with many alternative titles, such as De Jong's Function 2 (F2), Rosenbrock's "saddle" and the "Banana" function.

The function is continuous and unimodal, though the non-convex surface gradient can be deceptive to some search methods. The two-dimensional version is given in (1), with renderings shown in Figure 3 against a linear scale and Figure 4 against a logarithmic scale. (Generalisations to n dimensions exist but are not given here.)

$$f(x, y) = 100\left(y - x^2\right)^2 + (1 - x)^2 \tag{1}$$

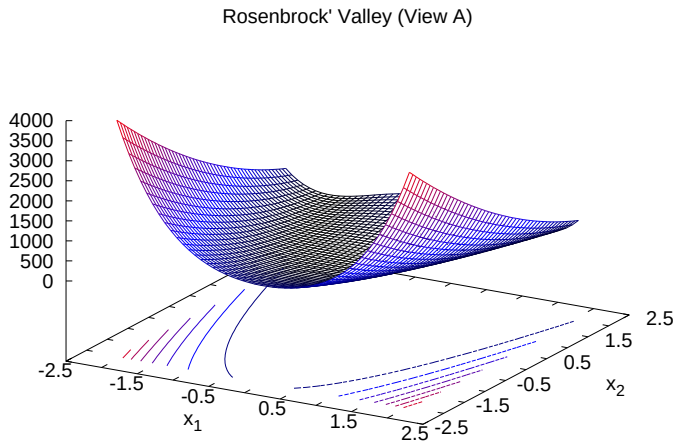

Figure 3: Rendering of the Rosenbrock function on a linear scale

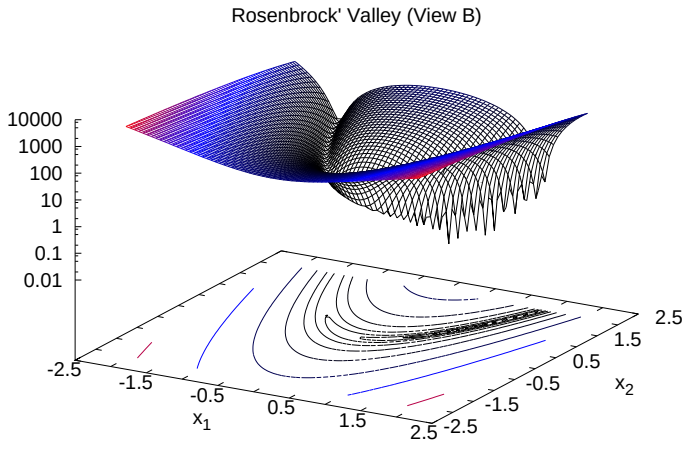

Figure 4: Rendering of the Rosenbrock function on a logarithmic scale

B esec Configuration

Listing 3 shows the configuration used for conducting the experiments in this report. Listing 4 shows the implementation of `mutate_DE` based on the pseudocode in Listing 2.

(The code shown has been simplified and comments removed from the full code available online at http://code.google.com/p/esec/.)

Listing 3: The `esec` configuration used

```
from esec.landscape import real

config = {
    'landscape': { 'class': real.Rosenbrock },
    'system': {
        'definition': r'''
  FROM random_real(length=2,lowest=-2.0,highest=2.0) \
    SELECT (size) population
  YIELD population

  BEGIN GENERATION
    targets = population

    # Stochastic Universal Sampling for bases
    FROM population SELECT (size) bases USING fitness_sus(mu=size)

    # Ensure r0 != r1 != r2, but any may equal i
    JOIN bases, population, population INTO mutators \
      USING random_tuples(distinct=True)

    FROM mutators SELECT mutants USING mutate_DE(scale=F)

    JOIN targets, mutants INTO target_mutant_pairs USING tuples
    FROM target_mutant_pairs SELECT trials \
      USING crossover_tuple(per_gene_rate=CR)

    JOIN targets, trials INTO targets_trial_pairs USING tuples
    FROM targets_trial_pairs SELECT population USING best_of_tuple

    YIELD population
  END GENERATION''',
        'size': 15,
        'mutate_DE': mutate_DE,
        'F': 0.8,
        'CR': 0.8,
    },
    'monitor': {
        'report': 'brief_float+local_float',
        'summary': 'status+brief_float+best_genome',
        'limits': {
            'generations': 1000,
            'fitness': -1.0e-6,
        }
    },
}
```

Listing 4: Python implementation of `mutate_DE`, based on Listing 2

```
def mutate_DE(source, scale):
    for joined_individual in source:
        base, parameter1, parameter2 = joined_individual[:]
        yield type(base)(
            [b + scale * (p1 - p2) for b, p1, p2 in \
                zip(base, parameter1, parameter2)],
            base)
```