Natural Computing Research & Applications Group
School of Computer Science & Informatics
University College Dublin
Technical Report NCRA-TR-2011-05-13

GEM - Grammatical Evolution in Matlab (v0.2)

Erik Hemberg, Michael O'Neill

erik.hemberg@ucd.ie, m.oneill@ucd.ie

School of Computer Science & Informatics
University College Dublin
Belfield, Dublin 4
Co. Dublin
Ireland

Abstract

GEM is an open source implementation of Grammatical Evolution in Matlab developed at UCD's Natural Computing Research & Applications group. As well as providing the characteristic genotype-phenotype mapper of GE a search algorithm engine and a GUI are also provided.

Contents

1 Introduction

Grammatical Evolution in Matlab (GEM) was developed at UCD's Natural Computing Research & Applications group[1]. It is an open source implementation of Grammatical Evolution (GE) [O'Neill and Ryan, 2003] released under

[1] http://ncra.ucd.ie

GNU GPL v3.0 , which provides a search engine framework in addition to the genotype-phenotype mapper of GE.

The goal of GEM is to create a simple, readable and practical Matlab toolbox for Grammatical Evolution. The source is developed in a imperative script style, to allow flexible use for beginners and experts. The toolbox aims to provide demos that allow the users to incorporate their own problems as efficiently as possibly. The intended users of GEM are those wanting to learn and explore Grammatical Evolutions capabilities. The GUI enables GEM to be used in both educational and research contexts. For improvement in speed a version in a compiled language could be faster, e.g. GEVA [O'Neill et al., 2008].

This report serves as an introduction to the GEM software providing guidelines on its installation and use. Some simple demonstration problems are provided to assist the user to gain an understanding of the code while reading the accompanying tutorials which are provided in this document and on the code's website [NCRA, 2010].

The structure of the report is the following. Section 2 has a user start guide. Section 3 describes the GUI and Section 4 on page 5 has an extended user guide. Then following a brief introduction to Grammatical Evolution in Section 5 on page 9, we describe the distribution contents of GEM in Section 6 on page 14, and demos in Section 7 on page 16. Finally release notes in Section 8 on page 18.

2 Getting Started Guide

This section contains a start guide for the user. The steps are:

1. Download *GEM.tgz*
2. Unpack *GEM.tgz*
3. Add the GEM directory to the Matlab path, e.g `addpath('GEM-v0.2')`
4. Open MATLAB (or run `matlab -r gem_gui`)
5. (Run the GUI from MATLAB with `gem_gui_export` (See Section 3))

Now you can try the demos in Section 7 on page 16.

3 GUI

The GUI allows the user access to GEM with some preconfigured settings. Fig. 1 on the next page shows a screenshot of the GUI when it is started. To run an experiment press `Run`. When a problem is selected some default parameters are set.

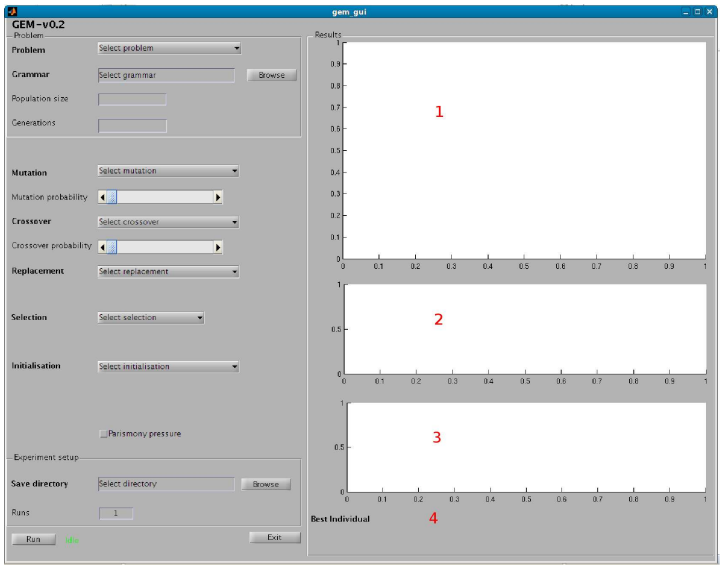

Fig. 1: GEM GUI

3.1 Input Fields

The GUI interacts with GEM sets the parameters described in Section 4.1 on the following page. The parameters available from the GUI are:

Problem to chose from, the default values can be found in the config file of the problem. (See Section 7 on page 16)

Grammar to use. `.bnf` files are listed. (See GRAMMAR_FILE, `param` 1 on the following page)

Population size (See POPULATION_SIZE, `param` 6 on page 6)

Generations (See GENERATIONS, `param` 7 on page 6)

Mutation (See MUTATION_OPERATION, `param` 13 on page 6)

Mutation probability (See MUTATION_PROBABILITY, `param` 12 on page 6)

Crossover (See CROSSOVER_OPERATION, `param` 11 on page 6)

Crossover probability (See CROSSOVER_PROBABILITY, `param` 10 on page 6)

Replacement Some replacement operators can use ELITE_SIZE (See REPLACEMENT, `param` 27 on page 7 and ELITE_SIZE, `param` 16 on page 6)

Selection Some selection operators can use TOURNAMENT_SIZE (See SELECTION_OPERATION, `param` 8 on page 6 and `param` 9 on page 6)

Initialisation Some initialisation operators can set MAX_DEPTH, TAIL_SIZE, and INITIAL_CHROMOSOME_SIZE (See INITIALISATION_OPERATION, `param` 24 on page 7, , `param` 26 on page 7 , `param` 23 on page 7, and `param` 22 on page 7)

Parsimony pressure (See PARSIMONY_PRESSURE, `param` 17 on the next page)

Save directory path to where the output is saved

Runs of the experiment that will be executed with different random seeds

3.2 Output Fields

The output values for the results that are plotted in different panels (denoted by the red digits in Fig. 1):

1. Best individual's fitness and average population fitness. **Note:** when running multi-objective experiments it is still only the best individual according to the ranking which is show.
2. Best individual's number of used codons, average population number of used codons, best individual chromosome size and average population chromosome size.
3. Ratio of number of calls to the mapping function over the number of generated individuals. GEM does not allow invalid individuals, but this shows how many mappings are nessecary in order to generate a valid individual.
4. The phenotype of the current best individual is shown

4 User Guide

This section describes how to setup a new problem and how to create new operators.

GEM mainly uses two structures for storing information, `param` for input parameters and `stats` for output and statistics.

4.1 Input Parameters

The parameters are set in the `param` structure. Currently available parameters in `param`:

1. GRAMMAR_FILE is path to grammar file
2. ORDER sorting order, set to: 'descend' or 'ascend'

3. MIN_FITNESS value of minimum fitness for problem, set to a real value ($-\infty \leq x \leq \infty$)
4. MAX_WRAPS number of wraps, set to a positive integer ($\mathbb{Z}^+$)
5. CODON_SIZE codon range, set to a positive integer ($\mathbb{Z}^+$)
6. POPULATION_SIZE number of individuals in population, set to a positive integer ($\mathbb{Z}^+$)
7. GENERATIONS number of iterations of the algorithm, set to a positive integer ($\mathbb{Z}^+$)
8. SELECTION name of selection operation, set to `@tournament_selection_nsga2` or `@tournament_selection`
9. TOURNAMENT_SIZE number of individuals competing when `tournament_selection` and `tournament_selection_nsga2` is used, set to a positive integer less or equal to POPULATION_SIZE (See, `param` 6) ($\mathbb{Z}^+, TOURNAMENT\_SIZE \leq POPULATION\_SIZE$)
10. CROSSOVER_PROBABILITY probability of crossover, set to a real value between zero and one ($x \in \mathbb{R}, 0 \leq x \leq 1$)
11. CROSSOVER_OPERATION operator used for crossover, set to `@single_point_crossover`
12. MUTATION_PROBABILITY probability of mutating a codon, set to to a real value between zero and one ($x \in \mathbb{R}, 0 \leq x \leq 1$) or a a positive integer ($\mathbb{Z}^+$) for an exact number of mutations per individual;
13. MUTATION_OPERATION mutation operation, set to `@integer_extended_nodal_mutation` or `@uniform_integer_mutation`
14. EXTENDED_NODAL_PROBABILITY probability of an extended nodal mutation event a structural otherwise a structural mutation occurs, set to a real value between zero and one ($x \in \mathbb{R}, 0 \leq x \leq 1$)
15. EXTENDED_NODAL_TRIES number of attempts to find a legal nodal site, set to a positive integer ($\mathbb{Z}^+$)
16. ELITE_SIZE number of elite individuals, set to a positive integer less than POPULATION_SIZE (See, `param` 6) ($\mathbb{Z}^+, ELITE\_SIZE < POPULATION\_SIZE$)
17. PARSIMONY_PRESSURE discriminate on shorter size if fitness is equal when selecting individuals. Set to integer where 0 is false otherwise true
18. FEVALS_TIMES_SINGLE used to set `maxEvalTimeSingle` when using `multicore`
19. NEVALS_AT_ONCE used to set `nrOfEvalsAtOnce` when using `multicore`
20. LOAD_POPULATION name of a saved population, set to a string

21. SAVE_POPULATION name of population to save, set to a string
22. INITIALISATION initialisation operation, set to `@ramped_half_half_initialisation` or `@uniform_initialisation`
23. INITIAL_CHROMOSOME_SIZE initial size of input(chromosome) when `uniform_initialisation` is used, set to a positive integer ($\mathbb{Z}^+$)
24. MAX_DEPTH maximum derivation tree depth when using `@ramped_half_half_initialisation`, set to a positive integer ($\mathbb{Z}^+$)
25. GROW_PROBABILITY probability of Grow algorithm when using the method `@ramped_half_half_initialisation` otherwise Full is used, set to a real value between zero and one ($x \in \mathbb{R}, 0 \leq x \leq 1$)
26. TAIL_SIZE ratio existing chromosome size appended consisting of uniformly random picked codons `@ramped_half_half_initialisation`, set to a a positive real value ($\mathbb{R}, 0 \geq x$)
27. REPLACEMENT name of replacement operation, set to `@replacement_nsga2` or `@generational_replacement`

4.2 Output

The structure `stats` is used for output and statistics. The output consists of a descriptions of the parameters used for the run. Some warnings and errors from the GE algorithm regarding mapping and fitness evaluation are registred:

NON_TERMINALS_LEFT the individuals is not mapped compeltely

PHENOTYPE_DUPLICATE the phenotype has already been evaluated

EVAL MATLAB `eval` error/exception is caught

PUIDs (the ratio of production choices used for each id and front, only for multi-objective problems)

The output for every generation is a row which starts with `ITR` were columns are separated by a "`,`" and the columns indicate:

1. `ITR`
2. Generation number
3. Average used codons in the population
4. Standard deviation of used codons in the population
5. Average depth in the population
6. Standard deviation of depth in the population

7. Number of extra calls to fitness evaluation
8. Number of calls to mapping function
9. Number of calls to expression checking
10. Size of fitness evaluation cache
11. Average fitness in the population
12. Standard deviation of fitness in the population
13. Phenotype of best individual
14. Fitness of best individual
15. Used codons in best individual
16. Depth of best individual

The output for every individual is a row which starts with `IND` were columns are separated by a "`,`" and the columns indicate:

1. `IND`
2. Individual number
3. Chromosome size
4. Phenotype size
5. Fitness value
6. Number of used codons
7. Max derivation tree depth
8. (Rank, if multi-objective)
9. (Distance, if multi-objective)

4.3 Creating A New Problem

When creating a new problem for GEM some things need to be considered.

Configuration The parameters for the problem configuration

Fitness Function The fitness function used to evaluate the solutions and assign fitness

Grammar The grammar used to create solutions

Valid solutions Which solutions are valid given the problem and the environment. This is for cases that cannot be encoded in the grammar, or are more easily implemented in a separate function.

4.4 Creating New Operators

Set the call to the operator in the `param` if there already is an existing operator, e.g. crossover and mutation.

5 Grammatical Evolution

Grammatical Evolution (GE) (e.g., [O'Neill and Ryan, 2003]) is a grammar-based form of Genetic Programming [Poli et al., 2008]. It is inspired by representation in molecular biology and combines this with formal grammars. The GE system is flexible and allows the use of alternative search strategies, whether evolutionary, deterministic or of some other approach. This system also includes the ability to bias the search by changing the grammar used. Since a grammar is used to describe the structures that are generated by GE, editing the grammar modifies the output structures. This constraining power is one of GE's main features. The genotype-phenotype, i.e. input-output mapping means that GE allows search operators to be performed on any representation in the algorithm, e.g. on the genotype (integer or binary chromosomes), as well as on partially generated phenotypes, and on the completely generated derivation trees or phenotypes.

The biological inspiration for GE comes from the generation of a protein from a sequence of DNA, which contains several mappings. A simplified description of the generation of a protein from DNA is described in Tab. 1 on the following page. In Biology, the genotype, DNA is transcribed to RNA, the RNA is translated to amino acids, the amino acids create proteins, and the proteins generate a phenotype. Analogously, for an individual in GE the genotype, binary string, is transcribed to an integer sequence, the integers are translated to production choices via a grammar, and the phenotype is the sentence generated from the grammar.

5.1 GE Control Flow

In GE the control flow of an EA in Fig. 2 on page 11 is extended with a genotype-phenotype mapping, this is the same as "decoding" in a GA.

The canonical GE uses a standard GA as a search engine, with crossover and mutation. The steps in a single iteration of GE are generally:

1. **Initialization** Input in the initial solutions is generated, e.g. uniformly randomly generated integer sequences
2. **Mapping** Mapping via a grammar, e.g. CFG (see Section 5.2 on page 11).
   (a) **Binary to Integer** (Transcription) Binary to integer translation
   (b) **Integer to String** (Translation) Grammar maps integer value to a sentential form (sequence of symbols).
3. **Evaluation** The individual solutions are evaluated.

Tab. 1: Comparison of a generation of a protein and the derivation of a sentence in GE. In Biology the genotype, DNA is transcribed to RNA, the RNA is translated to amino acids, the amino acids create proteins, and the proteins generate a phenotype. Analogously, for an individual in GE the genotype, binary string, is transcribed to an integer sequence, the integers are translated to production choices via a grammar, and the phenotype is the sentence generated from the grammar.

Biology		**Grammatical Evolution**
DNA		Binary string
⇓	*Transcription*	⇓
RNA		Integer sequence
⇓	*Translation*	⇓
Amino Acid		Production choice
⇓		⇓
Protein		Sentence(Program)
⇓		⇓
Phenotypic effect		Evaluated sentence

4. **Operators** Operations on input, e.g. mutation and crossover

   (a) **Selection** Some individuals from the current population are included in a new population. In tournament selection, a tournament size is chosen, and a number of individuals equal to the tournament size are randomly chosen from the population to compete in the tournament. The individual with the best fitness of the individuals selected for the tournament wins the tournament and is selected.

   (b) **Variation operators** Individuals are modified by some operators, e.g. crossover and mutation. In crossover one point in each parent's genotype is selected. The parts on each side of the point are joined to the opposing part from the other parent. This crossover creates two children consisting of one part from each parent. In mutation each input codon has a uniform probability of changing to a new uniform integer value.

   (c) **Replacement** A new population is created from the selected population and from the current population. If generational replacement is used, the entire population is replaced

5. **Termination** When the start symbol has generated a sentence, the genotype (input) is extended by wrapping. An individual that is not completely mapped, even after wrapping, is called an invalid individual.

These steps complete the algorithm.

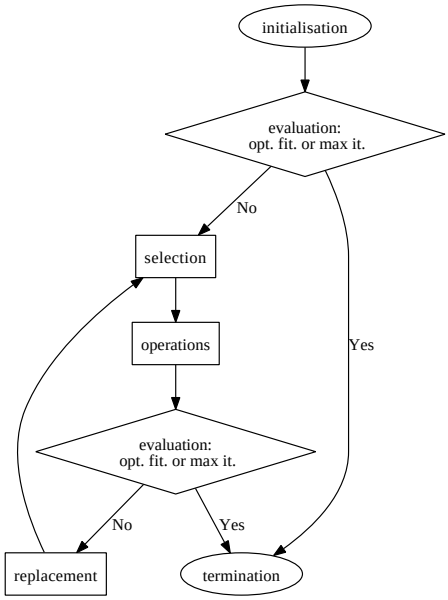

Fig. 2: Flow of a canonical Evolutionary Algorithm. First initialize a population, then evaluate the population, while not optimum found or not max iterations reached: select individual solutions from the population, apply operators to the selected solutions and replace the population.

5.2 Grammar Mapping in GE

The mapping of GE is shown in Fig. 3 on the next page. There are different spaces, genotype, phenotype and fitness.

In a Context-Free Grammar the generation of a word is not dependent on the surroundings, see Booth [Booth, 1967]. A CFG is a four tuple $G = \langle N, \Sigma, R, S \rangle$, where:

- N is a finite non-empty set of non-terminal symbols.
- Σ is a finite non-empty set of terminal symbols and $N \cap \Sigma = \emptyset$, the empty set.
- R is a finite set of production rules of the form $R : N \mapsto V^* : A \mapsto \alpha$ or (A, α) where $A \in N$ and $\alpha \in V^*$. V^* is the set of all strings constructed from $N \cup \Sigma$ and $R \subseteq N \times V^*$, $R \neq \emptyset$.
- S is the start symbol, $S \in N$.

"Context-Free" means that for a rule $A \to \alpha$, A can always be replaced by α, regardless of context [Harrison, 1978].

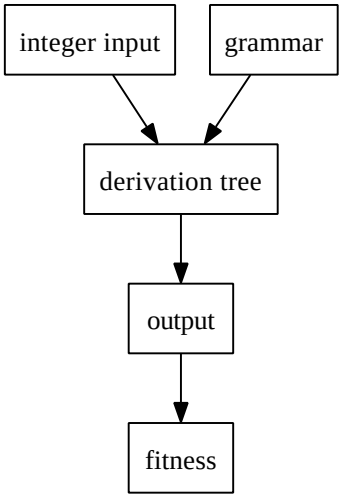

Fig. 3: GE mapping flow: input and grammar are mapped to output that is evaluated and assigned a fitness

5.2.1 The Grammar

For GE a suitable BNF grammar definition must exist. How much domain knowledge to incorporate is decided by the practitioner, who also defines how general or specific the Backus Naur Form (BNF) grammar is.

In GE, a BNF-grammar describes the output sentences that can be produced by the system, as well as the grammar bias.

The Grammar 1 on the following page can be used to generate boolean expressions, and `<expr>` can be transformed into one of three rules. It can become either `( <expr> <biop> <expr> )`, `<uop> <expr>`, or `<bool>`. A grammar can be represented by the tuple $\langle N, \Sigma, R, S \rangle$.

N = { `<expr>, <biop>, <uop>, <bool>` }
Σ = { `and, or, xor, nand, not, true, false, (, )` }
S = { `<expr>` }

The code produced after mapping a BNF-grammar in GE will consist of elements of the terminal set Σ. The grammar is used in a generative approach, whereby the evolutionary process evolves the production rules to be applied at each stage of a derivation process, starting from the start symbol, until a complete program is formed. The mapping (derivation) is complete when the sentence is one that is comprised of only elements of Σ.

5.2.2 The Mapping

The genotype is used to map the start symbol into a sentence, by the BNF-grammar. The mapping is done by reading input(*codons*) to generate a corre-

```
<expr> ::= ( <expr> <biop> <expr> )
| <uop> <expr>
| <bool>
<biop> ::= and
| or
| xor
| nand
<uop> ::= not
<bool> ::= true
| false
```

Grammar 1: Example of a grammar for boolean expressions. `<expr>` has three production choices, `<biop>` has four production choices, `<uop>` has one production choice and `<bool>` has two production choices.

sponding integer value, from which an appropriate production rule is selected by using the following mapping function:

$$Rule = c \ mod \ r \tag{1}$$

where c is the codon integer value, and r is the number of rule choices for the current non-terminal symbol.

Consider the following rule from the grammar in Grammar 1. Given the non-terminal `<biop>`, which describes the set of boolean operators that can be used, there are four production rules to select from. The choices are labeled from zero.

```
<biop> ::= and        (0)
         | or         (1)
         | xor        (2)
         | nand       (3)
```

If the codon being read produces the integer 6, then Eq. (1) gives $6 \ mod \ 4 = 2$, which would select rule `(2) xor`. In the derivation `<biop>` is replaced with `xor`.

Each time a production from a rule with more than one production choice has to be selected to transform a non-terminal, another codon is read. In this way the system traverses the genome.

The mapping is deterministic, i.e. the same input sequence will map to the same output sequence if the grammar is unchanged, each time the same codon is expressed it will generate the same integer value. But depending on the derivation context, i.e. the current non-terminal to which the codon is being applied, a different production rule may be selected, this is called intrinsic polymorphism [O'Neill et al., 2003].

While the mapping process in GE occurs and a sentence is being built, it can also be represented as a derivation tree. A concrete example of mapping in GE is shown in Fig. 4 on the next page.

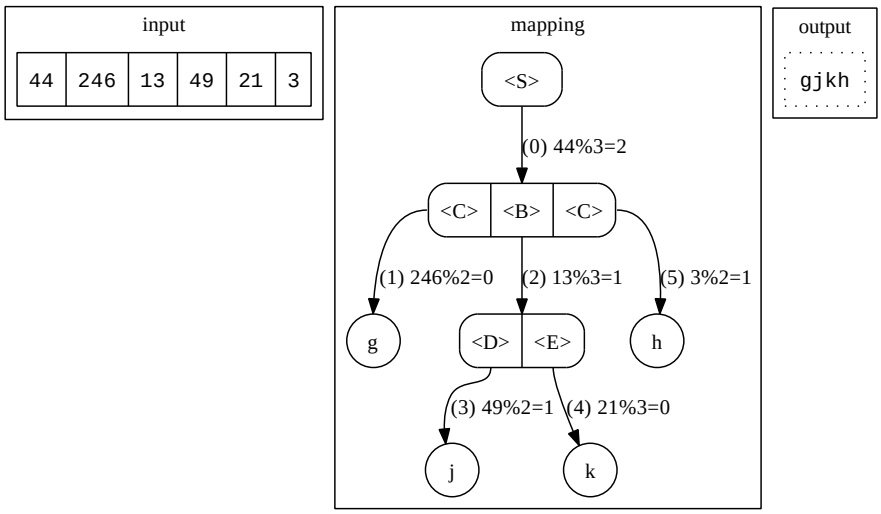

Fig. 4: Example of a derivation tree that generates a word, `gjkh`, using Grammar 2 on the next page

6 Distribution Contents

In the main distribution directory you will see the following files:

`Contents.m` Matlab contents file

`COPYING` contains the text of the GNU GPL version 3.

`create_individual.m` creating new individuals

`LICENSE` provides the Copyright notice and Licence information for the distribution.

`gema.m` the entry file, contains the algorithm

`gem_gui.m` the GUI

`gem_gui.mat` resource file for the GUI

`get_grammar.m` parse a BNF grammar

`map_individual.m` the GE mapping from genotype to phenotype of an individual

`financial_modelling` files for financial modelling example, see Section 7.4 on page 17

`table.csv` data file

```
<S> ::= <C>
      | <C><C>
      | <C><B><C>
<B> ::= <D>
      | <D><E>
      | <E>
<C> ::= g
      | h
<D> ::= j
      | k
<E> ::= k
      | l
      | m
```

Grammar 2: Example of a grammar for words.

`daily_return.m` Calculates the daily return $dr = (p(t) - p(t-1)/p(t-1)$

`financial_modelling_config.m` Configurations for financial modelling problem

`financial_modelling_ff.m` Calls the fitness function for financial modelling

`individual_validation.m` Check solution validity

`simple_moving_average.m` Calculate simple moving average of the data with the window size

`trading_fitness.m` Calculate the fitness when trading with buy and sell signals

`grammars` contains grammars

`sr.bnf` symbolic regression example grammar

`sr_xy.bnf` multiobjective example grammar

`trade_rules.bnf` financial modelling grammar

`docs` documentation for GEM

`multiobjective_function` files for multiobjective example, see Section 7.3 on page 17

`individual_validation.m` Check solution validity

`multiobjective_function_config` Configurations for multiobjective function

`multiobjective_function_ff.m` Calls the fitness function

`operators` the operators

`generational_replacement.m` Replace the entire old population with the new population

`integer_extended_nodal_mutation.m` Mutates the individuals n times on leafs or nodes

`ramped_half_half_initialisation` Ramps the population to full depth or grow to maximum depth.

`replacement_nsga2.m` Replacement using NSGA-II

`single_point_crossover.m` One crossover point is randomly picked for each parent.

`tournament_selection.m` Select a new population by selecting individuals using a tournament

`tournament_selection_nsga2.m` Select a new population using a tournament based on rank

`uniform_initialisation.m` Uniformly generate integer inputs

`uniform_integer_mutation.m` Mutates an individual uniformly with uniform probability

`symbolic_regression` files for the symbolic regression example, see Section 7.1 on the following page.

`symbolic_regression_config.m` Configurations for symbolic regression problem

`symbolic_regression_ff.m` Calls the fitness function

`individual_validation.m` Check solution validity

`symbolic_regression_multicore` files for the symbolic regression using multicore example, see Section 7.2 on the next page

`symbolic_regression_multicore_config.m` Configurations for symbolic regression multicore problem

`symbolic_regression_multicore_ff.m` Calls the fitness function

`individual_validation.m` Check solution validity

7 Demos

A number of different problems that highlight the features of GEM are presented in this section:

- Multicore use
- Mulitobjective function
- Financial application

By default all the demos disallow identical individuals.

NOTE: the default settings of the demos are non-optimal and used for purely illustrational purposes.

7.1 Symbolic Regression

The implemented problem is Symbolic Regression [Koza, 1992]. The sextic function is used here $\hat{f} = x + x^2 + x^3 + x^4 + x^5 + x^6$ with 20 fixed values of $x = [-1.0, -0.9, \ldots, 1.0]$. Fitness is calculated by the mean squared error, $F = 1/|x| \sum_{i=0}^{|x|}(f(x_i) - \hat{f}(x_i))^2$ and the fitness is minimized. The grammar used is:

```
<E> ::= ( <E> <O> <E> ) | <V>
<O>   ::= + | - | .*
<V> ::= x | 1 | 0
```

7.2 Symbolic Regression Multicore

This problem uses the multicore package `http://www.mathworks.com/matlabcentral/fileexchange/13775` by Markus Buehren to parallelize the fitness function evaluations. The setup is the same as in Section 7.1. See the multicore documentation for more information.

7.3 Multiobjective Function

From Zitzler et al. [Zitzler et al., 2000] Comparison of multiobjective EAs. The function is the first test function T_1. Minimize $F = [f_1(x_1), f_2(\mathbf{x})]$ where

$$\begin{aligned} f_1(x_1) =& x_1 \\ f_2(\mathbf{x}) =& g(\mathbf{x})h(f_1(x_1), g(\mathbf{x})) \\ g(x_2, \ldots, x_m) =& 1 + 9 \times \sum_{i=2}^{m} x_i \\ h(f_1, g) =& 1 - \sqrt{f_1/g} \end{aligned}$$

where $\mathbf{x} = [x_1, \ldots, m], m = 30, x \in [0, 1]$. The pareto optimal front is formed with $g(\mathbf{x}) = 1$ The grammar generates an array of simplified real values:

```
<A> ::= x = [<b>];
<b> ::= <d>,<d>,<d>,<d>,<d>,<d>,<d>,<d>,<d>,<d>,<d>,<d>,<d>,<d>,<d>,
        <d>,<d>,<d>,<d>,<d>,<d>,<d>,<d>,<d>,<d>,<d>,<d>,<d>,<d>,<d>,<d>
<d> ::= 0 | 0.1 | 0.2 | 0.3 | 0.4 | 0.5 | 0.6 | 0.7 | 0.8 | 0.9 | 1
```

The representation of real values in this grammar is simplified to values with one decimal.

7.4 Financial Modelling

In this problem you are evolving a rule to predict a buy or sell signal. The fitness function is maximized and calculated as follows. From the raw data you first get the daily return, which is given by:

$$return(t) = price(t) - price(t-1)/price(t-1)$$

Then you are applying the evolved rule using a moving-window to cover all the training cases, and you maintain a sum whose elements are calculated from each point in the time-series by multiplying the signal (1 or -1) that was generated in day $t-1$ with the daily return of day t. The you get the average by diving with the number of training days. This quantity should be maximised.

A moving average is a type of finite impulse response filter used to analyze a set of data points by creating a series of averages of different subsets of the full data set. It is commonly used with time series data to smooth out short-term fluctuations and highlight longer-term trends or cycles. The threshold between short-term and long-term depends on the application. Given a series of numbers and a fixed subset size, the moving average can be obtained by first taking the average of the first subset. The fixed subset size is then shifted forward, creating a new subset of numbers, which is averaged.

The grammar below is used, where the closing price of a stock in a particular day, and a simple moving average is compared to determine the signal. It is parametrised with the number of days used for smoothing - starting from 5 to 100 with a step of 5.

```
<CODE> ::= for t = 1:size(data,1) <expr> end;
<expr> ::= if <op>(smas(t,<integer>), data(t)) <expr> else <expr> end;
         | <signal>
<signal> ::= signals(t) = 1; | signals(t) = -1;
<integer> ::= 5 | 10 | 15 | 20 | 25 | 30 | 35 | 40 | 45 | 50 | 55
            | 60 | 65 | 70 | 75 | 80 | 85 | 90 | 95 | 100
<op> ::= le | ge
```

`smas(t,i,data)` is a data structure containing all the possible moving averages. This will increase the memory usage but significantly reduces the run time.

table.csv contains daily data from S&P 500 from 1980 to 2011[2]. This is loaded into `data` and then the adjusted closing price is used.

8 Release notes

This section contains release notes.

8.1 Improvements from GEM v0.1

- More demos
- Multiobjective fitness function and operators

[2] `http://finance.yahoo.com/q/hp?s=^GSPC&a=00&b=3&c=1980&d=04&e=30&f=2011&g=d`

8.2 Known issues

- The names of the rules in grammar can change the ordering. This is important since in BNF form the first rule is the start symbol, and most thevrefore be maintained by the parsing

Acknowledgments

We would like to thank past and present members of the UCD Natural Computing Research & Applications group. GEM is influenced by ponyGE [Hemberg and McDermott, 2010] and GEVA [O'Neill et al., 2008]. This work is done under Science Foundation Ireland Grant No. 08/IN1/I1868.

References

Taylor L. Booth. *Sequential machines and automata theory.* Wiley, 1967.

MA Harrison. *Introduction to formal language theory.* Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1978.

Erik Hemberg and James McDermott. ponyGE, November 2010. URL `http://code.google.com/p/ponyge/`.

John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection (Complex Adaptive Systems).* The MIT Press, December 1992. ISBN 0262111705.

NCRA. NCRA Software, November 2010. URL `http://www.ncra.ucd.ie/Site/Software.html`.

M. O'Neill, C. Ryan, M. Keijzer, and M. Cattolico. Crossover in Grammatical Evolution. *Genetic Programming and Evolvable Machines*, 4(1):67–93, 2003.

Michael O'Neill and Conor Ryan. *Grammatical Evolution: Evolutionary Automatic Programming in an Arbitrary Language.* Kluwer Academic Publishers, Norwell, MA, USA, 2003. ISBN 1402074441.

Michael O'Neill, Erik Hemberg, Conor Gilligan, Eliott Bartley, James McDermott, and Anthony Brabazon. Geva:grammatical evolution in java. *SIGEVOlution*, 3(2):17–23, Summer 2008.

Riccardo Poli, William B. Langdon, and Nicholas Freitag McPhee. *A field guide to genetic programming.* Published via `http://lulu.com` and freely available at `http://www.gp-field-guide.org.uk`, 2008. URL `http://www.gp-field-guide.org.uk`. (With contributions by J. R. Koza).

E. Zitzler, K. Deb, and L. Thiele. Comparison of multiobjective evolutionary algorithms: Empirical results. *Evolutionary computation*, 8(2):173–195, 2000. ISSN 1063-6560.