

perm_mateda: User's Manual

EKHINE IRUROSZKI, Intelligent Systems Group, Basque Center for Applied Mathematics

JOSU CEBERIO, Intelligent Systems Group, Faculty of Computer Science, University of the Basque Country UPV/EHU

JOSEAN SANTAMARIA, Intelligent Systems Group, Faculty of Computer Science, University of the Basque Country UPV/EHU

ROBERTO SANTANA, Intelligent Systems Group, Faculty of Computer Science, University of the Basque Country UPV/EHU

ALEXANDER MENDIBURU, Intelligent Systems Group, Faculty of Computer Science, University of the Basque Country UPV/EHU

This document describes how to install and use `perm_mateda`, a Matlab package for the optimization of permutation problems. `perm_mateda` is an extension of `MATEDA-2.0`, and it implements permutation-based probabilistic models (e.g. Mallows model) for different distances defined on permutations. The package is conceived for allowing the incorporation, by the user, of different combinations of selection, learning, sampling, and local search procedures. Similarly, additional permutation-based models can be added. Finally, the package includes the implementation of some of the most common permutation-based problems found in real-world applications.

Additional Key Words and Phrases: estimation of distribution algorithms, Mallows and Generalized Mallows models, optimization, permutation-based problems, Matlab

ACM Reference Format:

Ekhine Irurozki, Josu Ceberio, Josean Santamaria, Roberto Santana, Alexander Mendiburu, 2017. `perm_mateda`: A matlab toolbox of estimation of distribution algorithms for permutation-based combinatorial optimization problems. *ACM Trans. Math. Soft.* 0, 0, Article 0 (0), 10 pages.
DOI: <http://dx.doi.org/10.1145/0000000.0000000>

1. INTRODUCTION

Permutation problems are ubiquitous in the industry, society, and research. In transportation problems, sequence alignment, scheduling, or ranking determination, the goal is usually to find the permutation that optimizes a given criterion expressed as an objective function. There are a variety of optimization approaches to deal with these problems. One class of optimization algorithms that has recently deserved considerable attention comprises methods that learn a probability model on a set of selected solutions and use the model to organize a more efficient search. These algorithms are usually called Estimation of Distribution Algorithms (EDAs) [Larrañaga and Lozano 2002; Lozano et al. 2006; Pelikan et al. 2002].

This work has been partially supported by the Research Groups 2013-2018 (IT-609-13) and BERC 2014-2017 programs (Basque Government), and TIN2016-78365-R and Severo Ochoa excellence accreditation SEV-2013-0323 programs (Spanish Ministry of Economy and Competitiveness).

Author's addresses: E. Irurozki, Basque Center for Applied Mathematics (BCAM), Alameda de Mazarredo 14, 48009 Bilbao, Bizkaia, Spain; J. Ceberio, J. Santamaria and R. Santana, Computer Science and Artificial Intelligence Department, Faculty of Computer Science, UPV/EHU, 20018 Paseo M. Lardizabal 1, Donostia - San Sebastian, Gipuzkoa, Spain; A. Mendiburu, Computer Architecture and Technology, Faculty of Computer Science, UPV/EHU, 20018 Paseo M. Lardizabal 1, Donostia - San Sebastian, Gipuzkoa, Spain.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 0 ACM. 1539-9087/0/-ART0 \$15.00

DOI: <http://dx.doi.org/10.1145/0000000.0000000>

`perm_mateda` implements different variants of EDAs for optimizing permutation problems. This User's Manual focuses on the technical details needed to install and use this Matlab toolbox. The manual has been conceived as a supplementary material to the paper where `perm_mateda` is formally introduced¹. For an introduction to probabilistic models for permutations, discussion on related work, and consideration of the advantages and drawbacks of these algorithms, we recommend the reader to address the original paper. Furthermore, `perm_mateda` inherits several of the functions and procedures² of MATEDA-2.0. Therefore, for a detailed review of MATEDA-2.0 functionalities, we recommend the reader the original MATEDA-2.0 paper by Santana et al. [2010].

2. INSTALLING PERM.MATEDA

In order to use `perm_mateda`, the user has to download it (*perm_mateda.zip*) and unzip the file in a working directory. Then:

- (1) Run MATLAB.
- (2) Inside MATLAB, change to the directory containing `perm_mateda`. It can be done by using the left panel (Current Folder), or throughout the command window (`cd` command).
- (3) Run `InitEnvironments.m` by typing `InitEnvironments` in MATLABs command window.

Please note that steps 2 and 3 must be completed each time Matlab is started.

The folder `permutations/Scripts_Perm_Mateda/` contains three EDA examples for three permutation problems, which have been used to run the experiments in the original paper. If the reader is familiar with the Matlab environment and EDAs for permutation problems, these examples should be sufficient for a basic EDA implementation. Otherwise, the following sections provide a detailed explanation of the `perm_mateda` components.

3. DEFINING AND EXECUTING AN EDA FOR A PERMUTATION PROBLEM

`perm_mateda` inherits from MATEDA-2.0 the same organization of the EDA components. Algorithm 1 shows the pseudocode of a general EDA where each of the main methods that can be implemented by the practitioner are emphasized (*italics*).

3.1. Implementation of a general EDA

The general EDA program `RunEDA.m` is called as:

```
[AllStat,Cache] = RunEDA(PopSize,n,F,Card,cache,edaparams);
```

where the input and output parameters follow the description below.

3.1.1. Input parameters

- *PopSize*: Size of the population (number of individuals or solutions).
- *n*: Number of variables used to represent a solution.
- *F*: Name of the Matlab file that implements the objective function used to evaluate the quality of a solution (individual).
- *Card*: Cardinalities of the variables.

¹Irurozki et al. `perm_mateda`: A matlab toolbox of EDAs for permutation-based problems. 2017. Submitted for publication.

²Some MATEDA-2.0 functions that are not relevant for the solution of permutation problems, e.g., those require to create Bayesian network models, have not been included in `perm_mateda`

ALGORITHM 1: General pseudocode of Estimation of Distribution Algorithms (EDAs)

```

 $k = 0$ ;
Generate an initial population  $D_k$  using a seeding method;
If required, apply a repairing method to  $D_k$ ;
Evaluate (all the objectives of) population  $D_k$  using an evaluation method;
If required, apply a local optimization method to  $D_k$ ;
while The termination criterion is not met do
    Select a set  $D_k^S$  of solutions from  $D_k$  according to a selection method;
    Compute a probabilistic model of  $D_k^S$  using a learning method;
    Sample a  $D_{Sampled}$  population using a sampling method;
    If required, apply a repairing method to  $D_{Sampled}$ ;
    Evaluate (all the objectives of) population  $D_{Sampled}$  using an evaluation method;
    If required, apply a local optimization method to  $D_{Sampled}$ ;
     $k = k + 1$ ;
    Create  $D_k$  population from populations  $D_{k-1}$  and  $D_{Sampled}$  using a replacement method;
end

```

— *cache*: A vector specifying which components of the algorithm will be stored. $cache(i) = 1$ determines that the i -th component of EDA ($i = 1, 2, \dots, 5$) will be saved in each generation ($cache(i) = 0$ otherwise). The five components considered are the following:

- (1) Entire population.
- (2) Selected population.
- (3) Probabilistic model.
- (4) Fitness values of the entire population.
- (5) Fitness values of the selected population.

— *edaparams*: An array of cells specifying all the components and parameters used by the EDA. The i -th row of *edaparams* has the form: $\{type_of_method, name_of_implementation, implementation_parameters\}$.

+ *type_of_method* defines an EDA component. It is a string that can take one of the following values:

- 'seeding_pop_method'
- 'sampling_method',
- 'repairing_method'
- 'local_opt_method'
- 'replacement_method'
- 'selection_method'
- 'learning_method'
- 'statistics_method'
- 'verbose_method'
- 'stop_cond_method'

+ *name_of_implementation* is the name of a Matlab program where the EDA component has been implemented. It can be added by the user or be one of the methods included by default in perm_mateda.

+ *implementation_parameters* is a cell array containing the parameters used by the program *name_of_implementation* and are passed to it during the execution of RunEDA.

3.1.2. Output parameters

- *AllStat*: For each generation k , the cell array $AllStat\{k, : \}$ contains the following information:
 - + $AllStat\{k, 1\}$: Matrix of 5 rows and *number_objectives* columns. Each row shows information about maximum, mean, median, minimum, and variance values of the corresponding objective in the current population.
 - + $AllStat\{k, 2\}$: Stores the best individual.
 - + $AllStat\{k, 3\}$: Number of different individuals.
 - + $AllStat\{k, 4\}$: Matrix of 5 rows and n columns. Each row shows information about maximum, mean, median, minimum, and variance values of the corresponding variable in the current population.
 - + $AllStat\{k, 5\}$: Number of function evaluations until generation k .
 - + $AllStat\{k, 6\}$: Matrix with the time, in seconds, spent at the main EDA steps, each of the 8 columns stores the times elapsed in the following steps: sampling, repairing, evaluation, local optimization, replacement, selection, learning and total (which represents the time spent by the previous 7 steps and other EDA minor operations).
- If $cache(i) = 1$, for each generation k , $Cache\{i, k\}$ will store the corresponding component of the EDA. The order is the one presented in the explanation of the input parameter *cache*.

4. AN INTRODUCTORY EXAMPLE

Here we present an introductory example of how to define and execute an EDA for a permutation problem. Details on the methods used in this example are explained later in this manual. The example is intended to serve as a general overview of all the steps involved in the solution of the problem. In this section, we briefly explain the main stages of this code.

```

1 global LOPInstance
2 ReadLOPInstance('permutations/Problems/Instances/LOP/N-be75eec');
3 [matrix, NumbVar] = LOPInstance{:};
4 Card = [ones(1,NumbVar); NumbVar*ones(1,NumbVar)];
5 PopSize = 10*NumbVar;
6 F = 'EvalLOP';
7 cache = [0,0,1,0,0];
8 edaparams{1} = {'seeding_pop_method', 'InitPermutations', {}};
9 selparams(1:2) = {NumbVar/PopSize, 'fitness_ordering'};
10 edaparams{2} = {'selection_method', 'truncation_selection', selparams};
11 edaparams{3} = {'replacement_method', 'pop_aggregation_theta', {'fitness_ordering'}};
12 edaparams{4} = {'stop_cond_method', 'max_gen', {100}};
13 global FitnessImprovement
14 edaparams{5} = {'learning_method', 'Mallows_kendall_learning', {0.001, 10, 100, 'Borda', 0.1}};
15 edaparams{6} = {'sampling_method', 'Mallows_kendall_sampling', {PopSize-1, 1}};
16 FitnessImprovement = 0
17 [AllStat, Cache] = RunEDA(PopSize, NumbVar, F, Card, cache, edaparams)

```

The first step defines the optimization problem that will be addressed and uploads an instance of that problem. In this example, the Linear Ordering Problem (LOP) (see Section 7 for description of this and other problems) has been considered. In lines 1-3, a global variable that stores the characteristics of the instances is defined and the instance is loaded (from file 'N-be75eec'). This instance corresponds to a problem of size 50.

The next step deals with the general parameters of the EDA: size of the solution (individual) and cardinality of the variables (given by the instance), size of the population, the evaluation function to be used, and the cache (lines 4-7). The meaning of the *cache* parameter has been explained in Section 3.1.1. In this example, by setting `cache(3) = 1`, we indicate the algorithm to store only the probabilistic model learned at each generation.

Now, we tackle the design of our particular EDA. First, line 8, the method used to initialize the population is chosen. In this case, a random initialization has been preferred. With respect to the selection operator, lines 9-10, the `NumbVar/PopSize` percentage of the solutions with the best fitness are selected by truncation. In relation to the replacement operator, line 11, the population is updated by adding the newly created solutions and preserving the `PopSize` solutions with the best fitness. As regards the stopping criterion, line 12, this is set to 100 generations.

Finally, we specify the probabilistic model that is going to be used. In this example, we propose using the Mallows model under the Kendall- τ distance. To this end, the learning and sampling methods must be detailed and, accordingly, any global variable required by the learning or sampling procedure should be defined previously. In this example, the global variable `FitnessImprovement` (line 13) is used to exchange information between the replacement method and the learning procedure. This allows the implementation of adaptive learning algorithms that can modify its internal parameters if, for example, there has been any improvement in the population (which is the case shown in the example).

With respect to the learning method (line 14), the *Borda* function is used to compute an approximation of the central permutation. The spread parameters are estimated in the range $[0.001, 10]$ with a maximum of 100 iterations of the Newton-Raphson algorithm. An initial delta value for theta variation is set to 0.1. Every iteration without improvement of the best solution, the lower θ (set at the beginning as 0.001) is increased by 0.1. If a better solution is found, the lower θ is reset to the original value (0.001 in this example). For a list of other learning methods and a description of the parameters they use see Section 5.2. For a detailed explanation of these methods and relevant bibliographic references, we refer the interested reader the original paper.

According to the sampling step (line 15), `PopSize-1` new solutions are sampled from the model at each generation, using the sampling method conceived for the Mallows model with Kendall distance. For a list of sampling methods see Section 6.

In line 16, the `FitnessImprovement` variable is initialized and this step finishes the configuration of the algorithm.

Once the algorithm has been configured, it is executed by calling the *RunEDA* function (line 17) using the input and output parameters explained in Section 3.1.1 and Section 3.1.2.

5. LEARNING MODELS IN PERM_MATEDA

In an EDA, the learning procedure involves representing the most salient patterns of the selected solutions using a probabilistic graphical model. The model captures (or tries to capture) the dependence and independence relationships between the variables. In probabilistic models defined for permutation problems, the probabilities are associated to *distances* between permutations. Therefore, different distances determine different probabilistic models.

5.1. Probabilistic models for permutations: distances

In `perm_mateda`, three distances are implemented: Kendall's, Cayley, and Ulam³.

³See the original paper for a discussion and references on the different distances.

Kendall's- τ . The two auxiliary methods are (1) `vVector` for extracting the decomposition vector V associated to a given permutation, and (2) `GeneratePermuFromV` for obtaining a permutation consistent with V (see Table II). These conversions between σ and $X(\sigma)$ can be run in time $O(n^2)$.

Cayley. Two methods analogous to those included for the Kendall's- τ distance are introduced (see Table II), namely (1) `xVector`, which obtains the decomposition vector of a permutation and (2) `GeneratePermuFromX`, obtains a permutation associated to a given X decomposition (note that we will obtain one among the possibly many compatible permutations). These conversions between σ and $X(\sigma)$ can be run in time $O(n)$.

Ulam. In the case of the Ulam distance, for which no distance decomposition vector is defined, we need to take different strategies to work with the distribution, namely (1) `ComputeFerrerShapes`, computes the distribution of the Ulam distances (this function stores its result in files and are reusable, so it is computed only the first time), and (2) `GeneratePermuAtLIS`, to generate a permutation at a given Ulam distance.

Table I: Learning methods and related auxiliary functions for the different model and distance combinations.

Main Methods	Description
<code>Mallows_{kendall,cayley,ulam}.learning</code> <code>GMallows_{kendall,cayley}.learning</code>	Main methods for the learning step.
Methods called by the main learning method	Description
<code>{Borda,SetMedianPermutation,BestValue}</code> <code>CalculateThetaParameter{K,GK,C,GC,U}</code>	Methods to approximate the central permutation. Learn θ parameter(s).
<code>{Kendall,GKendall,Cayley,Ulam}ThetaFuncion</code> <code>{Kendall,GKendall,Cayley,Ulam}ThetaDevFuncion</code>	Auxiliary methods used to define the functions to be solved by applying Newton-Raphson.
Auxiliary functions	Description
<code>{Kendall,cayley,ulam}.distance</code> <code>Mallows_{kendall,cayley,ulam}.Probability</code> <code>GMallows_{kendall,cayley}.Probability</code> <code>Mallows_{kendall,cayley,ulam}.Probability_exp</code> <code>GMallows_{kendall,cayley}.Probability_exp</code> <code>CalculatePsiConstants{K,GK,C,GC}</code>	Distance between two permutations. Probability assigned by the model to a particular permutation. Value of the exponential part of the probability equation. Normalization term.

5.2. Learning methods

The parameters of the learning algorithms can be divided into two groups: first, those related to the EDA, and second, some specific parameters, which are passed in the *learning_params* vector and which shape the way that the learning algorithm is implemented. The following example shows how to call the method for learning the Generalized Mallows (GM) model under the Kendall's- τ distance:

```
[model] = GMallows_kendall_learning(k,NumbVar,Card,SelPop,AuxFunVal,learning_params)
```

where k is the current generation of the EDA, and *NumbVar* is the number of variables of the permutation. *Card* is a matrix with the dimensions of all the variables. *SelPop* parameter is the population of permutations from which the model is learned, *AuxFunVal* are the evaluation values of the solutions in the population for the selected problem, and *learning_params* is the set of additional learning parameters (which can be different for each model).

In all the cases –except for GM with Cayley– *learning_params* consists of the following: the first three parameters are related to the Newton-Raphson method used for

estimating the θ value: *initialTheta* and *upperTheta* are the interval values for θ , and *maxit* is the maximum number of iterations allowed. The fourth parameter, *RankingFun*, indicates the function to be used for approximating the central permutation (σ_0). In the case of GM with Cayley, as Newton-Raphson is not needed, only one parameter is used (*RankingFun*).

In Table I, the different learning methods, together with related auxiliary functions are presented⁴. Different functions have been defined for each step (or even sub-step) of the learning phase, taking into account the modular design of the Mateda-2.0 toolbox. Particularly, the main learning method calls two secondary methods, one for approximating the central permutation and the other for estimating the spread parameter(s). Moreover, as this last method uses Newton-Raphson, it calls *ThetaFunction* and *ThetaDevFunction* methods (except for GM-Cayley). This way, it is straightforward to modify some parts of the code or write new code: just change the particular method and the call to it. Replacing Newton-Raphson by another numerical method, or including a new proposal to obtain the central permutation are interesting extensions than can be made and, at the same time, good examples of the modularity of this toolbox.

6. SAMPLING MODELS IN PERM_MATEDA

The sampling step consists of generating permutations from the model obtained in the learning stage. The sampling algorithm also depends on the distance considered in the model. For instance, the method for sampling the GM model with the Kendall's- τ distance is called:

```
[pop] = GMallows_kendall_sampling(n,model,Card,AuxPop,AuxFunVal,sampling_params)
```

where k is the current generation of the EDA, and *NumbVar* is the number of variables of the permutation. *Card* is a matrix with the dimensions of all the variables, *AuxPop* is the population from which the model was learned, and *AuxFunVal* is the evaluation (fitness values) of the *AuxPop* data set. The *sampling_params* are the additional sampling parameters. In this case, our methods only require one parameter, N , which is the number of new individuals to be generated.

Table II shows the different sampling methods, together with related auxiliary functions⁵.

7. PERMUTATION PROBLEMS

Mateda-2.0 toolbox includes implementations of discrete and continuous optimization problems. Following the same idea, we have incorporated four problems defined on permutations: Traveling Salesman Problem (TSP), Permutation Flowshop Scheduling Problem (PFSP), Linear Ordering Problem (LOP), and Quadratic Assignment Problem (QAP). These problems are challenging and they appear frequently in the literature.

Table III describes the modules implemented for each optimization problem. Each problem is defined by two modules: one for reading the instance file and processing the parameters (*Read*{TSP,PFSP,LOP,QAP}instance), and the other for evaluating the solutions (permutations) (*Eval*{TSP,PFSP,LOP,QAP}).

⁴In order to avoid repeating each model - distance combination, we represent the different options between curly brackets. For example, the name of the learning method for GM with Cayley will be *GMallows.cayley.learning*.

⁵As with the learning phase, we represent the different options between keys, V (or v) is for Kendall's- τ and X (or x) is for Cayley

Table II: Sampling methods and related auxiliary functions for the different model and distance combinations.

Main methods	Description
Mallows_{kendall,cayley,ulam}_sampling GMallows_{kendall,cayley}_sampling	Main methods for the sampling step.
Auxiliary functions	Description
{v,x}Vector	Decompose the permutation
GeneratePermuFrom{V,X}	Obtain a new permutation from the {v,x}Vector.
ComputeFerrerShapes	Compute the auxiliary structures to get the distribution of the Ulam distances
GeneratePermuAtLIS	Generate a permutation at a given Ulam distance

Table III: Functions defined to read the instances and evaluate a solution for the four permutation-based problems implemented: TSP, PFSP, LOP, and QAP.

Methods	Description
Read{TSP,PFSP,LOP,QAP}Instance	Method to read an instance of a given problem
Eval{TSP,PFSP,LOP,QAP}	Method to evaluate a given solution (permutation)

The `Read{TSP,PFSP,LOP,QAP}Instance` module takes, as the only parameter, the filename of the instance to load. The output is a global variable with the name of the problem that contains the parameters needed by the EDA.

For example, to read a QAP problem, the method `ReadQAPInstance` is called:

```
ReadQAPInstance(InstanceName);
```

where *InstanceName* is the path and filename of the instance to read. This module creates a global variable, in this case called *QAPInstance*, that holds the data of the problem. The global variable is used in order to avoid unnecessary traffic of arguments across the main `RunEDA` process.

For the evaluation of a solution in the QAP problem, the method `EvalQAP` is called:

```
[val] = EvalQAP(permutation);
```

where *permutation* contains the permutation that will be evaluated. As previously mentioned, the instance data is taken from the global variable of the problem, in this case called *QAPInstance*. The output parameter *val* is the fitness of the permutation.

All problems described in Table III have the same input parameter. For the four problems, the methods to read and evaluate the problem are called in the same way. However, regarding the global variables, each problem has its own structure and, thus, the information stored for each problem is different.

- **Traveling Salesman Problem (TSP):** This problem is described by one matrix of size $n \times n$ containing the distances between the cities. The TSP implementation can work with both symmetric and asymmetric matrices. Once the data is read, the distance matrix of the problem and the number of cities are stored, in this order, in a global variable named *TSPInstance*.
- **Permutation Flowshop Scheduling Problem (PFSP):** The information about this problem is stored in one matrix of size $m \times n$ containing the processing times of executing job j , $j = 1, 2, \dots, n$ in machine i , $i = 1, 2, \dots, m$. The matrix that contains the processing times, the number of machines, and the number of jobs are stored, in this order, in a global variable named *PFSPInstance*.

- Linear Ordering Problem (LOP): It is described by one matrix of size $n \times n$ with arbitrary natural numbers. Once the data is read, the matrix and the problem size are stored, in this order, in a global variable named *LOPInstance*.
- Quadratic Assignment Problem (QAP): The information about this problem is contained in two matrices of sizes $n \times n$. The first matrix contains the flow between the facilities and the second one the distances between the locations. Once the data is read, the distance matrix, flow matrix and problem size are stored, in this order, in a global variable named *QAPInstance*.

8. ANALYZING THE RESULTS OF THE EDAS

As discussed in Section 3.1.2, in addition to the variable *Cache*, the output variable *AllStat* gathers diverse statistics of the EDA behavior. These statistics can be used to analyze the results of the algorithm. It is also possible to analyze the time spent by the algorithm in each of the main steps.

In order to illustrate the capabilities of the toolbox, we have run the code of the example presented in Section 4. In the following code we show how to extract, from the information stored in *AllStat* and *Cache*, the mean fitness of the population and the *theta* parameter learned at each generation.

```

1 for i=1:100,
2     MeanFitness(i) = AllStat{i,1}(2);
3     Theta(i) = Cache{3,i}{3};
4 end

```

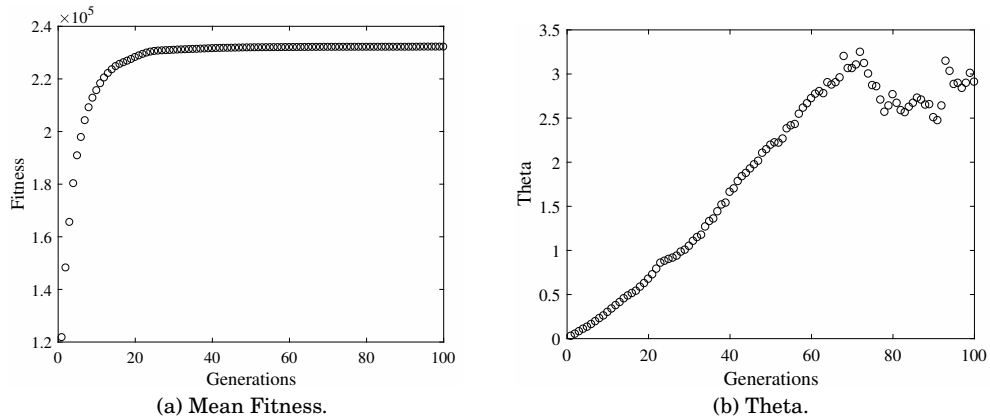


Fig. 1: Mean fitness of the population and θ values learned in each generation throughout the optimization.

Fig 1a shows the mean fitness of the individuals in the population across 100 generations. In addition, using the information about the probabilistic model stored in *Cache* (see line 8, where *cache(3)* was set to 1), a figure showing the evolution (convergence) of the θ variable has been also plotted (see Fig 1b).

REFERENCES

- Pedro Larrañaga and Jose A. Lozano. 2002. *Estimation of Distribution Algorithms: A New Tool for Evolutionary Computation*. Kluwer Academic Publishers.
- Jose A. Lozano, Pedro Larrañaga, Iñaki Inza, and Endika Bengoetxea. 2006. *Towards a New Evolutionary Computation: Advances on Estimation of Distribution Algorithms (Studies in Fuzziness and Soft Computing)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.
- Martin Pelikan, David E. Goldberg, and Fernando G. Lobo. 2002. A survey of optimization by building and using probabilistic models. *Computational Optimization and Applications* 21, 1 (2002), 5–20.
- Roberto Santana, Concha Bielza, Pedro Larranaga, Jose A Lozano, Carlos Echegoyen, Alexander Mendiburu, Ruben Armananzas, and Siddartha Shakya. 2010. Mateda-2.0: Estimation of distribution algorithms in MATLAB. *Journal of Statistical Software* 35, 7 (2010), 1–30.