Faculty of Technology
Department of Mathematics and Physics
Laboratory of Applied Mathematics

Differential Evolution approach and parameter estimation of chaotic dynamics.

by Vladimir Shemyakin

The topic of this Master's thesis was approved
by the faculty council of the Faculty of Technology on 25.10.2012

The examiners of the thesis were:
Prof. Heikki Haario and prof. Jari Hämäläinen
The thesis was supervised by: Prof. Heikki Haario
Supporting supervisor: Prof. Jari Hämäläinen

Lappeenranta, 2012

Vladimir Shemyakin
Punkkerikatu 7A, 12
53850, Lappeenranta
+ 358 417 059 605
vladimir.shemyakin [at] lut.fi

Abstract

Lappeenranta University of Technology
Faculty of Technology
Department of Mathematics and Physics

Vladimir Shemyakin

Differential Evolution approach and parameter estimation of chaotic dynamics.

Thesis for the Degree of Master of Science in Technology

2012 year

96 pages, 27 figures, 8 tables, 2 appendices

Supervisors: Prof. Heikki Haario, Prof. Jari Hämäläinen.
Examiners: Prof. Heikki Haario, Prof., Jari Hämäläinen.

Keywords: Evolutionary algorithm, Differential evolution, mutation, crossover, selection, chaotic dynamics, generation jumping, Lorenz system, EPPES, MCMC methods, importance sampling.

Parameter estimation still remains a challenge in many important applications. There is a need to develop methods that utilize achievements in modern computational systems with growing capabilities. Owing to this fact different kinds of Evolutionary Algorithms are becoming an especially perspective field of research. The main aim of this thesis is to explore theoretical aspects of a specific type of Evolutionary Algorithms class, the Differential Evolution (DE) method, and implement this algorithm as codes capable to solve a large range of problems. Matlab, a numerical computing environment

provided by MathWorks inc., has been utilized for this purpose. Our implementation empirically demonstrates the benefits of a stochastic optimizers with respect to deterministic optimizers in case of stochastic and chaotic problems. Furthermore, the advanced features of Differential Evolution are discussed as well as taken into account in the Matlab realization. Test "toy-case" examples are presented in order to show advantages and disadvantages caused by additional aspects involved in extensions of the basic algorithm. Another aim of this paper is to apply the DE approach to the parameter estimation problem of the system exhibiting chaotic behavior, where the well-known Lorenz system with specific set of parameter values is taken as an example. Finally, the DE approach for estimation of chaotic dynamics is compared to the Ensemble prediction and parameter estimation system (EPPES) approach which was recently proposed as a possible solution for similar problems.

Acknowledgements

There are a lot of people who has contributed to implementation of this Thesis. That is why it is almost impossible to mention all of them, thus I apologize in advance to those I have not mention personally.

First of all I would like to express my deep gratitude to my supervisors, namely Professor Heikki Haario and Professor Jari Hämäläinen for their patient guidance, enthusiastic encouragement, useful critiques of this Thesis and interesting research area. My grateful thanks are also extended to my home university, namely Southern Federal University and particularly to Faculty of Mathematics, Mechanics and Computer Sciences in the persons of associate professors Mikhail Karyakin and Konstantin Nadolin who have provided me with possibility to continue my study in Lappeenranta University of Technology and obtain double degree. My appreciation goes to staff of the Department of Mathematics who has provided useful facilities and made research time more convenient.

Besides, I wish to especially thank my girlfriend Alisa Zeleva who has been supporting me and waiting for me during my studies in Lappeenranta, and has been sympathetic to my studies far away from home.

Finally, I wish to thank my beloved family for their support and care about me.

Lappeenranta, 2012

Vladimir Shemyakin

Contents

List of Tables

List of Figures

1 Introduction

The topic of the research is devoted to the field of Differential Evolution (DE) algorithm which has become a demanded area due to the growth of efficiency of modern computational systems. Considerable attention for this framework from investigators is caused by potentially huge range of problems arises in many modern area of research such as parameter estimation problems particularly dedicated to weather forecasting problems which can be solved applying this method. First algorithmic description of DE algorithm was proposed by Kenneth Price and Rainer M. Storn and published as a technical report in 1995. The basic population structure and main stages of algorithm was discussed what had given a birth to a powerful probabilistic optimization tool.

Although there are plenty of implementations of classical DE approach, there is a lack of specific numerical experiments devoted to estimation of dynamic of chaotic systems. Moreover, modern advanced features dedicated to the background of the DE algorithm have been intensively establishing and they can be taken into account during implementation of such algorithm as well. Owing to the probabilistic nature of DE approach, the influence of advanced method which can be included to the extension of basic DE method should be tested on specific problem. Another challenging question which should be investigated is the performance advantages of DE method over the existed EPPES method which is proposed by Marko Laine, Antti Solonen, Heikki Haario and Heikki Järvinen as possible approach to numerical weather prediction (NWP) modeling.

Therefore, this study focuses on investigation of theory of the DE algorithm which is further implemented in Matlab. Also, two types of examples will be tested on applicability of DE algorithm, namely deterministic and stochastic cases of simple three-parameter estimation problem. Comparison between deterministic solver and DE solver is done and inapplicability of deterministic solver to estimate stochastic problem is demonstrated. Fur-

thermore possible directions of improvements of DE algorithm is discussed and the most profitable features are included into implementation of extended version of DE solver. The final part of the work is dedicated to the estimation of chaotic dynamics. As an example of the system exhibiting chaotic behavior the Lorenz system is used. The application of DE method is conducted and followed by Ensemble Prediction and Parameter Estimation (EPPES) approach discussion. Comparison between these two approaches sums performed research.

The paper is divided into four main sections and finalized by the Conclusion sections. The implementation of proposed algorithms and results are provided by numerical computing environment Matlab. The crucial parts of the code listings are included into proper chapters of paper, however full listings can be found in Appendix.

2 Theory of Differential Evolution

Differential evolution (DE) is a method of multidimensional mathematical optimization which belongs to the class of Evolutionary Algorithm (EA). This metaheuristic method tries to find optimum of the problem by iteratively improving of the candidate solution with respect to value of the objective function (function to be optimized). First of all, it is necessary to describe EA class of optimizers.

2.1 Evolutionary Algorithms: stochastic versus deterministic methods

Evolutionary Algorithm (EA) is a generic population-based *metaheuristic* optimization algorithm which tries to mimic the Darwinian Theory of Evolution through three basic elements, namely: *mutation, recombination, and survival of the fittest. Metaheuristic* algorithms in general make no assumption on the character of the cost function or the region where the solution is found which tends to the high diversity of possible space for searching of candidate solution. Nevertheless, disadvantage of such type of algorithms is that they do not ensure finding of solution of original problem. Moreover, EA belongs to the class of *stochastic optimizers* where parameters, functions and constraints, contrary to *deterministic optimizers*, can involve random variables.

Deterministic optimizers have a long history of intensive development and considerable success. Although they are perfect for educational purposes owing to mathematical elegance, most of deterministic algorithms set requirements on differentiability, continuity and convexity which strongly restrict field of applicability of such algorithms. Also, deterministic algorithms are quite sensitive to starting points. Although a choice of starting point is critical for the success of deterministic optimization algorithm, usually it cannot be done by chance and demands either reliable prior knowledge or trial and error approach to allocate appropriate starting point. Computa-

tionally efficiency of deterministic optimizers in the problems where they are applicable is the main advantage of such methods. However, at present computer technology is developing day by day and more powerful *stochastic optimization* tools have become more preferable for a huge range of applications. This statement can by confirmed by statistic which shows that the use of stochastic optimization is growing exponentially during last decade. Let us consider the main differences between deterministic and stochastic optimizers.

There are five key features that distinguish stochastic optimizers from deterministic. Most of them are considered somewhat controversial due to the fact that researches have opposite opinion about these properties[2].

1. **Randomness**. This property means that results obtained by execution of such algorithm in general cannot be predicted due to randomness occurred in the problem. It leads to controversy about applicability of such methods since proof of convergence to true optimal solution is a question demanding deep theoretical and numerical research and challenging to modern scientists. Despite the lack of guarantee of positive results, in practice stochastic optimizers demonstrate high level of success.

2. **Simplicity**. Typically, stochastic algorithms are easier in implementation because they do not require calculation of additional functions such as approximate or exact derivatives of objective functions. A background for the simplicity is that most of them were inspired by real natural phenomena like Darwinian Theory of Evolution. However, each stochastic optimizer has at least one control parameter, but performance of algorithm which can be obtained by tuning of this parameter is a complicated task.

3. **Efficiency**. Stochastic optimization algorithms usually require more evaluation of objective function to get expected result. That fact makes such algorithm more computationally expensive, time consuming or less efficient.

4. **Robustness**. This crucial property appears due to random-oriented origin of such algorithms. For instance, from one hand stochastic optimization algorithm may miss optimal solution even in favorable conditions, from the other hand it may find quasi-optimal solution which is close to real optimal solution under conditions where deterministic algorithms totally fail.

5. **Versatility**. The most of stochastic optimizers does not impose restriction on the optimization task, which means that they equally applicable for continuous, discrete and even symbolic data.

2.2 DE: basics and history

DE algorithm is originated by Kenneth Price and Rainer M. Storn and first publication of idea of this method was published as a technical report in 1995 (Storn and Price 1995). Just after inception of this method it has become an attractive field for research and after establishing by Storn in 1997 a website (`http://www.icsi.berkeley.edu/_storn/code.html`) an explosive expansion in differential evolution research took place. Moreover, the current progress in the field of computer computations makes in practice DE a powerful tool for stochastic optimization due to its parallelizable nature from the computational point of view. This situation is caused by the ability to perform calculations on each element of the population separately.

DE algorithm inherits common idea of EA that is idea of Natural selection which consists of following stages: maintaining a population of candidate solution, generation of new candidate solution by perturbation of present population according basic formulae, and then selection procedure of new generation which has the best (in general) value of the objective function. Moreover, this method is commonly used for multidimensional real-valued objective functions and does not take advantage of typical gradient-based optimization methods; thereby requirement for differentiability of the objective function does not take place. Hence, DE method is applicable for even non-continuous, noisy and non-deterministic problems.

Before giving a precise mathematical description of the steps of the DE algorithm, it is convenient to present the main idea of the algorithm in a graphical form [1], considering one full cycle of the algorithm and accompanying graphics by explanatory information. Let us consider two-dimensional objective function.

The Figure 2.1 demonstrates this function as contour lines, where optimum point is situated in the center ellipse. There are plenty of possible ways of initial population generation. Although the chosen way can influence on speed of algorithm convergence, the nature of DE approach makes possible

to move towards the optimum even if the initial population was generated out of optimum vicinity and did not cover whole exploring domain. The classical DE approach operates with uniform distribution within specified region. The preset parameters define this domain and the Np vectors of the initial population will be chosen out of this region. All vectors of population get unique index in range (0,Np-1) to be distinguished in conducted competition.

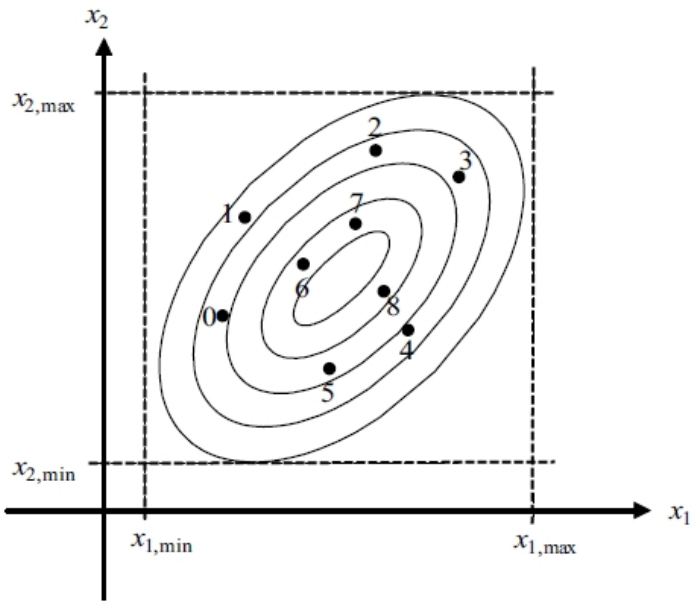

Figure 2.1: Initializing the DE population.

The distinctive feature of DE is the procedure of trial vector producing. For this purpose DE randomly chooses two vectors and calculates difference vector based on them (Figure 2.2). Then, difference vector is scaled and added to a third randomly selected vector forming trial vector (Figure 2.3). All these three randomly chosen vectors have to be different.

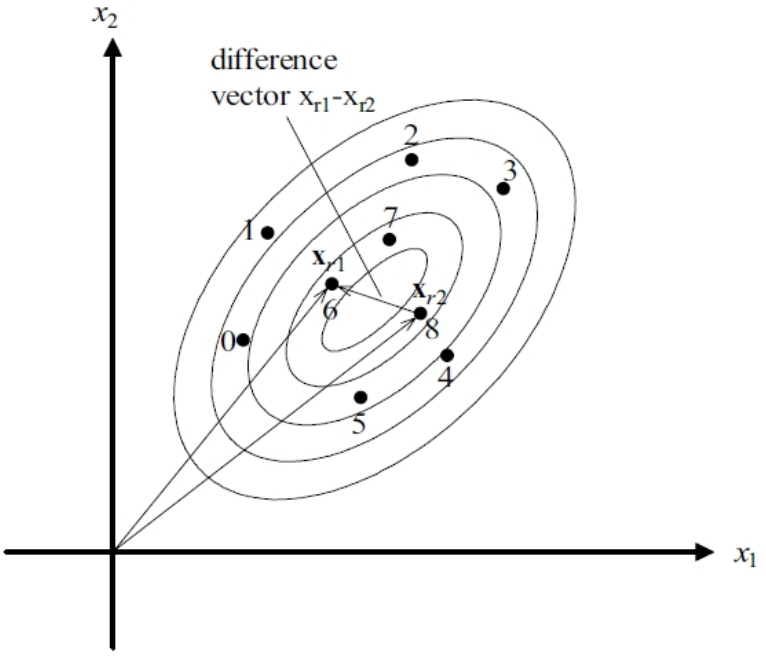

Figure 2.2: Generating the difference vector.

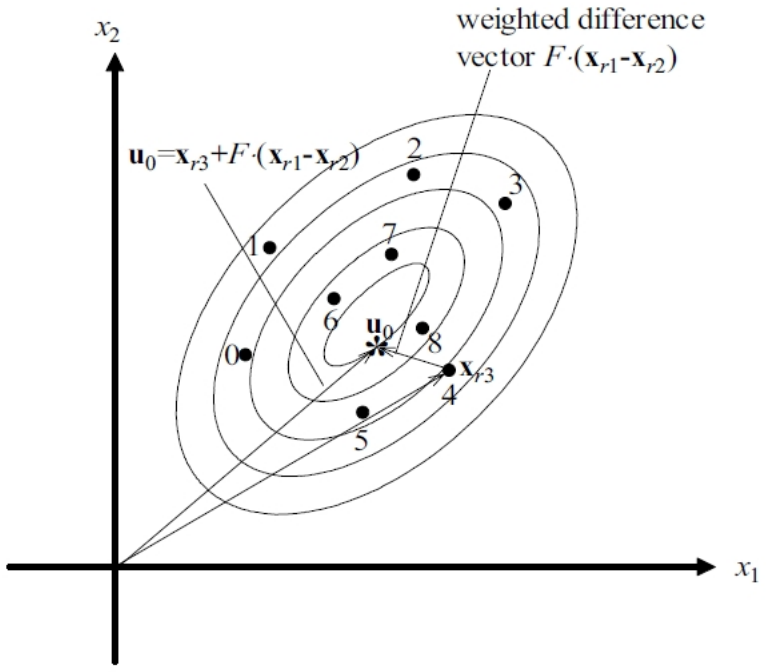

Figure 2.3: Mutation.

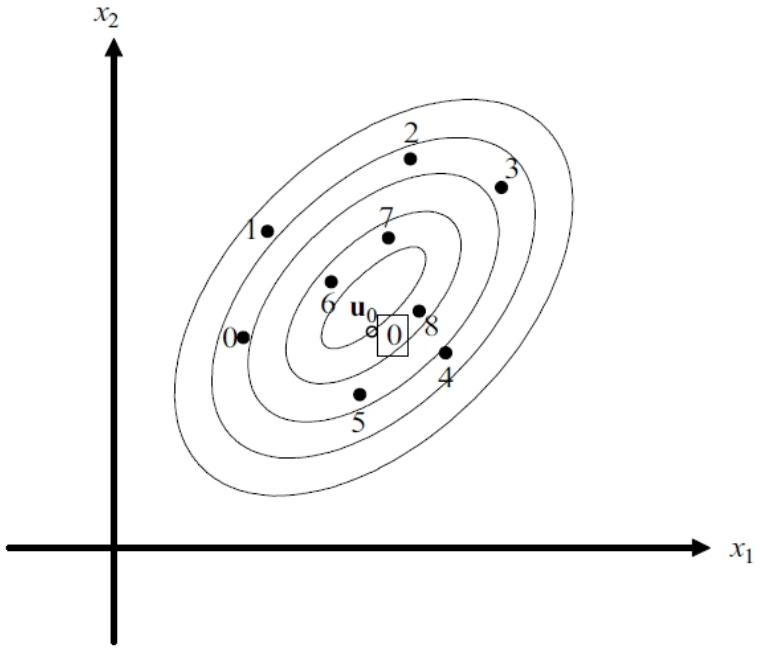

Figure 2.4: Selection. $\mathbf{u}_0$ replaces the vector with index 0 because it has lower value of objective function.

Recombination-mutation step is followed by the selection stage, where the trial vector competes against the population vector of the same index (Figure 2.4). Thus, Np pair wise competitions are conducted, the survivors of which become parents for next generation of evolutionary cycle.

2.3 Mathematical Description

This part is devoted to the mathematical description of typical structure of main DE algorithm stages.

Population structure. According to K. Price and R. M. Storn [1], classical population structure of DE algorithm has the following form. One of the vector populations, called the *current population*, consists of all approved either the initial points or selected by the competition points. All population vectors contain Np D-dimensional vectors of real-valued parameters. Index g shows the generation which the vector belongs to, two other indexes i and j are responsible for population and dimension indexes respectively (2.1):

$$\begin{aligned} P_{\mathbf{x},g} &= (\mathbf{x}_{i,g}), i = 0, 1, \ldots, Np-1, g = 0, 1, \ldots, g_{max}, \\ \mathbf{x}_{i,g} &= (x_{j,i,g}), j = 0, 1, \ldots, D-1. \end{aligned} \tag{2.1}$$

Once initialized, DE mutates randomly chosen vectors to generate intermediate population which consists of *mutant vectors* (2.2):

$$\begin{aligned} P_{\mathbf{v},g} &= (\mathbf{v}_{i,g}), i = 0, 1, \ldots, Np-1, g = 0, 1, \ldots, g_{max}, \\ \mathbf{v}_{i,g} &= (v_{j,i,g}), j = 0, 1, \ldots, D-1. \end{aligned} \tag{2.2}$$

Next step is recombination of current population with intermediate population to produce a trial population of *trial vectors* (2.3):

$$\begin{aligned} P_{\mathbf{u},g} &= (\mathbf{u}_{i,g}), i = 0, 1, \ldots, Np-1, g = 0, 1, \ldots, g_{max}, \\ \mathbf{u}_{i,g} &= (u_{j,i,g}), j = 0, 1, \ldots, D-1. \end{aligned} \tag{2.3}$$

After considering the population structure, it is possible to discuss the implementation of the main stages of DE.

Initialization. According to idea described earlier, it is necessary to specify the parameters characterizing boundary of searching space. The convenient way is to collect this data in two D-dimensional vectors where subscripts indicate the lower and upper bounds respectively. Once these parameters have been predefined, the initial population can be constructed using random number (e.g. rand-function in Matlab) generator under following formula (2.4):

$$\begin{aligned} (x_{j,i,0}) &= rand_j(0,1) \cdot (b_{j,U} - b_{j,L}) + b_{j,L}, \\ i &= 0, 1, \ldots, Np-1, \\ j &= 0, 1, \ldots, D-1. \end{aligned} \tag{2.4}$$

This random generator produces uniformly distributed random numbers from the range $(0, 1)$. The crucial difference here is that in spite of the required nature of parameters, the initialization is made by floating-point values, owing to DE internally treats all variables in such manner regardless of their true type.

Mutation. Every specific type of EA has its own mechanism of mutation. The procedure of mutation is applied for every population member in current population $P_{\mathbf{x},g}$ to produce intermediate population $P_{\mathbf{v},g}$. DE algorithm got his name from differential mutation, which for the current generation g has the following form (2.5):

$$\mathbf{v}_{i,g} = \mathbf{x}_{r0,g} + F \cdot (\mathbf{x}_{r1,g} - \mathbf{x}_{r2,g}), i = 0, 1, \ldots, Np-1, \tag{2.5}$$

where F is the scale factor controlling evolvement of population. It is a positive real number and although there is no upper limit of this constant, effective value is rarely greater than one. One can distinguish three types of vector contained in this formula, they are *target vector* with index i, *base vector* with index $r0$ and *difference vectors* with index $r1$ and $r2$. $r0$, $r1$ and $r2$ are three different randomly chosen numbers corresponding to indexes in current population which differ from index of target vector. The procedure of mutant generation in two-dimensional parameter space can be illustrated by the following graph Figure 2.5.

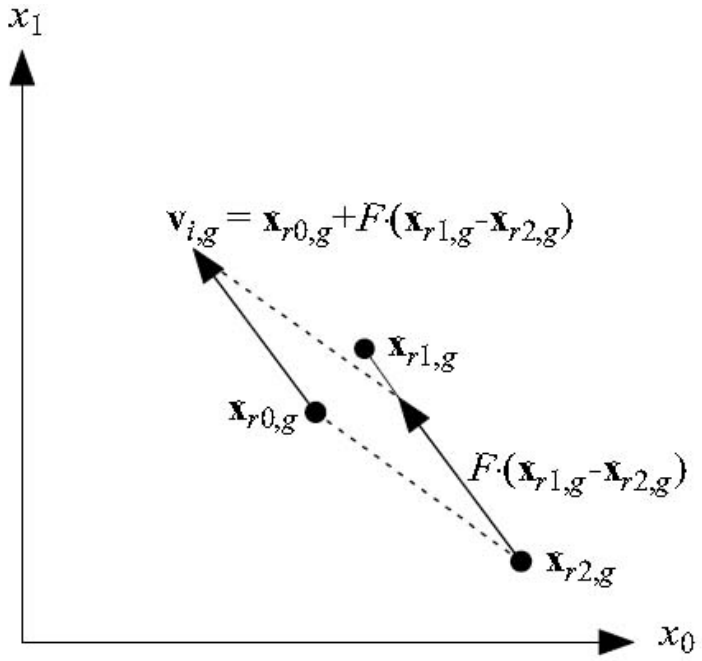

Figure 2.5: Differential mutation.

Crossover. Crossover is essential part of every EA methods, which is responsible for recombination of current with mutant population to produce trial population. Similarly to mutation, crossover mechanism is applied for every element in intermediate population ($i = 0, 1, \ldots, Np - 1, j = 0, 1, \ldots, D - 1$) in current generation g. Hence, the output trial population has the same size as the current population. Due to this circumstance the selection step of DE algorithm can be conducted element wise. Classical DE provides uniform crossover procedure according following formula (2.6):

$$\mathbf{u}_{i,g} = u_{j,i,g} = \begin{cases} v_{j,i,g}, & \text{if } (rand_j(0,1) \leq Cr \text{ or } j = j_{rand}); \\ x_{j,i,g}, & \text{otherwise.} \end{cases} \tag{2.6}$$

Thus, Parameter Cr here is the crossover probability which is user-defined parameter controlling fraction of parameters inherited from the mutant.In order to determine which source contributes a given parameter, a uniformly random-generated number is compared with the crossover probability. Also, additional condition ($j = j_{rand}$) ensures that trial vector differs from current vector. This scheme can be illustrated by following Figure 2.6:

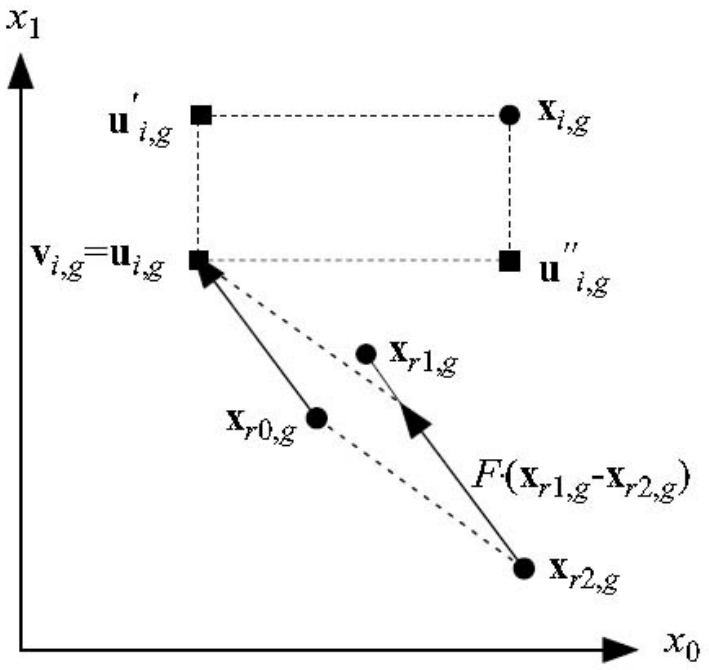

Figure 2.6: Crossover. Possible additional trial vectors $\mathbf{u}'_{i,g}, \mathbf{u}''_{i,g}$.

Selection. Procedure of surviving the fittest is realized in DE according to the value of objective functions for given trial and current vectors. Thus, every trial vector is compared with current vector element wise and in the case when objective function value for trial vector is lower or equal to such value of current vector, it replaces the current vector in next generation. This procedure can be explained by the following formula (2.7) where $i = 0, 1, \ldots, Np - 1$:

$$\mathbf{x}_{i,g+1} = \begin{cases} \mathbf{u}_{i,g}, & \text{if } f(\mathbf{u}_{i,g}) \leq f(\mathbf{x}_{i,g}) \\ \mathbf{x}_{i,g}, & \text{otherwise.} \end{cases} \tag{2.7}$$

Once the next generation is fully constructed, whole process starting from mutation to selection is repeated with selected population until the optimum is located or specified termination condition is satisfied.

Termination conditions. It is necessary to discuss issue concerning completion of DE algorithm. We describe three the most common termination criteria, namely [2]:

- **Objective met**. This criterion is used only for problems with known optima to test applicability of algorithm and rectify errors. Such condition is usually used while starting the implementation of DE algorithm or introducing new features to prevent damaging of algorithm or controlling the work in a proper way.
- **Limit on Number of Objective Function Evaluations**. This concept is widely used for real problems. Due to the fact that finding of optimal solution should take limited time, such termination criteria helps to control evolution of population in case when no more improvement takes place. The possible alternative to this criterion but with similar meaning is the limitation of the generations number. In classical DE approach this criterion can be controlled simpler, but in the case when distinguish between generations is vanished it becomes inappropriate.
- **Limit of Population Diversity**. This criterion is based on the nature of DE algorithm and connected to the notion of *premature populations* in evolutionary computations. *Premature population* is the population in which elements differ from each other slightly, thus, taking into account specificity of the next generation construction, it is practically pointless to continue computations.

Last to criteria will be considered as default termination condition in Matlab implementation.

3 Matlab implementation

This section will be devoted to describing the implementation of DE approach to the toy-cases of *inverse problem*. The *inverse problem* is a general class of mathematical problems that can be used for determination properties and information about physical object or system according measured data. Thus, one of the examples of inverse problem is estimating unknown parameters of mathematical model defining specific phenomena with the help of given measured data. A typical approach to test a an estimation algorithm is the following. Using known parameters of specific mathematical model it is possible to generate artificial data which will be handled as measured data during estimation procedure. Hence, it becomes possible to compare known true parameters of model with estimated ones using specific approach, e.g. widely used least square approach.

The essential feature in this approach is simulating of the noise in data. In real life every measurement can be conducted only with specific level of noise due to different reasons, for example human factor, finite limit of accuracy of measuring instruments or truncation error in digital representation of the data. Hence, the simple mathematical model for the parameter estimation problem in a general form can be presented as:

$$y = f(x, \theta) + \epsilon, \tag{3.1}$$

where x and y are *input* and *response*, respectively; θ is the *vector of parameters* to be estimated, and ϵ represents the *measurement noise*. It is assumed that the measurement noise is a random normally distributed variable with zero mean value and non-zero standard deviation ($\epsilon \sim \mathcal{N}(0, \sigma^2)$), which can be estimated using different techniques, for instance repeated measurements.

Taking into account the measurement noise we face with so-called *Bayesian approach* of parameter estimation problem. We are not going to provide deep description of Bayesian inference here, but the key idea of such approach is following [5]. Instead of estimation the fittest vector of parameters, we are interested in producing set of vectors which reasonably – statistically appropriate due to the noise in data – fit the data. Statistically appropriate means that a model with estimated parameters should fit the data with the same accuracy as the measurements are obtained. Hence, if we substitute estimated parameters into a model and calculate residual which is the difference between data and the model values (Equation 3.2), the basic statistical requirement should be satisfied, namely the residuals of the fit should be roughly equal to the estimated level of measurement noise (Equation 3.3).

$$res = y - f(x, \hat{\theta}) \tag{3.2}$$

$$res \simeq \epsilon \tag{3.3}$$

3.1 Deterministic "toy-case" problem

The first aim of research is to test applicability of DE approach for estimating parameters of *deterministic problem*. *Deterministic problem* means that no randomness involved into the behavior of future states. Thereby, a deterministic model always produces the same result under the same predefined parameters and initial values. As it was announced earlier, a numerical computing environment provided by Matlab which is developed by the MathWorks inc. will be used as a main programming tool for realization of considered algorithms. Owing to a high amount of built-in function available by default in Matlab, in this toy case we can compare the results obtained by DE with the results obtained by built-in optimizers, e.g. `FMINSEARCH` optimizer.

3.1.1 Mathematical model

Let us consider the following model containing three parameters:

$$y = f(x, \theta) = e^{\theta_0 + \theta_1 x + \theta_2 x^2} \tag{3.4}$$

It is expected to implement in Matlab the full procedure of solving inverse problem for estimation unknown parameters using DE approach. In order to handle this task, it is possible to divide implementation of assigned problem into several steps:

- The 1st step is "*DE environment implementation*". All functions and data structures which are necessary for DE algorithm will be implemented on this step.
- The 2nd step is "*Artificial data generation*". According method discussed above, we will produce some artificial data, generated using predefined parameters, and test DE approach with help of this data.
- The 3rd step is "*Application of DE approach to the artificial data*". This step will be devoted to testing of implemented DE environment on generated data.
- The 4th step is "*Comparison and analysis*". Considering concrete solutions, plotting graphs, comparison different aspects influencing on solution and analysis of obtained result will be discussed in this step.

3.1.2 DE environment implementation

First of all, it is necessary to define data structures which are responsible for allocation of algorithm parameters. Moreover, the essential thing here is generalized form of implementation, which means that all function written for DE environment have to work not only for specific problem, but be able to be reused for huge amount of problems without being changed dramatically. The main structure of parameters is called `paramsDE` and has following fields:

```
paramsDE.init_bounds     % bounds for initialization of DE
paramsDE.SoP             % size of population
paramsDE.F               % scale factor of mutation
paramsDE.CR              % crossover probability parameter
paramsDE.func            % name of cost function
paramsDE.D               % dimension of problem (number of ...
    unknown parameters)
paramsDE.Hl              % length of history of generations
paramsDE.Tol             % tolerance for standard deviation of ...
    Hl previous generations' objective values means
paramsDE.Gmax            % maximum number of generations
```

This structure should be specified by user. Moreover, there is only two compulsory input parameters; they are the name of cost function (`paramsDE.func`) and range of intervals for initialization of first generation (`paramsDE.init_bounds`). It is necessary to mention that an objective function to be minimized is usually called *cost function*. Thus, this notation will be utilized further. The parameter `paramsDE.Hl` corresponds to the 'Limit of Population Diversity' stopping criterion. Hence, when standard deviation of cost function values average of `paramsDE.Hl` previous population less then `paramsDE.Tol` the algorithm stops. The implementation of cost function will be described later in the 3rd step. Once someone of the rest of parameters is not implicitly specified by user, default values are used:

```
%% Default values:
paramsDE.D = size(paramsDE.init_bounds,1);
paramsDE.SoP = 10*paramsDE.D;
paramsDE.F = 0.5;
paramsDE.CR = 0.9;
```

```
paramsDE.Tol = 1e-10;
paramsDE.Hl = 10;
paramsDE.Gmax = 1e3;
```

Besides, there is a `data` structure which usually contains input and response of true model, but also may contain additional fields characterizing measured data:

```
data.xdata = x;                 % input
data.ydata = y;                 % response
```

Moreover, for convenience the model in-line function in stored in the same structure:

```
data.f = f;                   % model function
```

The main variable of program keeping current population is `pop`. This is `SoP` by *(D+1)* matrix, where every row represents member of population, first *D* columns saves parameter values and the last *D+1* column saves the value of objective function for every member of population. The classical DE implementation consists of four functions, namely: `initialization`, `mutation`, `crossover`, and `selection`. Let us provide Matlab code listings of these functions:

- `initialization`:

```
function pop = initialization(data,paramsDE)
    SoP = paramsDE.SoP;
    init_bounds = paramsDE.init_bounds;
    func = paramsDE.func;
    D = paramsDE.D;
    % generation of first population of parameters :
    pop = ones(SoP,1)*init_bounds(:,1)' + ...
          ones(SoP,1)*(init_bounds(:,2) - ...
              init_bounds(:,1))'.*rand(SoP,D);
    % calculation of cost function :
    tmp = func(pop,data);
    % pop = [<parameters>,cost]
```

```
    pop = [pop tmp];
end
```

- mutation:

```
function res = mutation(pop,paramsDE)
    SoP = paramsDE.SoP;
    F = paramsDE.F;
    D = paramsDE.D;
    cur_pop = pop(:,1:D);    % only parameters
    % choosing base  and difference vectors ...
    % differ from target vector:
    [tmp,ind] = sort(rand(SoP,SoP-1),2);
    ind = ind(:,1:3) + (ind(:,1:3) > [0:SoP-1]'*ones(1,3));
    % mutation itself
    res = cur_pop(ind(:,3),:) + F*(cur_pop(ind(:,1),:) - ...
        cur_pop(ind(:,2),:));
end
```

- crossover:

```
function res = crossover(new,pop,data,paramsDE)
    SoP = paramsDE.SoP;
    D = paramsDE.D;
    CR = paramsDE.CR;
    func = paramsDE.func;
    % generating of index vector of members to be tested ...
    % for being inherited from previous generation ...
    % including protection from exact copies.
    temp = rand(SoP,D);
    ind = randi(D,SoP,1);
    temp(sub2ind([SoP,D],[1:SoP]',ind)) = 0;
    % testing
    ind = (temp > CR);
    % crossover itself: generating trial vector
    new(ind) = pop(ind);
    % calculation of cost function of trial vector
    tmp = func(new,data);
    res = [new tmp];
end
```

- selection:

```
function res = selection(new,pop,paramsDE)
    D = paramsDE.D;
    % selection: comparison of objective function values:
```

```
    ind = (new(:,D+1) > pop(:,D+1));
    % construction of next generation:
    new(ind,:) = pop(ind,:);
    res = new;
end
```

Once these functions are implemented, it is convenient to implement additional function `de` which is responsible for initialization of missing fields of `paramsDE` structure and calling of previously realized functions in an appropriate order:

```
function [pop,G,evolution,fval,element] = de(data,paramsDE)
    %% default values
    if ~(isfield(paramsDE,'func') && ...
        isfield(paramsDE,'init_bounds'))
        display('Error!');
        pop = [];
        G = 0;
        evolution = {};
        return;
    end
    paramsDE.D = size(paramsDE.init_bounds,1);    % dimension ...
        of problem
    if ~(isfield(paramsDE,'SoP'))
        paramsDE.SoP = 10*paramsDE.D;             % size of ...
            population
    end
    if ~(isfield(paramsDE,'F'))
        paramsDE.F = 0.5;                         % scale ...
            factor in mutation
    end
    if ~(isfield(paramsDE,'CR'))
        paramsDE.CR = 0.9;                        % crossover ...
            probability
    end
    if ~(isfield(paramsDE,'Tol'))
        % tolerance for standard deviation of Hl previous ...
            generations'
        % objective values means:
        paramsDE.Tol = 1e-10;
    end
    if ~(isfield(paramsDE,'Hl'))
        paramsDE.Hl = 10;                 % length of history ...
            of generations
    end
    if ~(isfield(paramsDE,'Gmax'))
        paramsDE.Gmax = 1e3;              % maximum number of ...
            generations
```

```
    end
    %% initialization of variables
    D = paramsDE.D;
    Tol = paramsDE.Tol;
    Gmax = paramsDE.Gmax;
    history = 100*rand(1,paramsDE.Hl);
    G=1;
    pop = initialization(data,paramsDE);
    evolution = {};
    %% DE process
    while std(history)>=Tol && G<=Gmax
        mutant = mutation(pop,paramsDE);
        new = crossover(mutant,pop,data,paramsDE);
        pop = selection(new,pop,paramsDE);
        evolution{G} = pop;
        ind = pop(:,D+1)<Inf;
        history(mod(G,paramsDE.Hl)+1) =sum(pop(ind,D+1));
        G  = G+1;
    end
    [fval,element] = min(pop(:,D+1));
end
```

By default this function returns the last population and optionally the number of generations spent for solution and cell array containing the full evolution of the population.

3.1.3 Artificial data generation

This step as it was announced earlier is devoted to generating of artificial data in order to test implemented DE algorithm. We will consider mentioned model (Equation 3.4) in the interval $x \in [0, 10]$ with increment equal to 0.1 taking following parameter vector as true values:

$$\theta_{true} = (\theta_0, \theta_1, \theta_2) = (-6, 3, -0.3) \tag{3.5}$$

Hence, the true model has a form:

$$y = f(x, \theta) = e^{-6+3x-0.3x^2}, x \in [0, 10] \tag{3.6}$$

Once true response has generated, it is necessary to simulate the noise in data. According to basic statistical assumption, the noise is produced as a normally distributed random variable with zero mean value and predefined standard deviation. For convenience, it is sufficient to check the algorithm on perfect data without noise and then try to introduce noise to perfect data. Matlab code for this part is listed below:

```
%% Generating data
x = [0:0.1:10];    % input
% in-line model function
% applicable for theta (3 by n)-matrix:
f = @(theta,x)exp(theta*[ones(size(x));x;x.^2]);
theta_true = [-6 3 -0.3];    % true parameters
y_true = f(theta_true,x);    % response of input with true ...
    parameters
sigma = 0.1;                 % std for noise
y = y_true+sigma*randn(size(y_true)); % "measured data" (with ...
    noise);
% store the model data in structure
data.xdata = x;              % input
data.ydata = y;              % true response
data.fun = f;                % model function
```

To illustrate the problem, the plot of true data and noisy data has been generated Figure 3.1

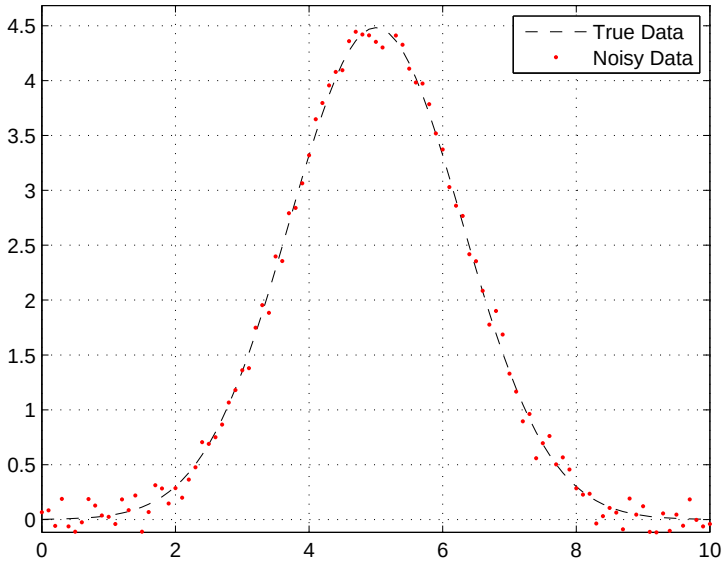

Figure 3.1: True and noisy data with standard deviation $\sigma = 0.1$.

3.1.4 Application of DE approach to the artificial data

The last compulsory part of the program is the definition of a cost function. To decide which element of the population produce the best result, the least squares principle is utilized. The idea of this principle is minimization of sum of squared difference between measured data and response of substituted parameters:

$$LSQ : min(l(\hat{\theta})) = \sum_{i=1}^{n} (y_i - f(x_i, \hat{\theta}))^2 \tag{3.7}$$

Hence, the natural criterion for selection is the value of the cost function, i.e. in selection procedure an element vector with lower value of cost function survives for the next population. The Matlab function responsible for calculation of cost function is called `fun_ss`. Due to the fact that the model function is implemented in the manner which allows calling with the whole matrix of parameters, the cost function can be implemented in a rather compact form:

```
function res = fun_ss(theta,data)
    % (Length-of-data.xdata by SoP)-matrix of responses
    y = data.fun(theta,data.xdata)';
    % calculation of LSQ
    res = sum((repmat(data.ydata',1,size(y,2))-y).^2);
    res = res';
end
```

It is important to mention that all functions are realized in separate files, at the same time generation of data, initialization of parameters by user, running DE algorithm and analysis of results are realized in the main script file `run.m`. Full listings of programs are presented in the appendix of this work. Thus, once all necessary functions are implemented and the artificial data are generated, it is required to initialize parameters of DE and run the program:

```
%% Initialization of Compulsory parameters of DE
paramsDE.init_bounds = [-10 -10 -3; 10 10 3]'; % bounds for ...
    initialization of DE
paramsDE.func = @fun_ss;                        % cost function
%% DE run
[pop,G,evolution] = de(data,paramsDE);
```

Due to probabilistic nature of the DE approach, it is crucial to conduct several calculations of the same problem to evaluate the mean number of generation required for finding solution with set accuracy. Because of the same reason there is probability to obtain absolutely incorrect results which will be considered as fail attempts for estimation. Moreover, it is possible to find the optimal population size as proportion between number of generations and accuracy of the solution. This can be done by introducing following part of Matlab code into `run.m`:

```
%% DE run
Gmean = 0; % number of generations
K = 100;   % number of simulations
Fails = 0; % number of fails in estimation
M = 10;    % estimate for mean cost function
for i = 1:K
    [pop,G,evolution] = de(data,paramsDE);
```

```
    Gmean = Gmean + G;
    Fails = Fails + (mean(pop(:,4))>=M);
    i
end
Gmean/K % mean number of generations
Fails/K % mean number of fails
```

3.1.5 Comparison and analysis

According to scheme discussed earlier, it is necessary to conduct several calculation of the task. Also we want to find optimal size population corresponding to accurate result and high efficiency. During every calculation we will compare solution obtain by DE approach and built-in solver to count number of fails in determination of problem parameters. Moreover, we can introduce different noise level into the exact data to analyze influence of noise into optimal size of population. Finally, the required relationship can be illustrated by the following graph Figure 3.2:

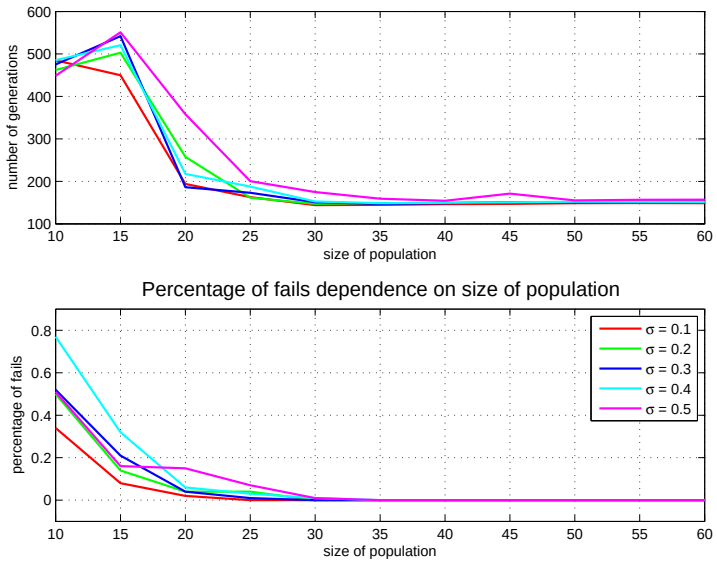

Figure 3.2: Number of generation and fails percentage dependence on size of population.

It is possible to conclude from this graph that optimal size of population lies in the region $Np \approx 10 \cdot D$. This empirical results agrees with theoretical estimation which will be described in the next section.

Once the optimal size of population is determined, it can be used to produce computations with different noise level and illustrate solutions by plotting graphs of fitting the data with DE approach(Figure 3.3, Figure 3.4, Figure 3.5). Furthermore, concrete values will be presented in Table 3.1, Table 3.2 and comparison of obtained results with built-in solver `fminsearch` solution can be made with the help of Table 3.4.

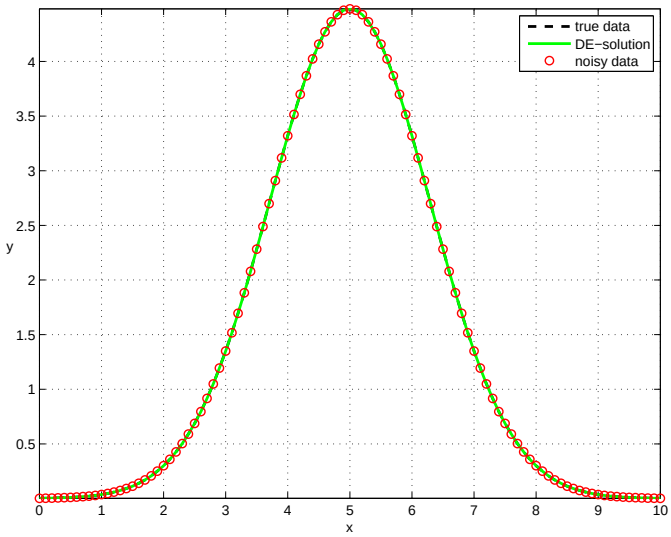

Figure 3.3: Fitting the data with noise level $\sigma = 0.0$.

Although there are Np different solutions from DE approach, the default stopping criterion in implemented algorithm is standard deviation of history of DE evolution is equal to 1e-10, which means in the case of deterministic problem that the population in the final generation has almost equal values of parameters. Owing to this it is possible to take into account only the best member of the population according to the value of objective function. This member is considered to be the DE-solution. Values of estimated parameters will be presented in the Table 3.1. Due to similarity of graphs, the illustrations of the case $\sigma = 0.1$ and $\sigma = 0.3$ are skipped.

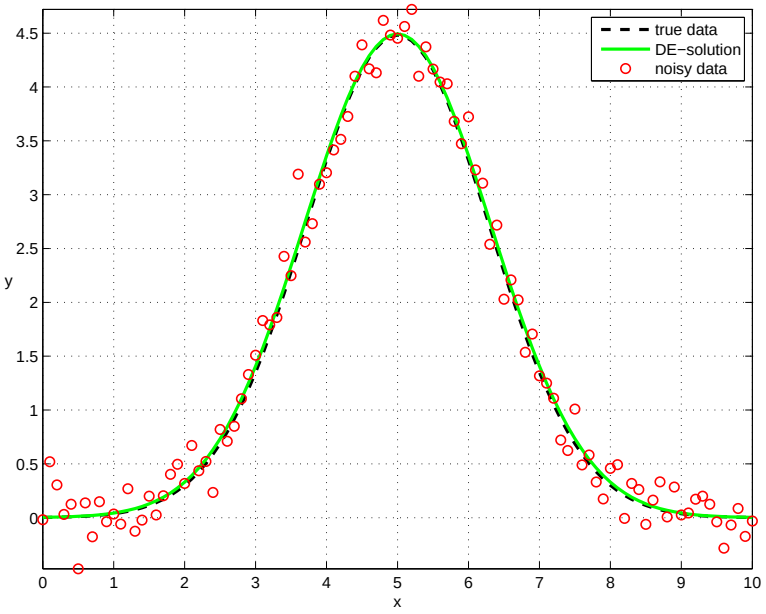

Figure 3.4: Fitting the data with noise level $\sigma = 0.2$.

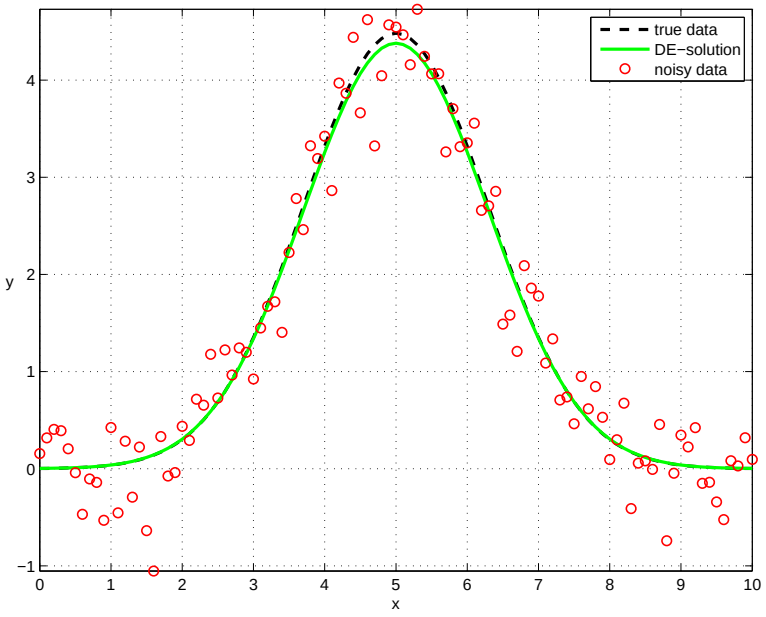

Figure 3.5: Fitting the data with noise level $\sigma = 0.4$.

The tables with concrete values of parameters and objective function are presented below. It is important to remind that the true vector of pa-

rameters of considered model is $\theta_{true} = (\theta_0, \theta_1, \theta_2) = (-6, 3, -0.3)$.

σ	θ_1	θ_2	θ_3	f_{val}
0.0	-5.9999998844	2.9999999501	-0.2999999951	3.3906658015e-014
0.1	-6.0221395147	2.9979231887	-0.2991487160	9.7582084808e-001
0.2	-5.7521311294	2.9016465779	-0.2901442080	3.5205629995e+000
0.3	-5.9252642622	2.9588615071	-0.2951914396	7.3744607077e+000
0.4	-5.9305662150	2.9625471950	-0.2962109595	1.3263261144e+001

Table 3.1: Best values of parameters and objective function values obtained by DE algorithm.

As it was mentioned earlier, the solution of parameter estimation problem in this case is considered as the best member of the population of the last generation. However, the mean values of population parameters can be treated as the solution as well. Thus, the mean parameters corresponding to population of last generation are computed and value of objective function should be calculated additionally. Finally, the solution of parameter estimation problem in the sense of mean parameter values has the following form:

σ	θ_1	θ_2	θ_3	f_{val}
0.0	-6.0000001400	3.0000000639	-0.3000000069	4.1154110407e-014
0.1	-6.0221395790	2.9979232049	-0.2991487166	9.7582084808e-001
0.2	-5.7521309442	2.9016465228	-0.2901442042	3.5205629995e+000
0.3	-5.9252640886	2.9588614478	-0.2951914351	7.3744607077e+000
0.4	-5.9305661771	2.9625471761	-0.2962109570	1.3263261144e+001

Table 3.2: Mean values of parameters and corresponded objective function values obtained by DE algorithm.

The Table 3.3 presents standard deviation of final generation population. Small values for standard deviation confirm idea of uniqueness of deterministic problem solution.

Finally, solutions obtained by built-in `fminsearch` solver is presented in Table 3.4 to compare correctness of stochastic optimizer with deterministic optimizer (`fminsearch` uses the Nelder-Mead simplex algorithm which can be found for instance in `http://en.wikipedia.org/wiki/Nelder\T1\textendashMead_method`):

σ	$std(\theta_1)$	$std(\theta_2)$	std(θ_3)
0.0	4.2051835838e-007	1.7350187042e-007	1.7440197092e-008
0.1	3.6830618356e-007	1.4501196782e-007	1.4284506860e-008
0.2	5.1680344874e-007	2.0661655439e-007	2.0639802688e-008
0.3	4.7644372814e-007	1.8993529210e-007	1.8722104250e-008
0.4	3.9492539809e-007	1.7014517267e-007	1.7884189936e-008

Table 3.3: Standard deviation of parameters of final generation population.

σ	θ_1	θ_2	θ_3	f_{val}
0.0	-6.0000206559	3.0000092031	-0.3000008172	9.1267068776e-009
0.1	-6.0221606272	2.9979317105	-0.2991494460	9.7582085210e-001
0.2	-5.7521626369	2.9016604062	-0.2901456669	3.5205630012e+000
0.3	-5.9252818250	2.9588676625	-0.2951918903	7.3744607101e+000
0.4	-5.9305956970	2.9625592788	-0.2962120935	1.3263261146e+001

Table 3.4: Values of parameters and objective function values obtained by built-in `fminsearch` function with default parameters.

Summing obtained result, it can be concluded that DE approach is applicable for deterministic problems although the performance is lower then built-in Matlab deterministic optimizers have. However this result is predictable due to the fact that stochastic DE optimization procedure is more universal and hence more computationaly consuming. Moreover, the main aim of the research is to apply DE algorithm to chaotic and stochastic problems where almost all deterministic optimizers fail. The next subsection is devoted to application of DE approach to the basic stochastic model.

3.2 Stochastic "toy-case" problem

In general, *stochastic problem* means that some stochasticity is involved to the behavior of the problem. It leads to the fact that system describing the problem can behave differently from run to run as opposed to deterministic problems. This property makes almost impossible to solve such type of problems by deterministic algorithms.

The simple stochastic model has the following form:

$$y = f(x, \theta) + \epsilon_{new} \tag{3.8}$$

This model seems to be similar to the deterministic model Equation 3.1. However the crucial distinguish here is the fact that in stochastic model ϵ_{new}, which in general sense is the measurement noise in the system, differs from run to run. Although ϵ_{new} can contain fixed measurement noise which remains for every run, it also contains adjustable component. Nevertheless the basic assumption still treated, namely $\epsilon_{new} \sim \mathcal{N}(0, \sigma^2)$ is normally distributed random variable.

3.2.1 Revision of Matlab implementation

The only difference which is needed to the existed implementation is stochastic component in the calculation of objective function value. It can be done by several ways but in order to test applicability of DE approach to the simple stochastic problem the global variable `global SIGMA` which is responsible for stochastic noise will be introduced. It is added to the true data in every call of objective function calculation. Thus objective function will produce different values for every single call even with the same parameter values. Taking into account such idea, definition of objective function for stochastic problem can be implemented in following form:

```
function res = fun_ss(theta,data)
    % (Length-of-data.xdata by SoP)-matrix of responses
    y = data.fun(theta,data.xdata)';
    % provide stochasticity
    global SIGMA;
    y = y + SIGMA*randn(size(y));
    % calculation of LSQ
    res = sum((repmat(data.ydata',1,size(y,2))-y).^2);
    res = res';
end
```

3.2.2 Results

Exploiting the same idea as for deterministic example, several series of calculations should be conducted to determine influence of the population size on accuracy and performance of DE algorithm. Moreover, comparison with built-in solver cannot be made in this case due to inapplicability of deterministic optimizers to solution of stochastic problems. However, fitting the data by solution obtained by DE will be illustrated and correlation between obtained parameters will be explored with the help of plotted data and covariance matrix.

Firstly, the graph illustrating required dependence is presented on Figure 3.6. It is possible to conclude out of this graph that optimal population size according to accuracy (percentage of fails) tends to $20 \cdot Np$ and increases with growing noise level. Behavior of generation number spent for getting the solution in stochastic case differs from behavior in deterministic case. In the deterministic case the function describing dependence between generations number and population size approaches specific constant almost independently on population size. At the same time, in the stochastic case there is no upper limit of number of generation with growth of population size. It can be explained by the fact that the main stopping criterion implemented here is devoted to standard deviation of members of population through specified length history. Thus, with rising of population size the growth of the diversity among population is introduced simultaneously. Hence, the stochasticity involved to the behavior of the system is responsible for diffi-

culty of meeting the stopping criteria, because the huge population will be updated more probable contributing in diversity of generation history.

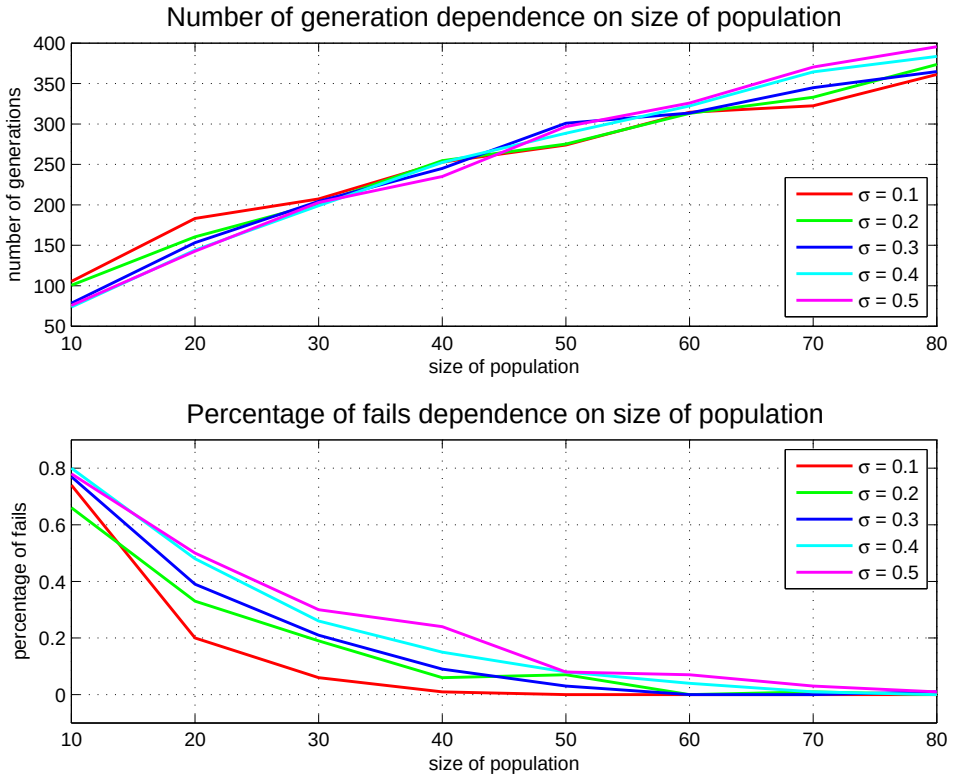

Figure 3.6: Number of generation and fails percentage dependence on size of population. Stochastic case.

Similarly to deterministic case, figures describing the original data fitting with different noise levels by the DE approach are presented in the Figure 3.7, Figure 3.8. Besides it is important to point out that in this case there is a distribution of possible optimal solutions contrary to the single optimal solution in the deterministic case. This fact also is presented in the graphs and covariance matrices are mentioned above the graphs. Two cases with low and high level of noise are used as an example:

$$mean(\theta) = [-5.9898965011, 2.9968101829, -0.2997310363];$$

$$cov(\theta) = \begin{pmatrix} 0.0109479212 & -0.0043066165 & 0.0004138307 \\ -0.0043066165 & 0.0017167433 & -0.0001668795 \\ 0.0004138307 & -0.0001668795 & 0.0000164162 \end{pmatrix};$$

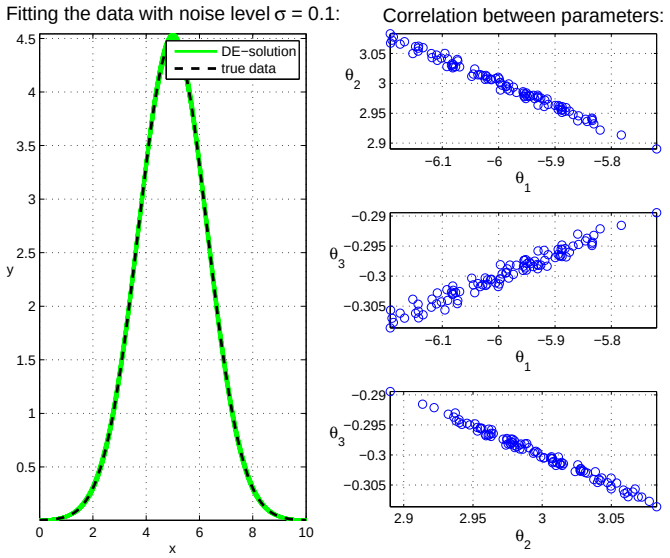

Figure 3.7: Fitting the data with noise level $\sigma = 0.1$.

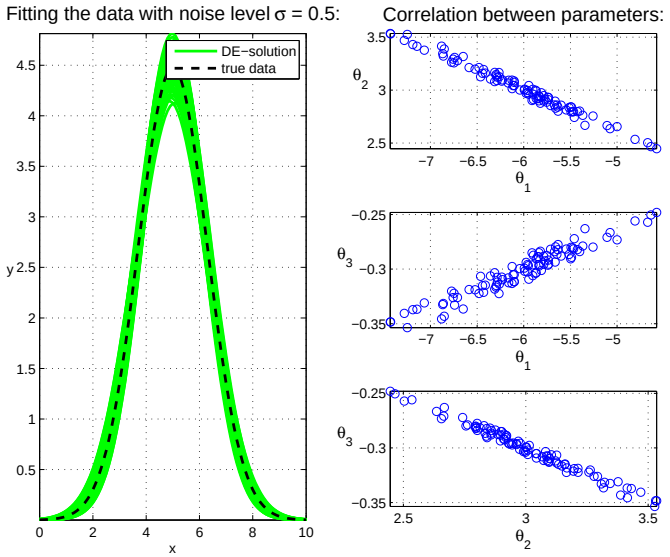

Figure 3.8: Fitting the data with noise level $\sigma = 0.5$.

$\text{mean}(\theta) =$ [-6.02119103, 3.010629061, -0.3013245289];

$$\text{cov}(\theta) = \begin{pmatrix} 0.3582556082 & -0.1382550489 & 0.01285172823 \\ -0.1382550489 & 0.0543814342 & -0.00515219543 \\ 0.0128517282 & -0.0051521954 & 0.00049825664 \end{pmatrix};$$

3.3 Advanced features in DE theory

After testing the DE approach on the simple example, it is possible to conclude that although this method is quite powerful tool even in the classical form, the full potential of this method is not revealed in this form. To determine the elements of the algorithm requiring development, it is necessary to turn to the theoretical aspects again [3]. One of the most essential properties of the DE algorithm is *contour matching*. *Contour matching* means that the population vector has capability to adapt to the objective function surface in such way that the challenging regions are explored automatically once they are found. Moreover, DE algorithm has ability of *basin-to-basin transfer*, where searching points can move from one vicinity of attraction to another. Thus, diversity yielded by specific amount of difference vectors allows basin-to-basin transfer causing the global optimum determination. However, there is a weak side of contour matching when deceiving nature of objective function matches vector of population leading away from the global minimum, hence one should keep balance related to contour matching. There are several ways which should be investigated to improve performance properties of DE algorithm; they can be divided into four groups:

1. Alternative ways for mutation procedure;
2. Investigation of influence of control parameters of algorithm;
3. Alternative selection schemes;
4. Dynamical implementation of DE approach.

Following subsection will describe all this aspects in details.

3.3.1 Alternative ways for mutation procedure

Perturbation of the base vector by mutation has been treated very early by researchers and appeared in literature ([3],[4]). Thereby, the notation

scheme of mutation was established and has had a form '$DE/x/y/z$', where x denotes the way of base vector choosing, y denotes the number of difference vectors used, and z denotes the crossover method. Thus, classical DE approach belongs to $DE/rand/1/bin$ class.There is a plenty of different schemes which can be handled as well, for instance:

- $DE/best/1/bin$:

$$\mathbf{v}_{i,g} = \mathbf{x}_{best,g} + F \cdot (\mathbf{x}_{r1,g} - \mathbf{x}_{r2,g}) \tag{3.9}$$

- $DE/current - to - best/2/bin$:

$$\mathbf{v}_{i,g} = \mathbf{x}_{r0,g} + F \cdot (\mathbf{x}_{best,g} - \mathbf{x}_{r0,g}) + F \cdot (\mathbf{x}_{r1,g} - \mathbf{x}_{r2,g}) \tag{3.10}$$

- The variant of $DE/best/2/bin$:

$$\mathbf{v}_{i,g} = \mathbf{x}_{best,g} + \frac{1}{2}F \cdot (\mathbf{x}_{r1,g} - \mathbf{x}_{r2,g} + \mathbf{x}_{r3,g} - \mathbf{x}_{r4,g}) \tag{3.11}$$

In fact there are many other linear combinations of vectors which can be used, the key point here to be maintained is that the base vector should be distinct from the other vectors. Thus, generalization of possible linear combinations can be presented as:

$$\mathbf{v}_{i,g} = \mathbf{y}_{i,g} + F \cdot \frac{1}{N} \sum_{n=0}^{N-1} (\mathbf{x}_{r(2n+1),g} - \mathbf{x}_{r(2n+2),g}) \tag{3.12}$$

In practice the most widely used formulas are (2.5) and (3.9) owing to the fact that other combinations are greedy. These schemes will be implemented in Matlab and tested after describing all other developing issues.

3.3.2 Investigation of influence of control parameters of algorithm

The crucial property of optimization algorithm is *rotationally invariance*. Basically, this property means, that the mean number of function evaluations for an objective function and its rotated counterpart is the same, i.e. algorithm does not depend on coordinate axes of problem. At the same time it was shown [1] that the mean number of function evaluations for an objective function and its rotated counterpart the same in DE algorithm only with crossover probability $Cr = 1$. Thus rotationally invariance property holds only for this particular case. Nevertheless, this statement does not mean that lower values of this parameter should be avoided. For instance, low values of Cr parameter are appropriate for separable objective functions. However, there is a rule for default values for control parameters proposed by Storn and Price which empirically has approved their usefulness, namely:

$$\begin{aligned} \text{scale factor}, F &\in [0.5, 1.0] \\ \text{crossover probability}, Cr &\in [0.8, 1.0] \\ \text{size of population}, Np &= 10 \cdot D \end{aligned} \tag{3.13}$$

Although they are applicable for huge range of practical purposes, tuning of control parameters should be made for specific problems additionally. Once it is required to find a general solution for this problem, a task of estimation of the best settings of F, Cr and Np automatically arises. One of the recent approaches is to consider F and Cr as supplementary parameters in population vector. Hence each parameter vector has $D + 2$ parameters, where last two parameters contain individual values for F and Cr. Thereby, if the trial vector wins in selection procedure then either both F and Cr inherited from the base vector to the next generation or individual parameter F and Cr generated randomly to particular vector. The results obtained by researches show that this schemes yields improving of objective function value after fixed set of function evaluation, compared to classical DE with $F = 0.5$ and $Cr = 0.9$.

Considering aspects which make DE algorithm a powerful tool for stochastic optimization [3], it is important to mention the fact that in the classical 'one difference vector' scheme the vector perturbations are based on $Np \cdot (Np - 1)$ nonzero difference vectors as opposite to other types of evolutionary algorithms where a predetermined probability density function is employed for that purpose. This fact provides contour matching property mentioned earlier. However, in the endeavor to obtain fast convergence the population size Np is usually kept low, which leads, in fact, to limit probability ($Np \cdot (Np - 1)$ possible directions) to find regions with possible improvements of objective function and so-called *stagnation* may occur. *Stagnation* here means that there is no improvements of cost function during several generations. Such dependence between size of population and speed of convergence demands alternative ways for diversity enhancement without increment of population size. It can be seen that one of the methods for diversity enhancement has always been an integral part of DE, crossover.

It is important to mention that there are two main deficiency of crossover, namely: loosing of contour matching property when strong crossover used (e.g. $Cr = 0.1$) and non-rotationally invariant nature. Nevertheless, strong crossover is perfect for separable functions where the main tendency is to search along the main parameter axes. Moreover, diversity enhancement impact of crossover is proved to outweigh disadvantages of this method especially for light crossover (Cr close to 1) for non-separable objective functions.

Scale factor F seems to be useful tool for solving the diversity enhancement task. There are two approaches to expand number of possible search direction tuning F parameter and maintaining size of population. Both of them exploit randomization of this parameter and called *dither* and *jitter* [3]. *Dither* employs following formula for randomization of parameter F:

$$F_{dither} = F_l + rand_g(0, 1) \cdot (F_h - F_l), \tag{3.14}$$

where F_h and F_l the highest and the lowest values of F. As it can be pointed out from this formula, F scale factor is randomized for whole generation;

however it can be generated for each of Np difference vector separately:

$$F_{dither} = F_l + rand_i(0,1) \cdot (F_h - F_l), i = 0, ..., Np - 1 \tag{3.15}$$

Besides improving the speed of convergence of DE algorithm, it has been proved that such method improves handling of problems with noisy objective functions. Thereby, due to the fact that the dither maintains contour matching and rotationally invariant property, it can be used for diversity enhancement.

Although *jitter* also randomizes scaling factor F, it makes it for every single parameter and every new mutant vector i according to:

$$F_{jitter,i} = F \cdot (1 + \delta \cdot (rand_j(0,1) - 0.5)) \tag{3.16}$$

The impact of jitter to diversity enhancement is not totally analyzed yet, whilst it has been shown that crucial thing in this method is that δ should be very small (~ 0.001). Moreover, δ itself can be randomized as well. It has been also shown that in general jitter is not rotationally invariant, but for small values of δ this deficiency negligible.

Generalization of principles mentioned above can be following formula, where both idea of dither and jitter are taken into account:

$$F_{jitter\&dither,i,g} = (F_l + rand_g(0,1) \cdot (F_h - F_l)) \cdot (1 + \delta \cdot (rand_j(0,1) - 0.5)) \tag{3.17}$$

Thereby, all advantages of proposed control parameter developing make these features appropriate for diversity enhancement and may be implemented as Matlab code.

3.3.3 Alternative selection schemes

There is another approach which may be useful for convergence acceleration of classical DE approach. It is usually called as *generation jumping* [3]. To

begin with, it is necessary to describe *opposition-based scheme* of DE. Idea of this scheme is to generate opposite population to current or mutant vector with respect to existing parameter vector. This concept can be illustrated by the following formula:

$$\mathbf{v}_{i,g,opposed} = \mathbf{x}_{g,min} + \mathbf{x}_{g,max} - \mathbf{v}_{i,g}, \tag{3.18}$$

where $\mathbf{x}_{g,min}$ and $\mathbf{x}_{g,max}$ denote minimum and maximum of parameter values for current generation respectively. For the initial generation the absolute bounds are used. It is necessary to mention that opposite point's generation scheme neither maintains rotationally invariance nor basin-to-basin transfer, that is why this approach is proved to be applicable only inside generation jumping scheme.

According to the generation jumping idea, the opposing points are not chosen on an individual basis but generated to the whole population with given probability Jp (for example $Jp = 0.25$). In this case there is two times more diversity in choosing direction: Np possible directions from the original population and Np possible directions from the opposite population. Out of this $2 * Np$ points the Np best ones are chosen to form the next generation. In the community connected with evolutionary computations this approach is called *elitist* or $(\lambda + \mu)$-selection, where λ best points are chosen out of $\lambda + \mu$ candidates. Although elitist scheme has the possibility to speed up convergence with comparison to classical one-to-one selection scheme, at the same time it can lead to premature convergence to improper solution.

Thus, owing to elitist selection is not used in every generation but only with given probability Jp, generation jumping scheme maintains proper balance between elitist and one-to-one scheme of selection candidates for the next generation. Thereby, generation jumping scheme collects both diversity expanding feature from opposition-based scheme and fast convergence counteracting the loss of focus towards the global minimum from $(\lambda+\mu)$-selection with $\lambda = \mu = Np$.

3.3.4 Dynamical implementation of DE approach

Once whole algorithm of DE has described specific influences of different mutation schemes and control parameters, it is possible to analyze the evolution mechanism itself. It can be seen from the existed implementation that the classical DE has two inherit defects, namely, *slow feedback to update of population status* and *extra memory requirement.*

From the point of view how classical DE algorithm updates population, it can be classified as static algorithm [2]. It means that the whole population $\mathbf{P}_{x,g}$ of generation g remains unchanged until it is replaced by the next population $\mathbf{P}_{x,g+1}$ of generation $g+1$. Thus, the classical DE method does not utilize any information which has taken place during current evolution loop. The classical DE approach keeps using current population to produce mutant vector until current evolution loop is completed, for example in the scheme DE/best/*/* the best vector maintained even when more dominant vector is found during this evolution loop. Therefore, classical DE method delays response on the progress of the population status making convergence slower.

It is possible to reformulate evolution mechanism of classical DE using well-known distinguish between the *Jacobi method* and the *Gauss–Seidel method* for solution of a set of linear algebraic equations as inspiration. It is necessary to remind these methods. The classical iteration numerical method for solution of system of linear equation $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$ is *Jacobi method* which can be presented as following formula:

$$x_i^{n+1} = \frac{b_i - \sum_{j \neq i} A_{ij} x_j^n}{A_{ii}}, \tag{3.19}$$

where x_i^0 is predefined initial guess. Like in the case of classical DE, this method does not overwrite anyone of component of the solution x^n before the new solution x^{n+1} is fully obtained. This method has a slow convergence and one of the most essential reasons for this is considered deficiency. The *Gauss-Seidel method* avoids this drawback taking into account new component of

solution x^{n+1} as soon as it is available being updated:

$$x_i^{n+1} = \frac{b_i - \sum_{j<i} A_{ij} x_j^{n+1} - \sum_{j>i} A_{ij} x_j^n}{A_{ii}} \tag{3.20}$$

Although iteration index n still remains in this formula, the only reason is convenience to emphasize the dynamic updating scheme. As it can be concluded out of this method, there is no more need in the memory for the next iteration, because updated solution is already included into current iteration. As a result better convergence of solution is obtained with weaker conditions and less memory requirements. Mimicking such development of static solution, dynamic DE method is proposed. Now, all modification of population during current evolution loop is being treated immediately. The most important drawback of dynamic DE is the fact that it cannot be simply parallelized any more. Summing up all possible modification of classic DE approach, the new implementation in Matlab is developed and tested on introduced both deterministic and stochastic toy-cases which can be found further. Also analysis of results obtained during these experiments is performed to be utilized in the more complex problems.

3.4 Application of advanced DE to "toy-case" problems

This subsection is devoted to implementation of discussed earlier modification of classical DE algorithm and determination of the most efficient schemes among possible combination of proposed changes. Since it is necessary to implement several new modifications, the structure of our functions will be changed. There are two main differences from the previous realization. The first distinguish is that now `dde.m` function contains all routines dedicated to initialization, mutation, crossover and selection stages of DE instead of having several separate function. The second distinguish is the drawback of the dynamical approach, i.e. it is necessary to introduce loop of calculation where changes in the population will be taken into account immediately. Moreover, it is essential to provide possibility for user to choose which mutation and selection scheme have to be used. Also, user can specify the value of parameter Jp which is responsible for generation jumping scheme. To compare productivity of proposed improvement it will be tried to estimate mean number of generation length required for approaching the specified accuracy for different possible combinations of mutation and selection schemes and for probability values of generation jump. Hence, we will consider sixty different combinations. Besides, the fail attempts to solve the problem may occur due to weak initial guesses and noise level in the stochastic case, thus it can be considered as additional parameter for comparison. Afterwards we can choose several best schemes in the meaning of appropriate relation between productivity and accuracy which will be used in the main problem of the paper. Therefore, in the following sections the key points of implementation of modifications will be discussed and the improved DE algorithm will be applied to the deterministic and stochastic cases of proposed example.

3.4.1 Implementation of advanced features in Matlab

First of all we will consider key points of generation jumping procedure. We skip most of auxiliary lines of Matlab code and emphasize on the most crucial parts.

```
%% generation jump
    ...
    Jp = paramsDE.Jp;
    cur_pop = pop(:,1:D+1); % parameters + cost
    if rand(1)<=Jp
        mins = [];          % minimum values of each parameter ...
            vector
        maxs = [];          % maximum values of each parameter ...
            vector

        ... % calculation of min/max

        % "oposite population":
        op_pop = repmat(mins,SoP,1) + repmat(maxs,SoP,1) - ...
            cur_pop(:,1:D);
        tmp = func(op_pop,data);
        op_pop = [op_pop tmp];

        % joined population:
        cur_pop = [cur_pop;op_pop];

        % the fittest SoP-size population from the joined ...
            population:
        [TMP,IX] = sort(cur_pop);
        ind = IX(1:SoP,D+1);
        res = cur_pop(ind',:);
        return;
    end
```

When algorithm have to make generation jump the minimum and maximum values of each D parameter vector in current population are calculated to be involved into computation of opposite generation. Once this generation has been constructed, the cost function should be calculated and then both populations are joined to select SoP best ones. It is important to mention that according to the theory of generation jumping approach, this procedure replaces all mutation-selection steps in current generation, i.e. in the case of generation jumping only procedure of survival the fittest joined population

elements are fulfilled.

The mutation routine remains the same whilst the crossover and the selection parts cannot be divided into independent parts due to Dynamical approach restrictions.

```
%% crossover and selection
temp = rand(SoP,D);
ind = randi(D,SoP,1);
temp(sub2ind([SoP,D],[1:SoP]',ind)) = 0;
indCrossover = (temp > CR);
...
switch F_type
    ...
    % Choosing the specified scheme of parameter F evaluation
end
% Population loop is drawback of Dynamical approach :
for i = 1:SoP

    switch mutation_type
        ...
        % Choosing the specified scheme of mutant vector ...
            generation
    end
    % Construction of trial vector:
    new(indCrossover(i,:)) = cur_pop(i,indCrossover(i,:));
    tmp = func(new,data);
    % Selection:
    if tmp<cur_pop(i,D+1)
        cur_pop(i,:) = [new tmp];
    end
end
```

This listing illustrates that one full step of population evolution is now divided into SoP sequential steps where all changes in current population are updated immediately and allowed for following elements to use updated values in computations. To solve the problem of the determination of the best DE scheme, it is necessary to modify main file of the program. It should provide possibility to generate information about mean number of generation and number of fails for every combination of control parameters. Therefore, several nested loops are needed and procedure of saving the data into log file will be implemented. It should be mentioned that repeated calculations for every combination of control parameters will be used for the purpose

of estimation the mean number of generations which is necessary to reach suitable accuracy.

3.4.2 Deterministic "toy-case" problem

Once the extended version of the DE approach has been taken into consideration in Matlab implementation, the analysis of improvement influence should be done according performance this algorithm with different control parameters combination in deterministic case. The results containing all possible combination of introduced tuning parameters with mean generation number and accuracy are presented in Table B.1 and Table B.2 in Appendix. This table provides the information about solution of deterministic case problem with noise level $\sigma = 0.2$ and population size $Np = 30$. The following abbreviation will be utilized to indicate different values of tuning parameters:

1. Mutation type:
   - $'Cls'$ corresponds to the scheme (2.5);
   - $'Bst'$ corresponds to the scheme (3.9);
   - $'CtB'$ corresponds to scheme (3.10).
2. F type:
   - $'Cls'$: F is equal to the predefined constant value for all members;
   - $'Dthr1'$: F is calculated according to (3.14);
   - $'Dthr2'$: F is calculated according to (3.15);
   - $'Jttr'$: F is calculated according to (3.16);
   - $'Mix'$: F is calculated according to (3.17).
3. Jp is generation jumping probability. It has four values, namely: 0.0, 0.1, 0.2, and 0.3.

According to this table the most efficient combination of parameters are:

- $['Bst','Cls', 0.0]$
- $['Bst','Dthr1', 0.0]$
- $['Bst','Dthr2', 0.0]$
- $['Bst','Jttr', 0.0]$
- $['Bst','Mix', 0.0]$

All these combinations demonstrate roughly equal performance in current example and improve the classical DE scheme almost three times which can be concluded out of Figure 3.2.

It should be mentioned that in deterministic case it is impossible to analyze the influence of generation jumping scheme and different scheme of F parameter evaluation on the performance of the algorithm because the main impact in this case is caused by the mutation scheme and dynamical implementation. Particulary, it can be seen that the best scheme in this case is $DE/Best/1/bin$.

Improvement in performance obtained by implementation of advanced features can be illustrated by Figure 3.9 which is obtained by introducing DE/best/1/bin mutation scheme, dither and jitter and generation jumping with probability 0.0. This figure can be compared with Figure 3.2.

The main conclusion that can be made out of this figure is that although introduced features demand higher size of population in comparison with classical implementation, this drawback is totally overweighed by the average number of generation which are needed to estimate parameters of the deterministic problem with specified accuracy. In the advanced implementation of DE the mean number of generation is two or even three times less then in classical scheme. Moreover, influence of the most of the tuning parameters remains unestimated due to the nature of deterministic problem and should be tested during application to stochastic case of the proposed example.

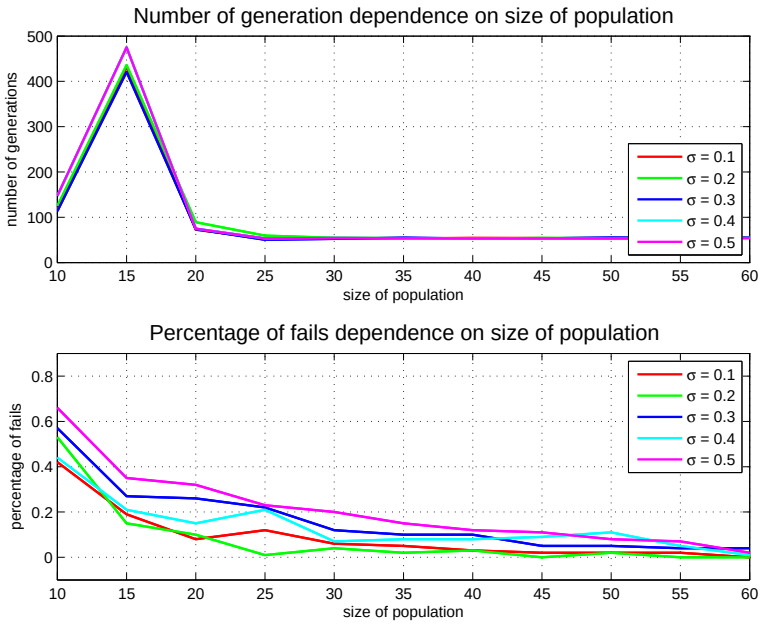

Figure 3.9: Number of generation and fails percentage dependence on size of population.

3.4.3 Stochastic "toy-case" problem

The same analysis as for deterministic case should be conducted here as well. The full table which contains different combinations of tuning parameters and corresponding performance and fail percentage can be found in the Appendix (Table B.3, Table B.4). This table was obtained with stochastic noise level $\sigma = 0.4$ and size of population $Np = 60$. Stochastic nature of problem and optimizer leads to appearance of fail attempts to estimate parameters. It should be pointed out from these tables that classical scheme is more accurate but time consuming. At the same time taking into account an approach which take an appropriate proportion between performance and accuracy as measure for goodness of tuning parameter set the following sets should be emphasized:

- $['Bst','Dthr1', 0.1]$
- $['Bst','Mix', 0.2]$

- $['CtB','Jttr',0.2]$

Crucial consequence of current example is the fact that when stochasticity involved into behavior of system, utilization of generation jumping approach stabilizes the population evolution making the estimation more accurate and reduces probability of fail attempts. The following figure illustrates influence of population size on performance and accuracy of DE algorithm during application to stochastic example. $['Bst','Mix',0.2]$ set of tuning parameters was used for this purpose:

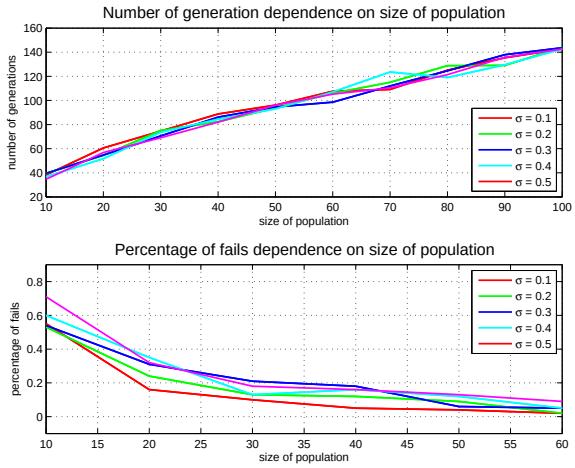

Figure 3.10: Number of generation and fails percentage dependence on size of population.

It can be noticed that the main tendency of tuning parameters influence for stochastic case is similar to deterministic case. Thus, drawback in average increasing of population size needed to avoid fail attempts of parameter estimation can be neglected owing two twice or triple growth in performance as measured by average generations needed to reach the specified accuracy. Thereby, advanced features involved into implementation of the DE algorithm shows strong benefits and will be maintained during estimation of chaotic dynamics.

4 Parameter estimation of chaotic dynamics

This section is devoted to exploring of the Lorenz system, which exhibits chaotic dynamics, using DE approach. The main aim is to apply the DE algorithm to the parameter estimation problem of the Lorenz system and solve additional problems which will be faced during this procedure. Thus proposed dynamical implementation with advanced techniques described and tested earlier on toy-case example will be taken into account for improving convergence properties of solution. For further investigation of the proposed problem, historical and theoretical background dedicated to Lorenz system will be presented.

4.1 Overwiew of Lorenz system

The *Lorenz system* is a specific system of ordinary differential equations and a classical example of a dynamical continuous system exhibiting chaotic behavior.

From the mathematical point of view a *dynamical system* is a concept where a fixed rule determines the time dependence of a point in a geometrical space. In other words, for every given time the behavior of the dynamical system is characterized by a *state* which is a vector and corresponded to a point in an appropriate state space. Besides, the evolution rule is a fixed rule describing future states of the system uniquely. Thus, the rule is deterministic leading to the fact that for given time interval only one future state follows from current state. *Chaotic behavior* of a dynamical system is a behavior which is highly sensitive to initial conditions. It means that in long time period two runs of the same dynamical system with small difference in initial conditions have totally different states. Although evolution of such dynamical system can be fully determined by initial conditions and the fixed evolution rule as in considering case, prediction of evolution is impossible in general due to the fact that small differences in initial conditions can be caused even by rounding errors during computations. In other words,

although no artificial randomness is not involved into the definition of a system, the behavior of the system remains unpredictable.

The Lorenz system was firstly published by Edward Lorenz in his paper in 1963 [6]. There he described a three parameter family of three-dimensional ordinary differential equations which show extremely complicated solution while applying to the computational system which was available in that time. These equations, now known as Lorenz equations, gave a rise to research of such system and have been studied by many authors which contributed a lot to investigation of this field. Lorenz's work is important even today because he had an insight of the essence of chaos although no one did not know it in those days and his work settled the today's chaos theory. Notability of these equations is caused by the fact that they were derived by Lorenz while searching for a three-dimensional set of ordinary differential equations which can be used as a model for unpredictable behavior normally associated with weather and atmosphere. Lorenz, a meteorologist as well as a mathematician, derived his equation from the Navier-Stokes equations with the Boussinesq approximation, which described the atmospheric convection. Although the Lorenz equation loses the correspondence to the actual atmosphere in the process of approximation, it is important that chaos appears from the equation which describes the dynamics of the nature. Hence, Lorenz system has a following form:

$$\begin{aligned} \frac{dx}{dt} &= \sigma(y - x), \\ \frac{dy}{dt} &= x(\rho - z) - y, \\ \frac{dz}{dt} &= xy - \beta z, \end{aligned} \tag{4.1}$$

where σ, ρ and β are three real positive constants.

Variables involved to this system has specific physical meanings related to the essence of the problem, namely the variable x measures the rate of convective overturning, the variable y determines the horizontal temperature variation and the variable z determines the vertical temperature variation. Three parameters σ, ρ and β contained in the system are respectively pro-

portional to the Prandtl number, the Rayleigh number, and some physical proportion of the region under consideration. The Prandtl and Rayleigh numbers are dimensionless numbers which are responsible for specific physical properties of a fluid.

It was shown that with the values $\sigma = 10$, $\rho = 28$ and $\beta = 8/3$ the Lorenz system exhibits chaotic behavior which can be illustrated by the three-dimensional plot of its trajectory:

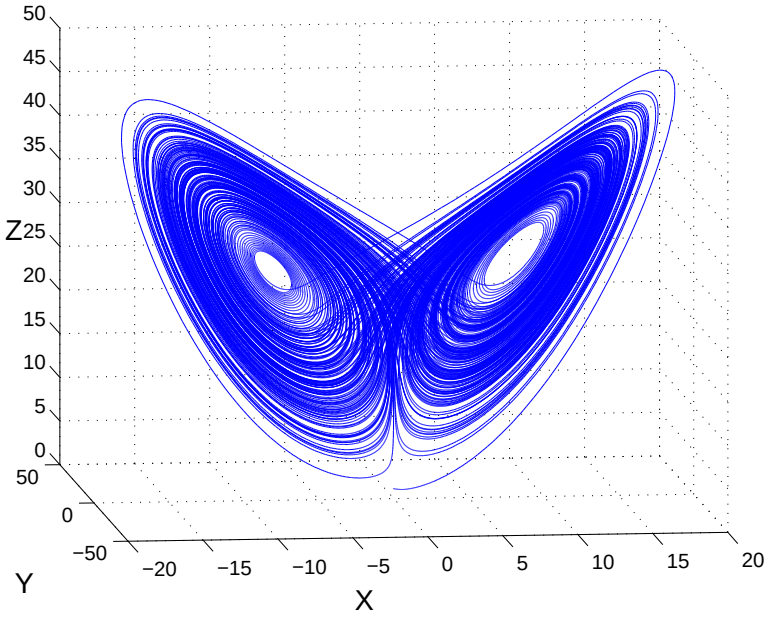

Figure 4.1: Lorenz system's phase portrait with initial values $[x_0, y_0, z_0] = [0, 1, 1.05]$.

It was also proved that this figure has different turbulence-type properties, they are [6]:

1. The trajectory presented in Figure 4.1 is not periodic.
2. However long the numerical integration is continued the trajectory does not approaches either stationary or periodic behavior.
3. The general form of the Figure 4.1 does not depend either on choice

of initial conditions if initial transient part of trajectory is ignored; or on choice of integrating routine being responsible for calculation of trajectory. This trajectory of Lorenz system is usually called *Lorenz Attractor*.

4. Contrary to the previous property, the details in Figure 4.1 is crucially dependent both on initial conditions and integration routine. As a consequence of such sensitivity, the property of unpredictability is taken place which is typical for chaotic behavior.

Once historical and theoretical aspects of Lorenz system is covered, study of parameter estimation of inverse problem related to Lorenz system can be discussed. The following subsection is devoted to this aim.

4.2 DE application to parameter estimation of Lorenz system

The aim of the parameter estimation problem is to determine unknown parameter values of given model according to measured data. Thus, in order to estimate the Lorenz system artificial data should be generated in advance. One of the most crucial problems to be faced is high sensitivity of presented system to initial values. It makes impossible to predict behavior of the system in long time period. This problem can be handled by dividing full time interval into smaller time intervals where system exhibits relatively deterministic behavior. Turn to the DE approach implementation, the basic idea to solve this problem is to apply one full evolutional step consisting of mutation, crossover and selection sequentially from one time subinterval to another. As an initial population for the every next time subinterval the 'survival' population from the previous subinterval is treated. The cost function is calculated by commonly used sum of squared differences between observation(measured noisy data) and the model. However the key point here is the fact that cost function is different in every subinterval because measured data differs. Moreover, it is assumed that the state of the system in the beginning of considered time window is known but noisy. Thus, these values can be used as mean values for initial values of the system in current time interval. Further, specific initial values for every population member are generated from Normal distribution with known mean value and specified standard deviation.

The state of Lorenz system S_i can be described by the following model:

$$S_i = F(t_i, X_0, \tilde{\theta}), \tag{4.2}$$

where $F(...) = (f_1(...), f_2(...), f_3(...))$ is Lorenz equations, $X_0 = (t_0, x_0, y_0, z_0)$ is a set initial conditions, $\tilde{\theta} = (\beta, \sigma, \rho)$ is a set of parameters of Lorenz system.

According to the scheme presented earlier, if there is a $j-$ time window of length w with data measurements $Y = (y_1, y_2, y_3)$ in every dt time points then the cost function LSQ of the model with current set of parameters can

be calculated using following formula:

$$LSQ_j = \sum_{i=w\cdot(j-1)+1:dt:w\cdot j} (F(t_i, X_{0j}, \theta) - Y_i)^2, \tag{4.3}$$

where $X_{0j} = (t_0, x_0, y_0, z_0) = (t_{w\cdot j-1}, \mathcal{N}(Y_{w\cdot j-1}, \sigma^2))$, and σ is specified standard deviation. Thus, whole procedure of estimation of parameters is divided into several step:

- Initialize the population for current time window by the population obtained on previous time window;
- Apply one full DE step for current population taking into account new data measurements and initial values influencing on calculation of cost function Figure 4.2.
- Continue this process until estimated parameters have been found with desired accuracy.

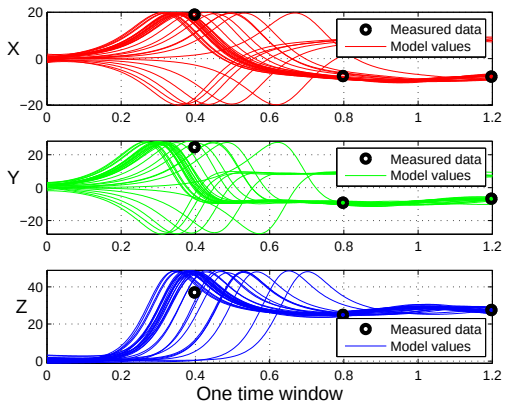

Figure 4.2: Example of one time window for Lorenz system.

During one evolutional step of the DE method, new parameter values which fit the data for current time window in the best way are proposed. Although there is probability of appearance of parameter sets exhibiting suitable fitting of the data only for current time window, they will be neglected during next time windows due to high values of cost function. According to this

idea, the DE approach adopts population with increasing number of time window taking into account the noise in measured data. As a result, true parameters of the system can be estimated. The presented concept is implemented in Matlab. Most of computational routines devoted to the DE approach are already implemented during investigation of previous toy-cases. Therefore, the implementation of solution of the current problem is required artificially measurements generation, Matlab-functions providing calculation of cost function, several tunings in DE routines and main file which is responsible for assembling all parts of realization to complete solver.

4.3 Matlab implementation

It is assumed here that performance which can be obtained by the dynamical implementation of DE algorithm overweighs the drawback in loss of parallelizability. It is truth under conditions of limited computational capacity and impossibility to provide parallel calculations. In the case when parallel computations can be conducted, the static implementation should be used. Thus, in this case the dynamical advanced implementation of DE method will be applied to the current problem.

Required data for parameter estimation problem of Lorenz system is generated using built-in Matlab solver with integration step $dt = 0.0025$ on the time interval [0:500]. Based on this data, measurements are chosen with analyze time interval $aint = 0.4$. Thus, the cost function for population members is calculated based on solution of Lorenz system with given integration step compared to the truth data in every $aint$ time points.

Calculation of cost function in this particular case can be implemented in following way. Firstly, function which produces evolution of dynamic of Lorenz system is needed. In presented project it is called `lorenzeq.m`:

```
function sdot = lorenzeq(t,s,params)
    % parameters
    beta  = params(1);
    sigma = params(2);
    rho   = params(3);
    % states values
    x = s(1);
    y = s(2);
    z = s(3);
    % equation
    dx = sigma*(y-x);
    dy = x*(rho - z) - y;
    dz = x*y-beta*z;
    % differential
    sdot = [dx;dy;dz];
end
```

This function has typical structure which is required for built-in Matlab

ode-solvers. It is important to mention that even for solution of autonomous systems (a system of ordinary differential equations which does not explicitly depent on the independent variable, in this case t) the independent variable has to be mentioned in the list of function parameters. Once such function has implemented, solution of a given system of differential equations with specified initial values and parameter set can be obtained using `lorenz_m.m`:

```
function y = lorenz_m(data,theta,y0)
    time    = data.xdata;
    % parameters of solver:
    opt     = odeset('RelTol',1e-11,'AbsTol',1e-11);
    [t,y]   = ode113(@lorenzeq,time,y0,opt,theta);
end
```

Finally, required cost function can be implemented in following way:

```
function [res,Y] = lorenz_ss(theta,data,init)
    res = [];
    aint = data.aint;   % analysis interval
    dt = data.dt;       % integration interval
    truth = data.truth; % truth data
    for i = 1:size(theta,1)
        y0  = init(i,:);% initial value
        Y(:,:,i)   = ...
            lorenz_m(data,[theta(i,1),theta(i,2),theta(i,3)],y0);
    end
    % solution in the point where truth values are calculated
    response = Y(aint/dt:aint/dt:end,:,:);
    % cost function calculated in LSQ-sense
    nens = size(response,3);
    ss    = (response-repmat(truth,[1 1 nens])).^2;
    for i = 1:nens
        res(i) = sum(sum(ss(:,:,i)));
    end
    res = res(:);
end
```

It should be pointed out that parameter `theta` in this function contains the whole population, e.g. Np-by-D matrix.

Further, `dde.m` routine should be slightly modified. According to the model and physical sense only positive parameter values are suitable, hence

to prevent appearance of negative parameter values checking the signs of parameters chosen during selection state is conducted and selection of only positive occurrence even with higher values of cost function is assumed.

Next step is implementation of the main function which is responsible for calling and treating of all subroutines to solve the problem. Before starting main part of estimation procedure, user should specify parameters of the DE algorithm like in toy cases. Once all required parameters have been specified, the main part of estimation process skipping detailed description can be coded as follows:

```
...
for i = 1:nSim
    % current time window
    a = (i-1)*range;
    b = (i)*range;
    time = a*aint:dt:b*aint;
    % analysis points for calc of cost function
    data.truth = truth(a+2:b+1,:);
    analysis = truth(a+1,:);
    % initial values generating
    INIT = repmat(analysis,SoP,1);
    init = INIT + init_std*randn(size(INIT));
    % encapsulation of data
    data.init = init;
    data.xdata = time;
    if i==1
        % initialization at the 1st step
        pop = initialization(data,paramsDE);
        ...
    end
    ...
    % one DE-step
    pop = dde(pop,data,paramsDE);
    % save evolution of parameter values
    history.beta = [history.beta pop(:,1)];
    history.sigma = [history.sigma pop(:,2)];
    history.rho = [history.rho pop(:,3)]
    save(['history.mat'],'history');
end
...
```

It should be mentioned that in context of this problem the time window and generation has the same meaning, e.g. the i-th time window corresponds

to the i-th generation. Thus Figure 4.3 illustrates solution for one time window obtained by DE approach. Color lines demonstrate solution of the problem with different population members using perturbed initial values. Black circles denote the true solution in given time points.

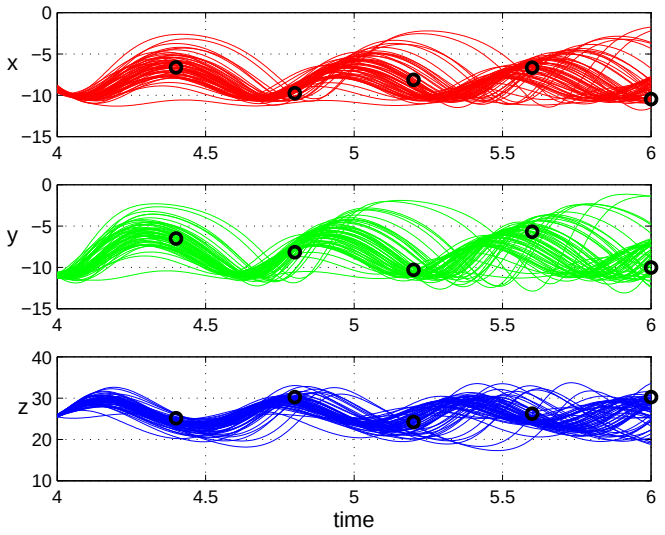

Figure 4.3: Solution for one time window of length `range*aint`.

The result of estimation and evolution of population members can be illustrated by Figure 4.4. Mean values of population members of the last generation are $\bar{\beta} = 2.674722, \bar{\sigma} = 10.110571, \bar{\rho} = 27.980898$, with standard deviations equal to 0.00666, 0.16551, and 0.03762 respectively.

Following set of DE parameters was used to perform presented estimation:

- Dynamical version of DE was used;
- Size of population $SoP = 30$;
- Crossover probability $Cr = 0.9$;
- Scale factor in mutation $F \in [0.45, 0.55]$;

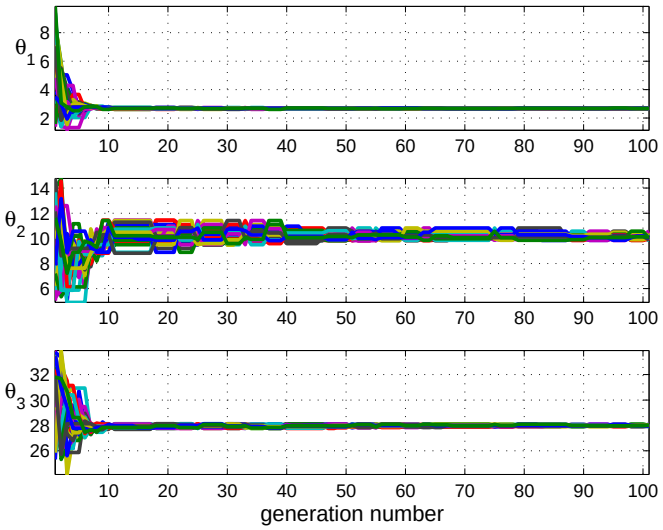

Figure 4.4: Parameter evolution.

- Mutation type is "DE/best/1/bin";
- Crossover scheme is mixed with both dither and jitter approach;
- Generation jumping technique with probability $Jp = 0.3$ was used;
- Standard deviation for generation of initial values $init\_std = 0.1$.

Also, it should be mentioned that the artificial data was generated with noise level $\sigma = 0.1$ and number of data points for analysis was used as $range = 3$.

Although implemented algorithm shows relatively high level of performance, it can be improved by additional adaptive step. The idea inspiring to propose such step is based on the fact that currently in every time window the values of cost function of trial population is compared to the values of cost function of current population calculated in the previous time window. Thus, speed of convergence of algorithm can be increased by introducing `recalc_cost.m` function. This function is responsible for recalculation of cost functions according to goodness of survived population in the current

time window. Moreover, cost function should be updated taking into account generation number i. Finally, the following formula for updating cost function values is proposed:

$$ss_{i-1,upd} = ss_{i-1} + \frac{ss_i - ss_{i-1}}{e^{\sqrt{i}}}, \tag{4.4}$$

where ss_{i-1} is value of cost function obtained in previous time window, ss_i is value of cost function calculated for current population in current window. Denominator is used for the purpose to reduce influence of difference between previous and current cost function values with increasing generation number. The meaning of such approach is based on the fact that the difference in the numerator in the time windows which are situated far from initial time point is mostly caused by the local behavior of the system while already chosen population members approved their suitability upon long time interval.

Once this additional step has been implemented, the estimation procedure can be conducted again and Figure 4.5 illustrates the evolution of population members with involved recalculation of cost function during first 20 steps. Mean values of population members of the last generation in this case are $\bar{\beta} = 2.677374, \bar{\sigma} = 10.299152, \bar{\rho} = 27.992075$, with standard deviations equal to 0.00596, 0.11498, and 0.02411 respectively.

Thereby, introduced additional step may improve performance properties of the implemented DE algorithm, though scheme proposed during previous research provides a powerful tool for estimating parameters of the system exhibiting chaotic behavior. The next and the final aim of the paper is to compare DE approach to one of already existed approach for estimating chaotic behavior. The next part of the thesis is devoted to this purpose.

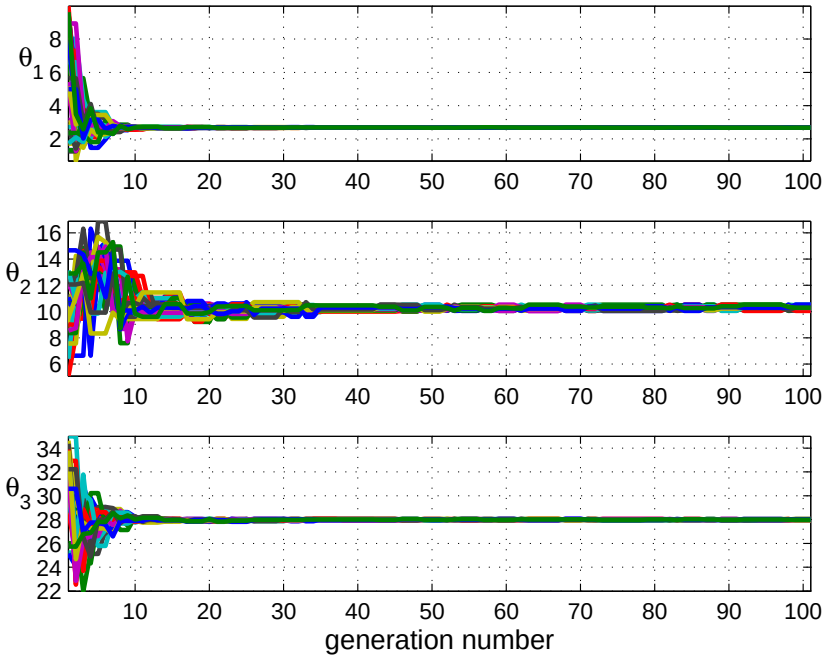

Figure 4.5: Parameter evolution with recalculation of cost function.

5 Comparison with EPPES solution

As it was mentioned earlier, one of the reasons to conduct current research dedicated to the DE approach for parameter estimation problems was the endeavor to find an alternative scheme for solution of such type of problem to the Ensemble Prediction and Parameter Estimation System (EPPES). EPPES algorithm was proposed by Marko Laine, Antti Solonen, Heikki Haario and Heikki Järvinen as a possible approach for the numerical weather prediction (NWP) modeling [7]. The weather forecasts today include the so called Ensemble Predictions: in addition to the main forecast, some 50 predictions are launched with perturbed initial values. The idea of EPPES is to use these calculations by adding parameter perturbations and estimating the performance of the parameters - at no additional CPU cost. From our point of view, the ensemble members present the different DE population members.

NWP models operate with so-called *closure parameters* approach. General assumption is that these *closure parameters* should define physical state of the considered NWP model independently of discretization details of specific problem, i.e. represent overall constants which should perform well in all weather types, times of day and year, etc. Manual tuning of these parameters becomes impossible with growth of models complexity. Moreover, some of closure parameters do not express directly observable quantities. However, the amount of observed data rises dramatically at present and it should be taken into account while estimating parameters. All these drawbacks have become a reason to propose algorithm which is responsible for estimating of closure parameters algorithmically and with increasing amount of observed data makes possible to provide suitable prediction for the model behavior.

5.1 EPPES concept

One of the main aims of proposed EPPES method is to provide information about posterior distribution in a Bayesian spirit even for nonlinear correlation between closure parameters. Although widely used for such type of problems *Markov chain Monte Carlo (MCMC)* methods are not directly applicable for this specific NWP case, the main idea of dealing with prior information and how to update it according to new observed data can be maintained. From the mathematical point of view MCMC methods are powerful tool for estimating parameters of the problem using Bayesian approach which considers data measurements and unknown parameters as random variables. According to Bayesian principle the full distribution of parameters is searched instead of finding the best fit [5]. It can be reached by combination of prior information about distribution of parameters which was adjusted regarding to evidence come from measurements through the likelihood (objective) function. This combination leads to the posterior distribution:

$$\pi(\theta) \equiv p(\theta|y) = \frac{p(y|\theta)p(\theta)}{\int p(y|\theta)p(\theta)d\theta}, \tag{5.1}$$

where $\pi(\theta)$ is posterior distribution, $p(\theta)$ is prior distribution, $p(y|\theta)$ is the likelihood function which contains the model itself, and the denominator is the normalizing constant due to which $\int_\theta \pi(\theta)d\theta = 1$.

One of the most challenging tasks here is the calculation of the normalizing constant in the denominator. To avoid necessity to calculate it, the most of MCMC algorithms utilize the idea of *Metropolis algorithm*. Basically this algorithm is consisted of two steps, namely the *proposal step* and the *acceptance step*. The first one is responsible for sampling candidate value, at the same time this value either accepted or rejected according to ratio of the posterior distribution at the candidate value and the present value:

$$\alpha = min(1, \frac{\pi(\hat{\theta})}{\pi(\theta_n)}), \tag{5.2}$$

where α denotes the probability to accept candidate value $\hat{\theta}$. Due to the fact that only ratio is needed, the problem of calculation of normalizing constant

is canceled out. Summing all mentioned above, the Metropolis algorithm can be divided into three steps:

1. Initialization. Choosing the starting point θ_1;
2. Sampling. Sample a new candidate value $\hat{\theta}$ from an appropriate proposal distribution $q(.|\theta_n)$;
3. Acceptance. Accept the candidate value $\hat{\theta}$ according to probability Equation 5.2. If candidate value is rejected then the previous value is repeated in the chain. Go to step 2.

The crucial point in the Metropolis algorithm is the fact that the acceptance of candidate value is proportional to the likelihood in the current step with previous step, thus the likelihood and the measurements have to remain the same from one step to another. But in on-line estimation problems when the new data appear during process it becomes impossible which makes the direct application of Metropolis algorithm inappropriate. Nevertheless, the key ideas of the algorithm can be maintained even for varying likelihood. In this case *Importance sampling approach* can be used [7]. The idea of this approach consists of two steps. The first step is weighting of each ensemble member in EPPES according to performance against measured data; and updating of proposal distribution for parameter perturbation by the weighted ensemble. Thus, in ideal situation during the on-line ensemble prediction process cumulatively new data are generated which leads to disappearing of uncertainties in closure parameters distribution. Although such approach is possible for testing the algorithm, taking into account non-ideal features of a process, the possible distribution of parameters is the main target of the estimation instead of individual value. This distribution is assumed to be a static. Moreover, nonlinear model can lead to the non-Gaussian correlation between the parameters. Thus, the history of parameter values evolution from one time window to another can be used to investigate this complex correlation. Moreover, instead of being directly interested in estimating of non-Gaussian posterior distribution of Q_i for every time window i, the vari-

ability of Q_i between different time windows is a main target for estimation and this distribution is assumed to be Gaussian.

To define theoretical aspects, the index i in variables will be used to refer to the time window i and index j is used to refer to the j-ensemble member. Hence, if the model in the i-time window is described by $F(x_i; \theta_i)$ and observations made for this time window is defined by y_i then applying ensemble structure to that model, it is possible to compare set of measurements y_i with each ensemble member $F(x_i^j; \theta_i^j)$ to obtain likelihood. Here it is also assumed that θ_i is a realization of random variable satisfying Gaussian distribution:

$$\theta_i \sim \mathcal{N}(\mu, \Sigma), i = 1, 2, \ldots, \tag{5.3}$$

where μ and Σ are static and unknown. Although the reason to treat target closure parameters as a random variable was discussed earlier, it should be mentioned here that μ parameter vector describes values of closure parameters which should perform optimally for the forecast model on average, at the same time variability in these values from one time window to another caused by the evident modelling errors are simulated by covariance matrix Σ. Thus, a sequential statistical approach is used to update the prior information according to incoming observation to produce posterior distribution [8]. Such procedure is repeated until an appropriate distribution is met and the posterior distribution of the current step becomes the prior distribution for the next step. In order to estimate the uncertainties related to the closure parameters θ, the sequential hierarchical statistical model should be proposed. Due to the fact that θ has unknown parameters of distribution, they also should be taken into account for this model as well. Thus, before analyzing the current time window i, the prior distribution of parameter vector θ_i is known but depends on previous steps:

$$\begin{aligned} \theta_i &\sim \mathcal{N}(\mu, \Sigma), \\ \mu &\sim \mathcal{N}(\mu_{i-1}, W_{i-1}), \\ \Sigma &\sim iWish(\Sigma_{i-1}, n_{i-1}). \end{aligned} \tag{5.4}$$

In this formulas parameters W_i and n_i control the influence of the new data

to process of estimation of μ and Σ respectively. Thus, while W_i goes to zero matrix and n_i to infinity, the effect of a single time window is decreasing. After generating a new ensemble θ_i^j according to prior distribution, the reweighted parameter ensemble is calculated using importance sampling procedure. Finally, the posterior distribution for μ and Σ can be evaluated according following conditional distribution approach:

$$\begin{aligned} \mu|\theta_i, \Sigma_{i-1} &\sim \mathcal{N}(\mu_i, W_i), && (5.5)\\ \Sigma|\theta_i, \mu_{i-1} &\sim iWish(\Sigma_i, n_i), && (5.6)\\ & && (5.7) \end{aligned}$$

where

$$\begin{aligned} W_i &= (W_{i-1}^{-1} + \Sigma_{i-1}^{-1})^{-1},\\ \mu_i &= W_i(W_{i-1}^{-1}\mu_{i-1} + \Sigma_{i-1}^{-1}\theta_i) \qquad (5.8)\\ n_i &= n_{i-1} + 1,\\ \Sigma_i &= (n_{i-1}\Sigma_{i-1} + (\theta_i - \mu_i)(\theta_i - \mu_i)')/n_i. \end{aligned}$$

These formulation gives an update formulas for unknown hyperparameters when moving from time window i to time window $i+1$. Also should be mentioned that these formulas can be calculated for every member of ensemble θ_i^j to produce $\mu_i^j, \Sigma_i^j, W_i^j$ and n_i^j. Further, next time window parameters can be obtained by calculating the mean values of these parameters.

Finally, the algorithm can be illustrated by the following scheme [8]:

1. Initialization of hyperparameters μ_0, Σ_0, W_0 and n_0. In this case the distribution $\mathcal{N}(\mu_0, \Sigma_0)$ corresponds to the prior and the proposal distribution for the first sample whereas W_0 and n_0 show the accuracy of proposed initial values for μ_0 and Σ_0;

2. For every time window i the sample of proposal values $\tilde{\theta}_i^j$ are generated from the multivariate Normal distribution:

$$\tilde{\theta}_i^j \sim \mathcal{N}(\mu_{i-1}, \Sigma_{i-1}), j = 1, \ldots, n_{ens}. \qquad (5.9)$$

3. Ensemble of prediction is generated according set of proposed parameters $\tilde{\theta}_i^j$;

4. The objective function is calculated for each ensemble and importance weights are calculated;

5. Resampled ensemble θ_i^j is generated out of $\tilde{\theta}_i^j$ taking into account importance weights;

6. Calculate the hyperparameters $\mu_i^j, \Sigma_i^j, W_i^j$ and n_i^j by applying values of just generated θ_i^j using Equation 5.8;

7. Assign hyperparameters values for the next time window taking the average of current values:

$$\mu_i = \sum_{j=1}^{n_{ens}} \mu_i^j / n_{ens}, W_i = \sum_{j=1}^{n_{ens}} W_i^j / n_{ens}, \Sigma_i = \sum_{j=1}^{n_{ens}} \Sigma_i^j / n_{ens}, n_i = \sum_{j=1}^{n_{ens}} n_i^j / n_{ens}.$$

8. Set the proposal distribution for the next time window as $\theta_{i+1} \sim \mathcal{N}(\mu_i, \Sigma_i)$. Repeat the procedure from the step 2.

The clarification of one step of EPPES algorithm is made by following Figure 5.1: Here the proposed new points $\tilde{\theta}_i^j$ are depicted by grey and black

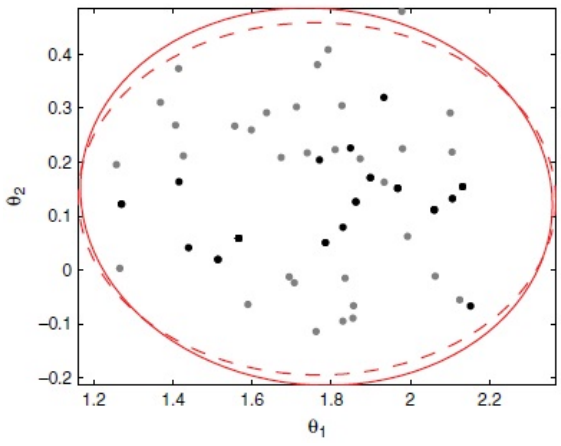

Figure 5.1: Illustration of the EPPES algorithm.

dots, resampled points θ_i^j by black dots, old prior is the solid line and updated prior is the dashed line.

Once EPPES algorithm have been briefly described, the simple example of Matlab implementation of such algorithm should be provided by the example. It should be pointed out that Matlab implementation already exists and can be found here: `http://helios.fmi.fi/~lainema/eppes/`.

5.2 EPPES linear case example

This example is taken from the provided Matlab implementation and demonstrates the ability of EPPES method to find the known true distribution in case of linear model. In this case it is assumed that the observation came from a hierarchical Gaussian model. At each iteration time window, the observations are generated from $y \sim \mathcal{N}(\theta, \Sigma_{obs})$. Moreover, the mean θ is random and satisfy Gaussian distribution $\theta \sim \mathcal{N}(\mu, \Sigma)$, where μ and Σ are unknown hyperparameters. The aim of the estimation is determine the values for that hyperparameters which can be done sequentially by EPPES method.

Let consider three-parameter problem and assume that the true values for hyperparameters are:

$$\mu_{true} = (1, 2, 3),$$

$$\Sigma_{true} = \begin{pmatrix} 0.3418 & 0.2102 & 0.2480 \\ 0.2102 & 0.3607 & 0.2291 \\ 0.2480 & 0.2291 & 0.3229 \end{pmatrix}.$$

Moreover, Σ_{obs} which is covariance matrix for distribution of θ is set to be 0.5. The solution obtained by EPPES approach can be illustrated by the Figure 5.2 and Figure 5.3. The first plot illustrates convergence of

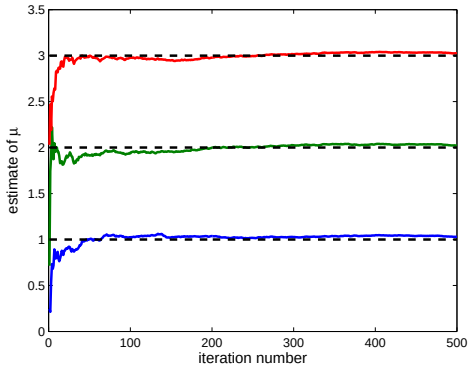

Figure 5.2: Convergence of hyperparameter μ.

the hyperparameter μ. It is possible to conclude that in this case solution obtained by EPPES algorithm starts to approache the true values of the

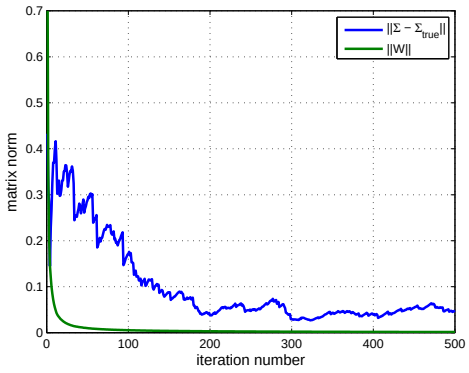

Figure 5.3: Convergence of hyperparameter Σ and W.

problem quickly even with specified high level of standard deviation. At the same time according to the second plot estimated Σ tends to correct one slowly and it takes 200-300 iterations to obtain an appropriate result. Also it is important to present numeric values for estimated Σ, namely

$$\Sigma = \begin{pmatrix} 0.3158 & 0.1952 & 0.2163 \\ 0.1952 & 0.3221 & 0.1932 \\ 0.2163 & 0.1932 & 0.2976 \end{pmatrix}.$$

How close the estimated Σ to Σ_{true} is can be illustrated by plotting this covariance matrices componentwise:

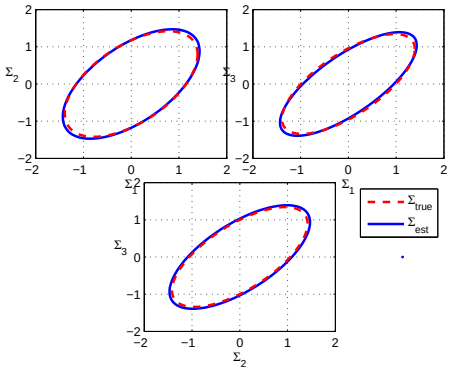

Figure 5.4: Convergence of hyperparameter Σ to Σ_{true} componentwise.

It can be seen that EPPES solution provide an suitable result on this test example and now it should be applied to the Lorenz system.

5.3 Comparison of EPPES and DE approaches to solution of Lorenz system

This section is devoted to comparison of EPPES and DE approach to the parameter estimation problem of chaotic dynamics of Lorenz system. For the purpose to conduct estimation procedure it is necessary to provide to EPPES solver Matlab routines devoted to Lorenz system. Since the required routines already exist from estimation procedure by DE approach, they can be treated here as well.

According to idea of EPPES initial values of hyperparameters should be specified. In general case close approximation for truth parameters is not known, hence hyperparameters W and n which controls accuracy of proposed initial values should be specified by huge numbers matrix and value 1 respectively. Also, noise in data is specified to be equal 0.01, initial values perturbation range is set to be equal 0.1, range for analysis is 5, and size of ensemble is 100 members. Evolution of estimation procedure can be illustrated by the following figure:

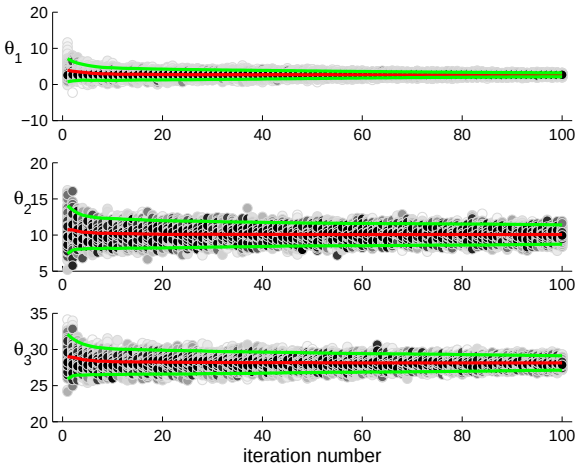

Figure 5.5: Solution of parameter estimation problem of Lorenz system obtained by EPPES approach.

The background idea of EPPES is that the estimated parameter is

stochastic, e.g. no 'true' value exists. However, in the test runs performed here it is assumed a fixed true value and compare the convergence properties of DE and EPPES. Thus, it should be provided the solution by the problem with the same parameters obtained by DE approach:

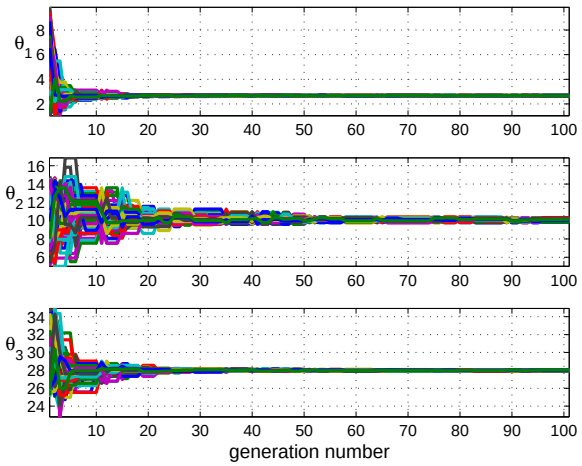

Figure 5.6: Solution of parameter estimation problem of Lorenz system obtained by DE approach.

Several runs with different initial values was conducted and EPPES solution demonstrates more sensitivity to initial values of model in general and particulary to initial values for parameters of algorithm. At the same time DE approach shows more stable behavior with respect to specified parameters of the model and converge to truth closure parameters even with default parameters of algorithm. This facts make possible to claim that for the case of parameter estimation task for low-dimensional chaotic system DE approach is an appropriate alternative to the existed EPPES approach, but applicability and performance of DE algorithm to the high-dimensional problem and advantages over EPPES algorithm should be tested additionally, due to the fact that EPPES algorithm was invented and tested for estimation more complex problem, for example for Lorenz-95 system.

6 Conclusion

This Master Thesis covers the main mathematical aspects regarding to DE algorithm. The classical mathematical description consisted of the basic population structure and main parts of DE algorithm such as initialization, mutation, crossover, and selection are introduced. Also possible stopping criteria and cases where they are mostly suitable are defined. Moreover, possible improvement in the classical DE structure is discussed and Matlab implementation which takes into account all these features is developed in this paper and additionally tested on test examples. Influence of tuning parameters, especially population size, is investigated as well as influence of noise level involved into the measured data. The key point of chaotic dynamic systems estimation is discussed and demonstrated on Lorenz system behavior. Concrete techniques devoted to enhancement of convergence of this specific problem are proposed and solutions are illustrated by plots of population evolution. Overview of EPPES method is also included and explained by the example. Finally, the comparison between DE and EPPES methods was conducted and result of this comparison makes possible to conclude that DE algorithm has several advantages as stability, less sensitivity on initial values and noise involved into the system, which makes it suitable alternative for EPPES approach at least for the low-dimensional problem. Nevertheless, applicability of DE algorithm in more complex systems related to modern NWP model remains possible direction of the future research.

References

[1] K.V. Price, R.M. Storn and J.A. Lampinen, "Differential Evolution: Practical Approach to Global Optimization", 2005 ed. Berlin, Heidelberg, New York: Springer, 2005, pp. 1-130.

[2] A. Qing, "Differential Evolution: Fundamentals and Applications in Electrical Engineering", Singapore: Wiley-IEEE Press, 2009, pp. 18-24, 41, 61-64.

[3] U.K. Chakraborty, "Advances in Differential Evolution", Verlag, Berlin, Heidelberg: Springer, 2008, pp. 1-15.

[4] V. Feoktiistov, "Differential Evolution: In Search of Solutions", New York: Springer, 2006, pp. 41-65.

[5] H. Haario, "Statistical Analysis in Modelling: MCMC methods", course presentation, Lappeenranta University of Technology, Finland, 2011. Available: `http://www.mafy.lut.fi/study/sam/statmode09.pdf`

[6] C. Sparrow, "The Lorenz Equations: Bifurcations, Chaos, and Strange Attractors" in Applied Mathematical Sciences, vol. 41, 1st ed., New York: Springer, 1989, pp. 1-3.

[7] H. Järvinen, M. Laine, A. Solonen and H. Haario "Ensemble prediction and parameter estimation system: The Concept", *Quarterly Journal of the Royal Meteorological Society*, 2011.

[8] M. Laine, A. Solonen, H. Haario and H. Järvinen, "Ensemble prediction and parameter estimation system: The Method", *Quarterly Journal of the Royal Meteorological Society*, 2011.

[9] M. Laine, Ensemble prediction and parameter estimation system [Online]. Available: `http://helios.fmi.fi/~lainema/eppes/`

A Matlab listings

A.1 Toy case. Classical DE

```
%% run.m  - main file
%% Test initialization
clear all, close all, format compact, format long, ...
    rng('shuffle'),clc;
%% Generating data
x = [0:0.1:10];    % input
% in-line model function
% applicable for theta (3 by n)-matrix:
f = @(theta,x)exp(theta*[ones(size(x));x;x.^2]);
theta_true = [-6 3 -0.3];   % true parameters
y_true = f(theta_true,x);   % response of input with true ...
    parameters
sigma = 0.1;                % std for noise
y   = y_true+sigma*randn(size(y_true)); % "measured data" ...
    (with noise);
data.xdata = x;             % input
data.ydata = y;             % response
data.fun = f;               % model function
global SIGMA;
SIGMA = 0.0;                % stochasticity
%% Initialization of Compulsory parameters of DE
paramsDE.init_bounds = [-10 -10 -3; 10 10 3]'; % bounds for ...
    initialization of DE
paramsDE.func = @fun_ss;                       % cost function
paramsDE.SoP = 35;                             % size of ...
    population
%% DE run
Gmean = 0; % number of generations
K = 100;   % number of simulations
Fails = 0; % number of fails in estimation
M = 10;    % estimate for mean cost function
for i = 1:K
    [pop,G,evolution] = de(data,paramsDE);
    Gmean = Gmean + G;
    Fails = Fails + (mean(pop(:,4))>=M);
    i
end
Gmean/K % mean number of generations
Fails/K % mean number of fails
%% plotting
D = size(pop,2)-1;
[xll,fval,exitflag,output] = fminsearch(@fun_ss,[0,0,0],[],data);
plot(x,y_true,'--k'); hold on;
plot(x,y,'.r');
plot(x,f(xll,x),'-.b');
plot(x,f(pop(:,1:D),x),'g');hold off;
```

```
axis tight;
grid on;
legend('true data','noisy ...
    data','FMINSEARCH-solution','DE-solution');
```

```
%% fun_ss.m - cost function
function res = fun_ss(theta,data)
    % (Length-of-data.xdata by SoP)-matrix of responses
    y = data.fun(theta,data.xdata)';
    % provide stochasticity
    global SIGMA;
    y = y + SIGMA*randn(size(y));
    % calculation of LSQ
    res = sum((repmat(data.ydata',1,size(y,2))-y).^2);
    res = res';
end
```

```
function [pop,G,evolution,fval,element] = de(data,paramsDE)
    %% default values
    if ~(isfield(paramsDE,'func') && ...
        isfield(paramsDE,'init_bounds'))
        display('Error!');
        pop = [];
        G = 0;
        evolution = {};
        return;
    end
    paramsDE.D = size(paramsDE.init_bounds,1);    % dimension ...
        of problem
    if ~(isfield(paramsDE,'SoP'))
        paramsDE.SoP = 10*paramsDE.D;             % size of ...
            population
    end
    if ~(isfield(paramsDE,'F'))
        paramsDE.F = 0.5;                         % scale ...
            factor in mutation
    end
    if ~(isfield(paramsDE,'CR'))
        paramsDE.CR = 0.9;                        % crossover ...
            probability
    end
    if ~(isfield(paramsDE,'Tol'))
        % tolerance for standard deviation of Hl previous ...
            generations'
        % objective values means:
        paramsDE.Tol = 1e-10;
    end
    if ~(isfield(paramsDE,'Hl'))
```

```
        paramsDE.Hl = 10;                % length of history ...
            of generations
    end
    if ~(isfield(paramsDE,'Gmax'))
        paramsDE.Gmax = 1e3;             % maximum number of ...
            generations
    end
    %% initialization of variables
    D = paramsDE.D;
    Tol = paramsDE.Tol;
    Gmax = paramsDE.Gmax;
    history = 100*rand(1,paramsDE.Hl);
    G=1;
    pop = initialization(data,paramsDE);
    evolution = {};
    %% DE process
    while std(history)>=Tol && G<=Gmax
        mutant = mutation(pop,paramsDE);
        new = crossover(mutant,pop,data,paramsDE);
        pop = selection(new,pop,paramsDE);
        evolution{G} = pop;
        ind = pop(:,D+1)<Inf;
        history(mod(G,paramsDE.Hl)+1) =sum(pop(ind,D+1));
        G  = G+1;
    end
    [fval,element] = min(pop(:,D+1));
end
```

```
function pop = initialization(data,paramsDE)
    SoP = paramsDE.SoP;
    init_bounds = paramsDE.init_bounds;
    func = paramsDE.func;
    D = paramsDE.D;
    % generation of first population of parameters :
    pop = ones(SoP,1)*init_bounds(:,1)' + ...
         ones(SoP,1)*(init_bounds(:,2) - ...
             init_bounds(:,1))'.*rand(SoP,D);
    % calculation of cost function :
    tmp = func(pop,data);
    % pop = [<parameters>,cost]
    pop = [pop tmp];
end
```

```
function res = mutation(pop,paramsDE)
    SoP = paramsDE.SoP;
    F = paramsDE.F;
    D = paramsDE.D;
    cur_pop = pop(:,1:D);    % only parameters
    % chosing base  and difference vectors ...
```

```
    % differ from target vector:
    [tmp,ind] = sort(rand(SoP,SoP-1),2);
    ind = ind(:,1:3) + (ind(:,1:3) > [0:SoP-1]'*ones(1,3));
    % mutation itself
    res = cur_pop(ind(:,3),:) + F*(cur_pop(ind(:,1),:) - ...
        cur_pop(ind(:,2),:));
end
```

```
function res = crossover(new,pop,data,paramsDE)
    SoP = paramsDE.SoP;
    D = paramsDE.D;
    CR = paramsDE.CR;
    func = paramsDE.func;
    % generating of index vector of memebrs to be tested ...
    % for being inherited from previous generation ...
    % including protection from exact copies.
    temp = rand(SoP,D);
    ind = randi(D,SoP,1);
    temp(sub2ind([SoP,D],[1:SoP]',ind)) = 0;
    % testing
    ind = (temp > CR);
    % crossover itself: generating trial vector
    new(ind) = pop(ind);
    % calcilation of cost function of trial vector
    tmp = func(new,data);
    res = [new tmp];
end
```

```
function res = selection(new,pop,paramsDE)
    D = paramsDE.D;
    % selection: comparison of objective function values:
    ind = (new(:,D+1) > pop(:,D+1));
    % construction of next generation:
    new(ind,:) = pop(ind,:);
    res = new;
end
```

A.2 Toy case. advanced DE

```
%% Test initialization
clear all, close all, format compact, format long, ...
    rng('shuffle'),clc;

%% Generating data
x = [0:0.1:10];    % input
% in-line model function
% applicable for theta (3 by n)-matrix:
f = @(theta,x)exp(theta*[ones(size(x));x;x.^2]);
theta_true = [-6 3 -0.3];   % true parameters
y_true = f(theta_true,x);   % response of input with true ...
    parameters
sigma = 0.1;                % std for measurement noise
global SIGMA;               % std for stochastic noise (used ...
    in fun_ss)
SIGMA = 0.1;                % if 0.0 then deterministic case
y    = y_true+sigma*randn(size(y_true)); % "measured data" ...
    (with noise);
data.xdata = x;             % input
data.ydata = y;             % response
data.fun = f;               % model function
%% Initialization of Compulsory parameters of DE
paramsDE.init_bounds = [-10 -10 -3; 10 10 3]'; % bounds for ...
    initialization of DE
paramsDE.func = @fun_ss;                       % cost function
paramsDE.SoP = 80;                             % size of ...
    population
paramsDE.Tol = 1e-5;
mutation_types = {'Cls','Bst','CtB'};
F_types = {'Cls','Dthr1','Dthr2','Jttr','Mix'};
JP = [0:0.1:0.3];
% methods = {'Bst'};
% mutation_types = {'Mix'};
% JP = 0.3;
name = ['Stoch-' datestr(now,1)];
name = [name '-' num2str(SIGMA) '-' num2str(paramsDE.SoP) '.log'];
for I = 1:length(mutation_types)
    for J = 1:length(F_types)
        for T = 1:length(JP)
            paramsDE.mutation_type = mutation_types{I};
            paramsDE.F_type = F_types{J};
            paramsDE.Jp = JP(T);
            %% DE run
            Gmean = 0;
            Fails = 0;
            K = 10;     % number of simulations
            fprintf('\n');
            disp(['DDE\' paramsDE.mutation_type '\' ...
                paramsDE.F_type '\' num2str(paramsDE.Jp) ':']);
```

```
            disp(['Mean populations(parameters, ss-value and ...
                number of generations):']);
            for i = 1:K
                [pop,G,evolution] = de(data,paramsDE);
                Gmean = Gmean + G;
                Fails = Fails + (mean(pop(:,4))>50);
                fprintf('%2d)%12.5f %12.5f %12.5f %12.5f %4d ...
                    \n',i,mean(pop),G);
            end
            disp(['Mean number of generations is ' ...
                num2str(Gmean/K)]);
            disp(['Percentage of fails is ' ...
                num2str(Fails/K*100) '%']);
            % Output for LaTeX.
            log = fopen(name,'at+');
            fprintf(log,'\\hline \n');
            fprintf(log,'%s & %5s & %5.3f & %6.2f & %5.3f \\\\ ...
                \n',paramsDE.mutation_type,paramsDE.F_type,...
                paramsDE.Jp,Gmean/K,Fails/K);
            fclose(log);
        end
    end
end
```

```
function [pop,G,evolution] = de(data,paramsDE)
    %% default values
    if ~(isfield(paramsDE,'func') && ...
        isfield(paramsDE,'init_bounds'))
        display('Error!');
        pop = [];
        G = 0;
        evolution = {};
        return;
    end
    paramsDE.D = size(paramsDE.init_bounds,1);    % dimension ...
        of problem
    if ~(isfield(paramsDE,'SoP'))
        paramsDE.SoP = 10*paramsDE.D;             % size of ...
            population
    end
    if ~(isfield(paramsDE,'F'))
        paramsDE.F = 0.5;                         % scale ...
            factor in mutation
    end
    if ~(isfield(paramsDE,'CR'))
        paramsDE.CR = 0.9;                        % crossover ...
            probability
    end
    if ~(isfield(paramsDE,'Tol'))
        % tolerance for standard deviation of Hl previous ...
            generations'
```

```
        % objective values means:
        paramsDE.Tol = 1e-10;
    end
    if ~(isfield(paramsDE,'Hl'))
        paramsDE.Hl = 10;                % length of history ...
            of generations
    end
    if ~(isfield(paramsDE,'Gmax'))
        paramsDE.Gmax = 1e3;             % maximum number of ...
            generations
    end
    if ~(isfield(paramsDE,'mutation_type'))
        paramsDE.mutation_type = 'Cls';  % {'Cls','Bst','CtB'}
    end
    if ~(isfield(paramsDE,'crossover_type'))
        paramsDE.crossover_type = 'Cls'; % ...
            {'Cls','Dthr1','Dthr2','Jttr','Mix'}
    end
    if ~(isfield(paramsDE,'Jp'))
        paramsDE.Jp = 0.0;               % generation jump ...
            probability
    end
    if ~(isfield(paramsDE,'F_range'))
        paramsDE.F_range = [0.45,0.55];     % [F_l, F_h]
    end

    %% initialization of variables
    D = paramsDE.D;
    Tol = paramsDE.Tol;
    Gmax = paramsDE.Gmax;
    history = 100*rand(1,paramsDE.Hl);
    G=1;
    pop = initialization(data,paramsDE);
    evolution = {}; % evolution of population
    %% DE process
    while std(history)>=Tol && G<=Gmax
        pop = dde(pop,data,paramsDE);
        evolution{G} = pop;
        ind = pop(:,D+1)<Inf;
        history(mod(G,paramsDE.Hl)+1) =sum(pop(ind,D+1));
        G  = G+1;
    end
end
```

```
function pop = initialization(data,paramsDE)
    SoP = paramsDE.SoP;
    init_bounds = paramsDE.init_bounds;
    func = paramsDE.func;
    D = paramsDE.D;
    % generation of first population of parameters :
    pop = ones(SoP,1)*init_bounds(:,1)' + ...
```

```
        ones(SoP,1)*(init_bounds(:,2) - ...
            init_bounds(:,1))'.*rand(SoP,D);
    % calculation of cost function :
    tmp = func(pop,data);
    % pop = [<parameters>,cost]
    pop = [pop tmp];
end
```

```
function res = dde(pop,data,paramsDE)
    %% generation jump
    D = paramsDE.D;
    Jp = paramsDE.Jp;
    SoP = paramsDE.SoP;
    func = paramsDE.func;
    cur_pop = pop(:,1:D+1); % parameters + cost
    if rand(1)<=Jp
        mins = [];          % minimum values of each parameter ...
            vector
        maxs = [];          % maximum values of each parameter ...
            vector
        for i=1:D
            tmp1=min(cur_pop(:,i));
            tmp2=max(cur_pop(:,i));
            mins = [mins,tmp1];
            maxs = [maxs,tmp2];
        end
        op_pop = repmat(mins,SoP,1) + repmat(maxs,SoP,1) - ...
        cur_pop(:,1:D); % "opposite population"
        tmp = func(op_pop,data);
        op_pop = [op_pop tmp];
        cur_pop = [cur_pop;op_pop];
        [TMP,IX] = sort(cur_pop);
        ind = IX(1:SoP,D+1);
        cur_pop = cur_pop(ind',:);
        res = cur_pop;
        return;
    end
    %% Params
    CR = paramsDE.CR;
    mutation_type = paramsDE.mutation_type;
    F_type = paramsDE.F_type;
    %% mutation
    [tmptmp,ind] = sort(rand(SoP,SoP-1),2);
    indMutation = ind(:,1:3) + (ind(:,1:3) > ...
        [0:SoP-1]'*ones(1,3));
    %% crossover and selection
    temp = rand(SoP,D);
    ind = randi(D,SoP,1);
    temp(sub2ind([SoP,D],[1:SoP]',ind)) = 0;
    indCrossover = (temp > CR);
    F_l = paramsDE.F_range(1);
```

```
    F_h = paramsDE.F_range(2);
    F_g = F_l + rand(1)*(F_h-F_l);
    switch F_type
        case 'Cls'
            F = paramsDE.F;
        case 'Dthr1'
            F = F_g;
        case 'Dthr2'
            F = F_l + rand(1)*(F_h-F_l);
        case 'Jttr'
            sigma = 0.001;
            F = paramsDE.F*(1+sigma*(rand(1,D)-0.5));
        case 'Mix'
            sigma = 0.001;
            F = F_g*(1+sigma*(rand(1,D)-0.5));
    end
    for i = 1:SoP
        switch mutation_type
            case 'Cls'
                new = ...
                    cur_pop(indMutation(i,3),1:D) + ...
                    F.*(cur_pop(indMutation(i,1),1:D) - ...
                        cur_pop(indMutation(i,2),1:D));
            case 'Bst'
                [best_tmp,best_ind] = min(cur_pop(:,D+1));
                if best_ind==indMutation(i,1)
                    indMutation(i,1) = indMutation(i,3);

                end
                if best_ind==indMutation(i,2)
                    indMutation(i,2) = indMutation(i,3);

                end
                new = ...
                    cur_pop(best_ind,1:D) + ...
                    F.*(cur_pop(indMutation(i,1),1:D) - ...
                        cur_pop(indMutation(i,2),1:D));
            case 'CtB'
                [best_tmp,best_ind] = min(cur_pop(:,D+1));
                new = ...
                    cur_pop(indMutation(i,3),1:D) + ...
                    F.*(cur_pop(best_ind,1:D)-cur_pop(indMutation(i,3),1:D)) ...
                        + ...
                    F.*(cur_pop(indMutation(i,1),1:D) - ...
                        cur_pop(indMutation(i,2),1:D));
        end
        new(indCrossover(i,:)) = cur_pop(i,indCrossover(i,:));
        tmp = func(new,data);
        if tmp<cur_pop(i,D+1)
            cur_pop(i,:) = [new tmp];
        end
    end
    res = cur_pop;
end
```

A.3 Lorenz system estimation

```
%% Parameter estimation with DE
clear all, close all, format compact, format long, clc;
% orginal parameters: [2.67,10,28];
addpath('DE','Model');
%% 'Original' data
load 'lorenz_0.4_0.0025_500_struct.mat';
truth = LORENZ.truth;
aint = LORENZ.aint;
dt = LORENZ.dt;
sigma = 0.01;
range = 5;
data.aint = aint;
data.dt = dt;
nSim = size(truth,1)-range-1;
nSim = 100; % for simplicity
%% Parameters of DE
mutation_types = {'Cls','Bst','CtB'};
crossover_types = {'Cls','Dthr1','Dthr2','Jttr','Mix'};
JP = 0:0.1:0.3;
paramsDE.init_bounds = [1 5 25; 10 15 35]';   % bounds for ...
    initialization of DE
paramsDE.SoP = 100;                            % size of ...
    population
paramsDE.F = 0.5;                              % scale factor ...
    in mutation
paramsDE.CR = 0.9;                             % crossover ...
    probability
paramsDE.init_std = 0.1;                       % std of ...
    initial values
paramsDE.func = @lorenz_ss;                    % cost function
paramsDE.D = size(paramsDE.init_bounds,1);     % demension of ...
    problem
paramsDE.mutation_type = mutation_types{2};    % mutation type
paramsDE.crossover_type = crossover_types{5};  % crossover type
paramsDE.Jp = JP(4);                           % generation ...
    jump probability
paramsDE.F_range = [0.45,0.55];                % [F_l, F_h]
paramsDE.update_cost = 1;                      % how long ...
    generations update cost
%% Parameters of Model
maxIter = 1; %additional optimization if maxIter > 1
% history of parameter evolution
history.beta = [];
history.sigma = [];
history.rho = [];
% name of log-file
method = ['DDE_DIPLOM_' paramsDE.mutation_type '_' ...
    paramsDE.crossover_type '_' num2str(paramsDE.Jp)];
% add noise to exact data
```

```
truth = truth + sigma*randn(size(truth));
for i = 1:nSim
    a = (i-1)*range;
    b = (i)*range;
    time = a*aint:dt:b*aint;
    data.truth = truth(a+2:b+1,:);
    analysis = truth(a+1,:);
    SoP = paramsDE.SoP;
    init_std = paramsDE.init_std;
    INIT = repmat(analysis,SoP,1);
    init = INIT + init_std*randn(size(INIT));
    data.init = init;
    data.xdata = time;
    if i==1
        % initialization at the 1st step
        pop = initialization(data,paramsDE);    % current ...
            population
        disp('Initial population:');
        history.beta = [history.beta pop(:,1)];
        history.sigma = [history.sigma pop(:,2)];
        history.rho = [history.rho pop(:,3)];
        l = 15+15+15+20+4+6;
        fprintf('%15s | %15s | %15s | %20s \n','Theta1   ...
            ','Theta2   ','Theta3   ','SS-values   ');
        fprintf('%s\n',repmat(['_'],1,l));
        fprintf('%15.10f | %15.10f | %15.10f | %20.10f \n',pop');
        fprintf('%s\n',repmat(['_'],1,l));
        disp(['Mean: ']);
        fprintf('%15.10f | %15.10f | %15.10f | %20.10f ...
            \n',mean(pop));
        fprintf('%s\n',repmat(['_'],1,l));
        disp(['Std: ']);
        fprintf('%15.10f | %15.10f | %15.10f | %20.10f ...
            \n',std(pop));
    else
        pop = recalc_cost(pop,data,paramsDE,i-1);   % ...
            recalculation
        maxIter = 1;
    end
    for j = 1:maxIter;
        fprintf('\n \n');
        disp(['Current generation is ' num2str(i)]);
        disp(['Time is [' num2str(a*aint) ',' num2str(b*aint) ...
            ']']);
        pop = dde(pop,data,paramsDE);
        l = 15+15+15+20+4+6;
        disp('Current population:');
        fprintf('%15s | %15s | %15s | %20s \n','Theta1   ...
            ','Theta2   ','Theta3   ','SS-values   ');
        fprintf('%s\n',repmat(['_'],1,l));
        fprintf('%15.10f | %15.10f | %15.10f | %20.10f \n',pop');
        fprintf('%s\n',repmat(['_'],1,l));
        disp(['Mean: ']);
```

```
        fprintf('%15.10f | %15.10f | %15.10f | %20.10f ...
            \n',mean(pop));
        fprintf('%s\n',repmat(['_'],1,1));
        disp(['Std: ']);
        fprintf('%15.10f | %15.10f | %15.10f | %20.10f ...
            \n',std(pop));
        history.beta = [history.beta pop(:,1)];
        history.sigma = [history.sigma pop(:,2)];
        history.rho = [history.rho pop(:,3)];
    end
    save(['history_for_comparison',method,'.mat'],'history');
end
%% plotting
% load history_DDE_new_Bst_Mix_0.3.mat
figure(1);
subplot(3,1,1);
plot(history.beta','LineWidth',2);
% h = title('Evolution of populations');
% set(h,'fontsize',14);
axis tight;
grid on;
h = ylabel('\theta_1  ','rot',0);
set(h,'fontsize',12);
subplot(3,1,2);
plot(history.sigma','LineWidth',2);
axis tight;
grid on;
h = ylabel('\theta_2  ','rot',0);
set(h,'fontsize',12);
subplot(3,1,3);
plot(history.rho','LineWidth',2);
axis tight;
grid on;
h = xlabel('generation number');
set(h,'fontsize',12);
h = ylabel('\theta_3  ','rot',0);
set(h,'fontsize',12);
```

```
function y = lorenz_m(data,theta,y0)
    time    = data.xdata;
    % parameters of solver:
    opt     = odeset('RelTol',1e-11,'AbsTol',1e-11);
    [t,y]   = ode113(@lorenzeq,time,y0,opt,theta);
end
```

```
function [res,Y] = lorenz_ss(theta,data,init)
    res = [];
    aint = data.aint;   % analysis interval
    dt = data.dt;       % integration interval
```

```
    truth = data.truth; % truth data
    for i = 1:size(theta,1)
        y0  = init(i,:);% initial value
        Y(:,:,i)   = ...
            lorenz_m(data,[theta(i,1),theta(i,2),theta(i,3)],y0);
    end
    % solution in the point where truth values are calculated
    response = Y(aint/dt:aint/dt:end,:,:);
    % cost function calculated in LSQ-sense
    nens = size(response,3);
    ss    = (response-repmat(truth,[1 1 nens])).^2;
    for i = 1:nens
        res(i) = sum(sum(ss(:,:,i)));
    end
    res = res(:);
end
```

```
function sdot = lorenzeq(t,s,params)
    % parameters
    beta  = params(1);
    sigma = params(2);
    rho   = params(3);
    % states values
    x = s(1);
    y = s(2);
    z = s(3);
    % equation
    dx = sigma*(y-x);
    dy = x*(rho - z) - y;
    dz = x*y-beta*z;
    % differential
    sdot = [dx;dy;dz];
end
```

```
function res = dde(pop,data,paramsDE)
    %% generation jump
    D = paramsDE.D;
    Jp = paramsDE.Jp;
    SoP = paramsDE.SoP;
    func = paramsDE.func;
    init = data.init;
    cur_pop = pop(:,1:D+1); % parameters + cost
    if rand(1)<=Jp
        mins = [];          % minimum values of each parameter ...
            vector
        maxs = [];          % maximum values of each parameter ...
            vector
        for i=1:D
            tmp1=min(cur_pop(:,i));
```

```
            tmp2=max(cur_pop(:,i));
            mins = [mins,tmp1];
            maxs = [maxs,tmp2];
        end
        op_pop = repmat(mins,SoP,1) + repmat(maxs,SoP,1) - ...
            cur_pop(:,1:D); % "oposite population"
        tmp = func(op_pop,data,init);
        op_pop = [op_pop tmp];
        cur_pop = [cur_pop;op_pop];
        [TMP,IX] = sort(cur_pop);
        ind = IX(1:SoP,D+1);
        cur_pop = cur_pop(ind',:);
        res = cur_pop(randperm(SoP),:);
        return;
    end
    %% Params
    CR = paramsDE.CR;
    mutation_type = paramsDE.mutation_type;
    crossover_type = paramsDE.crossover_type;


    %% mutation
    [tmptmp,ind] = sort(rand(SoP,SoP-1),2);
    indMutation = ind(:,1:3) + (ind(:,1:3) > ...
        [0:SoP-1]'*ones(1,3));

    %% crossover and selection
    temp = rand(SoP,D);
    ind = randi(D,SoP,1);
    temp(sub2ind([SoP,D],[1:SoP]',ind)) = 0;
    indCrossover = (temp > CR);
    F_l = paramsDE.F_range(1);
    F_h = paramsDE.F_range(2);
    F_g = F_l + rand(1)*(F_h-F_l);
    for i = 1:SoP
        switch crossover_type
            case 'Cls'
                F = paramsDE.F;
            case 'Dthr1'
                F = F_g;
            case 'Dthr2'
                F = F_l + rand(1)*(F_h-F_l);
            case 'Jttr'
                sigma = 0.001;
                F = paramsDE.F*(1+sigma*(rand(1,D)-0.5));
            case 'Mix'
                sigma = 0.001;
                F = F_g*(1+sigma*(rand(1,D)-0.5));
        end
        switch mutation_type
            case 'Cls'
                new = ...
                    cur_pop(indMutation(i,3),1:D) + ...
```

```
                    F.*(cur_pop(indMutation(i,1),1:D) - ...
                        cur_pop(indMutation(i,2),1:D));
            case 'Bst'
                [best_tmp,best_ind] = min(cur_pop(:,D+1));
                if best_ind==indMutation(i,1)
                    indMutation(i,1) = indMutation(i,3);

                end
                if best_ind==indMutation(i,2)
                    indMutation(i,2) = indMutation(i,3);

                end
                new = ...
                    cur_pop(best_ind,1:D) + ...
                    F.*(cur_pop(indMutation(i,1),1:D) - ...
                    cur_pop(indMutation(i,2),1:D));
            case 'CtB'
                [best_tmp,best_ind] = min(cur_pop(:,D+1));
                new = ...
                    cur_pop(indMutation(i,3),1:D) + ...
                    F.*(cur_pop(best_ind,1:D)-...
                    cur_pop(indMutation(i,3),1:D)) + ...
                    F.*(cur_pop(indMutation(i,1),1:D) - ...
                    cur_pop(indMutation(i,2),1:D));
        end
        new(indCrossover(i,:)) = cur_pop(i,indCrossover(i,:));
        tmp = func(new,data,init(i,:));
        if tmp<cur_pop(i,D+1) && sum(new<=0)==0 %% parametrs>0
            cur_pop(i,:) = [new tmp];
        end
    end
    res = cur_pop;

end
```

```
function pop = initialization(data,paramsDE)
    %   SoP - size of population (number)
    %   init_bounds - bounds for parameters (matrix)
    %   func - cost function
    %   data - additional data
    %   init_std - std of initial values
    SoP = paramsDE.SoP;
    init_bounds = paramsDE.init_bounds;
    func = paramsDE.func;
    D = paramsDE.D;         % demension of problem
    init = data.init;
    % generation of first population of parameters :
    pop = ones(SoP,1)*init_bounds(:,1)' + ...
         ones(SoP,1)*(init_bounds(:,2) - ...
             init_bounds(:,1))'.*rand(SoP,D);
    % cost function
```

```
    tmp = func(pop,data,init);
    % pop = [<parameters>,ss,init,ss_end]
    pop = [pop tmp];

end
```

```
function res = recalc_cost(pop,data,paramsDE,step)
    if step<=paramsDE.update_cost  % length of adapting
        func = paramsDE.func;
        D = paramsDE.D;
        init = data.init;
        tmp = func(pop,data,init);
        cur = pop(:,D+1) + (tmp - pop(:,D+1))./exp(sqrt(step));
        res = [pop(:,1:D) cur];
        l = 15+15+15+20+4+6;
        fprintf('%s\n',repmat(['_'],1,l));
        disp(['Recalculated mean: ']);
        fprintf('%15.10f | %15.10f | %15.10f | %20.10f ...
            \n',mean(res));
    else
        res = pop;
    end
end
```

B Tables

B.1 Deterministic case

Noise level $\sigma = 0.2$:

Mutation type	F type	Jp	$mean(G_{max})$
Cls	Cls	0.000	122.57
Cls	Cls	0.100	134.09
Cls	Cls	0.200	133.47
Cls	Cls	0.300	137.97
Cls	Dthr1	0.000	122.65
Cls	Dthr1	0.100	126.91
Cls	Dthr1	0.200	131.60
Cls	Dthr1	0.300	138.19
Cls	Dthr2	0.000	121.88
Cls	Dthr2	0.100	128.93
Cls	Dthr2	0.200	132.95
Cls	Dthr2	0.300	138.19
Cls	Jttr	0.000	122.63
Cls	Jttr	0.100	127.93
Cls	Jttr	0.200	137.39
Cls	Jttr	0.300	137.61
Cls	Mix	0.000	122.95
Cls	Mix	0.100	128.16
Cls	Mix	0.200	132.89
Cls	Mix	0.300	139.44
Bst	Cls	0.000	50.53
Bst	Cls	0.100	54.42
Bst	Cls	0.200	60.50
Bst	Cls	0.300	65.95
Bst	Dthr1	0.000	50.67
Bst	Dthr1	0.100	55.00
Bst	Dthr1	0.200	60.37
Bst	Dthr1	0.300	65.76

Table B.1: Different combinations of proposed methods.

Mutation type	F type	Jp	$mean(G_{max})$
Bst	Dthr2	0.000	50.26
Bst	Dthr2	0.100	55.19
Bst	Dthr2	0.200	59.65
Bst	Dthr2	0.300	66.16
Bst	Jttr	0.000	50.62
Bst	Jttr	0.100	55.02
Bst	Jttr	0.200	60.54
Bst	Jttr	0.300	65.27
Bst	Mix	0.000	49.80
Bst	Mix	0.100	54.82
Bst	Mix	0.200	60.38
Bst	Mix	0.300	65.86
CtB	Cls	0.000	72.58
CtB	Cls	0.100	91.06
CtB	Cls	0.200	91.16
CtB	Cls	0.300	89.96
CtB	Dthr1	0.000	73.84
CtB	Dthr1	0.100	68.36
CtB	Dthr1	0.200	77.22
CtB	Dthr1	0.300	87.15
CtB	Dthr2	0.000	82.67
CtB	Dthr2	0.100	71.31
CtB	Dthr2	0.200	78.69
CtB	Dthr2	0.300	88.92
CtB	Jttr	0.000	68.49
CtB	Jttr	0.100	69.97
CtB	Jttr	0.200	103.32
CtB	Jttr	0.300	82.41
CtB	Mix	0.000	75.74
CtB	Mix	0.100	78.75
CtB	Mix	0.200	77.76
CtB	Mix	0.300	79.43

Table B.2: Different combinations of proposed methods. Continue.

B.2 Stochastic case

Stochastic noise level $\sigma = 0.4$:

Mutation type	F type	Jp	$mean(G_{max})$	Fails percentage, %
Cls	Cls	0.000	375.90	0.000
Cls	Cls	0.100	414.00	0.000
Cls	Cls	0.200	349.40	0.010
Cls	Cls	0.300	384.50	0.000
Cls	Dthr1	0.000	358.60	0.010
Cls	Dthr1	0.100	392.10	0.000
Cls	Dthr1	0.200	376.60	0.000
Cls	Dthr1	0.300	361.00	0.000
Cls	Dthr2	0.000	324.50	0.000
Cls	Dthr2	0.100	332.60	0.010
Cls	Dthr2	0.200	383.10	0.000
Cls	Dthr2	0.300	358.20	0.000
Cls	Jttr	0.000	412.20	0.000
Cls	Jttr	0.100	393.00	0.000
Cls	Jttr	0.200	412.90	0.000
Cls	Jttr	0.300	384.50	0.000
Cls	Mix	0.000	416.10	0.000
Cls	Mix	0.100	380.10	0.020
Cls	Mix	0.200	431.00	0.000
Cls	Mix	0.300	350.00	0.010
Bst	Cls	0.000	280.70	0.030
Bst	Cls	0.100	274.00	0.040
Bst	Cls	0.200	291.60	0.000
Bst	Cls	0.300	251.90	0.030
Bst	Dthr1	0.000	230.10	0.050
Bst	Dthr1	0.100	228.70	0.040
Bst	Dthr1	0.200	269.10	0.020
Bst	Dthr1	0.300	260.50	0.020

Table B.3: Different combinations of proposed methods.

Mutation type	F type	Jp	$mean(G_{max})$	Fails percentage, %
Bst	Dthr2	0.000	255.20	0.050
Bst	Dthr2	0.100	247.80	0.060
Bst	Dthr2	0.200	272.70	0.020
Bst	Dthr2	0.300	249.00	0.010
Bst	Jttr	0.000	263.20	0.050
Bst	Jttr	0.100	255.90	0.030
Bst	Jttr	0.200	276.50	0.020
Bst	Jttr	0.300	249.80	0.030
Bst	Mix	0.000	261.20	0.080
Bst	Mix	0.100	240.80	0.050
Bst	Mix	0.200	225.80	0.020
Bst	Mix	0.300	264.70	0.010
CtB	Cls	0.000	284.30	0.010
CtB	Cls	0.100	296.20	0.020
CtB	Cls	0.200	264.70	0.030
CtB	Cls	0.300	303.40	0.010
CtB	Dthr1	0.000	308.90	0.010
CtB	Dthr1	0.100	299.70	0.010
CtB	Dthr1	0.200	254.60	0.040
CtB	Dthr1	0.300	269.00	0.040
CtB	Dthr2	0.000	287.80	0.400
CtB	Dthr2	0.100	284.40	0.020
CtB	Dthr2	0.200	295.50	0.000
CtB	Dthr2	0.300	319.80	0.020
CtB	Jttr	0.000	290.40	0.050
CtB	Jttr	0.100	259.60	0.030
CtB	Jttr	0.200	274.20	0.020
CtB	Jttr	0.300	266.30	0.020
CtB	Mix	0.000	299.20	0.010
CtB	Mix	0.100	285.70	0.020
CtB	Mix	0.200	275.10	0.010
CtB	Mix	0.300	304.60	0.000

Table B.4: Different combinations of proposed methods. Continue.