Induction Motor Design by Use of Genetic Optimization Algorithms

prof. N. Zablodskiy [1)], prof. J. Lettl [2)], doc. V. Pliugin [3)], ing. K. Buhr [4)], stud. S. Khomitskiy [5)]

[1)] Donbas State Technical University/Automation of electro-technical systems, Alchevsk, Ukraine, *info@dmmi.edu.ua*
[2)] Czech Technical University in Prague/Faculty of Electrical Engineering, Prague, Czech republic, *lettl@fel.cvut.cz*
[3)] Donbas State Technical University/Automation of electro-technical systems, Alchevsk, Ukraine,
vlad.plyugin@gmail.com
[4)] Czech Technical University in Prague/Faculty of Electrical Engineering, Prague, Czech republic, *buhr@fel.cvut.cz*
[5)] Donbas State Technical University/Automation of electro-technical systems, Alchevsk, Ukraine,
stas.blitzkrieg@mail.ru

Abstract — **The problem of the automated calculation and optimal design of an induction motor is presented. The problem of optimization by use of genetic algorithms is set and solved. The analysis of the obtained results is executed.**

Keywords — induction motor, optimization, varied variables, genetic algorithm, criteria, limitations, effective variant, EvoJ library

I. INTRODUCTION

Neuron networks, being one of perspective trends of researches in the artificial intelligence area, as a result of watching processes going on in the nervous system of man were created. Approximately by the same way genetic algorithms were also «invented», but watched over the man nervous system, by the process of living organisms evolution.

Genetic algorithms - one of research trends in the artificial intelligence area, engaging in creation of the simplified evolution models of living organisms for the of optimization task decision [1].

A classic genetic algorithm (GA) consists of following steps:

1) initializing, or choice of initial chromosomes population;
2) an estimation of chromosomes adjustment in a population - calculation of adjusted function for every chromosome;
3) verification of algorithm stop condition;
4) chromosomes selection - choosing of chromosomes, participated in descendants for a next population creation;
5) application of genetic operators are mutations and crossing;
6) forming of new population;
7) choosing of the «best» chromosome.

The block-diagram of GA is represented in Fig. 1.

A simple GA generates an initial population by a random way. Working of the GA is an iteration process which proceeds until the generations set number or some another stop criterion will not be executed. On every generation, a proportional selection on adjusted, crossing and mutation is realized.

The simplest proportional selection is roulette. The wheel of roulette contains one sector for every member of population. The size of every sector is proportional to the corresponding size of adjusted function. At such selection, members of population with higher adjustment will be chosen with greater probability than individuals with subzero adjustment. The next step is using of crossing and mutation.

A previous population, obtained after a mutation, is overwritten and the cycle of one generation is completed. Subsequent generations i.e. selection, crossing and mutation obtained as a GA working result are processed in the same way.

II. THEORY AND PROGRAM REALIZATION

In the examined task a GA provides one criterion of optimality only, by virtue of the program realization of a calculation function minimum search [2 - 4].

The order of optimization will be different from considered Carthesian product (CP) in the previous article [5]:

1) setting the range of the varied variables;
2) setting limitations;
3) choosing the criterion of optimality;
4) calling the CA function for optimization and getting the optimal varied variables set;
5) calling the function of the induction motor (IM) automatic calculation for the found set.

We will consider an example of the IM optimization programmatic realization using GA on Java in NetBeans IDE [2]. For a decision the problem we will use free Java-library EvoJ (Evolution Java) [3]. A project EvoJ is designed as upgradable framework of Java classes for the decision of various optimization tasks by use of evolutional (genetic) algorithms. For the use of EvoJ a programmer must implement one simple interface, consisting of one method only. All other steps undertake EvoJ algorithm.
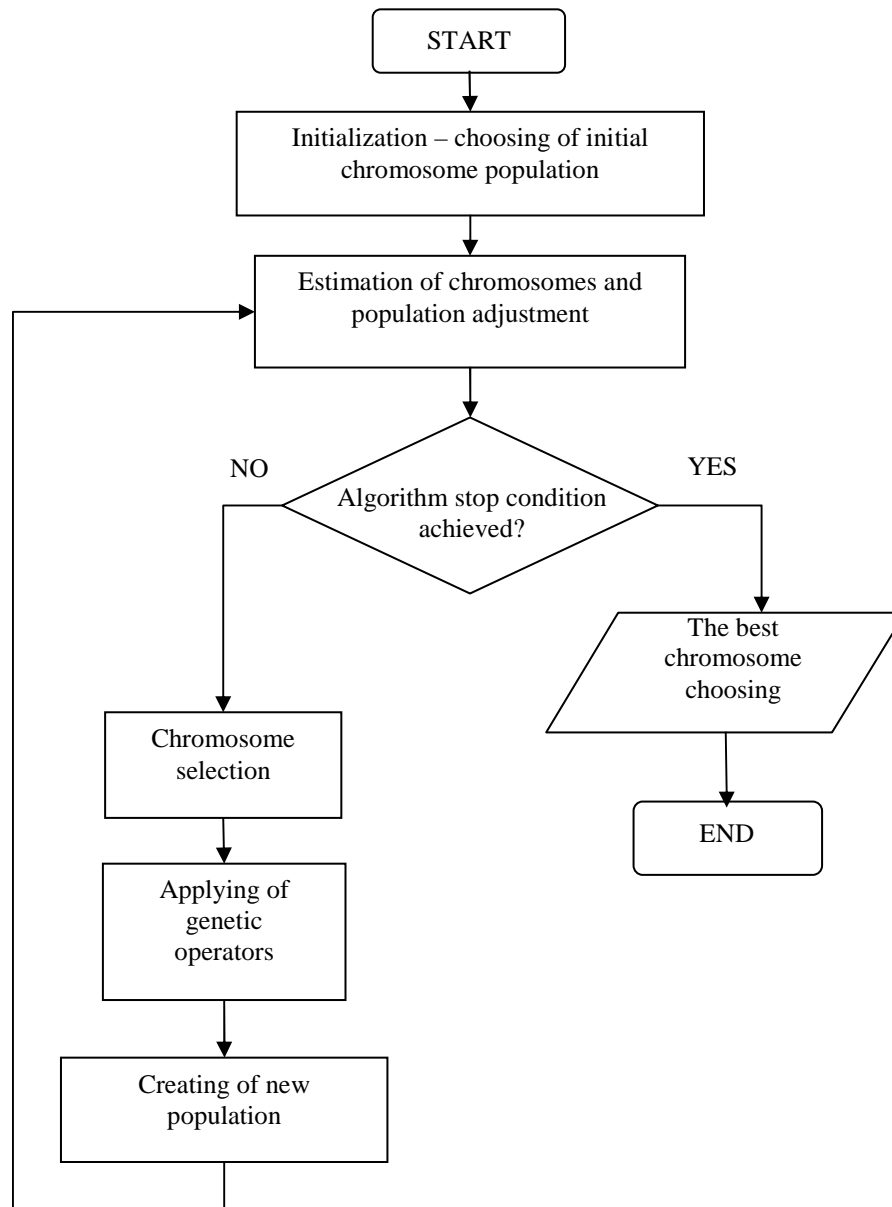
```
                    ┌──────────────┐
                    │    START     │
                    └──────┬───────┘
                           │
           ┌───────────────▼───────────────┐
           │ Initialization – choosing of  │
           │   initial chromosome          │
           │        population             │
           └───────────────┬───────────────┘
                           │
           ┌───────────────▼───────────────┐
    ┌─────►│ Estimation of chromosomes and │
    │      │    population adjustment       │
    │      └───────────────┬───────────────┘
    │                      │
    │          NO      ◇───▼───◇      YES
    │         ◄──── Algorithm stop ────►
    │              condition achieved?
    │                                    │
    │      ┌──────────────┐     ┌────────▼────────┐
    │      │  Chromosome  │     │    The best     │
    │      │  selection   │     │   chromosome    │
    │      └──────┬───────┘     │    choosing     │
    │             │             └────────┬────────┘
    │      ┌──────▼───────┐              │
    │      │  Applying of │      ┌───────▼──────┐
    │      │   genetic    │      │     END      │
    │      │  operators   │      └──────────────┘
    │      └──────┬───────┘
    │      ┌──────▼───────┐
    │      │ Creating of  │
    └──────┤     new      │
           │  population  │
           └──────────────┘
```

Fig. 1. Magnetization as a function of applied field**.**

In the example two varied variables will be considered: the internal diameter of the stator core and the length of the stator core.

We shall create Java-interface with the name "Solution" in which we set the range (minimum and maximum values) of the varied variables. The code of the Solution interface is down in the text.

EvoJ is able to change the variables without a setting of a range. However if it is needed to implement an own mutation strategy, one have to declare setters. In other case we shall not have a possibility to change the variable range.

Pay attention to annotation @of Range - it sets the range of values which a variable can accept. Variables of random values from the set range are initiated.

```java
package MotorClasses;
import net.sourceforge.evoj.core.annotation.MutationRange;
import net.sourceforge.evoj.core.annotation.Range;
public interface Solution {
    //Diameter of stator core
    String smin1 = "165";
    String smax1 = "205";
    //Length of stator core
    String smin2 = "115";
    String smax2 = "145";

    @Range(min = smin1, max = smax1)
    double getX(); //return the optimal diameter
      @Range(min = smin2, max = smax2)
    double getY(); //return the optimal length
}
```

However as a result of mutation they potentially can go out from the indicated range. It can be prevented using the parameter of strict=«true», that will not allow a variable to take on an impermissible value, even if we make an effort to propose them using setter-method.

Another case on which it is needed to pay attention here, is that all parameters of all limitations in EvoJ are strings. It allows both to specify the value of parameter directly and to specify the name of property instead of concrete value, to specify the value of limitation parameters at the compile-time.

Now we have an interface with variables and we shall write a fitness-function. Fitness-function in EvoJ is implemented as the following interface:

```
public interface SolutionRating <T> {
    Comparable calcRating(T solution);
}
```

Here parameter <T> is our interface with variables. The greater value return (according the contract *Comparable*) the more suitable solution is considered. Null can be returned and it is the smallest value of a fitness-function.

It is recommended to implement this interface as mediated using helper-classes. They undertake some service functions: elimination of the old decisions (if the maximal life term of decision is set), cashing of function value for decisions which were not sifted from in the previous GA iteration.

A Fitness-function for our case will look like the following (we create a new class with the name Rating):

```
package MotorClasses;
import
net.sourceforge.evoj.strategies.sorting.AbstractSimpleRating;
public class Rating extends AbstractSimpleRating <Solution> {
    static AMotor mot;//motor object
    static int krit;//index of optimality criterion
    static int iter_numb;//number of iterations
    //Constructor
    public void set_motor(AMotor mot, int krit){
        this.mot = mot;
        this.krit = krit;
        this.iter_numb = 0;
    }

 //reception of iterations number
    public int get_iter(){return this.iter_numb;};

    public static double calcFunction(Solution solution){
        iter_numb++;//increase of iterations count
        double x = solution.getX();//reception of new diameter
        double y = solution.getY();//reception of new length
        mot.stator.set_D(x/1000);//setting of new diameter
        mot.stator.set_ld(y/1000);//setting of new length
        double fn = mot.auto(krit);//automatic motor calculation

        return fn;//return a criterion of optimality
    }
```

```
    @Override
    public Comparable doCalcRating(Solution solution){
        double fn = calcFunction(solution);//call motor function
        boolean flag = mot.control();//control of limitations

        if (Double.isNaN(fn) | flag == false){
            return null;//sift-out false variant
        } else {
            return - fn;//return an effective variant
        }
    }
}//end of class
```

All code lines above are obviously enough. We simply take and count our function, using variables from the *Solution* interface:

```
double fn = calcFunction(solution);//call motor function
```

Because we search minimum and the contract of class supposes that the best decisions must have a greater rating, we return the value of function, multiplied by 1:

```
return - fn;//return an effective variant
```

The population with the highest value of variable *fn* will be considered as the most close to optimum result.

In addition, we sift-out false decisions (if NaN turned out or motor limitations were not passed), returning null.

Override of function *Comparable* realize the mechanism of genetic populations, using as the achieved result the value returned by the function *calcFuction()*.

In the IM class Motor we create the function of automatic calculation, which accepts as an argument the index of optimality criterion and returns the got criterion after the motor calculation:

```
double auto(int krit){
int res = 0;
//Code body of motor calculation
//…
switch (krit){
        case 1://1 is efficiency
            res = 1/kpdnr;
            break;
        case 2://2 – power factor
            res = 1/cosFinr;
            break;
        case 3://3 – starting current
            res = I1pn;
            break;
        case 4://4 – starting torque
            res = 1/Mpo;
            break;
}
    return res;//return a criterion depending on its index
}
```

Further in the motor class Motor we create the function of GA realization (a code structure is explained in comments):

```
void optimization(int krit){
DefaultPoolFactory pf = new DefaultPoolFactory();
//creation of populations with the amount "populations"
GenePool<Solution> pool = pf.createPool(populations,
Solution.class, null);
Rating rtg = new Rating();//Constructor of Rating class
rtg.set_motor(this, krit);//getting of Motor object and criterion
//factory of initial decisions set generation
DefaultHandler handler = new DefaultHandler(rtg, null, null,
null);
//implementation of iterations number "iterations"
//over the population "populations"
handler.iterate(pool, iterations);
//reception of the best found decision
Solution solution = pool.getBestSolution();
D_opt = solution.getX()/1000; //optimal diameter
L_opt = solution.getY()/1000; //optimal length
int iter = rtg.get_iter(); //reception of iterations count number
}
```

From a NetBeans form, the code of the GA optimization implementation consists of two lines:

```
motor.limits();//setting of limitations
motor.optimization(krit);//optimization with the criterion "krit"
```

Functions *limits()* consist restrictions on motor geometric sizes, temperature limits, starting currents and etc. In the fitness-function limitations are checked by the control function

```
boolean flag = mot.control(); //control of limitations
```

This function return *false* if even one restriction will be broken. In *control()* function there are 16 motor variables limits have been set. In particular we can avoid high temperatures of the stator that is probably resulted because of increasing of the stator winding cross-section.

If a decision will not arranged (do not satisfy to motor restrictions according to limitation function) it is possible to continue the GA iterations (increasing the populations number "populations" and iterations "iterations"), while the desired quality of decision will not be attained.

So to solve a GA task using EvoJ it is necessary:

1) to create an interface with variables;
2) to implement the interface of the fitness-function;
3) to create the population of decisions and carry out the necessary amount of the GA iterations above them, using a code, given above.

The results of the GA solution at a choice of maximum efficiency as a criterion of optimality are shown in Tab. 1.

It is obvious from Tab. I, an optimal motor efficiency is higher than a base value and other parameters are satisfying limitation range.

TABLE I
GENETIC ALGORITHM: TABLE OF PARAMETERS BEFORE AND AFTER OPTIMIZATION

Name	Base value	Optimal value
Induction in the air-gap, T	0.748	0.807
Internal diameter of stator core D, mm	185	194
Length of stator core Lδ, mm	130	115
Relative size $\lambda = L\delta/\tau$ ($\tau = \pi D/2p$, where p - number of pole pairs)	0.895	0.755
Height of stator slot, mm	21.9	14.6
Height of rotor slot, mm	32.2	33.2
Width of the upper line of stator slot, mm	7.7	7.8
Width of the down line of stator slot, mm	10.2	9.3
Upper diameter of rotor slot, mm	7.9	7.8
Down diameter of rotor slot, mm	3.7	3.4
Efficiency	0.885	0.891
Power factor	0.893	0.9
Starting current relative value	5.84	6.52
Starting torque relative value	1.4	1.62
Overload torque capability	2.65	2.88
Overall stator winding temperature, C	93.25	95.69

III. CONCLUSIONS

Algorithm of the previous considered CP [5] in comparison with the GA, allows to execute multi-criterion optimization, that is its undoubted advantage. In addition, the CP always gives only the synonymous best variant among the existing ones. However, in the range of varying of two variables ± 20 % from a base value (3976 combinations) the calculation time is approached up to 48 min.

Implementation of CA gives stunning results. At the same varied variables and range of their change ± 100 % (!) from a base value, the calculation time is only 40 sec!

However, GA, at least, in the present article task, does not allow to execute optimization for a few criteria.

In the GA number of the varied variables and a range of their change is not important from the point of view of the productivity, because a set of the varied variables is created dynamically, but not beforehand, as in the CP method. In addition, all combinations of the variables and values of objective function are realized in a binary form. However, time of the GA work is very critical to the number of the created populations and number of iterations in the populations.

The choice of population's number and iterations realized by an experienced way increases until an acceptable result will not be obtained. A result of the optimization with the use of the GA will always be the best for the chosen criterion, but there is not a guarantee, that a better variant can exist. Actually, herein there is a genetic selection logic - we get a result, approaching the best among the random created populations. A most reliable result depends on population's number. Thus, the degree of authenticity can be estimated by the variation of

the obtained results in repeated calculations at the same population's number.

Therefore, the GA productivity at effective variant populations number is determined but not by the number and range of the varied variables.

The result of the optimization is ambiguous and close to the best. When production of approximate calculations in maximum compressed terms is needed and quality of the obtained results is written with a permissible error, then using of the GA optimization will be the irreplaceable instrument for designers.

REFERENCES

[1] Yemelyanov V.V., Kureychik V.V., Kureychik V.M. Theory and practice of evolution modeling. M.: PHYSMATLIT, 2003. – 432 p.

[2] N. Zablodskiy, V. Pliugin, K. Buhr. CAD of electromechanic devices: educational tutorial, part 2, 2013. - 330 p. (will be printed).

[3] *http://evoj-frmw.appspot.com* [ONLINE].

[4] N.K. Vereshchagin, A. Shen. Lectures on mathematical logic and theory of algorithms. Beginning of sets theory. MCNMO, 2008. – 198p.

[5] N. Zablodskiy, V. Pliugin, K. Buhr, S. Khomitskiy. Asynchronous motor optimal design with using of Cartesian product (will be printed).

REFERENCES ON RUSSIAN:

[1] Емельянов В.В., Курейчик В.В., Курейчик В.М. Теория и практика эволюционного моделирования. М.: ФИЗМАТЛИТ, 2003. – 432 с.

[4] Верещагин Н.К., Шень А. Лекции по математической логике и теории алгоритмов. Начала теории множеств. МЦНМО, 2008. – 198с.