Innovative Lifelong e-Learning for Professional Engineers (e-ProfEng)

586391-EPP-1-2017-1-SE-EPPKA2-CBHE-JP

Training in Electrical Engineering Discipline

Modeling and Simulation in Electrical Engineering

Teaching Materials for Topic 3

Theory and implementation of metaheuristic optimization methods for optimizations in the distribution power system

Authors:

Marinko Barukčić, P9 FERIT

Contents

Optimization in brief

Optimization is the process of finding the value of something that satisfied (match) some criteria as best as possible. From a mathematical point of view, it is the procedure of finding the value of the variable (parameter) gives the optimal value of some objective (minimum or maximum extrema).

Mathematics notation of the optimization problem

The general mathematical description of the optimization problem can be expressed as:

$$\begin{aligned}
&f_{obj} = f(\vec{X}) \rightarrow \text{minimize(maximize)} \\
&\vec{X} = [x_1, \dots, x_m] \\
&\text{subject to:} \\
&h_i(\vec{X}) = 0,\ i = 1, \dots, n_e \\
&g_j(\vec{X}) \geq 0,\ j = 1, \dots, n_i \\
&\vec{X}_l \leq \vec{X} \leq \vec{X}_u
\end{aligned} \tag{3.1}$$

where f_{obj} is an objective function, X is a vector of the solution, h and g are equality and inequality constraints respectively.

Elements of the solution vector are decision variables. Decision variables values are the solutions of the optimization problem. The size (number of elements) of the decision variable vector represents problem dimensionality also. Depending on the features of the objective function and constraints as well as problem dimensionality the optimization problem can be very hard to solve.

Overview of the optimization techniques

There are different optimization techniques that can be applied to solve the optimization problem (3.1). Schematic overview of the optimization procedure is given in Fig. 3.1.

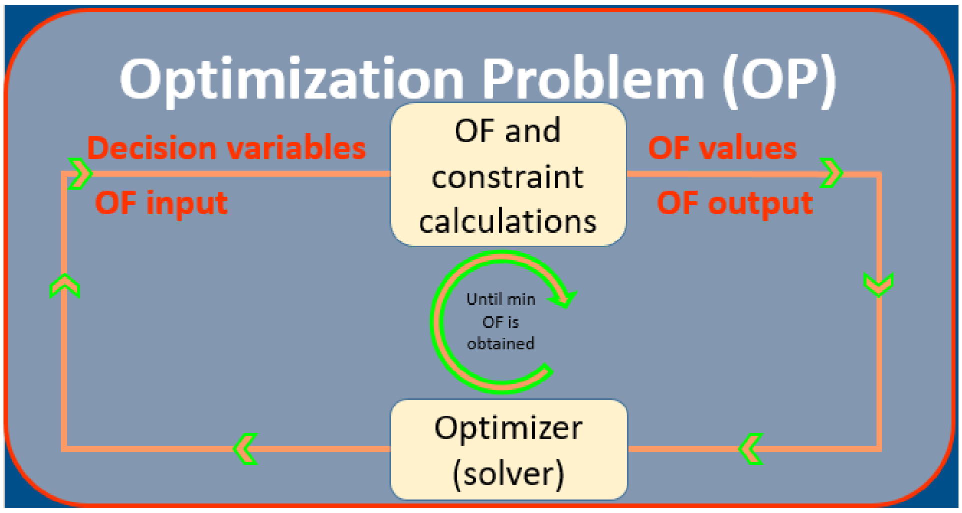


Figure 3.1: Representation of the optimization procedure

The main classification of the optimization techniques is on classical and metaheuristic (heuristic) methods[1]–[3].

Both classes of the optimization techniques have a number of methods.

The main group inside the classical optimization techniques are mathematical programming methods. Some of the classical optimization procedures are:

- Linear programming
- Nonlinear programming
- Quadratic programming
- Combinatorial optimization
- …

The metaheuristic (nonclassical, advanced) optimization methods are procedures for solving optimization problems using partly knowledge about the optimization problem and stochastic based operators. The area of metaheuristic optimization techniques has been researcher very intensive recent time (last few decades). Most of these optimization methods belong to the class of the nature-inspired optimization algorithms. There are three subsets inside this overall set of metaheuristic methods such are bio-inspired algorithms, physics, and chemistry based methods. Special subsets of the metaheuristic method of bio-inspired methods are swarm intelligence (SI) based methods and evolutionary algorithms (EA) [4]–[6]. Examples of metaheuristic methods are:

- Simulated annealing
- Genetic algorithm
- Evolutionary strategy
- Differential evolution (do not has a background in nature but has features of EA)
- Particle swarm optimization
- Cuckoo search optimization
- …

Most of the metaheuristic algorithms are population-based methods. Because they have not deterministic operators, usually these optimization techniques are described as methods available to find a solution near to global optimum. Also, metaheuristic optimizations algorithms are capable to be global optimization procedures.

On figure 3.2 the general scheme of population-based optimization techniques is given.

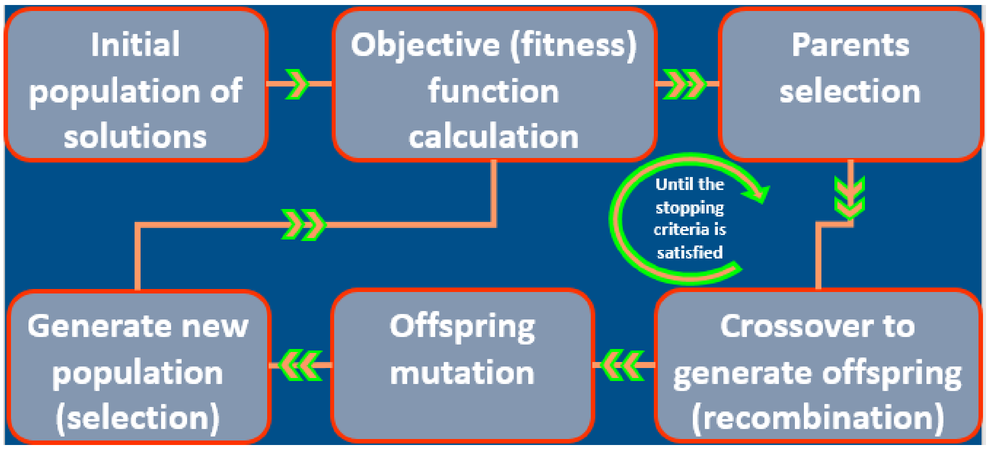

Initial population of solutions
Objective (fitness) function calculation
Parents selection
Until the stopping criteria is satisfied
Crossover to generate offspring (recombination)
Offspring mutation
Generate new population (selection)

Figure 3.2: Procedure of population-based metaheuristic optimization algorithm

Metaheuristic optimization methods search the problem solution space using stochastic operators and need minimum information about the objective function. Classical optimization algorithms required knowing the analytic form of the objective function because they usually use some of the gradient-based procedure to search the solution space. On the other hand, metaheuristic optimizations require only numerical values of the objective. In Table 3.1 main features of the classical and metaheuristic optimizations are given.

Table 3.1: Comparison of main features of the metaheuristic and classical optimization techniques

Optimization /OP properties	**Conventional (classic)**	**Metaheuristic**
Objective, Constraints	analytic expression, continuous, differentiable (smooth)	continuous, discontinuous, differentiable, non-differentiable, only values are required
Decision variables	real, integer (some methods)	real, integer (some methods), linguistic
Start point in the solution space	one	multiple

Based on the properties given in Table 3.1, the advantages and drawbacks of each optimization method group are shown in Table 3.2.

Table 3.2: Advantages and drawbacks of the optimization method classes

	Conventional (classic)	**Metaheuristic**
Advantages	The deterministic solution, proved convergence, law number of objective function and constraints calculation, faster (lower computational time)	Can work using only numeric data (analytic expression is not required), avoid stuck in local optimum (parallel search in solution space, use multiple starting points), can be used for black-box optimization, can be used for global optimization
Drawbacks	The solution can depend on starting point, high possibility of stuck in local optimum, can't be used for black-box optimization	Stochastic operators (can occur different solution when optimization is repeated), high computational effort and time (due to the high number of objective and constraints values calculations)

The differential evolution (DE) will be used here as a metaheuristic optimization method.

Optimization in electric power networks

Generally speaking, each problem in power system which can be formulated in the form of the optimization problem can be solved by using the optimization techniques. Some of the most solved optimization problems in power systems are:

- Allocation of devices in a power network
- System and devices control
- Optimal Power Flow
- Parameters identification
- System and device design
- Network configuration and reconfiguration
- …

The optimization in the distribution power system becomes very actual today due to the development of the smart grid concept and increasing of energy efficiency. Because the objective function is often calculating by using iterative numerical method or simulation, the metaheuristic optimization algorithms are very applicable for this purpose. Some additional examples of metaheuristic optimization applied to problems in the power system can be found in [7], [8].

Used computational tools for simulation and optimization in the power system

Thanks to the development of computer techniques and methods of numerical mathematics, there are a lot of the simulation tools for modeling and simulation of the distribution power system. Use of such simulation tools makes able to perform more accurate calculations decreasing approximations in the model. There are a number of both commercial and non-commercial (mostly open source) simulation tools [9]. Generally speaking, for academic purposes, commercial software have some advantages such are: more user-friendly interface, offer "all in one" solution, tested and validate calculation procedures, have technical support but the main drawback is usually high price. On the other hand, costless tools can be achieved for free or with symbolic price, have possibilities to user modifications and changes but require more work for model building and result presentation. The commercial tools generally have more simulation possibilities then non-commercial but there are a few open source tools have features comparable to the commercial tools. One of such tool is OpenDSS simulation tool [10]. Speaking about existing metaheuristic optimization tools there are a few very specialized commercial solutions [11], [12] as well as tools for general using [13]–[15] but a lot of the tools can be found as open source available in different programming environments (Python, Java, C++, …). One of open source tool implementing metaheuristic optimization by differential evolution in Python is used here.

Differential evolution optimization using SciPy package

SciPy is an open source Python package intended for scientific programming and calculations [16]. The package has submodule for optimization performing called *scipy.optimize* package [17]. The package has functions for classical optimizations and some heuristic methods. Here differential evolution optimization function will be used [18]. The scheme for using *scipy.optimize.differential_evolution* the procedure is shown in Figure 3.3.

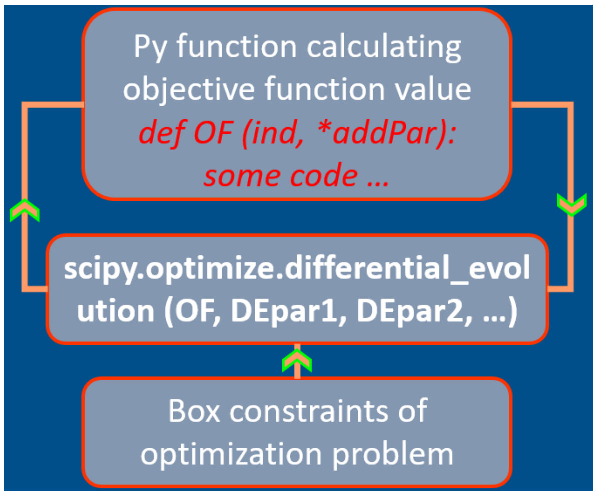

Py function calculating objective function value
def OF (ind, *addPar):
some code …
scipy.optimize.differential_evolution (OF, DEpar1, DEpar2, …)
Box constraints of optimization problem

Figure 3.3: Using differential evolution from scipy

The DE search solution in the solution space using stochastically based evolutionary operators crossover, mutation and selection (Figure 3.2). These operators are defined in DE as follows.

Mutation generate mutant vector:

$$\vec{S}m_g^j = \vec{S}_g^{r1} + F \cdot \left(\vec{S}_g^{r2} - \vec{S}_g^{r3}\right) \qquad r1 \neq r2 \neq r3 \tag{3.2}$$

the S_g^{r1} is the base vector and S_g^{r2}, S_g^{r3} are difference vectors.

Recombination is applied on a mutant vector to get the trial vector:

$$St_{g,i}^j = \begin{cases} Sm_{g,i}^j & if \ \left(rand_i[0,1) \leq Cr\right) \\ S_{g,i}^j & otherwise \end{cases} \tag{3.3}$$

$St^j_{g,i}$ is the i^{th} element of the j^{th} trial vector for the population. The S^j_g is the j^{th} individual (usually called the target vector) from the population and $S^j_{g,i}$ is the i^{th} element of the S^j_g. The *Cr* is DE parameter, called crossover rate in the range [0, 1].

The next generation is populated by the solutions chosen by the selection operator:

$$\vec{S}_{g+1}^j = \begin{cases} \vec{S}t_g^j & if \ \left(f\left(\vec{S}t_g^j\right) \leq f\left(\vec{S}_g^j\right)\right) \\ \vec{S}_g^j & otherwise \end{cases} \tag{3.4}$$

$\vec{S}_{g+1}^j$, $\vec{S}t_g^j$ and $\vec{S}_g^j$ are the individual in the new generation, the trial vector and the target solution vector.

The scipy DE function is called by next code:

scipy.optimize.differential_evolution(OF, BoxCnstr., maxiter=100, popsize=15, mutation=(0.5, 1), recombination=0.7, disp=True, polish=False)

The first argument of the DE function call function calculates value of the objective *"OF"*, the second argument is list contains variable decision limits *"BoxCnstr"*, rest of the arguments are parameters of DE method, number of DE generations, size of the population, mutation factor, crossover factor, "*maxiter=100, popsize=15, mutation=(0.5, 1), recombination=0.7*", displaying solution evolution through the generations can be set by "*disp=True*", and application of the local optimizer can be disabled by *"polish=False"*.

The objective function (OF) is defined as a standard function in Python. The first argument of this function is individual (*ind*) of DE and other (optional) arguments (not directly used by DE) can be added on other positions of function arguments. The objective function, in this case, can be coded as:

```
def OF(ind, *additionalArguments):
        v1, v2 = ind[0], ind[1]
        OFvalue = v1+v2**2
        return OFvalue
```

Exercise 1: Optimization of the problem with an analytically objective function

<u>Task:</u> Find optimal powers of the generating units such that power (energy) production cost per hour is as low as possible (unit commitment problem).

Problem description: The electric power system consists of four generating units supplying the loads with energy. Value of total load is constant in time. Each source produces power (energy) at a price described by the production cost function. The optimal amount of each source needs to be found by the metaheuristic optimization procedure.

System data:

The production cost function is given by:

$$C_i = a_i P_i^2 + b_i P_i + c_i \quad i \in (0,1,2,3) \tag{3.5}$$

The cost coefficients values in (3.5) are given in Table 3.3.

Table 3.3: Cost coefficients for each production unit

Coefficient\Unit	0	1	2	3
a [EUR/kW^2]	3e-4	2e-4	8e-4	1e-4
b [EUR/kW]	0.02	0.015	0.024	0.018
c [EUR]	7	5	4	7

Power limit per production unit: 1000 kW

Amount of total system load is P_{load} = 2500 kW.

Specific tasks to solve the optimization problem by using the metaheuristic optimization procedure:

- Formulate the optimization problem (expressions for the objective function and constraints)
- Coding decision variables in the method individual
- Defining ranges (limits) of the decision variables-box constraints
- Coding the problem in the specific optimization tool (SciPy DE).

Guidance for problem solving:

The objective function: summation of cost functions of each unit

The constraint: system power balance equation – equality constraint, unit production range – box constraints

Syntax of ScyPy Differential Evolution function: https://docs.scipy.org/doc/scipy-0.17.0/reference/generated/scipy.optimize.differential_evolution.html

Solution: jupyter notebook ***Optim01.ipynb*** in additional materials, OF = 402.1 EUR, P1 = 522.68 kW, P2 = 786.92 kW, P3 = 190.53 kW, P4 = 1000 kW.

Self exercise 1:

- All units produce in range 300 – 1000 kW (all other data are same as in the basic problem), *solution: OF = 412.51 EUR, P1 = 477.24 kW, P2 = 722.96 kW, P3 = 300 kW, P4 = 1000 kW.*
- Total load is 500 kW (all other data are same as in the basic problem), *solution: OF = 44.63 EUR, P1 = 80.86 kW, P2 = 135.83 kW, P3 = 29.1 kW, P4 = 254.2 kW.*

OpenDSS simulation software – basics for model generating

The OpenDSS simulation tool is, above all, intended for simulation of the modern distribution system and it is maintained by EPRI institute [19]. Detailed instructions for using of the tool can be found in the manual (also available after software installation) [20] and very useful materials for starting in use of the tool are available on [21] such are [22]–[24]. One of the very useful features of the OpenDSS will be used here, is its interfacing possibilities to other simulation tools and programming environments. The OpenDSS can be achieved and manipulate from MATLAB, PYTHON, Visual Basic for Application, C…[20]. Some of the basic things needed for model building in the OpenDSS simulation toll are shown below.

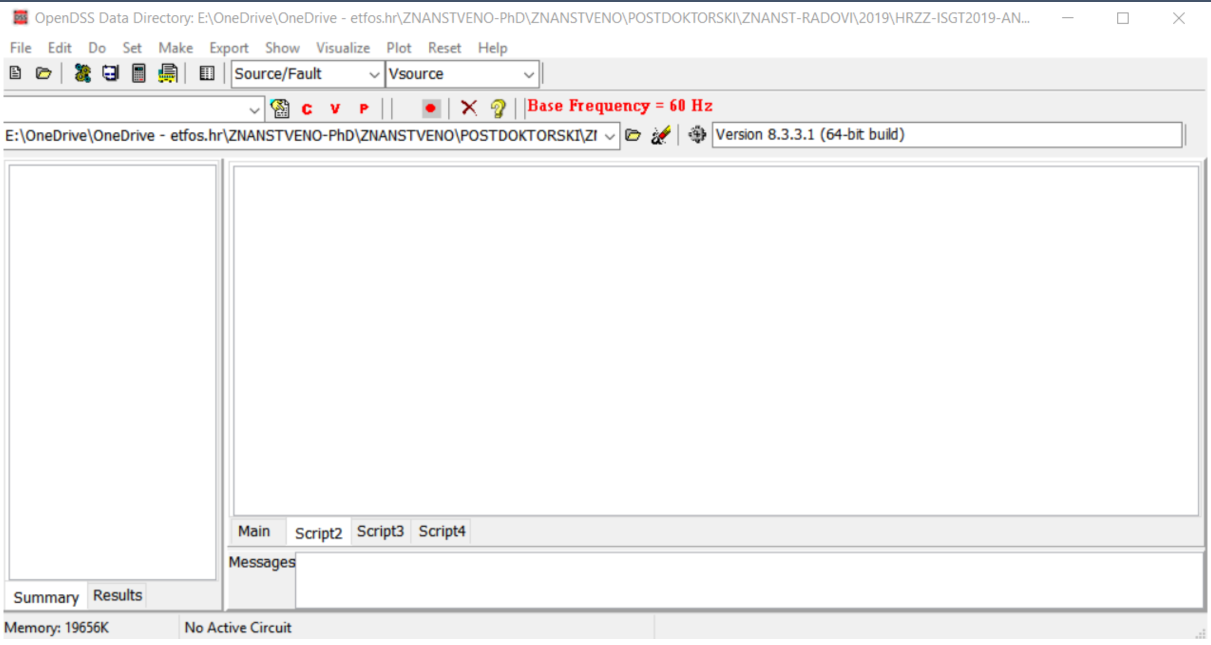
After the OpenDSS is started the main program window is opened (Figure 3.4). The distribution system (network) model is built by writing a textual script so there is no graphical interface (GUI) with drag and drop approach. However, recently OpenDSS version including GUI for system modeling OpenDSS-G is presented [25]. The text script approach is very useful for modification network model in OpenDSS through the external programming environment (MATLAB; PYTHON…) making the OpenDSS very flexible for specific user usage. Thanks to this whole model in OpenDSS can be generated from an external tool. Also, the model in the form of the script allows building the model in any text editor (Notepad, Word …) and simply copying it in the OpenDSS window. The program execution and model solving is done by selecting all text in the script and choosing the command "*Do selected*" as it shows Figure 3.5. The main approach to build a model in OpenDSS is based on modeling each the real network element by the command in the script. When some code is written comments are usually added in it for purpose of code explanation. The text in OpenDSS will be considered as a comment if the exclamation mark is inserted on the beginning of the text line:

! All write after exclamation mark is a comment (limited od one line in the script)

If the command in the script is longer than one line it can be continued in the next line by inserting tilde (~) mark at the beginning:

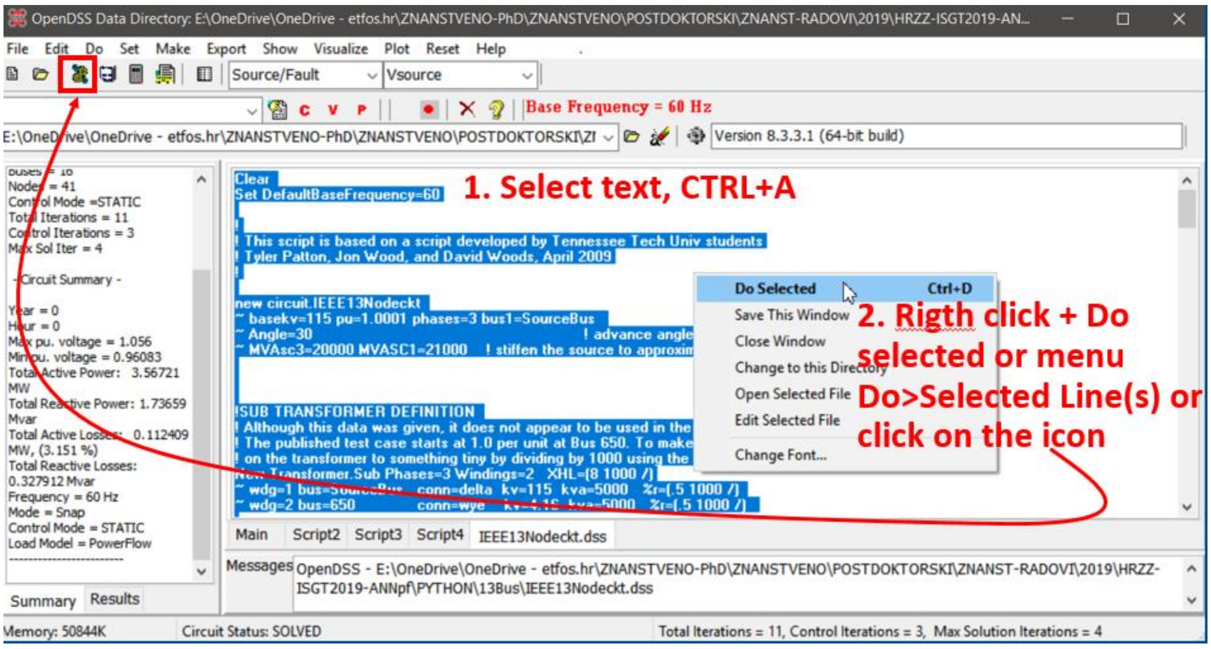
This command is in

~ two lines


OpenDSS Data Directory: E:\OneDrive\OneDrive - etfos.hr\ZNANSTVENO-PhD\ZNANSTVENO\POSTDOKTORSKI\ZNANST-RADOVI\2019\HRZZ-ISGT2019-AN...
File Edit Do Set Make Export Show Visualize Plot Reset Help
Source/Fault
Vsource
Base Frequency = 60 Hz
Version 8.3.3.1 (64-bit build)
Main Script2 Script3 Script4
Messages
Summary Results

Memory: 19656K No Active Circuit

Figure 3.4 OpenDSS user interface



```
Clear
Set DefaultBaseFrequency=60

!
! This script is based on a script developed by Tennessee Tech Univ students
! Tyler Patton, Jon Wood, and David Woods, April 2009
!

new circuit.IEEE13Nodeckt
~ basekv=115 pu=1.0001 phases=3 bus1=SourceBus
~ Angle=30                              ! advance angle
~ MVAsc3=20000 MVASC1=21000    ! stiffen the source to approxim


!SUB TRANSFORMER DEFINITION
! Although this data was given, it does not appear to be used in the
! The published test case starts at 1.0 per unit at Bus 650. To make
! on the transformer to something tiny by dividing by 1000 using the
New Transformer.Sub Phases=3 Windings=2   XHL=(8 1000 /)
~ wdg=1 bus=SourceBus   conn=delta  kv=115  kva=5000   %r=(.5 1000 /)
~ wdg=2 bus=650         conn=wye    kv=4.16 kva=5000   %r=(.5 1000 /)
```

```
Buses = 16
Nodes = 41
Control Mode =STATIC
Total Iterations = 11
Control Iterations = 3
Max Sol Iter = 4

 - Circuit Summary -

Year = 0
Hour = 0
Max pu. voltage = 1.056
Min pu. voltage = 0.96083
Total Active Power:   3.56721
MW
Total Reactive Power: 1.73659
Mvar
Total Active Losses:   0.112409
MW, (3.151 %)
Total Reactive Losses:
0.327912 Mvar
Frequency = 60 Hz
Mode = Snap
Control Mode = STATIC
Load Model = PowerFlow
------------------------
```

Messages: OpenDSS - E:\OneDrive\OneDrive - etfos.hr\ZNANSTVENO-PhD\ZNANSTVENO\POSTDOKTORSKI\ZNANST-RADOVI\2019\HRZZ-ISGT2019-ANNpf\PYTHON\13Bus\IEEE13Nodeckt.dss

Memory: 50844K Circuit Status: SOLVED Total Iterations = 11, Control Iterations = 3, Max Solution Iterations = 4

Figure 3.5 Start simulation in OpenDSS

The model of the real system element is represented by the object in OpenDSS. This object is programmed in the form of the OpenDSS class [20]. The general representation of the network (system) element model in OpenDSS is done by giving keyword (program command) at the beginning of the script line, after that instantiate the object class and defining an object:

Command verb *Object class* *Class attributes*

According to this syntax the model of the network element in the OpenDSS script has three main parts and can be added as:

New *element.SomeName* *Bus1=Name Par1=value Par2=value …*

Space (obtained by keyboard backspace or tab) or comma are used to designate different parts of the above general syntax scheme. Depending on the action in OpenDSS each script line has to have the first part (Command verb) of the given syntax scheme at least.

Based on the above explanation the distribution system model can be generated. The first **reference node (slack node)** need to be modeled by using the Thevenin equivalent:

New circuit.TestNet basekv=11 pu=1.0 phases=3 bus1=SourceBus Angle=0

The command verb for adding the element in the model is "*New*", object class represents the Thevenin equivalent is "*circuit*", the arbitrary name of the element (class instance) in this case is "*TestNet*", attributes of the class are parameters of the element defined in form "*parameter name = parameter value*" – source nominal voltage = amount in kV – "*basekv=11*", place of the source installation in the system is defined as bus1=bus name (arbitrary) – "bus1=SourceBus", reference voltage in p.u. is given by "*pu=1.0*", number of phases is defined as "*phases=3*", voltage phase shift in degrees is given by "*Angle=0*".

The network lines can be added into the OpenDSS model as follow:

New line.L1 bus1=SourceBus bus2=B2 R1=0.117 X1=0.048 Phases=3

In case of the **line** the buses on which line terminals are connected (buses connected by line) need to be defined "*bus1=SourceBus bus2=B2*", line parameters can be defined in more ways [20] and here it is done by defining line series impedance per length (resistance and inductive reactance in Ohm/m) as "*R1=0.117 X1=0.048*". If the length of the line is not defined the default length multiplier is 1.

The **load** is defined by the next script line:

New load.LD1 bus1=B2 kv=11 kw=230 kvar=142.5 model=1 Phases=3 conn=wye

Because the **load** is connected as shunt in the network only one bus is defined "*bus1=B2"*, nominal voltage in kV, nominal active (in kW) and reactive (in kvar) powers of the load are given by "*kv=11 kw=230 kvar=142.5*" respectively, mathematical model of the load (constant power, constant impedance…) is defined as "*model=1*" and type of the load connection by "*conn=wye*".

At the end of the model all nominal voltage values existing in the different parts of the system are given with the purpose of expressing simulation results in p.u.:

Set voltagebases=[11]

The generating list of bus names is done giving the command:

Calcv

After the network model is coded, the simulation can be performed by:

Solve

Writing and showing some results the command starting with command "*Show*" is used:

Show voltages

Exercise 2: Coding distribution networks in OpenDSS

Task: For given example of the distribution network feeder in Figure 3.6, make a model in OpenDSS simulation tool. Data of the network are given in Table 3.4. Simulate a given network and show total losses and phase voltages in all nodes. Loads are modeled as constant power load model.

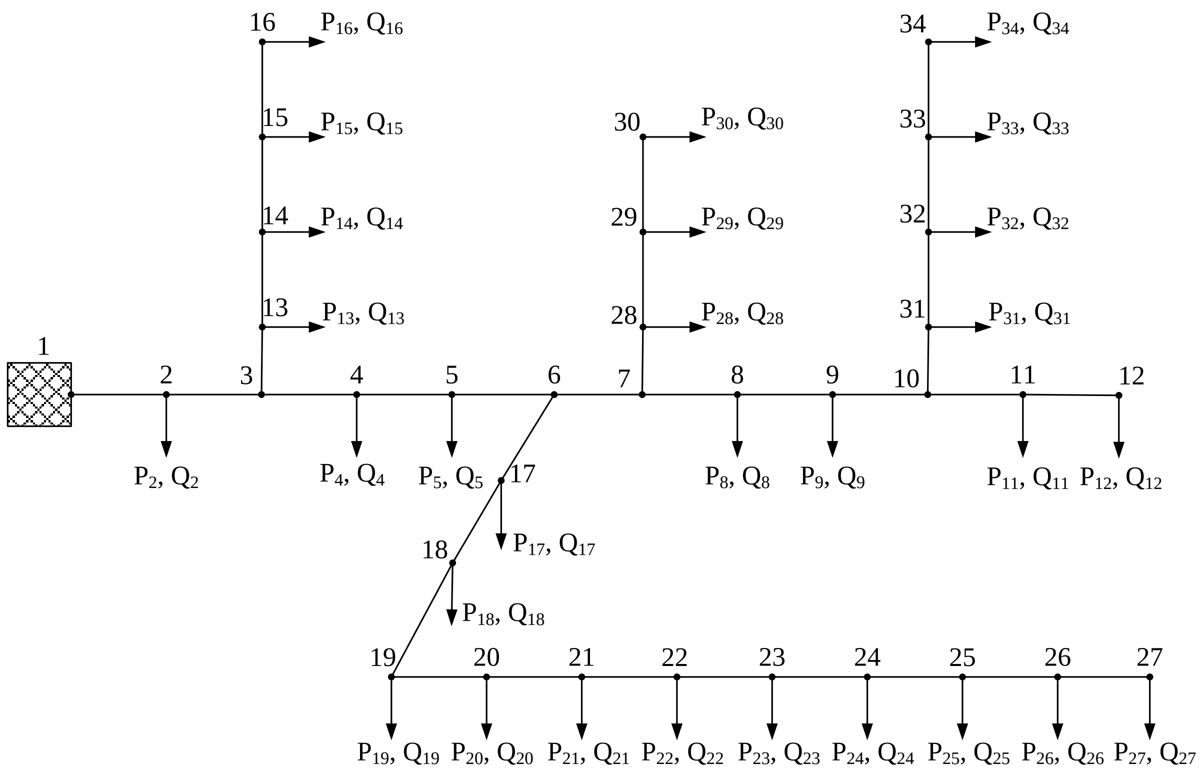


Figure 3.6: 11 kV Distribution network

Table 3.4: Data of the network from Figure 3.6.

Line	Line impedance		Node	Load power	
	R [Ω]	X_L [Ω]		P [kW]	Q [kvar]
1 – 2	0,117	0,048	2	230,00	142,50
2 – 3	0,107	0,044	3	0,00	0,00
3 – 4	0,164	0,046	4	230,00	142,50
4 – 5	0,150	0,042	5	230,00	142,50
5 – 6	0,150	0,042	6	0,00	0,00
6 – 7	0,314	0,054	7	0,00	0,00
7 – 8	0,210	0,036	8	230,00	142,50
8 – 9	0,314	0,054	9	230,00	142,50
9 – 10	0,210	0,036	10	0,00	0,00
10 – 11	0,131	0,023	11	230,00	142,50
11 – 12	0,105	0,018	12	137,00	84,00
3 – 13	0,157	0,027	13	72,00	45,00
13 – 14	0,210	0,036	14	72,00	45,00
14 – 15	0,105	0,018	15	72,00	45,00
15 – 16	0,052	0,009	16	13,50	7,50

6 – 17	0,179	0,050	17	230,00	142,50
17 – 18	0,164	0,046	18	230,00	142,50
18 – 19	0,213	0,047	19	230,00	142,50
19 – 20	0,194	0,043	20	230,00	142,50
20 – 21	0,194	0,043	21	230,00	142,50
21 – 22	0,262	0,045	22	230,00	142,50
22 – 23	0,262	0,045	23	230,00	142,50
23 – 24	0,314	0,054	24	230,00	142,50
24 – 25	0,210	0,036	25	230,00	142,50
25 – 26	0,131	0,023	26	230,00	142,50
26 – 27	0,105	0,018	27	137,00	85,00
7 – 28	0,157	0,027	28	75,00	48,00
28 – 29	0,157	0,027	29	75,00	48,00
29 – 30	0,157	0,027	30	75,00	48,00
10 – 31	0,157	0,027	31	57,00	34,50
31 – 32	0,210	0,036	32	57,00	34,50
32 – 33	0,157	0,027	33	57,00	34,50
33 – 34	0,105	0,018	34	57,00	34,50

Specific tasks to solve the optimization problem by using the metaheuristic optimization procedure:

- Formulate the optimization problem (expressions for the objective function and constraints)
- Coding decision variables in the method individual

Solution: *OpenDSS script in additional materials, P_{loss} = 217.5 kW, V_{min} = 0.9399 p.u. in node b27*.

Self-exercise 2:

- Add two distributed generation (DG) units of sizes P1 = 500 kW, Q1 = 150 kvar, P2 = 800 kW, Q2 = 400 kvar in network nodes 9 and 24. Compare total network losses to case without DGs. *solution: OpenDSS script in additional materials, P_{loss} = 99.3 kW, V_{min} = 0.9631 p.u. in node b27.*
- Add two distributed generation (DG) units of sizes P1 = 500 kW, Q1 = 150 kvar, P2 = 800 kW, Q2 = 400 kvar in network nodes 2 and 16. Compare total network losses to the case without DGs and DGs installed in nodes 9 and 24. *solution: OpenDSS script in additional materials, P_{loss} = 198.2 kW, V_{min} = 0.9427 p.u. in node b27.*

Co-simulation approach to metaheuristic optimizations in electric power system

Using simulation tools for modeling and analyzing the power distribution networks is ordinary today. Modeling and calculations of the system using the specialized computing tools make able to make analysis with less neglecting and approximations in the model. Also, there are specialized computing tools for implementing metaheuristic optimizations. In the case of the calculating, some quantities in the system model by simulating them in simulation software the output data are obtained numerically without explicitly expressed relation between inputs and outputs. So, if some optimization of the power system wants to be done with obtaining objective values through simulations the metaheuristic algorithms are imposed as optimization methods due to their features (Tables 3.1 and 3.2). This approach in optimization is known as black-box optimization. The co-simulation setup between simulation tool for power system and metaheuristic optimization tool is set to perform this black-box optimization. The co-simulation approach makes able to do more accurate results due to more realistic modeling of the distribution power system in a specialized simulation tool. A general overview of such setup is shown in Figure 3.7. Main drawbacks of this approach are high computational time (due to a number of objective function and constraints calculations) and obtaining near to global optimum solutions. Also, the choice of the software tools will be co-simulated depends on tools features regarding interfacing with external software. Fortunately, most (especially open source) of the power system simulation tools has an interface to the most used programming environment for scientific and engineering calculations such are MATLAB, PYTHON, C++. Some of these programming tools such are MATLAB and PYTHON have built-in metaheuristic optimizations what makes them very suitable for co-simulation approach for solving complex optimization problems in the distribution networks. Apart from these built-in metaheuristic optimization tools, there are standalone tools especially in PYTHON, Java, C programming environments or that have an interface to these programs.

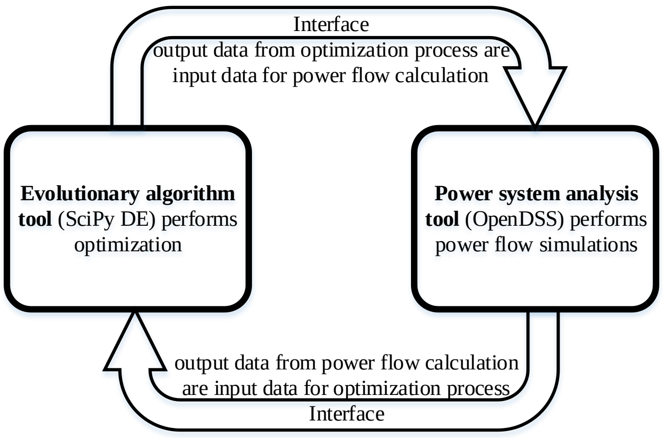

Interface
output data from optimization process are input data for power flow calculation
Evolutionary algorithm tool (SciPy DE) performs optimization
Power system analysis tool (OpenDSS) performs power flow simulations
output data from power flow calculation are input data for optimization process
Interface

Figure 3.7: Co-simulation setup OpenDSS – SciPy DE

OpenDSS – SciPy DE co-simulation

The used co-simulation setup here includes SciPy Python package and OpenDSS power system simulation tool that co-work inside Python environment. Because SciPy package is Python native there is no need any procedure to interface it with Python. The OpenDSS has an interface to different programming languages as mentioned above. Interfacing OpenDSS to Python can be done in two ways: through COM interface or using OpenDSSdirect Python package [26]. Below procedure for interfacing OpenDSS with Python using COM interface is presented.

The first access to OpenDSS COM from Python need to be established:

import win32com.client

dssObj = win32com.client.Dispatch("OpenDSSEngine.DSS")

where *"dssObj"* is an instance of the main OpenDSS object and *"OpenDSSEngine.DSS"* is OpenDSS solution engine.

After the main OpenDSS object is instantiated other instances to other model objects can be established, for example:

dssText = dssObj.Text # interface to text editing in the script *dssCircuit = dssObj.ActiveCircuit # interface to the circuit model*

dssSolution = dssCircuit.Solution # interface to the solution method

dssElem = dssCircuit.ActiveCktElement # interface to model element

Before starting with manipulation inside the OpenDSS model, the OpenDSS script needs to be compiled:

dssText.Command = r"Compile 'E:\Google_Drive_on_E\ZNANSTVENO-PhD\ZNANSTVENO\POSTDIPLOMSKI\ZNANST-RADOVI\2017\ELMAkonf-Sofija-Bulgaria\PYTHON\IEEE13estLOAD.dss'"

Performing a few above code lines in Python makes the preparation for handling network model built in OpenDSS from Python.

In addition, each of the modeled network element reached from Python and its properties value can get or set:

dssCircuit.Loads.Name = 'L1' # set load L1 as active load

pf0 = dssCircuit.Loads.pf # get load power factor value of L1

*dssCircuit.Loads.pf= 0.9*pf0 # set new pf value of L1*

After the wanted modifications in the OpenDSS script are done simulation can be started and results can be accessed from Python:

dssSolution.Solve() # execute simulation

loss = dssCircuit.Losses # get total network losses

Stot = dssCircuit.TotalPower # get power injected from (in) reference node

Vph = dssCircuit.AllBusVolts # get components in [V] of voltage phasors (complex form) for all network nodes

Vpu = dssCircuit.AllBusVmagPU # get RMS values in [p.u] of nodal voltages

Based on the above-given information of basic usage of OpenDSS and SciPy DE tools and their co-simulation setup the optimization problem in the power distribution network (system) can be codded and solved in the used programming environment.

Exercise 3: Optimal allocation of distributed generation (DG) in the distribution network

Task 1: Using OpenDSS – SciPy DE co-simulation find the optimal allocation of DG in distribution network from Figure 3.6 (network data in Table 3.4). DG power limits are PDG ∈ (0, 5000) kW, QDG ∈ (-2000, 2000) kvar. Use the OpenDSS script generated in Exercise 2. All modifications in the OpenDSS script do only from Python except adding DG unit (add it in OpenDSS script in any network node). Nodal voltage values required to be within the limits of 0.9 p.u. ≤ V ≤ 1.1 p.u. The problem objective is the minimization of total active power losses. (Tip: repeat the optimization procedure more time changing parameters of DE each time to find their values guarantee convergence of the solution near to the global optimum).

Task 2: As Task 1 but with loads changing in time according to the dynamic given in Figure 3.8. Production of DG is required to be constant over time.

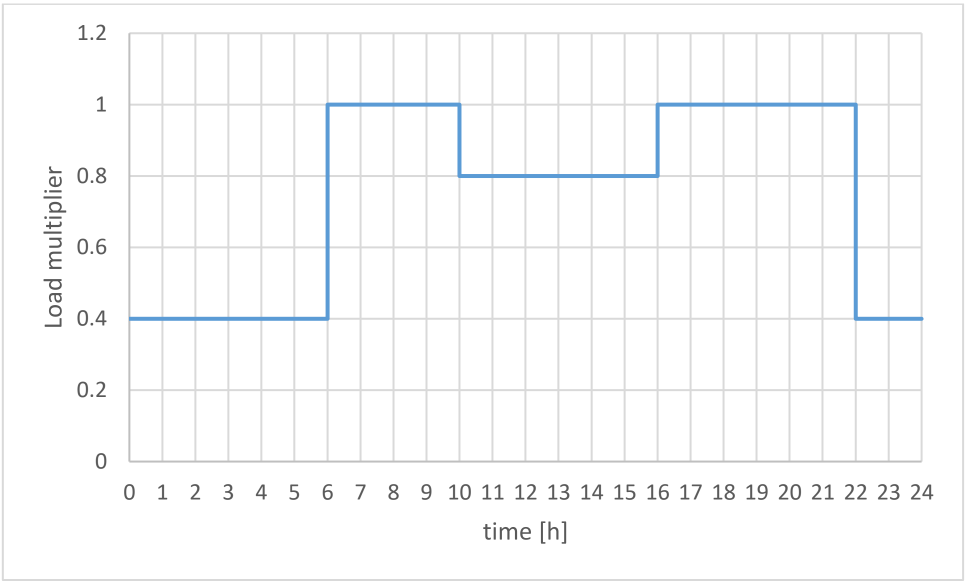


Figure 3.8: Load shape in Task 2 (Exercise 3)

Specific tasks to solve the optimization problem by using co-simulation approach:

- Set COM interface to OpenDSS
- Coding decision variables as DE individual
- Add DG unit to prepared OpenDSS script through COM interface from Python
- Formulate the optimization problem in black-box approach – function in Python including an interface to OpenDSS
- Formulate and code constraints – simultaneously with coding the objective function
- Defining ranges (limits) of the decision variables
- Execute optimization and show results

Solution:

Add DG unit in OpenDSS script prepared in Exercise 2:

New Generator.DG1 bus1=LOCd …

DE individual can be formulated as ind = [DG_L, DG_P, DG_Q] with decision variables of location, active and reactive powers of DG. So, this is a three-dimensional problem - 3 decision variables.

Because SciPy DE works with continuous variables the first decision variable need to be coded as integer values represents the network node (location of the DG):

DGL = round(ind[0])

Location represents position in bus list:

BusN = [bus 1, bus2, … busN]

It will be decoded to OpenDSS syntax as:

LOCd = BusN[DGL]

dssText.Command = 'Edit generator.DG1' + 'bus1='+LOCd+' kw='+str(ind[1])+' Kvar='+str(ind[2])

Solution of Task 1: Complete solution in form of OpenDSS and Python scripts can be found in additional materials, and optimal allocation is: *bus b21, PDG = 2937 kW, QDG = 1800 kvar,* P_{LOSS} *= 46.93 kW,* V_{min} *= 0.9823 p.u. ,* V_{max} *= 0.9993 p.u.*

Solution of Task 2: Complete solution in form of OpenDSS and Python scripts can be found in additional materials, and optimal allocation is: *bus b21, PDG = 2217.47 kW, QDG = 1358.77 kvar,* P_{LOSS} *= [27.25349751651501, 30.15595202862467, 56.55781826046511] kW,* E_{LOSS} *= [218.0279801321201, 180.93571217174804, 565.5781826046511] kWh,* E_{LOSST} *= 964.54 kWh,* V_{min} *= [1.000, 0.9849, 0.9770] p.u. ,* V_{max} *= [1.016, 0.9994, 0.99907]p.u.*

Self-exercise 3:

Solve Task 1 from exercise 3 with possible installation of 5 DG units. What is the optimal number of DGs?. *Solution: OpenDSS script and PY scripts in additional materials, buses ['b21', 'b17', 'b25', 'b9', 'b30'], PDG = [899.52, 916.98, 954.98, 796.53, 492.49] kW, QDG = [558.18, 557.77, 588.11, 495.09, 307.61] kvar, PDGt = 4060.5 kW,* P_{LOSS} *= 3.71 kW,* V_{min} *= 0.9979 p.u. ,* V_{max} *= 0.9998 p.u.*

Basics of multiobjective optimization

Multiobjective optimization is defined as the optimization of a number of objectives simultaneously. Solving such optimization problem can be very challenging if objectives are conflicted what is ordinary in the case of multiobjective optimization problems (MOOP) [27]–[30]. The objective functions in case of MOOP are represented by the function vector consists of individual objective functions:

$$\min \boldsymbol{F}(\boldsymbol{X}) = [f_1(\boldsymbol{X}), f_2(\boldsymbol{X}), \cdots, f_i(\boldsymbol{X}), \cdots, f_z(\boldsymbol{X})\,]^T$$

$$\text{Subject to:}$$

$$g_j(\boldsymbol{X}) \leq 0, \qquad j = \{1, \dots, m\}$$

$$h_k(\boldsymbol{X}) \ = \ 0, \qquad k = \{1, \dots, p\} \tag{3.6}$$

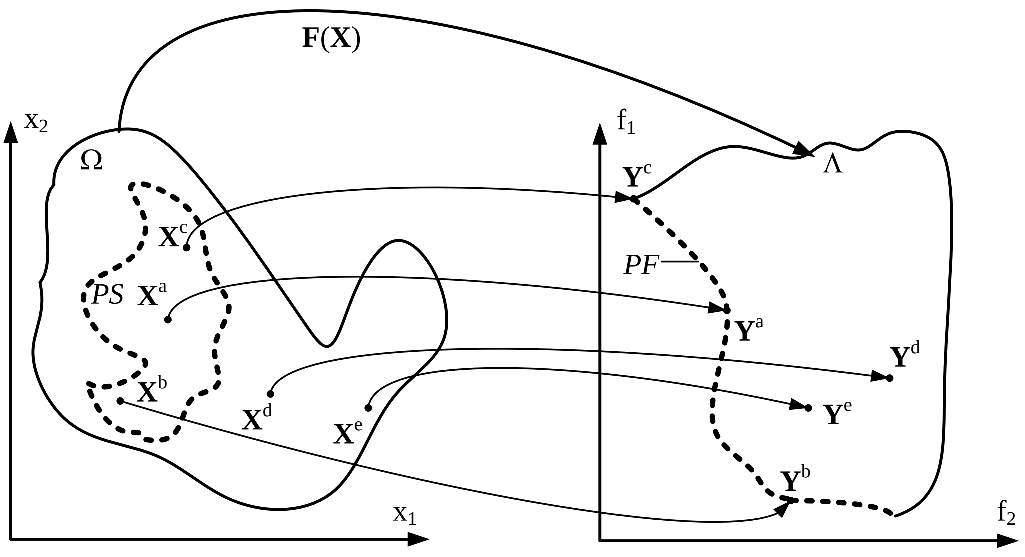
$$\text{solution:}$$

$$\boldsymbol{X} = [x_1,\ x_2, \cdots, x_r, \cdots, x_n]^{\boldsymbol{T}}$$

The solution of MOOP is the solution set as opposed to the one solution in case of single optimization. When solution set is obtained decision maker can choose the optimal solution according to currently specific criteria. The MOOP solutions included in the solution set represent trade-off solutions between objectives in conflict with different levels of trade-off.

The solution of the MOOP is based on Pareto definitions [31]. Some of the definitions are mentioned below.

The MOOP solution set is often called the Pareto set and corresponding values of all objectives are called the Pareto front, Figure 3.8.



$\mathbf{X}^a$, $\mathbf{X}^b$, $\mathbf{X}^c$ – non-dominated solutions

$\mathbf{Y}^a$, $\mathbf{Y}^b$, $\mathbf{Y}^c$ – objective values for non-dominated solutions

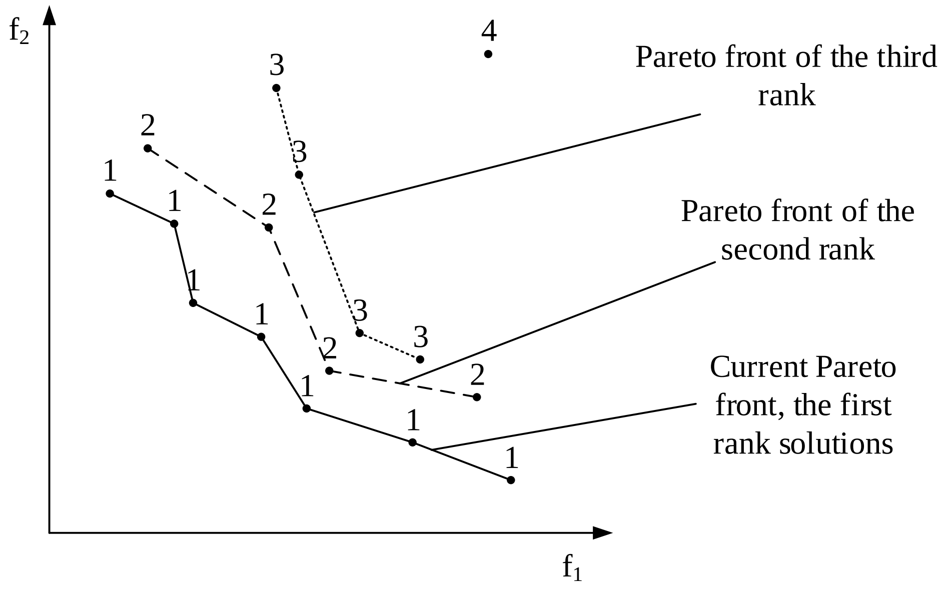
$\mathbf{Y}^d$, $\mathbf{Y}^e$ – objective values for dominated solutions

Figure 3.8: Pareto set and Pareto front for two objectives MOOP

The procedure of solving MOOP in different optimization method use Pareto dominance defined as:

$$\begin{gathered} \forall i \in \{1 \ldots n\} : f_i(dv_1) \leq f_i(dv_2) \wedge \\ \exists k \in \{1 \ldots n\} : f_k(dv_1) < f_k(dv_2) \end{gathered} \tag{3.7}$$

According to (3.7) one solution from Pareto set is non-dominated by other solution if its objective values are equal or less than objective values for other solution and at least one objective values are less then the values for the other solution. Non-dominated solutions are more quality ("better") than the dominated solution and all non-dominated solutions have the same solution quality in sense of the Pareto dominance definition. Such defined relations between solutions of MOOP are base for the so-called Pareto ranking method in MOOP. The fitness and sorting of solutions in metaheuristic optimization are based on the ranking sorting method given in Figure 3.9.



f₂
f₁
Pareto front of the third rank
Pareto front of the second rank
Current Pareto front, the first rank solutions
1
2
3
4

Figure 3.9: Representation of ranking method in case of two objectives MOOP

There are more metaheuristic optimization techniques such are [31]:

- Vector Evaluated Genetic Algorithm (VEGA)
- Niched Pareto Genetic Algorithm (NPGA)
- Nondominated Sorting Genetic Algorithm (NSGA)
- Strength Pareto Evolutionary Algorithm (SPEA)

DEAP – Python package for multiobjective optimization

There are tools for solving MOOP in different programming languages. In the Python programming environment, more different tools using metaheuristic optimizations are available such are DEAP, PyGMO, Platypus, pymoo [31]–[34]. Here DEAP is used as an example of solving MOOP. DEAP – distributed evolutionary algorithms in Python is very flexible tools allowing the user to develop costume EA by combining different evolutionary operators and methods.

DEAP module is prepared for use with the next command in Python:

```
import deap
```

```
from deap import algorithms, base, creator, tools
```

The first procedure in DAEP application is creating types of metaheuristic optimization parts:

```
creator.create("FitnessMin", base.Fitness, weights=(-1.0,-1.0))
```

```
creator.create("Individual", list, fitness=creator.FitnessMin)
```

These above two commands create fitness function as minimization of two objectives. Both objectives have weight 1.0 and minus (-) sign define minimization type of the problem (without – the maximization problem is defined). Individual of EA is declared to be “list” type (in form of Python list) and goodness of individual is based on before defined fitness function.

Instantiate DEAP toolbox class is done by:

toolbox = base.Toolbox()

Now the type of decision variables responsible for the construction of EA individual can be defined by registration type of the individual:

toolbox.register("dv1", random.randint, L1, U1)

toolbox.register("dv2", random.random, L2, U2)

The first decision variable (represents decision variable 1 of the problem) is the type of integer number ("*random.randint*") in a range lower – upper variable limit ("*L1-U1*", box constraints). The second decision variable is real-valued (variable of float type) inside its range.

After all decision variables are registered the EA individual can be constructed as:

toolbox.register("ind", tools.initCycle, creator.Individual, (toolbox.Nl, toolbox.Nn), n=1)

The EA individual can be named arbitrary ("*ind*"). In this example, the individual is generated by successive repeating (*"tools.initCycle"*) n times ("*n=1*") the defined sequence of the decision variables

When EA individual in DEAP is constructed the population can be generated as follow:

toolbox.register("population", tools.initRepeat, list, toolbox.ind)

Name of the EA population is arbitrary ("*population*") and the population will be populated with defined individuals ("*toolbox.ind*") by generating the individual given times ("*tools.initRepeat*").

Now the population of size "*nP*" can be generated as:

pop = toolbox.population(n=nP)

The next step is registration of the EA operators, mutation, crossover and selection:

toolbox.register("mate", tools.cxUniform, indpb = 0.5)

toolbox.register("mutate", tools.mutPolynomialBounded, low=[L1, L2], up=[U1, U2], eta=20.0, indpb=0.05)

toolbox.register("select", tools.selNSGA2)

In above example recombination ("*mate*") is preformed by uniform crossover type ("*tools.cxUniform"*) and crossover probability value is set to 0.5 ("*tools.cxUniform*"). The mutation operator ("*tools.mutPolynomialBounded*") will generate feasible mutant inside decision variable range ("*low=[L1, L2], up=[U1, U2],*"). Each decision variable in individual has probability of mutation of 5% ("*indpb=0.05*"). Because the multi objective problem is considered here the DEAP selection operator based on the Pareto ranking is used ("*tools.selNSGA2*") represents selection used in NSGA2 optimization method.

The objective function used by the DEAP optimization algorithm is registered as:

toolbox.register("evaluate", OF)

Where "*OF*" is the name of the problem objective function defined in the form of ordinary Python function.

When all parts of the EA are defined the optimization procedure can be started:

res, log = algorithms.eaMuPlusLambda(pop, toolbox, mu=toolbox.pop_size, lam=toolbox.pop_size, cxpb=1-toolbox.mut_prob, mutpb=toolbox.mut_prob, stats=None, ngen=nG, verbose=True)

Type of the EA is defined by the name of the DAEP algorithm class ("*algorithms.eaMuPlusLambda*"). Here used (μ+λ) EA type includes the elitism principle (surviving the best individual through evaluation) by making population in the next generation for parents and offspring. With arguments "pop" and "toolbox" previously defined, EA population and operators are forwarded to the optimization algorithms. Number of parent "*mu*" and offspring ("*lam*") individual can be defined here with "*mu=toolbox.pop_size, lam=toolbox.pop_size*". In case it is needed the previously defined crossover and mutation probabilities can be changed ("*cxpb=1-toolbox.mut_prob, mutpb=toolbox.mut_prob*"). A maximal number of generation is defined as "*ngen=nG*" and storage of the evolution statistic can be enabled (or disabled) with "*verbose=True*".

A solution of the optimization problem after the optimization is finished can be simply obtained with:

DVsol = np.array(res)

Exercise 4: Optimal allocation of distributed generation (DG) in the distribution network for two objective optimization problem

Task: Using OpenDSS – DEAP co-simulation find the optimal allocation of DG in distribution network from Figure 3.6 (network data in Table 3.4). DG power limits are PDG $\in$ (0, 5000) kW, QDG $\in$ (-2000, 2000) kvar. Use the OpenDSS script generated in Exercise 2. All modifications in the OpenDSS script do only from Python except adding DG unit (add it in OpenDSS script in any network node). Nodal voltage values required to be within limits of 0.9 p.u. ≤ V ≤ 1.1 p.u. The problem objectives are the minimization of total active power losses and minimization of active DG power. Loads are constant over time. (Tip: repeat the optimization procedure more time changing parameters of EA each time to find their values guarantee convergence of the solution near to the global optimum).

Specific tasks to solve the optimization problem by using co-simulation approach:

- Set COM interface to OpenDSS
- Coding decision variables as DEAP EA individual
- Add DG unit to prepared OpenDSS script through COM interface from Python
- Formulate the optimization problem in black-box approach – function in Python including an interface to OpenDSS
- Formulate and code constraints – simultaneously with coding the objective function
- Defining ranges (limits) of the decision variables
- Execute optimization by using DEAP and show results

Solution:

Add DG unit in OpenDSS script prepared in Exercise 2:

New Generator.DG1 bus1=LOCd …

EA individual can be formulated as ind = [DG_L, DG_P, DG_Q] with decision variables of location, active and reactive powers of DG. So, this is a three-dimensional problem - 3 decision variables.

DG location represents position in bus list:

LOC = [bus 1, bus2, … busN]

It will be decoded to OpenDSS syntax as:

LOCd = LOC[DGL]

dssText.Command = 'Edit generator.DG1' + ‘bus1='+LOCd+' kw='+str(ind[1])+' Kvar='+str(ind[2])

Parts of EA should be coded for using DEAP based on the guidance given in previous subsection.

Complete solution in form of OpenDSS and Python scripts can be found in additional materials, and optimal allocation is: *optimal location in Pareto set are in busses b21, b22 and b23, P_{DG} is in range 1.21-2886.4 kW, Q_{DG} is in range 1522.97-1787.62 kvar, Ploss is in range 46.97-171.41 kW.*

Homework:

- Solve Task from exercise 4 with the possible installation of 10 DG units. What are optimal allocations of DGs for edges of the Pareto front (minimal losses and maximal voltage profile flatness)?

References as sources for further self-learning

The examples of application metaheuristic application in power systems and electrical engineering as inspiration for further research can be found in [35]–[48]. Besides this, a number of sources of interests for this topic (metaheuristic methods, simulation tools, optimization in electrical system) are given through the previous sections and listed below in reference list.

Reference list

[1] R. Wehrens and L. M. C. Buydens, “Classical and Nonclassical Optimization Methods,” in *Encyclopedia of Analytical Chemistry*, 2006.

[2] R. A. Meyers, “Classical and Nonclassical Optimization Methods Classical and Nonclassical Optimization Methods 1 Introduction 1 1.1 Local and Global Optimality 2 1.2 Problem Types 2 1.3 Example Problem: Fitting Laser-induced Fluorescence Spectra 3 1.4 Criteria for Optimization 4 1.5 Multicriteria Optimization 4,” in *Encyclopedia of Analytical Chemistry*, Chichester: John Wiley & Sons Ltd, 2000, pp. 9678–9689.

[3] O. Akkus and E. Demir, “COMPARISON OF SOME CLASSICAL AND META-HEURISTIC OPTIMIZATION TECHNIQUES IN THE ESTIMATION OF THE LOGIT MODEL PARAMETERS,” *Int. J. Adv. Res*, vol. 4, no. 10, pp. 1026–1042.

[4] S. Bandaru and K. Deb, “Metaheuristic Techniques.” [Online]. Available: https://pdfs.semanticscholar.org/c28f/cda2bccc75b10e1d540031269e82a5d9c9b0.pdf?_ga=2.130438726.859446823.1553504960-1592470437.1516282346. [Accessed: 20-Jan-2019].

[5] I. Fister, X.-S. Yang, I. Fister, J. Brest, and D. Fister, “A Brief Review of Nature-Inspired Algorithms for Optimization,” *Elektroteh. Vestn.*, vol. 80, no. 3, pp. 1–7, 2013.

[6] S. Binitha and S. Siva Sathya, “A Survey of Bio inspired Optimization Algorithms 138,” *Int. J. Soft Comput. Eng.* , vol. 2, no. 2, pp. 137–151, 2012.

[7] J. Radosavljevic, *Metaheuristic Optimization in Power Engineering*. Institution of Engineering and Technology, 2018.

[8] M. Gavrilas, “Heuristic and metaheuristic optimization techniques with application to power systems,” in *Proceedings of the 12th WSEAS international conference on Mathematical methods and computational techniques in electrical engineering*, 2010, pp. 95–103.

[9] “Power Systems Analysis Software - Open Electrical.” [Online]. Available: https://wiki.openelectrical.org/index.php?title=Power_Systems_Analysis_Software. [Accessed: 28-Feb-2019].

[10] “EPRI | Smart Grid Resource Center &gt; Simulation Tool – OpenDSS.” [Online]. Available: http://smartgrid.epri.com/SimulationTool.aspx. [Accessed: 01-Mar-2019].

[11] “modeFrontier | www.esteco.com.” [Online]. Available: https://www.esteco.com/modefrontier. [Accessed: 21-Mar-2019].

[12] “HEEDS Design Exploration Software.” [Online]. Available: https://www.redcedartech.com/. [Accessed: 21-Mar-2019].

[13] “Global Optimization Toolbox - MATLAB.” [Online]. Available: https://www.mathworks.com/products/global-optimization.html. [Accessed: 08-Mar-2019].

[14] D. Hadka and P. Reed, “Borg: An Auto-Adaptive Many-Objective Evolutionary Computing Framework,” *Evol. Comput.*, vol. 21, no. 2, pp. 231–259, May 2013.

[15] “MIDACO-SOLVER.” [Online]. Available: http://www.midaco-solver.com/. [Accessed: 28-Mar-2019].

[16] https://www.scipy.org, “SciPy.org — SciPy.org.” [Online]. Available: https://www.scipy.org/. [Accessed: 28-Feb-2019].

[17] “SciPy.optimize.” [Online]. Available: https://docs.scipy.org/doc/scipy/reference/tutorial/optimize.html. [Accessed: 28-Jan-2019].

[18] “scipy.optimize.differential_evolution — SciPy v1.2.1 Reference Guide.” [Online]. Available: https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.differential_evolution.html. [Accessed: 03-Jan-2019].

[19] “EPRI Home.” [Online]. Available: https://www.epri.com/#/?lang=en-US. [Accessed: 15-Jan-2019].

[20] R. Dugan and D. Montenegro, “OpenDSS manual,” 2018. [Online]. Available: https://sourceforge.net/projects/electricdss/files/OpenDSS/OpenDSSManual.pdf/download. [Accessed: 03-Mar-2019].

[21] “OpenDSS / Code / [r2573] /trunk/Distrib/Doc.” [Online]. Available: https://sourceforge.net/p/electricdss/code/HEAD/tree/trunk/Distrib/Doc/. [Accessed: 07-Mar-2019].

[22] “OpenDSS / Code / [r2573] /trunk/Distrib/Doc/OpenDSS Solution Interface.pdf.” [Online]. Available: https://sourceforge.net/p/electricdss/code/HEAD/tree/trunk/Distrib/Doc/OpenDSS Solution Interface.pdf. [Accessed: 07-Mar-2019].

[23] “OpenDSS / Code / [r2573] /trunk/Distrib/Doc/OpenDSS Cheatsheet.pdf.” [Online]. Available: https://sourceforge.net/p/electricdss/code/HEAD/tree/trunk/Distrib/Doc/OpenDSS

Cheatsheet.pdf. [Accessed: 07-Mar-2019].

[24] "OpenDSS / Code / [r2573] /trunk/Distrib/Doc/OpenDSS Circuit Interface.pdf." [Online]. Available: https://sourceforge.net/p/electricdss/code/HEAD/tree/trunk/Distrib/Doc/OpenDSS Circuit Interface.pdf. [Accessed: 07-Mar-2019].

[25] "OpenDSS-G download | SourceForge.net." [Online]. Available: https://sourceforge.net/projects/dssimpc/. [Accessed: 25-Feb-2019].

[26] "OpenDSSDirect.py 0.3.1." [Online]. Available: https://dss-extensions.org/OpenDSSDirect.py/. [Accessed: 19-Mar-2019].

[27] J. Andersson, "A survey of multiobjective optimization in engineering design," 2000.

[28] "EMOO Home Page." [Online]. Available: http://neo.lcc.uma.es/emoo/. [Accessed: 09-Jan-2019].

[29] L. S. de Oliveira and S. F. P. Saramago, "Multiobjective optimization techniques applied to engineering problems," *J. Brazilian Soc. Mech. Sci. Eng.*, vol. 32, no. 1, pp. 94–105, Mar. 2010.

[30] M. O. W. Grond, N. H. Luong, J. Morren, and J. G. Slootweg, "Multi-objective optimization techniques and applications in electric power systems," in *2012 47th International Universities Power Engineering Conference (UPEC)*, 2012, pp. 1–6.

[31] "pymoo." [Online]. Available: https://www.egr.msu.edu/coinlab/blankjul/pymoo/. [Accessed: 19-Jan-2019].

[32] "DEAP documentation — DEAP 1.2.2 documentation." [Online]. Available: https://deap.readthedocs.io/en/master/.

[33] "Pagmo &amp; Pygmo — pagmo 2.11 documentation." [Online]. Available: https://esa.github.io/pagmo2/. [Accessed: 01-Mar-2019].

[34] "Platypus - Multiobjective Optimization in Python — Platypus documentation." [Online]. Available: https://platypus.readthedocs.io/en/latest/. [Accessed: 29-Jan-2019].

[35] X.-S. Yang, Ed., *Nature-Inspired Algorithms and Applied Optimization*, vol. 744. Cham: Springer International Publishing, 2018.

[36] X.-S. Yang and X. He, "Nature-Inspired Optimization Algorithms in Engineering: Overview and Applications," 2016, pp. 1–20.

[37] M. Barukčić, Ž. Hederić, and T. Benšić, "Analytical estimation of switched reluctance motor flux linkage profile by using evolutionary algorithm and numerical simulations," *J. energy Technol.*, vol. 10, no. 2, pp. 19–34, Jun. 2017.

[38] M. Barukcic, M. Zgela, and D. Buljic, "The Python-OpenDSS co-simulation for the evolutionary multiobjective optimal allocation of the single tuned passive power filters," in *2017 International Conference on Smart Systems and Technologies (SST)*, 2017, pp. 209–215.

[39] F. Halak, T. Bensic, and M. Barukcic, "Induction motor variable inductance parameter identification," in *2017 International Conference on Smart Systems and Technologies (SST)*, 2017, pp. 315–320.

[40] N. B. Raicevic, S. R. Aleksic, Ž. Hederic, M. Barukcic, and I. Iatcheva, "Optimal selection of coaxial ring systems in environmental electrostatic shielding," *COMPEL - Int. J. Comput. Math. Electr. Electron. Eng.*, vol. 37, no. 4, pp. 1418–1435, Jul. 2018.

[41] M. Barukči´, B.-S. Nikolovski -Franjo, and J. Jovi´c, “HYBRID EVOLUTIONARY-HEURISTIC ALGORITHM FOR CAPACITOR BANKS ALLOCATION,” *J. Electr. Eng.*, vol. 61, no. 6, pp. 332–340, 2010.

[42] M. Barukčić, Ž. Hederić, and M. Hadžiselimović, “Determination of optimal capacitor bank allocation by an adapted evolutionary algorithm with use of constraint fuzzification,” *J. Energy Technol.*, vol. 4, no. 1, pp. 23–38, Jan. 2008.

[43] M. Barukcic, P. Maric, and S. Nikolovski, “Applying an evolutionary strategy for multiobjective optimization of capacitor banks allocation in distribution feeders,” in *Eurocon 2013*, 2013, pp. 1262–1269.

[44] M. Barukcic, Z. Hederic, and K. Miklosevic, “Multi objective optimization of energy production of distributed generation in distribution feeder,” in *2014 IEEE International Energy Conference (ENERGYCON)*, 2014, pp. 1325–1333.

[45] M. Barukčić, Ž. Hederić, and Ž. Špoljarić, “The estimation of I – V curves of PV panel using manufacturers’ I – V curves and evolutionary strategy,” *Energy Convers. Manag.*, vol. 88, pp. 447–458, Dec. 2014.

[46] D. Bilandžija, M. Barukčić, and V. Ćorluka, “Using an Evolutionary Algorithm for Harmonic Load Modeling by Norton and Thevenin Equivalents,” *Int. J. Electr. Comput. Eng. Syst.*, vol. 5, no. 2, pp. 39–45, Apr. 2015.

[47] M. Barukcic, M. Vukobratovic, and T. Bensic, “Evolutionary optimization approach for performing interval power flow considering uncertainties in electric power systems,” in *2016 International Conference on Smart Systems and Technologies (SST)*, 2016, pp. 185–190.

[48] M. Barukcic, M. Vukobratovic, D. Masle, D. Buljic, and Z. Herderic, “The evolutionary optimization approach for voltage profile estimation in a radial distribution network with a decreased number of measurements,” in *2017 15th International Conference on Electrical Machines, Drives and Power Systems (ELMA)*, 2017, pp. 26–31.