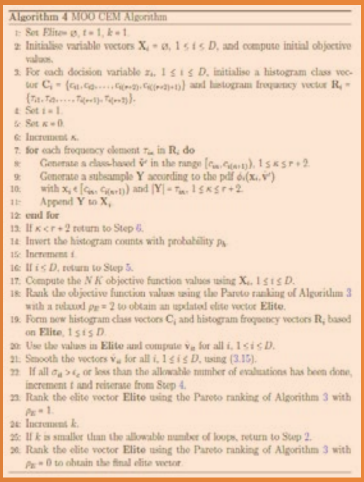
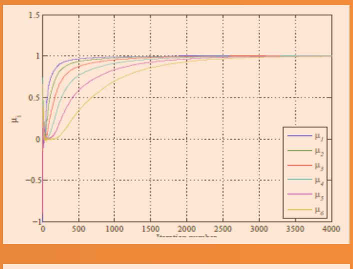
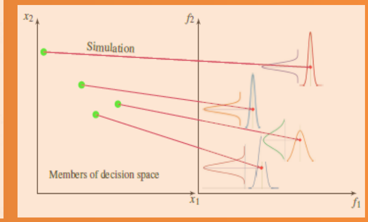


Algorithm 4 MOO CEM Algorithm

```
1: Set Elite= $\varnothing$, $t = 1$, $k = 1$.
2: Initialise variable vectors $\mathbf{X}_i = \varnothing$, $1 \le i \le D$, and compute initial objective values.
3: For each decision variable $x_i$, $1 \le i \le D$, initialise a histogram class vector $\mathbf{C}_i = \{c_{i1}, c_{i2}, \dots, c_{i(r+2)}, c_{i((r+2)+1)}\}$ and histogram frequency vector $\mathbf{R}_i = \{\tau_{i1}, \tau_{i2}, \dots, \tau_{i(r+1)}, \tau_{i(r+2)}\}$.
4: Set $i = 1$.
5: Set $\kappa = 0$.
6: Increment $\kappa$.
7: for each frequency element $\tau_{i\kappa}$ in $\mathbf{R}_i$ do
8:     Generate a class-based $\hat{\mathbf{v}}^r$ in the range $[c_{i\kappa}, c_{i(\kappa+1)})$, $1 \le \kappa \le r+2$.
9:     Generate a subsample $\mathbf{Y}$ according to the pdf $\phi_i(\mathbf{x}_i, \hat{\mathbf{v}}^r)$
10:    with $\mathbf{x}_i \in [c_{i\kappa}, c_{i(\kappa+1)})$ and $|\mathbf{Y}| = \tau_{i\kappa}$, $1 \le \kappa \le r+2$.
11:    Append $\mathbf{Y}$ to $\mathbf{X}_i$
12: end for
13: If $\kappa < r+2$ return to Step 6.
14: Invert the histogram counts with probability $p_h$.
15: Increment $i$.
16: If $i \le D$, return to Step 5.
17: Compute the $N\,K$ objective function values using $\mathbf{X}_i$, $1 \le i \le D$.
18: Rank the objective function values using the Pareto ranking of Algorithm 3 with a relaxed $\rho_E = 2$ to obtain an updated elite vector Elite.
19: Form new histogram class vectors $\mathbf{C}_i$ and histogram frequency vectors $\mathbf{R}_i$ based on Elite, $1 \le i \le D$.
20: Use the values in Elite and compute $\hat{\mathbf{v}}_{it}$ for all $i$, $1 \le i \le D$.
21: Smooth the vectors $\hat{\mathbf{v}}_{it}$ for all $i$, $1 \le i \le D$, using (3.15).
22: If all $\sigma_{it} > \epsilon_c$ or less than the allowable number of evaluations has been done, increment $t$ and reiterate from Step 4.
23: Rank the elite vector Elite using the Pareto ranking of Algorithm 3 with $\rho_E = 1$.
24: Increment $k$.
25: If $k$ is smaller than the allowable number of loops, return to Step 2.
26: Rank the elite vector Elite using the Pareto ranking of Algorithm 3 with $\rho_E = 0$ to obtain the final elite vector.
```





6. Optimisation with simulation

In the previous chapter we learnt that one can use ANOVA or the K-N method, among others, to find the best scenario from a small number of scenarios in a simulation study. In this chapter we address the problem when we have a large decision space to investigate with simulation – spaces that are so large it would be impossible, due to time, to define and evaluate all solutions in search of the optimum.

6.1 Introduction to simulation-optimisation

Many simulation problems are of *combinatorial nature*, *i.e.* the decision variables can take on many values, resulting in a large decision space. The Buffer-allocation problem (BAP) is a good example of a combinatorial optimisation problem. Suppose we have four buffers, and we allow the size to vary between one and 10, all inclusive, then the decision space consists of 10^4 possibilities. If we want to evaluate each of these, we need to run $10^4 \times n^*$ replications. This could result in very long, impractical simulation times. Suppose the BAP takes one minute to run its n^* replications for a given buffer allocation, then it would take almost seven days to cover the full decision space of this example. Of course, if we have strong computational capability, the time can be reduced.

In optimisation we may use *exact algorithms* and *heuristics*. The exact algorithm finds the guaranteed optimal solution in finite time, provided the problem is not too large in terms of the number of decision variables. Linear programming is an example of an exact method, provided that certain assumptions are adhered to. If we cannot formulate a problem to solve it using exact methods, we often revert to simulation, and we call this ***simulation-optimisation*** (SO). In simple terms we can say that the simulation model becomes the '$f(x)$' of an optimisation problem, *i.e.* it is the objective function, and it represents the complex but unknown relationships among decision variables.

To address this problem, meta-heuristics are used in conjunction with simulation to intelligently search the decision space and find good answers while evaluating combinations of decision variables. Because the computational time required to solve many real-life problems with large decision spaces may grow extremely quickly as the problem size increases, it is often better to try to find a solution for these problems that is approximate and almost as good as the best answer than to try to find the absolute best answer. Techniques that yield good, though not necessarily optimal solutions, are known as *heuristic techniques*. The word 'heuristic' is from the Greek word 'heuriskein', which means to search. Heuristics are problem-specific, while an extension of them, namely *meta-heuristics*, is more generic and can solve a wide variety of problems, often without the need to understand the problem in detail. The term *meta-heuristic* is defined by Sörensen (2015) as

Definition 6. *A metaheuristic is a high-level problem-independent algorithmic framework that provides a set of guidelines or strategies to develop heuristic optimization algorithms. The term is also used to refer to a problem-specific implementation of a heuristic optimization algorithm according to the guidelines expressed in such a framework.*

For a discussion and formal definition, see Fred Glover (2018). A meta-heuristic thus uses a heuristic ('search guidance' if you want to) in a specific way, hence the term *meta* which means 'high-level' or 'beyond'. Meta-heuristics reduce the number of decision variable combinations so that a smaller subset of the decision space needs to be explored to find good objective function values. The meta-heuristic provides values for the decision variables in the simulation model, the model runs a number of replications with these settings, then returns point estimators for the output parameters (the '$f(x)$'s). This process is repeated a number of times, until the meta-heuristic (usually) converges. If at least one of the decision variables is stochastic, we can only estimate "$f(x)$", and in particular, $\mathbb{E}[f(x)]$.

Rosen, Harmonosky, and Traband (2007) define the traditional simulation optimisation problem as

$$\begin{aligned} &\text{Minimise} \quad && f(\theta) && (6.1)\\ &\text{subject to} \quad && \theta \in \Theta, && (6.2) \end{aligned}$$

where $f(\theta) = E[\psi(\theta, \xi)]$ is the expected system performance value, and is estimated by $\hat{f}(\theta)$ from samples of a simulation model using instances of discrete or continuous feasible and possibly constrained input $\theta \in \Theta \subset \mathbb{R}^D$. The stochastic elements of the model are represented by ξ.

The search algorithm mentioned previously uses the current response value(s) to adjust the control variables (parameters) towards a (hopefully) better solution, and it terminates when the response value(s) do not show significant improvement. The search algorithms contain procedures to prevent the algorithms getting stuck into local optima. Once the search algorithm terminates, the values of the control variables are considered the near-optimum combination of values.

Many simulation packages now include optimisation procedures: Simio is integrated with OptQuest, which uses scatter- and tabu searches as well as neural networks, while the WITNESS Optimizer includes simulated annealing and tabu searches.

The simulation-optimisation process using a metaheuristic is shown in Figure 6.1. A trajectory-based optimisation (*e.g.* simulated annealing) works similarly, but no population is used.

■ **Example 6.1** Consider the 7-11 example to illustrate the size of the decision space (Michalewicz and Fogel, 2004). Here is a story: I went to the 7-11 shop last night and bought four items. I noticed the cashier multiplied the prices of the four items, then she said I owe her R7-11. I objected and insisted that she added the four prices; she did so and again found that I owe her R7-11. What were the prices of the four items?

To answer this, first note that the prices can be expressed in cents, so the problem is discrete and the prices can range from 1¢ to 708¢. Let the prices be X_i, $i = 1,2,3,4$. The following

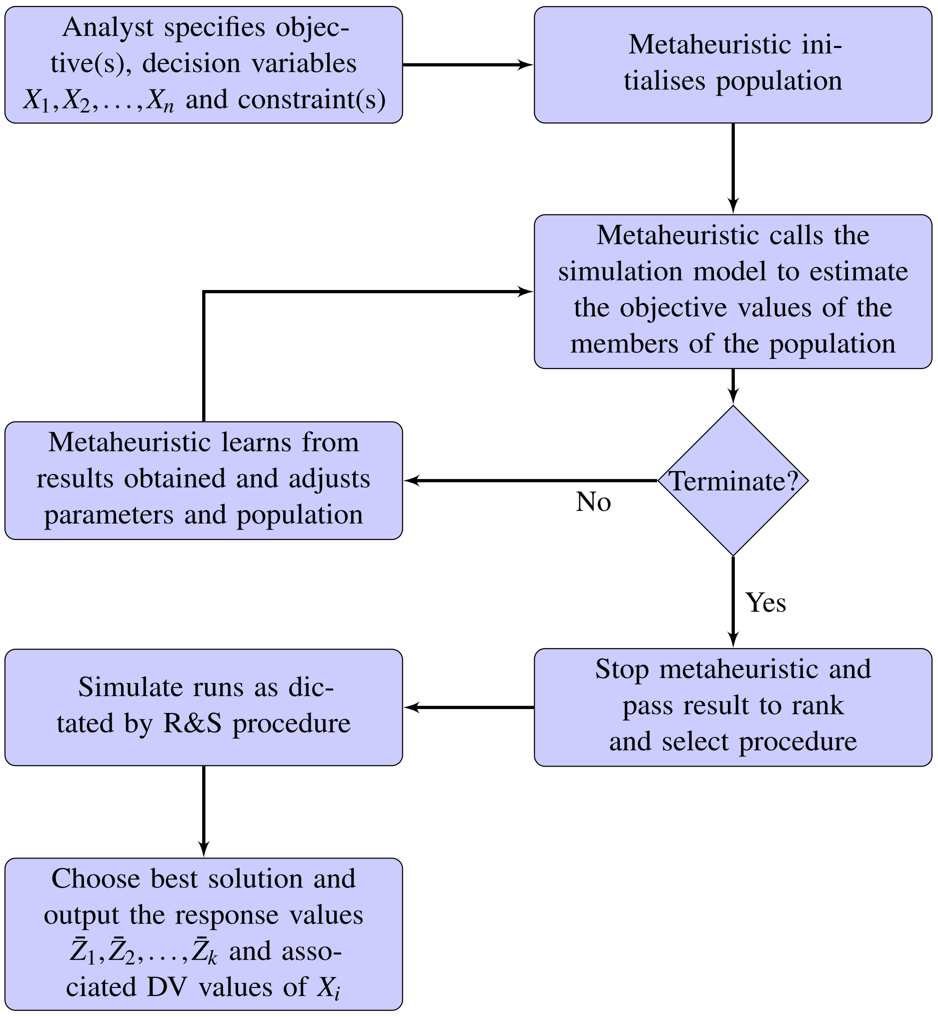

Analyst specifies objective(s), decision variables $X_1, X_2, \ldots, X_n$ and constraint(s)
Metaheuristic initialises population
Metaheuristic calls the simulation model to estimate the objective values of the members of the population
Terminate?
No
Metaheuristic learns from results obtained and adjusts parameters and population
Yes
Stop metaheuristic and pass result to rank and select procedure
Simulate runs as dictated by R&S procedure
Choose best solution and output the response values $\bar{Z}_1, \bar{Z}_2, \ldots, \bar{Z}_k$ and associated DV values of X_i

Figure 6.1: Simulation-optimisation process with a population-based metaheuristic

equations must be satisfied:

$$\begin{aligned}\sum_{i=1}^{4} X_i &= 711 \text{ and}\\ \prod_{i=1}^{4} X_i &= 711\,000\,000.\end{aligned}$$

Since a product must have a price, the minimum value for X_i is 1¢, and therefore the maximum price is 708¢. The size of the decision space of this problem is thus $708 \times 708 \times 708 \times 708 = 708^4 = 251\,265\,597\,696 > 2.5 \times 10^{11}$. This is a large space to search, and if each solution is calculated, it could take a long time to find the answer. We call problems of this nature *combinatorial problems*, because we have to use different combinations of decision variable values to find the optimum. (The answer is X_1=R1.20, X_2=R1.25, X_3=R1.50, X_4=R3.16, you can determine it at home.)

There are many meta-heuristics available, and many of them are nature-inspired. The best-known meta-heuristic is the genetic algorithm (GA, Goldberg (1989)), which will be used in this module. It is a built-in feature of Tecnomatix. Other examples of meta-heuristics are

- Simulated annealing,

- Ant colony optimisation,
- Tabu search,and
- Particle swarm optimisation.

A list of these algorithms is available at `https://en.wikipedia.org/wiki/List_of_metaphor-based_metaheuristics`.

6.2 Some optimisation problems

A few (nasty) optimisation test problems, for GAs and other meta-heuristics are now presented. These problems are included here to show the reader some of the challenging shapes of optimisation problems that meta-heuristics have to deal with.

6.2.1 Rosenbrock function

The Rosenbrock function is an accepted benchmark for optimisation algorithms. It must be minimised and one variant is defined as

$$f(\mathbf{x}) = \sum_{i=1}^{D-1} [100(x_{i+1} - x_i^2)^2 + (x_i - 1)^2] \tag{6.3}$$

with D variables $x_1, \ldots, x_D$, where $-2 \leq x_i \leq 2$ for all $i = 1, \ldots, D$. A plot for $D = 2$ is shown in Figure 6.2. The exact minimum is a vector of ones, that is $\mathbf{x}^* = (1,1,1,\ldots,1)$, with $f(\mathbf{x}^*) = 0$, while a local minimum exists at $(-1,1,1,\ldots,1)$ for $4 \leq D \leq 7$.

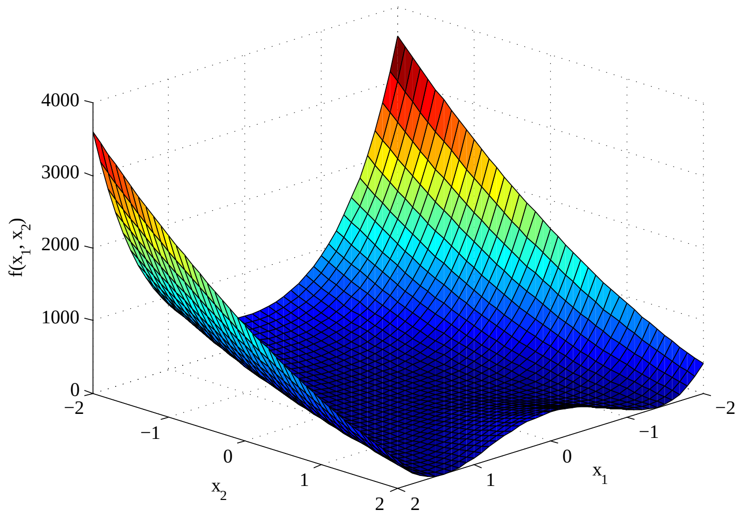


Figure 6.2: The Rosenbrock function with $D = 2$ and $-2 \leq x_i \leq 2$

The function has a deceptive optimum at (0,0), while many other coordinates seem to be optimal. They are, however, all local optima.

6.2.2 Rastrigin function

The Rastrigin function has many local optima. It is the function

$$f(\mathbf{x}) = 10D + \sum_{i=1}^{D} [x_i^2 - 10(\cos(2\pi x_i))], \tag{6.4}$$

which has to be minimised. A plot for the case of $D = 2$ decision variables, where $-5.12 \leq x_i \leq 5.12$ for $i = 1,2$, is shown in Figure 6.3. The absolute minimum is at $(0,0)$ with $f(0,0) = 0$.

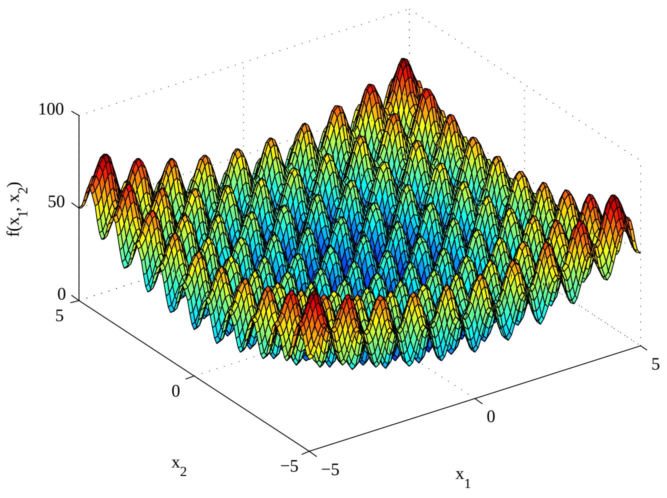


Figure 6.3: Rastrigin function with $D = 2$

6.2.3 Shekel function

The Shekel function

$$f(\mathbf{x}) = -\sum_{i=1}^{m_s} \frac{1}{c_i + \sum_{j=1}^{D} (x_j - a_{ij})^2} \tag{6.5}$$

is another widely accepted benchmark for evaluating optimisation algorithms. It has to be minimised and depends on D variables $x_1, \ldots, x_D$ with $0 \le x_i \le 10$, m_s being the number of local minima and can take, by definition, values of 5, 7 or 10.

The values of a_{ij} and c_i are determined via an algorithm for each problem instance. A plot of $-f(\mathbf{x})$ for the case $m_s = 10$ and $D = 2$ is shown in Figure 6.4 (the negative of the function is plotted for clarity). The absolute minimum of $f(\mathbf{x})$ is at $(4,4)$ with $f(4,4) = -11.0298$.

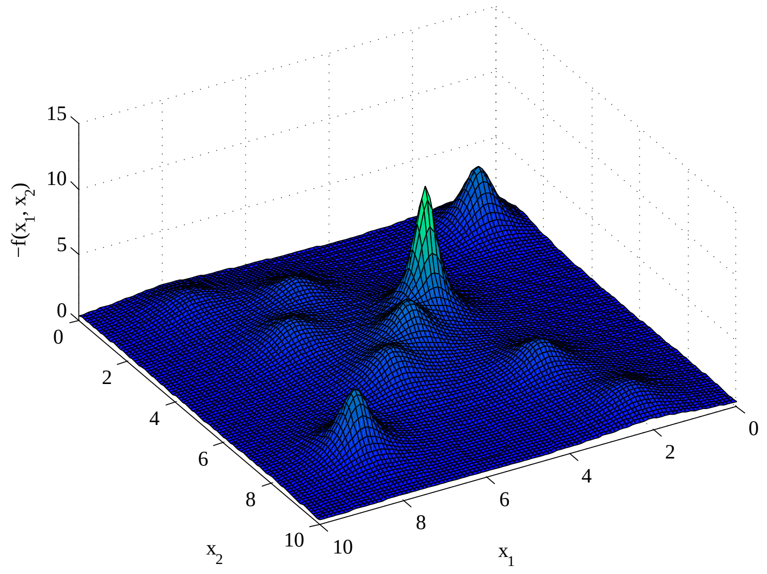


Figure 6.4: Negative Shekel function with 10 peaks

Tecnomatix includes a GA for optimisation, and we must understand the GA principles in order to do simulation-optimisation. The interested student may consult books by Goldberg (1989) and Mitchell (1997) for more information on GAs.

6.3 The genetic algorithm

Genetic algorithms are based on the biological principle of evolution with the survival of the fittest being a fundamental property. It is an iterative procedure that operates on a constant-size population. The GA was developed by John Holland and others (Mitchell (1997), Goldberg (1989)).

When applied to an optimisation problem the analogy is as follows: A *population* of possible solutions is created from the problem's *solution space* by varying the values of the *decision variables* (search space). Each individual (referred to as a *chromosome*) in the population is an *encoding* of a solution. Some of the individuals combine with each other to form new individuals or *offspring*, similar to parents producing children. Good individuals are more likely to be used during this combination phase (*crossover* phase). The process is also subject to random variation (known as *mutation*) where some part of the chromosome is randomly changed. The new (on average 'better' or 'fitter') individuals replace some or all of the individuals in the old population thus causing the population of solutions to improve over time. A given population represents a *generation*, and once new (improved) individuals are added to the population, a new generation is created. This process of combination and replacement continues until a sufficiently good solution has evolved.

The GA thus effectively searches in the decision space of the problem, and learns from the outcomes in the solution space to adjust the search. The improvement of the population is associated with forming chromosomes that drive the fitness function to a near-optimum (approximate) value. [Note: 'approximate' means we think it is a good guess of the true value of the solution, while 'approximation' means we know (we can prove it) how close to the true value of the solution we are.]

The GA is included under the term *evolutionary algorithms*, together with evolution strategies, evolutionary programming and genetic programming (GP (2018), MathWorks (2018)). There are many variations of GAs, while other types of optimisation algorithms also exist, including population-based incremental learning (PBIL), simulated annealing and neural networks. Each algorithm is more suitable or less suitable for a specific problem. Problems addressed in industrial engineering include complex scheduling problems, layout problems, vehicle routing problems and multi-objective optimisation problems. When exact techniques like linear programming and goal programming can address a problem (they can find the exact solution in finite time), we should use these rather than meta-heuristics.

The pseudo-code for a GA is outlined in Algorithm 1. There are many variants of the code given, but the principles are similar. A description of the components follows.

Problem representation

We recognise that our objective function, *i.e.* the function we want to maximise/minimise is a function of several decision variables that we may change until the objective reaches an accepted level of near-optimality. We may *e.g.* want to schedule a number of tasks on a finite number of machines so that the time to produce a number of products is the minimum.

To utilise a GA, we have to *encode* our decision variables to a form that facilitates the GA processes of *crossover* and *mutation*. The obvious question is how to encode a set of decision variables (which translates back to a solution of the objective function). One way to encode a solution is to represent it as a binary string (*i.e.* a string of 0's and 1's). If, for example, we were searching for an integer value that maximises the function

$$f(x) = x^3 - 60x^2 + 900x + 100 \tag{6.6}$$

Algorithm 1 Basic Genetic Algorithm

```
1: Generate an initial random population and evaluate members. Set $t = 1$.
2: Repeat
3:   If the cross-over probability is satisfied
4:     Randomly select two individuals to reproduce
5:     Do cross-over between the members
6:   Else
7:     Choose any of the parents to be the offspring
8:     Replace worst individual in population by new offspring
9:   End If
10:  For Each gene in the offspring
11:    If mutation probability is satisfied
12:      Flip value of gene
13:    End If
14:  End For Each
15:  Evaluate offspring with the objective function
16:  If offspring is better than least fit member in population
17:    Replace the least fit member with the offspring
18:  End If
19:  $t \leftarrow t + 1$
20: Until Termination condition
```

in the range between 0 and 31, with x the only decision variable, we could code the value of x as a binary string of length 5. For example, the string “01010” corresponds to the value $x = 10$. If we had a problem involving many variables then we would represent the decision variables as one long binary string. If we wanted to work with fractions then we could allocate extra bits to get the required precision. Note that the search space in this case is linear and discrete, *i.e.* all integers from 0 to 31.

As mentioned before, a specific encoded string of our set of decision variables is called a *chromosome*. In the simple example above, the chromosome consists of an encoding for x only, so that $x = 5$ is represented by the chromosome “00101”, while $x = 19$ is represented by chromosome “10011”. The components of a chromosome are referred to as *genes*. In a binary representation, each 0 or 1 is equivalent to a gene. If we have, say, three decision variables represented by x, y and z, then we encode them as a string:

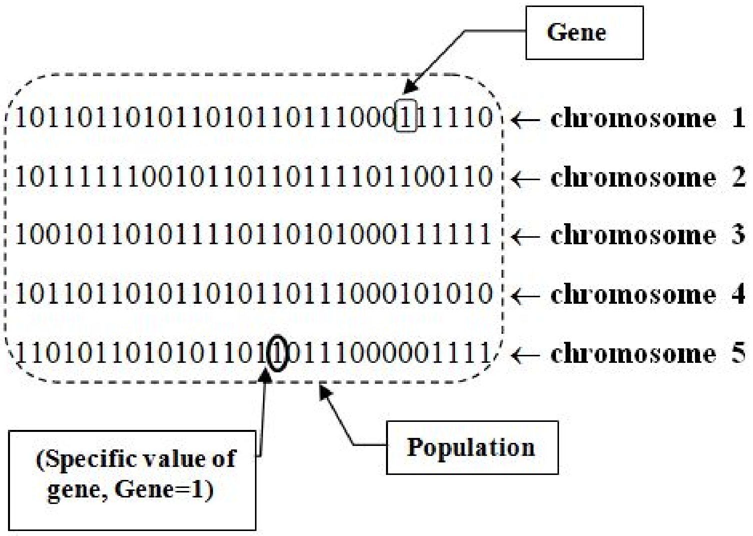
$$\underbrace{1001011010}_{x(=602)}\ \underbrace{0111011011}_{y(=475)}\ \underbrace{1000111110}_{z(=574)} \leftarrow \text{chromosome}$$

The vertical lines in between are added to distinguish between each variable’s representation; the representation is thus one single string consisting of many ‘0’s and ‘1’s. Note that it is implied that each variable is represented by 10 bits, *i.e.* it consists of 10 genes, the latter assuming values of either ‘0’ or ‘1’.

Initial population and population size

The GA is initialised by forming a *population* of initial chromosomes, either via a random or a heuristic process. The population is thus made up of a collection of chromosomes. The *size of the population* is another parameter of the GA. In general, a larger population size will increase the likelihood of finding very good solutions. However, it will also increase the computation time. Typically, population sizes between 30 and 100 chromosomes are used.

Initially, the chromosomes need to be filled with data before the GA can operate on them. This can be done using random values for each bit in a gene by assigning it a value of ‘0’ or ‘1’ with a probability of 0.5 in each case. A population of five chromosomes consisting of three decision variables, each 10 bits (genes) long, may thus look as follows:


Gene
101101101011010110111000111110 ← chromosome 1
101111110010110110111101100110 ← chromosome 2
100101101011110110101000111111 ← chromosome 3
101101101011010110111000101010 ← chromosome 4
110101101010110101011100000111 1 ← chromosome 5
(Specific value of gene, Gene=1)
Population

A specific instance of the population is referred to as a *generation*.

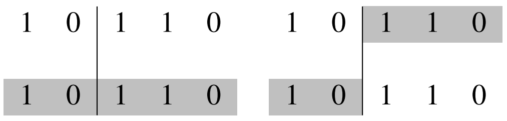
Evaluating the population

Every GA needs a so-called *fitness function* (objective function) to evaluate the fitness of the chromosomes in the population. In the example in (6.6), $f(x)$ is the fitness value, because it is this function that we want to maximise; the value of this function for each chromosome also drives the improvement of the GA. We thus need to decode the chromosomes into values for the decision variables again so that the fitness function can be evaluated with these values. Decoding thus comprises the translation of the binary substrings into representations understandable by man, for example real numbers.

Now suppose that we are using a genetic algorithm to try to determine the best schedule of tasks for a group of machines in a factory. In this case we might want to minimise the time it takes to complete the tasks and the fitness function can be the completion time of the last task.

Crossover

The crossover phase combines the genes of two chromosomes. There are several methods of performing crossover. The common method of *one-point crossover* works as follows (a bit different from what is shown in Algorithm 1). Two chromosomes (parents) are randomly chosen from the population, then, of these two, the one with the best objective function value is selected. Another chromosome (parent) is again randomly selected and it replaces the second chromosome. A random position, the crossover point, is then chosen between two genes. Crossover forms two offspring, the first of which is formed by combining the first parent’s genes that are before the crossover point, with the second chromosome’s genes that follow after the crossover point. The second chromosome is formed in the reverse manner. The next schematic depicts one-point crossover and uses grey colours to see how the combinations are formed.



Crossover is applied to pairs of chromosomes as determined by the selection mechanism of the GA (explained later). A *crossover probability* is applied to these pairs of chromosomes to determine if they are to be subjected to crossover. This crossover probability is a parameter that is set by the decision-maker. Typical crossover probabilities are in the region 0.7 – 1. Crossover is required if we choose a random number U so that $U \leq P(\text{Crossover})$.

■ **Example 6.2** Suppose $U = 0.451$, and $P(\text{Crossover}) = 0.8$, then we would do crossover, since $U \leq P(\text{Crossover})$.

Mutation

Mutation is an important part of a genetic algorithm because it introduces new genetic material into the population and prevents the algorithm from getting trapped in a local optimum. The method of flip mutation as applied to a binary coded chromosome changes the value of the contents of a gene to a '0' if it contained a '1', or it changes the gene content to a '1' if it contained a '0'.

Mutation is applied to a chromosome by considering each gene in turn and mutating the gene according to a certain probability (in other words each gene has this probability of being mutated). This mutation probability is a parameter of the algorithm. Typical mutation rates lie in the region 0.001 – 0.03. Mutation is applied to all offspring that are generated, but since the probability is low, only a few bits are flipped at a time. Mutation is done if we choose a random number U so that $U \leq P(\text{Mutation})$.

■ **Example 6.3** Suppose $U = 0.311$, and $P(\text{Mutation}) = 0.01$, then we would not do mutation on the specific gene (bit), since $U \geq P(\text{Mutation})$. An example of a mutated bit is shown below:

$$\begin{array}{cccccccccc}
1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 \\
 & & & & & \downarrow & & & & \\
1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0
\end{array}$$

Selection and replacement

The selection method used governs which chromosomes will be selected for crossover. There are several selection methods, for example roulette wheel, Boltzman, tournament and rank selection. The method of tournament selection works in the following way: Two (or more) different chromosomes are selected at random from the population. Of these selected chromosomes, the chromosome with the best fitness function is selected to be the first parent for crossover. The second parent is selected randomly from the population.

The replacement method determines how the offspring are to be inserted into the population. Some types of genetic algorithms replace the entire population with new offspring while others keep some chromosomes from the old population. One method of replacement is to replace only one of the members of the population. GAs that employ this method of replacement are known as *incremental GAs*. Only one offspring is generated in each generation and it replaces a chosen chromosome in the population. This chromosome is chosen in many ways, including

- Replace the chromosome with the worst fitness value in the population.
- Randomly replace a chromosome, which implies that a "good" chromosome can be replaced. This improves diversity of the population but results in possible longer times to convergence.

It does not matter which of the two offspring generated from crossover are used for this purpose because crossover chooses the parents in a random manner in any case.

Stopping conditions

The stopping conditions determine when the algorithm terminates. The algorithm could terminate after a certain number of iterations have been completed, which is called a hard stopping

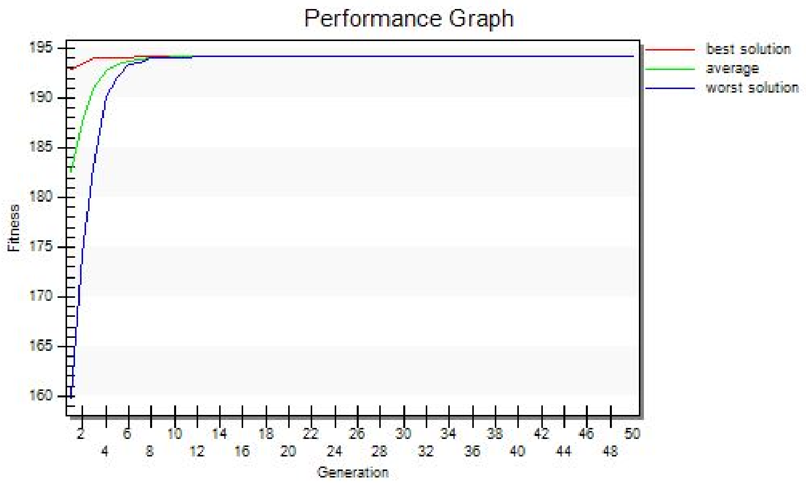


Figure 6.5: Generation progress of a GA as developed by Tecnomatix

							Fitness
1	0	1	0	1	0	1	11.2
1	0	1	0	1	0	1	12.4
0	1	0	1	0	1	1	14.9
1	0	1	0	1	0	1	10.9
0	0	0	0	1	1	0	19.7
1	1	1	1	0	0	1	12.3
0	1	1	0	0	1	1	21.4
1	1	0	1	0	0	1	18.4
0	0	0	0	0	1	0	14.3
1	0	1	1	1	1	1	13.2
Average for this generation:							14.87

condition: the algorithm thus stops after T iterations have been executed. The algorithm may also terminate due to some other condition such as when the rate of improvement slows down. One method is to determine the average value of the fitness function for each generation. If the current iteration yields an average of the fitness function that is within $\delta\%$ of the average of the last n averages, then we terminate the algorithm. We should also add a second stopping criterion T, as above. That prevents the algorithm from running indefinitely.

The graph in Figure 6.5 shows a plot of the average of the fitness function per generation (this example was taken from Tecnomatix). After some generations we see that the average stabilises.

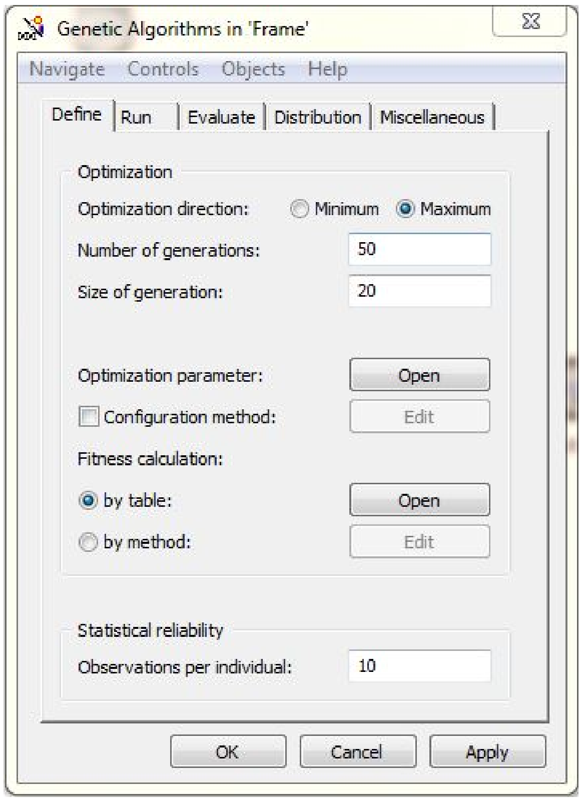
Assuming that a local optimum is not reached, we can determine

$$\left| \frac{\overline{M}_c - \overline{\overline{M}}_n}{\overline{M}_c} \right| \leq \delta$$

where

$\overline{M}_c$	$=$	Average of the objective function values in current generation.
$\overline{\overline{M}}_n$	$=$	Average of the previous n averages of the objective function.

Suppose we have the following population:


Genetic Algorithms in 'Frame'

Navigate Controls Objects Help

Define | Run | Evaluate | Distribution | Miscellaneous

Optimization

Optimization direction: Minimum [✓ Maximum]

Number of generations: 50

Size of generation: 20

Optimization parameter: Open

☐ Configuration method: Edit

Fitness calculation:

[✓ by table:] Open

by method: Edit

Statistical reliability

Observations per individual: 10

OK Cancel Apply

Figure 6.6: Tecnomatix GA Wizard settings

The value of 14.87 is the average for this generation, *i.e.* it is $\overline{M}_c$. We should thus record each average before replacing the current generation with the next. Also, if we decide to consider the averages of the previous n generations, we can only do so after we have formed at least $n+1$ generations.

We can now use the GA in conjunction with simulation to solve large problems and find us near-optimal solutions.

6.4 Application of GA principles in Tecnomatix

Several simulation packages use meta-heuristics for optimisation, and Tecnomatix uses the genetic algorithm. We shall now see how the concepts mentioned are implemented in Tecnomatix. We now learn how the wizard is used. The main fields for the GA settings are shown in Figure 6.6.

The decision-maker should know whether the optimisation is minimisation or maximisation – we typically minimise machine down-times or maximise throughput. The fields of the GA Wizard are subsequently related to some of the GA principles.

1. The **Number of generations** is the maximum number of times you will allow the GA to create a new, improved generation. A small number specified here will often prevent the GA enough opportunity to 'learn' from previous generations. A large number will be advantageous, but it could take a very long time before the simulation-optimisation terminates.
2. The **Size of the generation** specifies how many individuals you allow in your *population* (Tecnomatix means 'population' here). A population with only a few individuals will not exhibit sufficient diversity to allow the GA to find good answers. A large population is

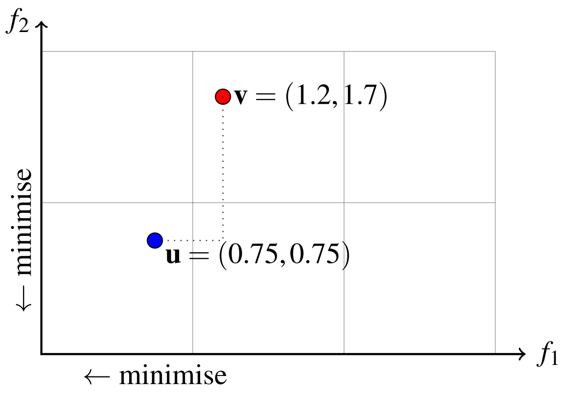


Figure 6.7: Vector dominance illustrated

good, but remember that each member must be evaluated using n^* replications. This can thus take long.

3. The **observations per individual** specifies your choice for n^*. When doing SO, one can usually not afford to enforce very small confidence intervals because of the resulting computational burden. This is a trade-off that the simulation analyst must make – if you have good computing power, you may increase the statistical reliability and the values for the GA optimisation parameters in 1. and 2. My experience has shown that it is better to assign more computing power to the number of generations and the size of the generation than to require ultimate statistical accuracy. (Why am I right?)
4. The **optimis[z]ation parameter** in Figure 6.6 refers to the set of decision variables (the X_i in Figure 6.1).
5. The **fitness calculation** refers to the set of response parameters (output values) (the $\overline{Z}_i$ in Figure 6.1). (Note that we must strictly-speaking refer to $\overline{\overline{Z}}_i$, because these are the parameters that we usually estimate.)

This concludes the brief introduction to simulation-optimisation. You should know how it works in principle, and also how to use the settings in the Tecnomatix GAWizard. Next, multi-objective optimisation with simulation is presented.

6.5 Multi-objective optimisation in large decision spaces with simulation

In multi-objective optimisation (MOO), two or more *conflicting* objectives are *simultaneously* optimised. The objectives are often *non-commensurate*, meaning they are measured in different units. Think, for example, of an investment: we want to *maximise* profit, but we also want to *minimise* the risk of losing money. Another example is: you want to buy as much airtime as possible but want to minimise the cost. MOO problems that can be solved do not yield a single best solution, but a set of optimal solutions called the *Pareto set*. We shall only deal with two objectives in this module, called *bi-objective optimisation*. For simplicity, we shall use 'multi-objective optimisation' throughout the text.

When doing MOO, we need to understand *vector dominance*. This is necessary to separate good and no-good solutions. Consider Figure 6.7 and the following definitions.

Definition 7. *Assuming minimisation, given two vectors* $\mathbf{u} = (u_1, \ldots, u_K)$ *and* $\mathbf{v} = (v_1, \ldots, v_K) \in \mathbb{R}^K$, *then* $\mathbf{u} \leq \mathbf{v}$ *if* $u_i \leq v_i$ *for* $i = 1, 2, \ldots, K$, *and* $\mathbf{u} < \mathbf{v}$ *if* $\mathbf{u} \leq \mathbf{v}$ *and* $\mathbf{u} \neq \mathbf{v}$.

Definition 8. *Given two vectors* $\mathbf{u}$ *and* $\mathbf{v}$ *in* $\mathbb{R}^K$, *then* $\mathbf{u}$ *dominates* $\mathbf{v}$ *(denoted by* $\mathbf{u} \prec \mathbf{v}$*) if* $\mathbf{u} < \mathbf{v}$.

Definition 9. *A vector of decision variables* $\mathbf{x}^* \in \Omega$ *(Ω is the feasible region) is Pareto optimal if there does not exist another* $\mathbf{x} \in \Omega$ *such that* $\mathbf{f}(\mathbf{x}) \prec \boldsymbol{f}(\mathbf{x}^*)$.

The concept of domination is illustrated with the simple Pareto front plot in Figure 6.8 (both objectives are to be minimised). Note that we now focus on the *solution space*.

Assuming the blue and red dots represent all possible solutions to the problem shown in Figure 6.8, then the blue dots represent members of the Pareto set, and these are not dominated by the red dots. At 'A', $f_1(x) = 2, f_2(x) = 2$, but at 'B', $f_1(x) = 1, f_2(x) = 1$, so the point at 'B' is far 'better' than the point at 'A'. Although $f_1(x)$ at 'A' is less than $f_1(x)$ at 'C', many other points have $f_1(x)$ less than that of 'A', so 'A' is dominated.

▶ In MOO, the aim is to find the blue dots ● and their associated green dots ● in Figure 6.9; the problem in this case has two decision variables x_1 and x_2, and two objectives, f_1 and f_2, both to be minimised. The blue dots represent non-dominated solutions and form the *Pareto set*. For a given solution (blue dot) there is an associated set of decision variable values (the green dots). These are the values at which a system must be operated to ensure near-optimality.

We can now state the MOO problem formulation.

6.6 Multi-objective optimisation problem formulation

The MOO problem with K objectives and $M+Q$ constraints are formulated as follows (Tsou, 2008):

$$\begin{aligned} \text{Minimise } \mathbf{f}(\mathbf{x}) &= [f_1(\mathbf{x}), f_2(\mathbf{x}), \ldots, f_K(\mathbf{x})]^T && (6.7)\\ \text{subject to } \mathbf{x} &\in \Omega && (6.8)\\ \Omega &= \{\mathbf{x} \mid g_i(\mathbf{x}) \le 0, i = 1,2,\ldots,M; && (6.9)\\ h_j(\mathbf{x}) &= 0, j = 1,\ldots,Q\}. && (6.10) \end{aligned}$$

The symbol $\mathbf{f}$ implies a vector, *i.e.* two or more objective functions, while the symbol $\mathbf{x}$ implies a vector of decision variables. In (6.8), the feasible region is defined. The constraints are defined in (6.9), which are the inequality constraints, and the equality constraints are defined in (6.10).

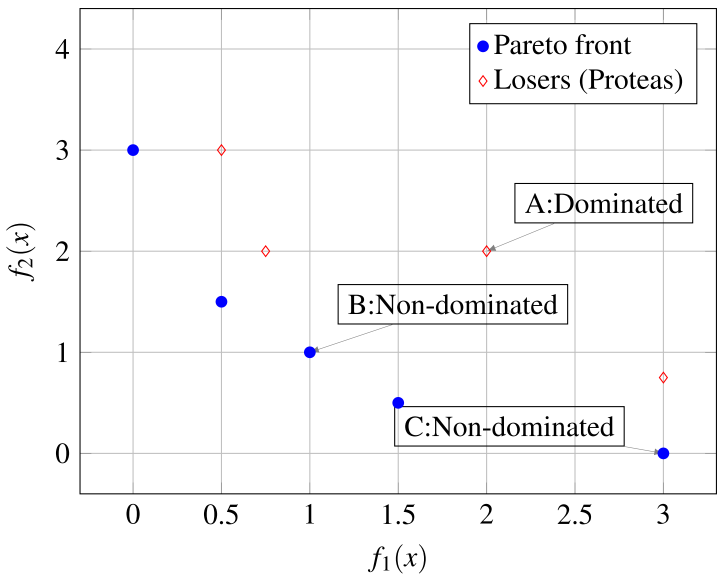


Figure 6.8: Concept of domination in MOO (both objectives to be minimised)

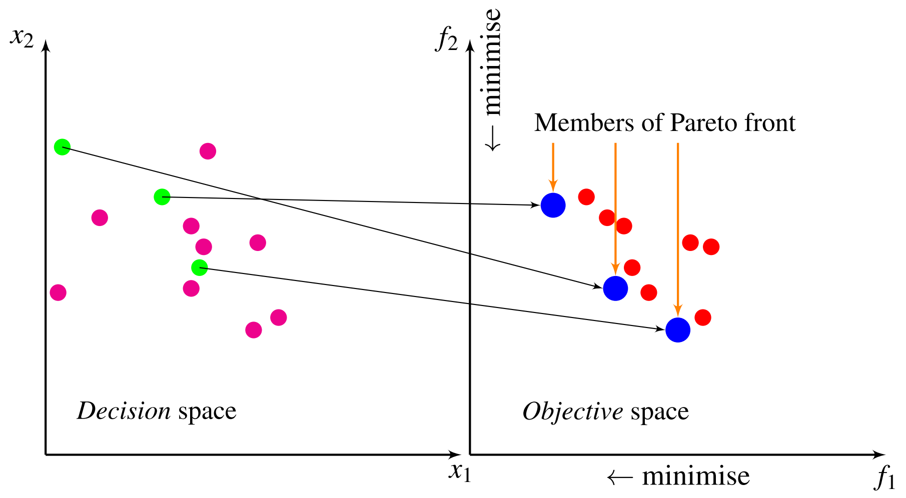


Figure 6.9: The decision space and objective space

MOO deterministic example

■ **Example 6.4** Consider the following simple deterministic, continuous example with one decision variable and two objective functions:

Minimise

$$f_1(x) = x^2 \tag{6.11}$$
$$f_2(x) = (x-2)^2 \tag{6.12}$$

with $-10^5 \leq x \leq 10^5$. The decision space is one-dimensional but large. We can find by inspection that $0 \leq x \leq 2$, while $0 \leq f_1(x) \leq 4$ and $0 \leq f_2(x) \leq 4$. We see that the objectives have many values. Indeed, we can plot a subset of them on a Cartesian coordinate system, as shown in Figure 6.10. The red curve shows the Pareto set, and it can be seen that the solutions on the blue part of the curve are dominated by those on the red curve.

Also note that, if we decrease $f_1(x)$, then $f_2(x)$ increases, illustrating the conflicting nature of the two functions.

MOO stochastic example

■ **Example 6.5** In simulation-optimisation, $\mathbf{f}(\mathbf{x})$ is estimated with a simulation model. We can reformulate the multi-objective simulation optimisation (MOSO) problem as (Moonyoung Yoon, 2017):

$$\begin{aligned} \min \quad & \mathbf{f}(\mathbf{x}) = [f_1(\mathbf{x}), f_2(\mathbf{x}), \ldots, f_p(\mathbf{x})]^T \\ & \quad = [\mathrm{E}(Y_1(\mathbf{x},\xi)), \ldots, \mathrm{E}(Y_p(\mathbf{x},\xi))]^T \\ \text{s.t.} \quad & \mathbf{x} \in S_F, \end{aligned} \tag{6.13}$$

where $Y_i(\mathbf{x},\xi)$ represents the random variable of the simulation output for the i^{th} objective $(i = 1,\ldots,p)$. The symbol ξ indicates the stochastic elements of the optimisation problem and S_F the feasible region.

Multi-objective simulation-optimisation (MOSO) *is very complex and challenging due to the stochastic output*. The principle is illustrated in Figure 6.11. For each combination of (x_1,x_2), the simulation model must execute a number of replications to determine a ‘good’ value for the point estimator, and this often takes time. Each combination also has its unique distribution, although

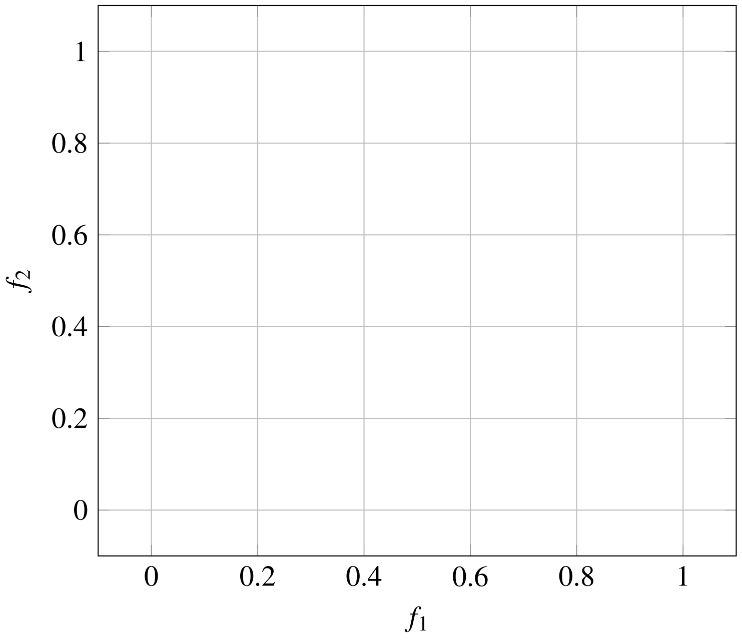


Figure 6.10: Solutions to (6.11) and (6.12) including the Pareto front

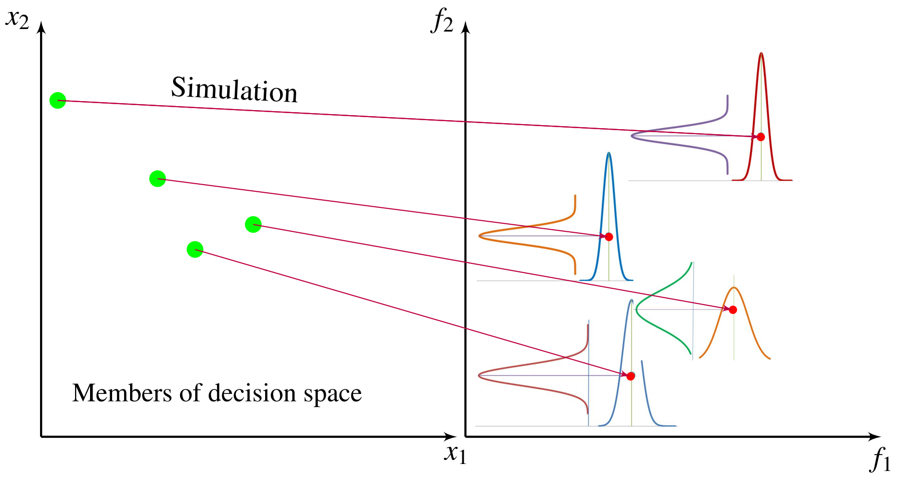


Figure 6.11: The computational burden of simulation-optimisation illustrated

one can assume that estimations of expected values are approximately normally distributed. Finding the approximate Pareto set while doing the smallest number of simulation replications is currently being actively researched (Yoon and Bekker, 2020; Yoon and Bekker, 2022).

In the classic buffer-allocation problem (BAP), one wants to *maximise throughput* and *minimise work-in-progress*, so a typical (approximate) Pareto front is shown in Figure 6.12, for 10 machines and 25 buffers. In this problem then, $f_1(x)$ represents the throughput and $f_2(x)$ the work-in-progress, while $S = \{1,2,3,\ldots,25\}$. The stochastic elements ξ in this example include the processing times on the machines, the number of cycles until failure per machine, and the times to repair the machines.

The blue 'cloud' of circles shows all solutions developed in search of the Pareto set (via simulation), while the red dots show the good solutions. Note the shape of the Pareto front – we *maximise* throughput. (How would a Pareto front look like if we maximised both objectives?)

For further practical applications of MOSO see Scholtz, Bekker, and Toit (2012), Stadler and Bekker (2014), Snyman (2019) and Yoon and Bekker (2022).

6.7 Ranking of solutions

When doing MOSO, many solutions may emerge, especially if meta-heuristics are used to search the decision space. To find the solutions of the approximate Pareto set, some form of *ranking* is needed. This 'ranking' should separate non-dominated solutions from those that are dominated. A version of the Pareto ranking method proposed by Fonseca and Fleming (1998) will be discussed, although it only applies to *deterministic* results. Suppose one wants to rank the data values of columns $f_1(\mathbf{x})$ and $f_2(\mathbf{x})$ in Table 6.1(a), which were created from (6.11) and (6.12).

The method is applied by *sorting* the data set, with the result shown in Table 6.1(b), where a rank column has been added. The data was sorted in *descending order* with the column of $f_1(\mathbf{x})$ as sort key, so that the *largest* values are at the top. The sort order of $f_1(\mathbf{x})$ is

- **descending** if $f_1(\mathbf{x})$ is to be **minimised**, or
- **ascending** if $f_1(\mathbf{x})$ is to **maximised**.

The values of column $f_2(x)$ are now considered: if the element in the first row is larger than the element in the second row, the rank of row 1 is incremented by one. This is repeated until the last row of column $f_2(x)$ is reached. The process is repeated, but one now starts at row 2, and does the comparison from row 3 downwards. The rank value of the last row is defaulted to 0. Once all rows (except the last row) have been used as reference, all the rows having a rank of zero are taken out (Table 6.1(c)) and the $f_i(x)$ form the Pareto set. A rank value of 0 indicates that no other solution set $(f_1(x), f_2(x))$ dominates any solution set in the group (the Pareto set). The minimum and maximum values for x can also be retrieved from this set. In this example, x was not indicated as a vector although it strictly is one. Note that the process to extract the x-values is a bit more complicated when the Pareto set is discontinuous.

For a Pareto set obtained with stochastic methods, one must ensure that the solutions in the Pareto set are statistically significantly different, similar to the SOSO case (Subsection

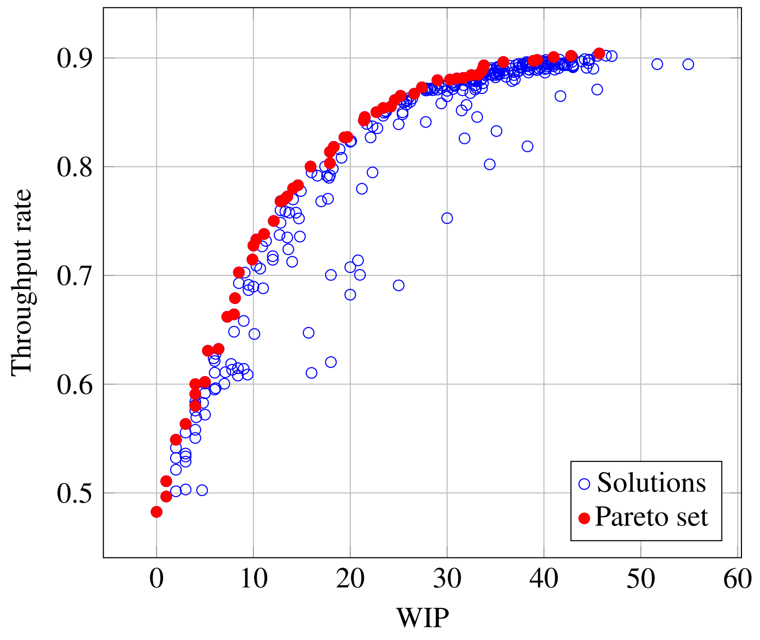


Figure 6.12: Approximate Pareto set with dominated vectors for the BAP

Table 6.1: Example values for Pareto ranking

Original data set		
x	$f_1(x)$	$f_2(x)$
-2.0	4.00	16.00
-1.8	3.24	14.44
-1.6	2.56	12.96
-1.4	1.96	11.56
-1.2	1.44	10.24
-1.0	1.00	9.00
-0.8	0.64	7.84
-0.6	0.36	6.76
-0.4	0.16	5.76
-0.2	0.04	4.84
0.0	0.00	4.00
0.2	0.04	3.24
0.4	0.16	2.56
0.6	0.36	1.96
0.8	0.64	1.44
1.0	1.00	1.00
1.2	1.44	0.64
1.4	1.96	0.36
1.6	2.56	0.16
1.8	3.24	0.04
2.0	4.00	0.00
2.2	4.84	0.04
2.4	5.76	0.16
2.6	6.76	0.36
2.8	7.84	0.64
3.0	9	1.00

(a)

Ranked data set			
x	$f_1(x)$	$f_2(x)$	Rank
3.0	9.00	1.00	10
2.8	7.84	0.64	7
2.6	6.76	0.36	6
2.4	5.76	0.16	5
2.2	4.84	0.04	2
-2.0	4.00	16.00	20
2.0	4.00	0.00	0
-1.8	3.24	14.44	18
1.8	3.24	0.04	0
-1.6	2.56	12.96	16
1.6	2.56	0.16	0
-1.4	1.96	11.56	14
1.4	1.96	0.36	0
-1.2	1.44	10.24	12
1.2	1.44	0.64	0
-1.0	1.00	9.00	10
1.0	1.00	1.00	0
-0.8	0.64	7.84	8
0.8	0.64	1.44	0
-0.6	0.36	6.76	6
0.6	0.36	1.96	0
-0.4	0.16	5.76	4
0.4	0.16	2.56	0
-0.2	0.04	4.84	2
0.2	0.04	3.24	0
0.0	0.00	4.00	0

(b)

Pareto set			
x	$f_1(x)$	$f_2(x)$	Rank
2.0	4.00	0.00	0
1.8	3.24	0.04	0
1.6	2.56	0.16	0
1.4	1.96	0.36	0
1.2	1.44	0.64	0
1.0	1.00	1.00	0
0.8	0.64	1.44	0
0.6	0.36	1.96	0
0.4	0.16	2.56	0
0.2	0.04	3.24	0
0.0	0.00	4.00	0

(c)

5.5.3). The work by Pierro, Khu, and Savić (2007) and Lee et al. (2010) explains and applies multi-objective ranking and selection (MORS) and their methods to ensure statistically different solutions in the Pareto set. The MORS field is actively researched, but falls outside of the scope of this module.

6.8 Selecting a solution for implementation

Assume that the Pareto set has been formed with an R&S procedure, and one of these solutions must now be selected for implementation. You must now advise Management. The solutions in the Pareto set are all 'good', and the decision-maker may select any of these. However, to determine which one to select for implementation, some preferences need to be defined, *e.g.* from a business perspective. There are methods like simple additive-weighting (SAW) and TOPSIS available, but these are outside of the scope of this module. The interested reader is referred to Hwang, Lai, and Liu (1993). One can also use business requirements to select a single solution from the approximate Pareto set. Consider the (r, Q) inventory problem, in which we want to minimise inventory cost, but maximise service level. A typical Pareto front is shown in Figure

6.13.

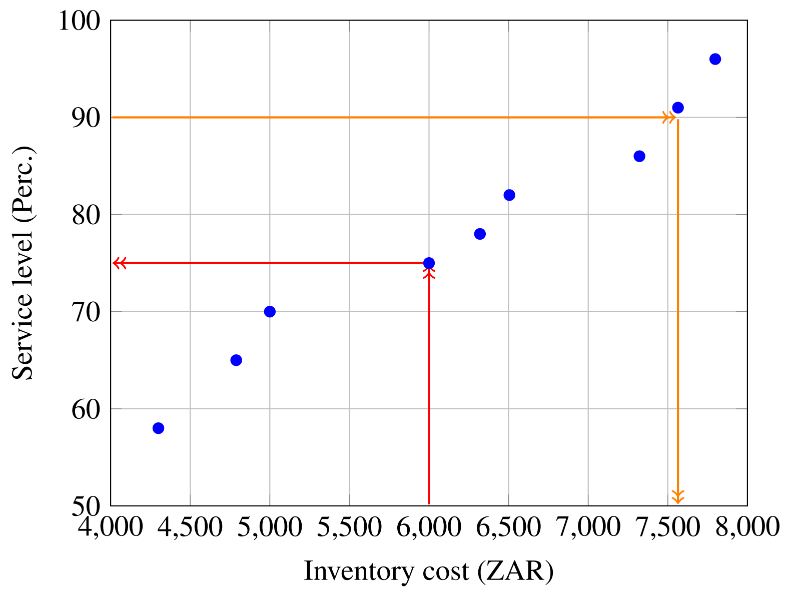


Figure 6.13: Example Pareto front for the (r, Q) inventory problem

Suppose Management decided that the maximum inventory cost they can afford per period is R6 000. The analyst reads the graph by starting on the horizontal axis at R6 000, then goes up to the dots along the red line. When a dot or nearby dot is found, the analyst goes horizontally to the left, still along the red line, and reads off the service level (75% in this case). The analyst can then tell Management that the maximum expected service level will be 75% if R6 000 is invested per period. Management will have to accept that 25% of the customers will not be served or increase the allowed cost.

On the other hand, suppose Management insists on a 90% service level, then the analyst can follow the orange line, starting at 90 on the vertical axis, then projecting a horizontal line to the right towards the (nearest) 'dot' on the Pareto front. From that point, by going down, the analyst determines that the cost is approximately R7 560. Management must now decide if this cost justifies the desired service level. The lesson for the industrial engineer here is that in MOO and MOSO, the best solutions are often found via methods other than mathematical methods.

6.9 Some MOO algorithms

Research is actively done to develop algorithms that can find the Pareto set with as few as possible simulation evaluations (Bekker and Aldrich, 2011). Examples include (Coello Coello, Lamont, and Veldhuizen, 2007):

1. Multi-objective genetic algorithm (MOGA, Fonseca & Fleming).
2. Multi-objective optimisation with the cross-entropy method (MOO CEM) ((Bekker and Aldrich, 2011))
3. Niched-Pareto Genetic Algorithm (NPGA, Erickson *et al.*).
4. Strength Pareto Evolutionary Algorithm (SPEA, Zitzler & Thiele).
5. Pareto Archived Evolution Strategy (PAES, Knowles & Corne).
6. Non-dominated sorting genetic algorithm (NSGA-II, Deb *et al.*).

Many other algorithms are available in literature, and commercial packages like OptQuest are used with several simulation software packages (*e.g.* Simio) for multi-objective simulation-optimisation.

Survival kit:

Optimise your SO skills and ensure you have the following in your assessment pack:

1. Understand the combinatorial nature of optimisation problems.
2. Understand the principles of simulation-optimisation (SO) and define it.
3. Understand meta-heuristics, and that they provide good, but not necessarily optimal solution(s), in relatively short time. It is important to understand Figure 6.1.
4. You must know the basic working of the GA, its parameters (population size, number of generations, termination conditions, mutation and selection and cross-over).
5. You must define the MOO problem, and understand the concept of dominance and the Pareto set.
6. Understand and define MOSO.
7. Always keep in mind that in SO, we can only estimate an approximate Pareto set, and that a *computational burden* is associated with SO. The concepts illustrated in Figure 6.11 are very important to understand.
8. You must be able to do Pareto ranking of solutions (although it is not strictly correct in the case of MOSO).

R Notice on the door to the School of the Gifted: *Pull.* I pushed and I pushed . . .