Part II – Decision Systems for Engineering Design
Lecture 3: Multi-objective Optimization I

Robin Purshouse
University of Sheffield

Spring Semester 2023/24

1 Lecture overview

This lecture introduces multi-objective optimization as a core element of a decision system for engineering design.

In this lecture, we will cover:

- Multi-objective optimization—notions of optimality and the aims of a multi-objective optimizer;
- Population-based optimizers—the main components of generic population-based optimizers that are often used in decision systems for engineering design;
- Pareto-based algorithms—one of the three classes of population-based optimizer that can be used as the engine for a multi-objective optimization process.

2 Multi-objective Optimization

2.1 The multi-objective optimization problem

Consider a decision systems problem where we seek to find a design $\mathbf{x}$ from design space $\mathcal{D}$ that optimizes some performance criteria $\mathbf{z}$. If no constraints exist for the problem (other than the bounds on the design space—so-called *box constraints*) then the problem consists purely of objectives and can be formulated as an unconstrained multi-objective optimization problem (MOP):

$$\begin{aligned} \underset{\mathbf{x}}{\text{minimize}} \quad & \mathbf{z} = \mathbf{f}(\mathbf{x}, \mathbf{p}) \\ & \mathbf{x} \in \mathcal{D} \\ & \mathbf{p} \leftarrow \mathcal{P} \end{aligned} \tag{1}$$

To simplify things further in this lecture, we will assume that a single baseline parameterisation of the evaluation function is used throughout and drop $\mathbf{p}$ from the notation. We will also assume minimization of $\mathbf{z}$, without loss of generality.

2.2 Notions of optimality

2.2.1 Pareto dominance relations

Consider a pair of candidate designs, $\mathbf{u}$ and $\mathbf{v}$, from $\mathcal{D}$. Clearly design $\mathbf{u}$ can be considered superior to design $\mathbf{v}$ if $\mathbf{u}$ performs better than $\mathbf{v}$ across all the objectives $\mathbf{z}$. In the language of multi-objective optimization, design $\mathbf{u}$ *strictly dominates* solution $\mathbf{v}$, denoted as $\mathbf{u} \prec\prec \mathbf{v}$. Formally:

$$\mathbf{u} \prec\prec \mathbf{v} \iff z_i(\mathbf{u}) < z_i(\mathbf{v}), \forall i \in \{1, \ldots, M\} \tag{2}$$

where M is the number of objectives.

Design $\mathbf{u}$ can also be considered superior to design $\mathbf{v}$ if $\mathbf{u}$ performs no worse than $\mathbf{v}$ across all the objectives and better than $\mathbf{v}$ in at least one objective. In the language of multi-objective optimization, design $\mathbf{u}$ *dominates* design $\mathbf{v}$, denoted as $\mathbf{u} \prec \mathbf{v}$. Formally:

$$\mathbf{u} \prec \mathbf{v} \iff (z_i(\mathbf{u}) \leq z_i(\mathbf{v}), \forall i \in \{1, \ldots, M\}) \wedge (\exists i \in \{1, \ldots, M\} : z_i(\mathbf{u}) < z_i(\mathbf{v})) \tag{3}$$

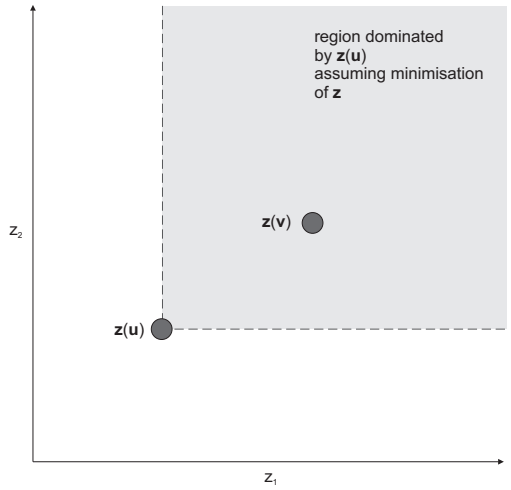

Figure 1: Example of Pareto dominance in two objectives. The shaded area denotes strict dominance by design $\mathbf{u}$, the dashed lines indicate dominance. Design $\mathbf{u}$ is seen to strictly dominate an alternative design $\mathbf{v}$.

Besides dominance and strict dominance, two further relationships are sometimes useful when comparing a pair of designs. If design $\mathbf{u}$ performs no worse than design

$\mathbf{v}$ across all objectives then $\mathbf{u}$ is said to *weakly dominate* $\mathbf{v}$. Essentially this loosens the dominance relation to include the possibility that the two designs exhibit equal performance. Formally:

$$\mathbf{u} \preceq \mathbf{v} \iff z_i(\mathbf{u}) \leq z_i(\mathbf{v}), \forall i \in \{1, \ldots, M\} \tag{4}$$

Further, designs $\mathbf{u}$ and $\mathbf{v}$ are defined as being *incomparable* in relation to each other if $\mathbf{u}$ does not weakly dominate $\mathbf{v}$ whilst also $\mathbf{v}$ does not weakly dominate $\mathbf{u}$:

$$\mathbf{u} || \mathbf{v} \iff (\mathbf{u} \npreceq \mathbf{v}) \wedge (\mathbf{v} \npreceq \mathbf{u}) \tag{5}$$

2.2.2 Pareto optimality

If no other feasible design $\mathbf{v}$ exists that dominates design $\mathbf{u}$, then $\mathbf{u}$ is classified as a *non-dominated* or *Pareto optimal* design. Formally:

$$\mathbf{u} \in \mathbf{X}^\star \iff \nexists \mathbf{v} \in \mathcal{D} : \mathbf{v} \prec \mathbf{u} \tag{6}$$

where $\mathbf{X}^\star$ is the *efficient set* of Pareto optimal designs. In addition to thinking about global optimality, it is often also useful to think about non-dominance in a more local context (such as the set of designs found so far)—in this case, we replace the conditionality on $\mathcal{D}$ with some local space $\mathcal{L}$ in Equation 6.

The objective vectors that correspond to the efficient set are described as the *Pareto front* or *trade-off surface*. An example of the relationship is shown in Figure 2.

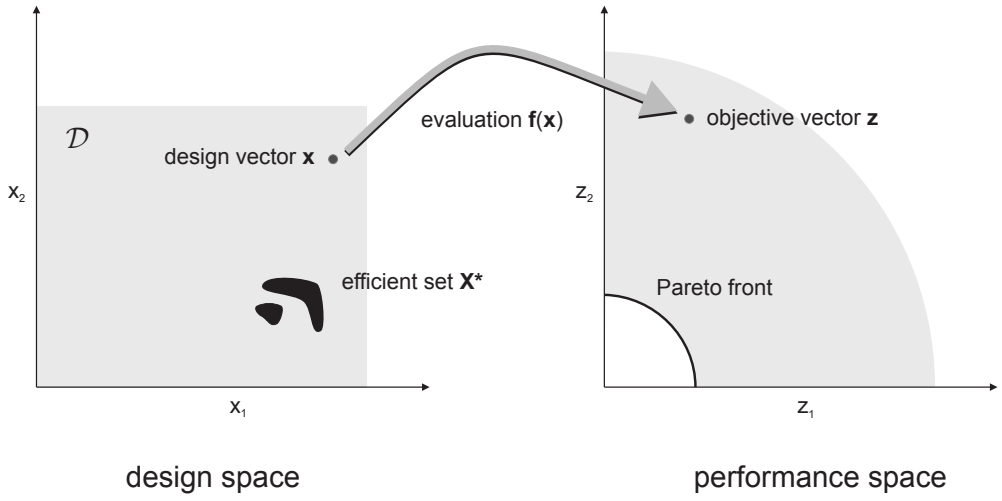

Figure 2: The multi-objective problem domain

The strict dominance relation can also be used to define *weakly non-dominated* (or *weakly Pareto optimal* or *weakly efficient*) designs: $\nexists \mathbf{v} \in \mathcal{D} : \mathbf{v} \prec\prec \mathbf{u}$. These

are designs for which it is not possible to improve in all objectives simultaneously by switching to a different design.

2.2.3 *A posteriori* decision-making

Without further preference information about the objectives it is not possible to say which of the Pareto optimal designs are better than the others. Further, if the design variables are continuous in nature, then the trade-off surface may contain a potentially infinite number of candidate designs. The job of a decision system in this context is thus to identify a usefully representative set of these designs. This process is known as *a posteriori* decision-making because the final design is only selected after information about the Pareto optimal set has been revealed.

The final set of candidate designs generated by the decision system is known as an *approximation set*. Three aspects of approximation set quality can be considered. These are listed below, and shown graphically in Figure 3.

- **Proximity.** The approximation set should contain designs whose corresponding objective vectors are close to the true Pareto front.
- **Diversity.** The approximation set should contain a good coverage of designs, in terms of both extent and uniformity. Good diversity is commonly of interest in performance space, but may also be required in design space. In performance space, the approximation set should extend across the entire range of the true Pareto front with an even distribution across the surface.
- **Pertinency.** The approximation set should only contain solutions in the decision-maker (DM) region of interest (ROI). In practice, and especially as the number of objectives increases, the DM is interested only in a sub-region of objective space. Thus, there is little benefit in representing trade-off regions that lie outside the ROI. Focusing on pertinent areas of the search space helps to improve optimization efficiency and reduces unnecessary information that the DM would otherwise have to consider.

3 Population-based optimizers

Decision systems for engineering design require some kind of search engine that is able to identify a suitable approximation set of promising candidate designs. *Population-based optimizers*, such as **evolutionary algorithms** are natural choices for such a search engine because they explicitly work with a set of solutions as the search progresses. In this section, we will briefly introduce evolutionary computing methods.

3.1 Fundamental concepts

Evolutionary computation is a search discipline based on the evolutionary biology concepts of natural selection and population genetics. The term ‘evolutionary algorithm’

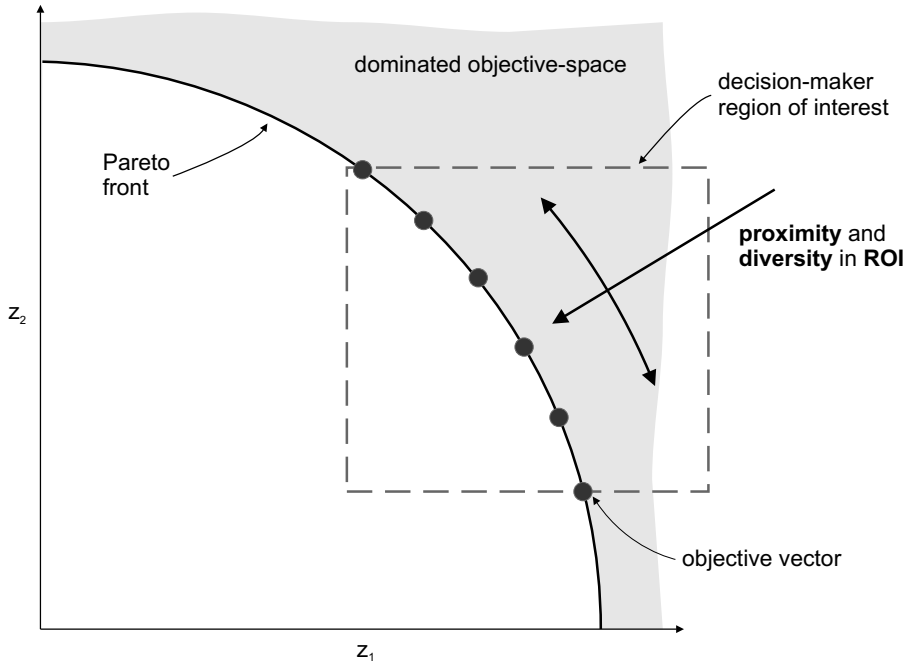

Figure 3: *A posteriori* solution to a multi-objective optimization problem

(EA) has no rigorous definition, but can generally be used to describe any population-based, stochastic, direct search method. Many such algorithms exist.

Contemporary evolutionary algorithms have their origins in two originally independent and highly influential evolutionary models: the German *evolution strategy* (ES) and the American *genetic algorithm* (GA). The distinctions between these models have weakened considerably over recent decades.

The fundamental idea in an evolutionary search is to iteratively apply *variation* (*v*) and *selection* (*s*) processes to a *population* of candidate solutions (i.e. designs) until some termination criterion is satisfied (e.g. the budget for evaluations is exhausted). This core concept is described in Equation 7, in which $P[t]$ is the population at time t.

$$P[t+1] = s(v(P[t])) \tag{7}$$

Probabilistic changes are made to solutions in the population to yield a set of new candidate solutions. Solutions are then selected for inclusion in the population at the next iteration (or *generation*), $P[t+1]$, based on their ability to solve the problem at hand. Variation is then applied to the new population to generate yet another set of new solutions, and so on. The aim of the EA is thus to progressively develop better solutions to the problem by making modifications to previous solutions that exhibited good performance within their peer group.

3.2 A generalised evolutionary algorithm

Expanding slightly on the fundamental evolutionary search mechanism given in Equation 7, the general form of a modern EA can be captured as shown in Equation 8. The central difference is that the overall selection operation has been split into two processes, *selection-for-variation* (s_v) and *selection-for-survival* (s_s), which are described in more detail below.

$$P[t+1] = s_s(v(s_v(P[t])), P[t]) \tag{8}$$

3.2.1 Selection-for-variation

At the selection-for-variation stage, solutions from $P[t]$ are chosen for inclusion in the so-called *mating pool*, to which variation operators are applied to create new solutions. The performance of every solution is evaluated for the problem at hand, by whatever means. This raw evaluation is then transformed into a scalar *fitness* value that is an overall measure of solution performance. Larger fitness values correspond to better solutions. All EA selection operators use fitness value as the selection discriminator. Fitness ultimately represents the expected number of times that a solution will be selected. The mapping between the two quantities is often implicit to the selection procedure used.

Several different selection operators have been proposed in the literature, including *tournament selection* and *stochastic universal sampling*. The ideal selection mechanism has a low computational complexity, is parallelisable, has minimal spread (defined as the maximum sample deviation from the expected number of selections of an

individual), and minimal bias (defined as the difference between the sampling probability of selection for an individual and the true expected value).

All selection operators have a property known as *selective pressure*. This is the difference in the number of expected selections between the best performing solution and the worst. In proportionate selection schemes this value is explicitly defined by the user, whilst it is implicit to tournament selection mechanisms.

3.2.2 Variation

The variation operators use the genetic material of the solutions in the mating pool to create new candidate solutions. The operators work on a representation of the solutions (known as the *genotype*) rather than the actual solutions themselves (known as the *phenotype*). Thus, coding and decoding processes are required between genotype and phenotype. In genetic algorithms, the genotype is traditionally a binary string, whereas in evolution strategies it is a concatenation of real numbers (direct operation on the phenotype). However, there is no restriction as to representation, providing that variation operators can be devised to handle it. Contemporary wisdom is to use whatever representation seems most appropriate for the task at hand. For instance, genetic algorithms solving real-parameter function optimization problems generally use real-parameter variation operators. Block diagram representations, and other structures such as trees, are also popular.

Variation operators can be classified according to whether one solution or multiple solutions (known as *parents*) are used to create the new solutions (known as *children* or *offspring*). Single parent operators are commonly described as *mutations*. The classic example for the original genetic algorithm is known as *bit-flipping mutation*, in which each element, or *locus*, of the genotype is probabilistically tested for possible mutation. Consider the parent genotype `1111`: if only the third locus of the genotype is to be mutated then the resulting child genotype is `1101`. In the classic evolution strategies approach, mutation is achieved by means of a Gaussian probability density function (PDF) centred on the parent decision variable. The PDF is sampled to obtain a child value on the real number range.

Multi-parent variation operators are commonly known as *recombination* or *crossover* operators. This can be illustrated by the original genetic algorithm technique known as *single-point binary crossover*, in which two parents are partitioned at randomly determined equivalent positions and then the genetic material is swapped between the two genotypes in one of the segments to form two children. Consider an example with parents `1111` and `0000`: if the crossover site is chosen as the third locus, then the children are `1100` and `0011`.

Every variation operator has an associated probability of application. Good values for this probability depend on the problem to be solved and the interaction with selection mechanisms. In genetic algorithms, general heuristics exist for both crossover and mutation: the probability of crossover between a parent pair is usually quite high (values such as 0.7 or 0.8 are common); conversely, the probability of applying mutation to a particular locus on the genotype is usually quite low (often set to be equivalent to an expectation of one mutation per solution). These values are based on

the assumption that both operators are to be applied at each generation. In evolution strategies, a *self-adaptive* approach is usually employed, in which the variation probability is itself subject to evolution.

3.2.3 Selection-for-survival

Following the application of the variation operators, two sets of solutions exist: the old population, $P[t]$, and the child population, $v(s_v(P[t]))$. Since the size of an EA population over the generations is typically static, more solutions now exist than can be retained. Thus, a selection-for-survival stage is required in order to determine the new population, $P[t+1]$. In ES, this concept is embodied in the so-called (μ, λ) and $(\mu+\lambda)$ schemes, where μ is the current population and λ is the child population. In the (μ, λ) method, only child solutions can survive to $P[t+1]$, whereas in the $(\mu+\lambda)$ approach the child and $P[t]$ solutions compete for inclusion in the subsequent population. In the genetic algorithm field the concept of a *generational gap* was conceived, in which sufficient children were generated to replace a certain percentage of $P[t]$. Selection-for-survival is often a deterministic process in which the best solutions are selected from a fitness-based hierarchy. However, stochastic schemes are also possible. An s_s scheme that forces the retention of high-performance members of $P[t]$ is described as *elitist*.

3.2.4 Exploration versus Exploitation

One of the fundamental issues in any search algorithm is the trade-off between *exploration* of the undiscovered regions of search space and detailed *exploitation* of promising areas already identified. Good performance can be heavily dependent on an appropriate choice of exploration-exploitation trade-off. Obtaining the correct balance for the task at hand is the black art of EA design.

4 Pareto-based decision systems

4.1 Overview of methods

Pareto-based algorithms represent a major class of multi-objective optimizer. There are many variants of these algorithms but they typically contain two components that inform the selection mechanisms s_v and s_s of the algorithm:

1. *Dominance-based component*—uses concepts related to Pareto dominance (Equation 3) to determine the relative merits of different candidate solutions;
2. *Density-based component*—uses concepts related to the density of solutions to discriminate between solutions that cannot be discriminated in terms of weak dominance.

Amongst the well-known Pareto-based algorithms that have been proposed are the *multi-objective genetic algorithm* (MOGA), the *non-dominated sorting genetic algorithm* (NSGA), and the *strength Pareto evolutionary algorithm* (SPEA). Note that MOGA was developed in ACSE (by Carlos Fonseca and Peter Fleming) and two examples of its use in engineering design applications were given in Lecture 1 (for the systems architecture and gasifier control system problems).

In the remainder of this lecture we will focus in detail on what is unarguably the most well-known of the Pareto-based algorithms—the *elitist non-dominated sorting genetic algorithm* (NGSA-II). This algorithm can be found in many of the major engineering design optimization software products, such as Esteco's modeFRONTIER and Dassault's iSight.

4.2 The NSGA-II algorithm

4.2.1 Core components

The NSGA-II algorithm begins with an initial population of solutions $P[0]$, which is conventionally generated at random in the design space, but could be generated using any of the space-filling approaches from Lecture 2. NSGA-II then uses a *non-dominated sorting* procedure to rank the alternative solutions in the population in terms of Pareto dominance. The non-dominated sorting procedure is an iterative process that first identifies all locally non-dominated designs in the population using Equation 6 where $\mathcal{D}$ is replaced by $P[0]$. These non-dominated solutions are assigned rank $r = 1$, are temporarily removed from consideration, and the process is repeated to identify a new set of locally non-dominated solutions. These are assigned rank $r = 2$, are removed from consideration, and the process is repeated until all solutions in the population have been assigned a rank. This process is illustrated in Figure 4.

To distinguish between solutions that have been assigned the same rank by the non-dominated sorting process, a density estimator is used. The density estimator proposed in NSGA-II is the so-called *crowding distance*, which is based on distances between designs in objective space. For each design, nearest neighbour solutions are identified based on adjacency in each objective dimension, but only considering those solutions of equal rank. The crowding distance d_i of design $\mathbf{x}_i$ is then calculated as the dimensions of the hypercuboid formed by placing nearest neighbour solutions $\mathbf{x}_{i-1}$ and $\mathbf{x}_{i+1}$ at the vertices of the hypercuboid:

$$d_i = \sum_{k=1}^{K} \left| \bar{z}_k^{(\mathtt{sort}_k(i+1))} - \bar{z}_k^{(\mathtt{sort}_k(i-1))} \right| \tag{9}$$

where K is the number of objectives, $\bar{z}_k^{(s)}$ represents the (normalised) performance of design s according to the kth objective, and $\mathtt{sort}_k(s)$ is a function that sorts the solutions of equal rank to i according to the kth objective and returns the sth solution in the sorted list.

For designs that lie on the edge of the performance space, nearest neighbours do not exist in all directions and so the crowding distance is set to infinity. An ex-

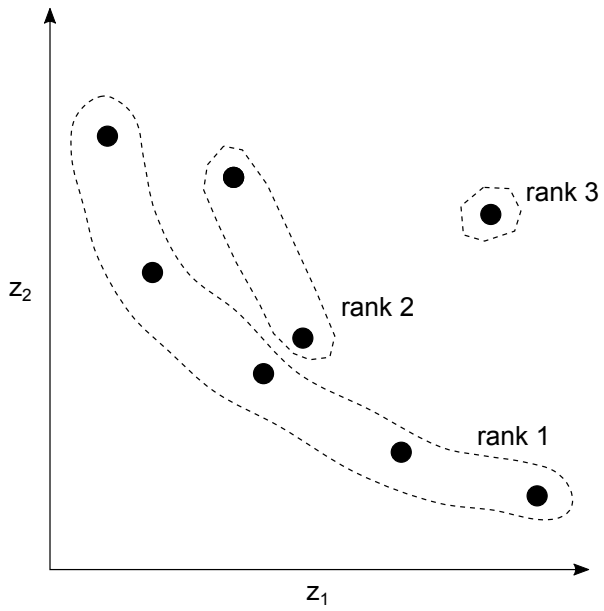

Figure 4: Non-dominated sorting procedure

ample crowding distance calculation in two-objective performance space is shown in Figure 5.

Algorithm listings for the non-dominated sorting and crowding distance operators can be found in the NSGA-II paper included as further reading.

4.2.2 Selection-for-variation

The next stage of the algorithm is to perform selection-for-variation s_v on the ranked population. This is achieved using *binary tournament selection with replacement*. To select a solution, we randomly choose two solutions $\mathbf{x}_i$ and $\mathbf{x}_j$ from the population $P[0]$ and then pick solution i over solution j if:

1. If solution i has the better rank, i.e. $r_i < r_j$;
2. If the ranks are the same, then if solution i has a better crowding distance, i.e. $(r_i = r_j) \wedge (d_i > d_j)$;
3. If the ranks are the same and the crowding distances are the same, then randomly, i.e. $(r_i = r_j) \wedge (d_i = d_j) \wedge (U(0,1) < 0.5)$.

Sufficient solutions need to be selected in order to construct the next iteration of the optimizer's population. Typically, this will be the same number of solutions as the population size itself.

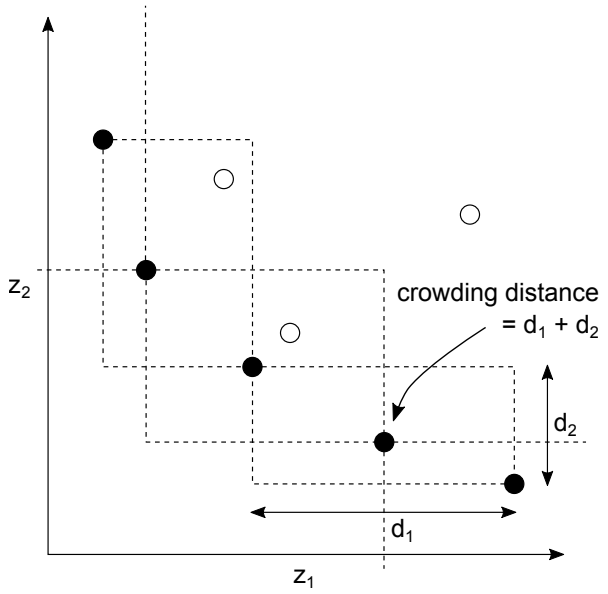

Figure 5: Crowding procedure

4.2.3 Variation

Once solutions have been selected, variation operators are applied to generate new candidate designs. The variation operators will be problem-dependent. For problems with continuously-defined variables, then *simulated binary crossover* (SBX) and *polynomial mutation* are popular choices.

Polynomial mutation. Polynomial mutation is a one-parent variation operator that takes a parent design and creates a new child design. Variable-wise perturbations to existing designs (i.e. parents) are performed according to a probability distribution function centred over the parent value. The operator is defined in Equation 10, where x_i is the parent value for the ith decision variable, u_i and l_i are the upper and lower bounds on the ith decision variable, η_m is a distribution parameter, ρ_i is a number generated uniformly at random from [0 1], and c_i is the resulting child value for the ith decision variable.

$$\left.\begin{array}{rcl} c_i & = & \left\{ \begin{array}{ll} x_i + (x_i - l_i)\,\delta_i & \text{if } \rho_i < 0.5, \\ x_i + (u_i - x_i)\,\delta_i & \text{otherwise.} \end{array} \right. \\ \delta_i & = & \left\{ \begin{array}{ll} (2\rho_i)^{1/(\eta_m+1)} - 1 & \text{if } \rho_i < 0.5, \\ 1 - [2\,(1-\rho_i)]^{1/(\eta_m+1)} & \text{otherwise.} \end{array} \right. \end{array}\right\} \quad (10)$$

Polynomial mutation has two controllable parameters: (i) the probability of applying

mutation to a chromosome element, p_m, and (ii) a mutation distribution parameter, η_m. The latter parameter controls the magnitude of the expected mutation of the candidate solution variable. The normalised variation is likely to be of $\mathcal{O}(1/\eta_m)$. Thus, relatively speaking, small values of η_m should produce large mutations whilst large values of η_m should produce small mutations.

Simulated binary crossover. Unlike polynomial mutation, SBX is a two-parent variation operator that produces two new solutions. SBX is defined in Equation 11, where $x_i^{(1)}$ and $x_i^{(2)}$ are the parent values for the ith decision variable, η_c is a distribution parameter, ρ_i is a number generated uniformly at random from [0 1], and $c_i^{(1)}$ and $c_i^{(2)}$ are the resulting child values for the ith decision variable.

$$\left.\begin{aligned} c_i^{(1)} &= 0.5\left[(1+\beta_i)\,x_i^{(1)} + (1-\beta_i)\,x_i^{(2)}\right] \\ c_i^{(2)} &= 0.5\left[(1-\beta_i)\,x_i^{(1)} + (1+\beta_i)\,x_i^{(2)}\right] \\ \beta_i &= \begin{cases} (2\rho_i)^{1/(\eta_c+1)} & \text{if } \rho_i < 0.5, \\ \left[1/\left(1-\rho_i\right)\right]^{1/(\eta_c+1)} & \text{otherwise.} \end{cases} \end{aligned}\right\} \quad (11)$$

SBX generates child values from a probability distribution, with standard deviation derived from the distance between parent values and a distribution parameter η_c. The distance determines the overall magnitude of the distribution, whilst η_c determines the shape of the distribution. To generate child values for a decision variable, the distribution is centred over each parent and a random value is generated from the distribution to create one child. The second child is generated symmetrically to the first child about the mid-point between the parents. The child values for a decision variable are then exchanged between the complete child solutions with probability p_e. Note that child values that are generated outside the range of a decision variable are cropped to the nearest feasible value.

4.2.4 Selection-for-survival

At this point in the algorithm, we have an initial population $P[0]$ and a new set of designs generated by variation operators, $Q[0]$. We now need to decide which solutions will be carried over to the next iteration of the algorithm and which will be removed from further consideration (note that these latter solutions will in practice be stored in an off-line archive to enable them to be interrogated by decision-makers later).

To achieve the necessary reduction in the number of solutions from $|P \cup Q|$ to $|P|$, a further selection process is undertaken. First, the non-dominated sorting operator is applied to the combined set of designs $P[0] \cup Q[0]$. Then, the next iteration of the population $P[1]$ is initialised to a null set. $P[1]$ is then filled with solutions of rank $r = 1$. If space remains, then solutions of rank $r = 2$ are included, and so on. If any 'spill over' occurs when trying to fill a population with a particular rank then the crowding

distance operator is applied to the solutions of that rank and only the solutions with better crowding distances are included.

The algorithm then proceeds to the selection-for-variation stage using the updated population $P[1]$ until a convergence criterion is achieved.

5 Further reading

- Giagkiozis I., Purshouse R.C., Fleming P.J. An overview of population-based algorithms for multi-objective optimization. *International Journal of Systems Science* 2015;46:1572–1599.
- Deb K., Pratap A., Agarwal S., Meyarivan T., A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation* 2002;6:182–197.