UNIVERSIDADE BEIRA INTERIOR

Gradient-Free Optimization

Otimização em Engenharia (15235)

2º Ciclo/Mestrado em Engenharia Aeronáutica

2024

Pedro V. Gamboa

Departamento de Ciências Aeroespaciais

Faculdade de Engenharia

0. Topics

- When to use gradient-free optimization methods
- Non-population based methods
- Population based methods
- Hybrid meta-heuristics
- Penalty functions

1. Introduction

- Gradient-free algorithms fill an essential role in optimization.
- The gradient-based algorithms introduced in Chapter 4 are efficient in finding local minima for high-dimensional nonlinear problems defined by continuous smooth functions
- However, the assumptions made for these algorithms are not always valid, which can render these algorithms ineffective.
- Also, gradients might not be available when a function is given as a black box.
- In this chapter, only a few popular representative gradient-free algorithms are introduced.
- Most are designed to handle unconstrained functions only, but they can be adapted to solve constrained problems by using penalty or filtering methods.

2. When to use gradient-free algorithms

- Gradient-free algorithms can be useful when gradients are not available, such as when dealing with black-box functions.
- Although gradients can always be approximated with finite differences, these approximations suffer from potentially significant inaccuracies.
- Gradient-based algorithms require a more experienced user because they take more effort to set up and run.
- Overall, gradient-free algorithms are easier to get up and running but are much less efficient, particularly as the dimension of the problem increases.

2. When to use gradient-free algorithms

- One significant advantage of gradient-free algorithms is that they do not assume function continuity.
- For gradient-based algorithms, function smoothness is essential when deriving the optimality conditions, both for unconstrained functions and constrained functions.
- More specifically, the Karush–Kuhn–Tucker (KKT) conditions require that the function be continuous in value (C^0), gradient (C^1), and Hessian (C^2) in at least a small neighbourhood of the optimum.
- If, for example, the gradient is discontinuous at the optimum, it is undefined, and the KKT conditions are not valid.
- Away from optimum points, this requirement is not as stringent.

2. When to use gradient-free algorithms

- Although gradient-based algorithms work on the same continuity assumptions, they can usually tolerate the occasional discontinuity, as long as it is away from an optimum point.
- However, for functions with excessive numerical noise and discontinuities, gradient-free algorithms might be the only option.
- Many considerations are involved when choosing between a gradient based and a gradient-free algorithm.
- One problem characteristic often cited as a reason for choosing gradient-free methods is multimodality.
- Design space multimodality can be a result of an objective function with multiple local minima.
- In the case of a constrained problem, the multimodality can arise from the constraints that define disconnected or nonconvex feasible regions.

2. When to use gradient-free algorithms

- Some gradient-free methods feature a global search that increases the likelihood of finding the global minimum.
- This feature makes gradient-free methods a common choice for multimodal problems.
- However, not all gradient-free methods are global search methods; some perform only a local search.
- Additionally, even though gradient-based methods are by themselves local search methods, they are often combined with global search strategies.
- It is not necessarily true that a global search, gradient-free method is more likely to find a global optimum than a multi-start gradient-based method.
- As always, problem-specific testing is needed.

2. When to use gradient-free algorithms

- Furthermore, it is assumed far too often that any complex problem is multimodal, but that is frequently not the case.
- Although it might be impossible to prove that a function is unimodal, it is easy to prove that a function is multimodal simply by finding another local minimum.
- Therefore, we should not make any assumptions about the multimodality of a function until we show definite multiple local minima.
- Additionally, we must ensure that perceived local minima are not artificial minima arising from numerical noise.
- Another reason often cited for using a gradient-free method is multiple objectives.
- Some gradient-free algorithms, such as the genetic algorithm, can be naturally applied to multiple objectives.

2. When to use gradient-free algorithms

- However, it is a misconception that gradient-free methods are always preferable just because there are multiple objectives.
- Another common reason for using gradient-free methods is when there are discrete design variables.
- Because the notion of a derivative with respect to a discrete variable is invalid, gradient-based algorithms cannot be used directly (although there are ways around this limitation).
- Some of the gradient-free algorithms (but not all) can handle discrete variables directly.
- Although multimodality, multiple objectives, or discrete variables are commonly mentioned as reasons for choosing a gradient-free algorithm, these are not necessarily automatic decisions, and careful consideration is needed.

2. When to use gradient-free algorithms

- Assuming a choice exists (i.e., the function is not too noisy), one of the most relevant factors when choosing between a gradient-free and a gradient-based approach is the dimension of the problem.
- Figure 5.1 shows how many function evaluations are required to minimize the *n*-dimensional Rosenbrock function for varying numbers of design variables.

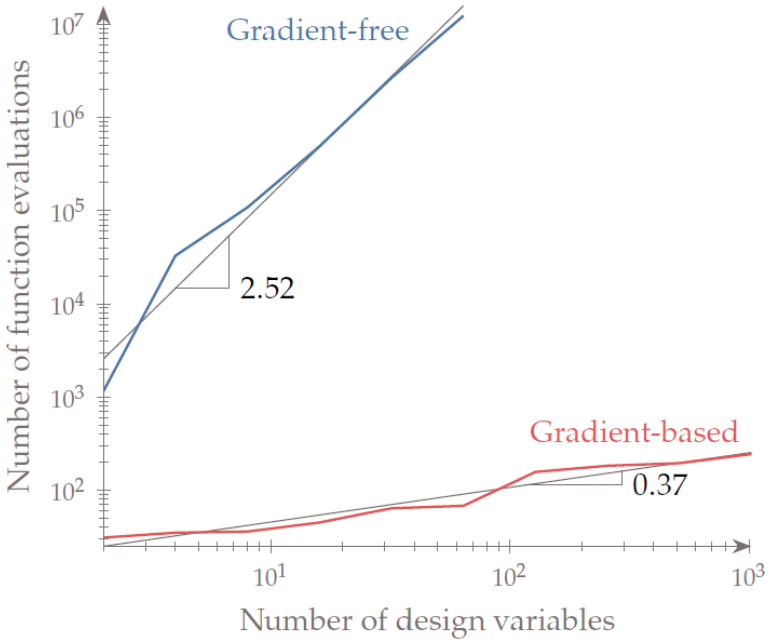

Figure 5.01 Cost of optimization for increasing number of design variables in the *n*-dimensional Rosenbrock function.

Source: Martins et Ning, 2021

2. When to use gradient-free algorithms

- Two classes of algorithms are shown in the plot: gradient-free and gradient-based algorithms.
- The general takeaway is that for small-size problems (usually ≤ 30 variables), gradient-free methods can be useful in finding a solution.
- Furthermore, because gradient-free methods usually take much less developer time to use, a gradient-free solution may even be preferable for these smaller problems.
- However, if the problem is large in dimension, then a gradient-based method may be the only viable method despite the need for more developer time.

3. Classification of gradient-free algorithms

- There is a much wider variety of gradient-free algorithms compared with their gradient-based counterparts.
- Although gradient-based algorithms tend to perform local searches, have a mathematical rationale, and be deterministic, gradient-free algorithms exhibit different combinations of these characteristics.
- Some of the most widely known gradient-free algorithms are shown in Table 5.01 and classify them according to the characteristics introduced in Figure 1.20.

3. Classification of gradient-free algorithms

	Search		Algorithm		Function eval.		Stochas-ticity	
	Local	Global	Mathematical	Heuristic	Direct	Surrogate	Deterministic	Stochastic
Nelder–Mead	•			•	•		•	
GPS	•		•		•		•	
MADS	•		•		•			•
Trust region	•		•			•	•	
Implicit filtering	•		•			•	•	
DIRECT		•	•		•		•	
MCS		•	•		•		•	
EGO		•	•			•	•	
Hit and run		•		•	•			•
Evolutionary		•		•	•			•

Table 5.01 Classification of gradient-free optimization methods using the characteristics of Fig. 1.20.

Source: Martins et Ning, 2021

3. Classification of gradient-free algorithms

- Local search, gradient-free algorithms that use direct function evaluations include the Nelder–Mead algorithm, generalized pattern search (GPS), and mesh-adaptive direct search (MADS).
- Although classified as local search in the table, the latter two methods are frequently used with globalization approaches.
- The Nelder–Mead algorithm is heuristic, whereas the other two are not.
- GPS and MADS are examples of derivative-free optimization (DFO) algorithms, which, despite the name, do not include all gradient-free algorithms.
- DFO algorithms are understood to be largely heuristic-free and focus on local search.

3. Classification of gradient-free algorithms

- GPS is a family of methods that iteratively seek an improvement using a set of points around the current point.
- In its simplest versions, GPS uses a pattern of points based on the coordinate directions, but more sophisticated versions use a more general set of vectors.
- MADS improves GPS algorithms by allowing a much larger set of such vectors and improving convergence.

3. Classification of gradient-free algorithms

- Model-based, local search algorithms include trust-region algorithms and implicit filtering.
- The model is an analytic approximation of the original function (also called a surrogate model), and it should be smooth, easy to evaluate, and accurate in the neighbourhood of the current point.
- The trust-region approach can be considered gradient-free if the surrogate model is constructed using just evaluations of the original function without evaluating its gradients.
- This does not prevent the trust-region algorithm from using gradients of the surrogate model, which can be computed analytically.
- Implicit filtering methods extend the trust-region method by adding a surrogate model of the function gradient to guide the search.

3. Classification of gradient-free algorithms

- This effectively becomes a gradient-based method applied to the surrogate model instead of evaluating the function directly.
- Global search algorithms can be broadly classified as deterministic or stochastic, depending on whether they include random parameter generation within the optimization algorithm.

3. Classification of gradient-free algorithms

- Deterministic, global search algorithms can be either direct or model based.
- Direct algorithms include Lipschitzian-based partitioning techniques—such as the “divide a hyperrectangle” (DIRECT) algorithm and branch-and-bound search - and multilevel coordinate search (MCS).
- The DIRECT algorithm selectively divides the space of the design variables into smaller and smaller *n*-dimensional boxes — hyperrectangles.
- It uses mathematical arguments to decide which boxes should be subdivided.
- Branch-and-bound search also partitions the design space, but it estimates lower and upper bounds for the optimum by using the function variation between partitions.

3. Classification of gradient-free algorithms

- MCS is another algorithm that partitions the design space into boxes, where a limit is imposed on how small the boxes can get based on the number of times it has been divided.

3. Classification of gradient-free algorithms

- Global-search algorithms based on surrogate models are similar to their local search counterparts.
- However, they use surrogate models to reproduce the features of a multimodal function instead of convex surrogate models.
- One of the most widely used of these algorithms is efficient global optimization (EGO), which employs kriging surrogate models and uses the idea of expected improvement to maximize the likelihood of finding the optimum more efficiently.
- Other algorithms use radial basis functions (RBFs) as the surrogate model and also maximize the probability of improvement at new iterates.

3. Classification of gradient-free algorithms

- Stochastic algorithms rely on one or more nondeterministic procedures; they include hit-and-run algorithms and the broad class of evolutionary algorithms.
- When performing benchmarks of a stochastic algorithm, you should run a large enough number of optimizations to obtain statistically significant results.
- Hit-and-run algorithms generate random steps about the current iterate in search of better points.
- A new point is accepted when it is better than the current one, and this process repeats until the point cannot be improved.

3. Classification of gradient-free algorithms

- What constitutes an evolutionary algorithm is not well defined.
- Evolutionary algorithms are inspired by processes that occur in nature or society.
- There is a plethora of evolutionary algorithms in the literature, thanks to the fertile imagination of the research community and a never-ending supply of phenomena for inspiration.
- These algorithms are more of an analogy of the phenomenon than an actual model.
- They are, at best, oversimplified models and, at worst, barely connected to the phenomenon.
- Nature-inspired algorithms tend to invent a specific terminology for the mathematical terms in the optimization problem.
- For example, a design point might be called a “member of the population”, or the objective function might be the “fitness”.

3. Classification of gradient-free algorithms

- The vast majority of evolutionary algorithms are population based, which means they involve a set of points at each iteration instead of a single one.
- Because the population is spread out in the design space, evolutionary algorithms perform a global search.
- The stochastic elements in these algorithms contribute to global exploration and reduce the susceptibility to getting stuck in local minima.
- These features increase the likelihood of getting close to the global minimum but by no means guarantee it.
- The algorithm may only get close because heuristic algorithms have a poor convergence rate, especially in higher dimensions, and because they lack a first-order mathematical optimality criterion.

3. Classification of gradient-free algorithms

- This chapter covers five gradient-free algorithms:
  - Nelder–Mead algorithm,
  - generalized pattern search (GPS)
  - DIRECT method
  - genetic algorithms (GA)
  - particle swarm optimization (PSO)
- There are a few other algorithms that can be used for continuous gradient-free problems (e.g., simulated annealing and branch and bound) which are more frequently used to solve discrete problems.

4. Nelder-Mead algorithm

- The simplex method of Nelder and Mead is a deterministic, direct search method that is among the most cited gradient-free methods.
- It is also known as the nonlinear simplex - not to be confused with the simplex algorithm used for linear programming, with which it has nothing in common.
- The Nelder–Mead algorithm is based on a simplex, which is a geometric figure defined by a set of n+1 points in the design space of n variables, $X=\{x^{(0)}, x^{(1)}, \ldots, x^{(n)}\}$.
- Each point $x^{(i)}$ represents a design (i.e., a full set of design variables).
- In two dimensions, the simplex is a triangle, and in three dimensions, it becomes a tetrahedron (see Fig. 5.02).
- Each optimization iteration corresponds to a different simplex.

4. Nelder-Mead algorithm

- The algorithm modifies the simplex at each iteration using five simple operations.
- The sequence of operations to be performed is chosen based on the relative values of the objective function at each of the points.

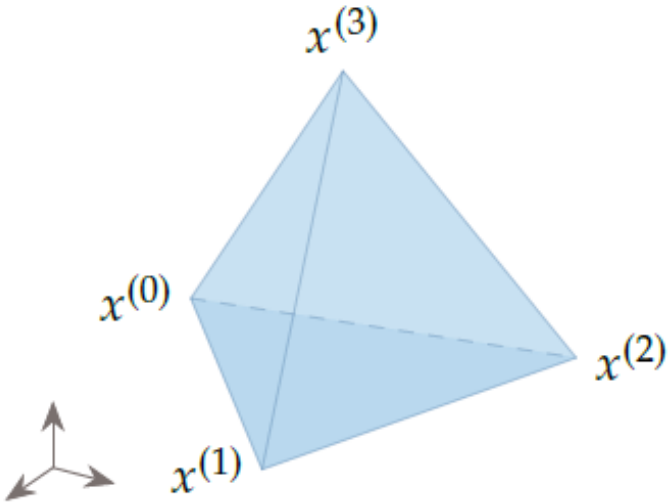

Figure 5.02 A simplex for n=3 has four vertices.

Source: Martins et Ning, 2021

4. Nelder-Mead algorithm

- The first step of the simplex algorithm is to generate n+1 points based on an initial guess for the design variables.
- This could be done by simply adding steps to each component of the initial point to generate n new points.
- However, this will generate a simplex with different edge lengths, and equal-length edges are preferable.
- Suppose we want the length of all sides to be l and that the first guess is $x^{(0)}$.
- The remaining points of the simplex, $\{x^{(1)}, \ldots, x^{(n)}\}$, can be computed by

$$x^{(i)} = x^{(0)} + s^{(i)} \qquad (5.01)$$

where $s^{(i)}$ is a vector whose components j are defined by

4. Nelder-Mead algorithm

$$s_j^{(i)} = \begin{cases} \frac{1}{n\sqrt{2}}\left(\sqrt{n+1}-1\right) + \frac{1}{\sqrt{2}}, & if\ j = i \\ \frac{1}{n\sqrt{2}}\left(\sqrt{n+1}-1\right), & if\ j \neq i \end{cases} \qquad (5.02)$$

- At any given iteration, the objective *f* is evaluated for every point, and the points are ordered based on the respective values of *f*, from the lowest to the highest.
- Thus, in the ordered list of simplex points $X=\{x^{(0)}, x^{(1)}, \ldots, x^{(n-1)}, x^{(n)}\}$, the best point is $x^{(0)}$, and the worst one is $x^{(n)}$.
- The Nelder–Mead algorithm performs four main operations on the simplex to create a new one: reflection, expansion, outside contraction, inside contraction, and shrinking.
- The operations are shown in Fig. 5.03.

4. Nelder-Mead algorithm

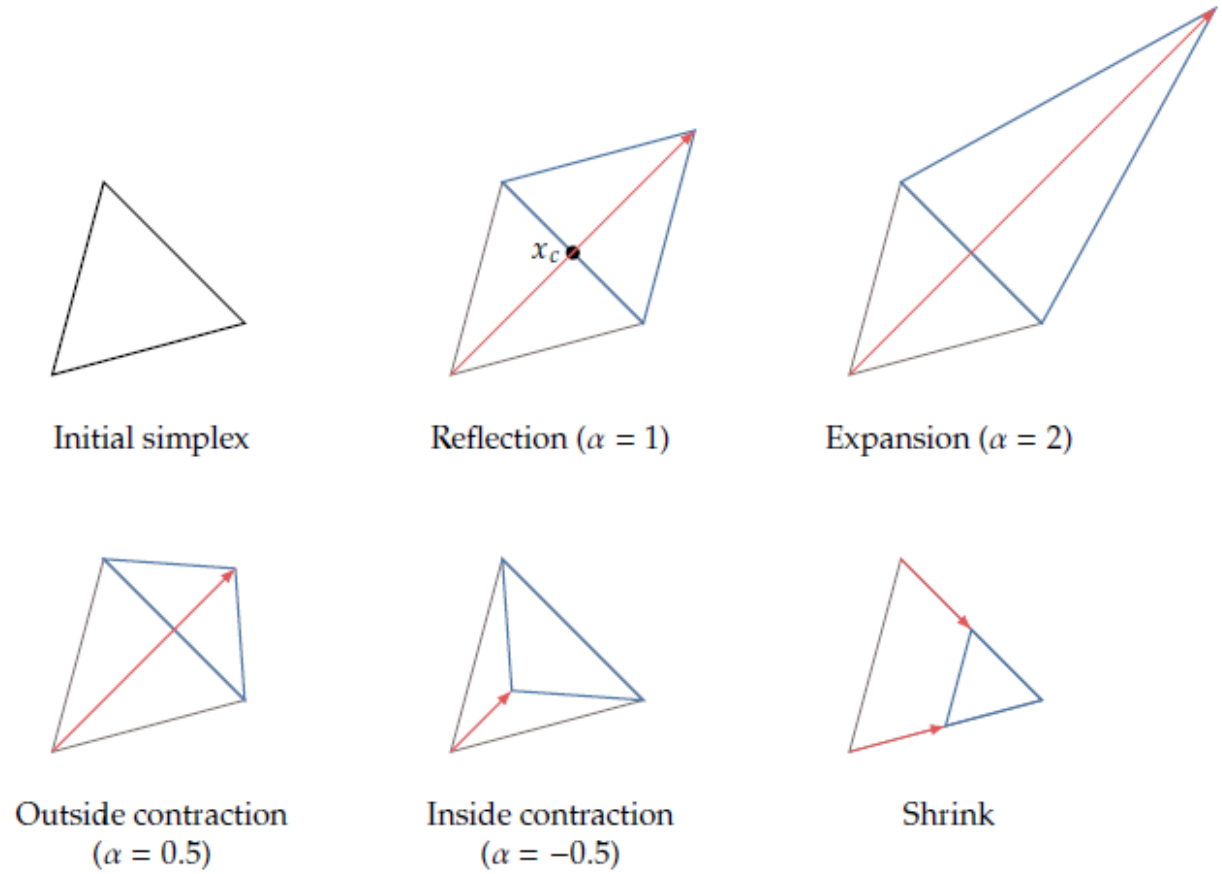

Figure 5.03 Nelder–Mead algorithm operations for $n = 2$.

Source: Martins et Ning, 2021

4. Nelder-Mead algorithm

- Each of these operations, except for the shrinking, generates a new point, given by

$$x = x_C + \alpha\left(x_C - x^{(n)}\right) \tag{5.03}$$

where α is a scalar, and x_C is the centroid of all the points except for the worst one, that is,

$$x_C = \frac{1}{n}\sum_{i=0}^{n-1} x^{(i)} \tag{5.04}$$

- This generates a new point along the line that connects the worst point, $x^{(n)}$, and the centroid of the remaining points, x_C.
- This direction can be seen as a possible descent direction.

4. Nelder-Mead algorithm

- Each iteration aims to replace the worst point with a better one to form a new simplex.
- Each iteration always starts with reflection, which generates a new point using Eq. 5.03 with α=1, as shown in Fig. 5.03.
- If the reflected point is better than the best point, then the "search direction" was a good one, and we go further by performing an expansion using Eq. 5.03 with α=2.
- If the reflected point is between the second-worst and the worst point, then the direction was not great, but it improved somewhat.
- In this case, we perform an outside contraction (α=0.5).
- If the reflected point is worse than our worst point, we try an inside contraction instead (α=-0.5).

4. Nelder-Mead algorithm

- Shrinking is a last-resort operation that we can perform when nothing along the line connecting $x^{(n)}$ and x_C produces a better point.
- This operation consists of reducing the size of the simplex by moving all the points closer to the best point,

$$x^{(i)} = x^{(0)} + \gamma\left(x^{(i)} - x^{(0)}\right) \; for \; i = 1, \ldots, n \qquad (5.05)$$

where γ=0.5.

- Algorithm 5.1 details how a new simplex is obtained for each iteration.
- In each iteration, the focus is on replacing the worst point with a better one instead of improving the best.
- The corresponding flowchart is shown in Fig. 5.04.

4. Nelder-Mead algorithm

Algorithm 5.1: Nelder-Mead algorithm

Inputs:

$x^{(0)}$: Starting point

τ_x: Simplex size tolerances

τ_f: Function value standard deviation tolerances

Outputs:

x^*: Optimal point

---

for $j = 1$ **to** n **do** — Create a simplex with edge length l

  $x^{(j)} = x^{(0)} + s^{(j)}$ — $s^{(j)}$ given by Eq. 7.2

end for

while $\Delta_x > \tau_x$ **or** $\Delta_f > \tau_f$ **do** — Simplex size (Eq. 7.6) and standard deviation (Eq. 7.7)

  Sort $\left\{x^{(0)}, \ldots, x^{(n-1)}, x^{(n)}\right\}$ — Order from the lowest (best) to the highest $f(x^{(j)})$

  $x_c = \frac{1}{n}\sum_{i=0}^{n-1} x^{(i)}$ — The centroid excluding the worst point $x^{(n)}$ (Eq. 7.4)

  $x_r = x_c + \left(x_c - x^{(n)}\right)$ — *Reflection*, Eq. 7.3 with $\alpha = 1$

  if $f(x_r) < f(x^{(0)})$ **then** — Is reflected point is better than the best?

    $x_e = x_c + 2\left(x_c - x^{(n)}\right)$ — *Expansion*, Eq. 7.3 with $\alpha = 2$

    if $f(x_e) < f(x^{(0)})$ **then** — Is expanded point better than the best?

      $x^{(n)} = x_e$ — Accept expansion and replace worst point

    else

      $x^{(n)} = x_r$ — Accept reflection

    end if

  else if $f(x_r) \leq f(x^{(n-1)})$ **then** — Is reflected better than second worst?

    $x^{(n)} = x_r$ — Accept reflected point

  else

    if $f(x_r) > f(x^{(n)})$ **then** — Is reflected point worse than the worst?

      $x_{ic} = x_c - 0.5\left(x_c - x^{(n)}\right)$ — *Inside contraction*, Eq. 7.3 with $\alpha = -0.5$

      if $f(x_{ic}) < f(x^{(n)})$ **then** — Inside contraction better than worst?

        $x^{(n)} = x_{ic}$ — Accept inside contraction

      else

        for $j = 1$ **to** n **do**

          $x^{(j)} = x^{(0)} + 0.5\left(x^{(j)} - x^{(0)}\right)$ — *Shrink*, Eq. 7.5 with $\gamma = 0.5$

        end for

      end if

    else

      $x_{oc} = x_c + 0.5\left(x_c - x^{(n)}\right)$ — *Outside contraction*, Eq. 7.3 with $\alpha = 0.5$

      if $f(x_{oc}) < f(x_r)$ **then** — Is contraction better than reflection?

        $x^{(n)} = x_{oc}$ — Accept outside contraction

      else

        for $j = 1$ **to** n **do**

          $x^{(j)} = x^{(0)} + 0.5\left(x^{(j)} - x^{(0)}\right)$ — *Shrink*, Eq. 7.5 with $\gamma = 0.5$

        end for

      end if

    end if

  end if

end while

4. Nelder-Mead algorithm

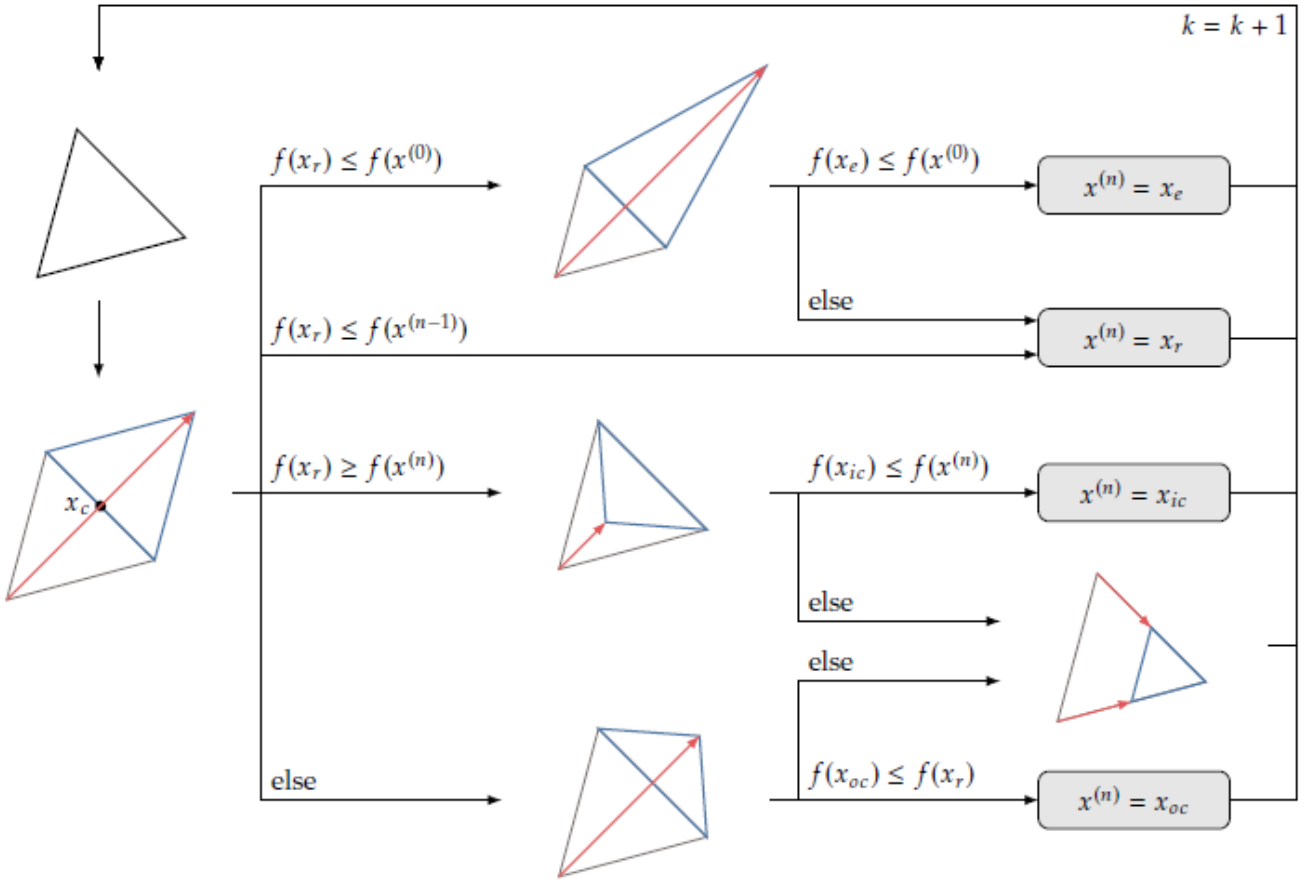

Figure 5.04 Flowchart of Nelder–Mead.

Source: Martins et Ning, 2021

4. Nelder-Mead algorithm

- The cost for each iteration is one function evaluation if the reflection is accepted, two function evaluations if an expansion or contraction is performed, and $n+2$ evaluations if the iteration results in shrinking.
- Although the n evaluations when shrinking can be parallelized, it would not be worthwhile because the other operations are sequential.
- There are a number of ways to quantify the convergence of the simplex method.
- One straightforward way is to use the size of simplex, that is,

$$\Delta_x = \sum_{i=0}^{n-1} \left\| x^{(i)} - x^{(n)} \right\| \qquad (5.06)$$

and specify that it must be less than a certain tolerance.

4. Nelder-Mead algorithm

- Another measure of convergence that can be used is the standard deviation of the function values,

$$\Delta_f = \sqrt{\frac{\sum_{i=0}^{n}\left(f^{(i)} - \bar{f}\right)^2}{n+1}} \qquad (5.07)$$

where $\bar{f}$ is the mean of the n+1 function values.

- Another possible convergence criterion is the difference between the best and worst values in the simplex.
- Nelder–Mead is known for occasionally converging to non-stationary points, so you should check the result if possible.
- Like most direct-search methods, Nelder–Mead cannot directly handle constraints.

4. Nelder-Mead algorithm

- One approach to handling constraints would be to use a penalty method (discussed in Section 10) to form an unconstrained problem.
- In this case, the penalty does not need to be differentiable, so a linear penalty method would suffice.

4. Nelder-Mead algorithm

Example 5.1: Nelder–Mead algorithm applied to the bean function.

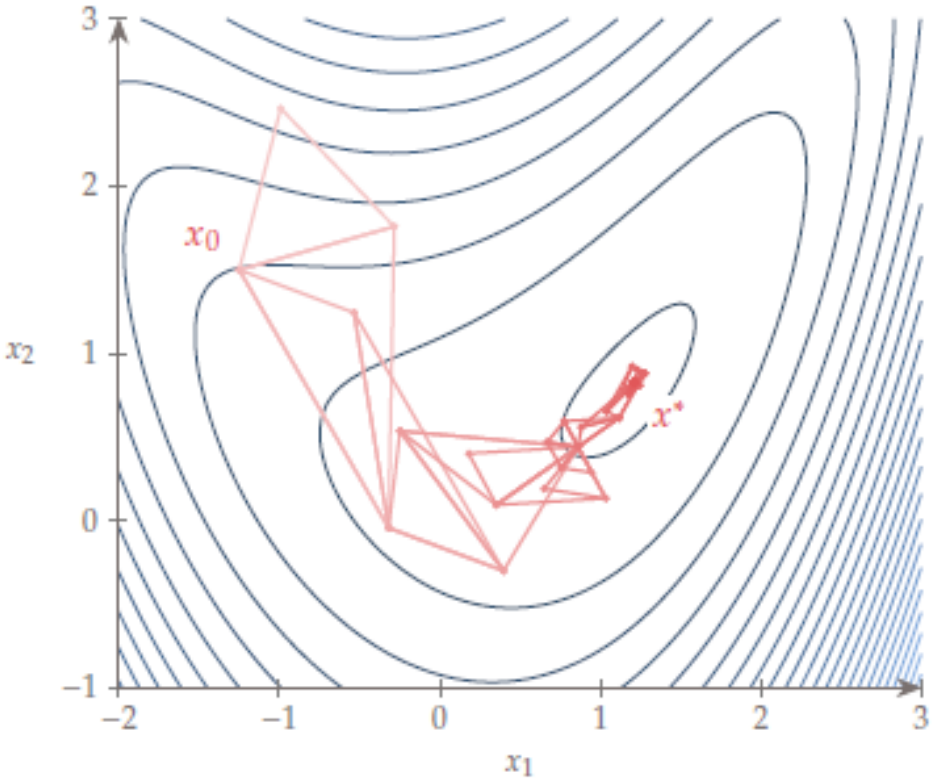

Figure 5.05 Sequence of simplices that minimize the bean function.

Source: Martins et Ning, 2021

5. Generalized Pattern Search

- GPS builds upon the ideas of a coordinate search algorithm.
- In coordinate search, we evaluate points along a mesh aligned with the coordinate directions, move toward better points, and shrink the mesh when we find no improvement nearby.
- Consider a two-dimensional coordinate search for an unconstrained problem.
- At a given point x_k, points that are a distance Δ_k away in all coordinate directions are evaluated, as shown in Fig. 5.06.

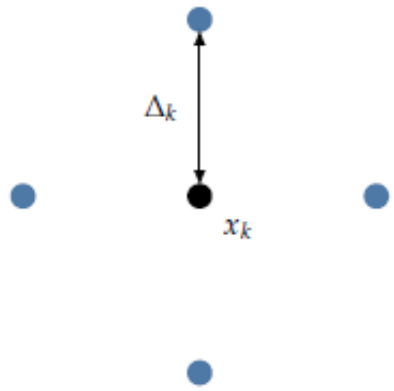

Figure 5.06 Local mesh for a two-dimensional coordinate search at iteration *k*.

Source: Martins et Ning, 2021

5. Generalized Pattern Search

- If the objective function improves at any of these points (four points in this case), the point is recentered with x_{k+1} at the most improved point, the mesh size is kept the same at $\Delta_{k+1}=\Delta_k$, and the next iteration is started.
- Alternatively, if none of the points offers an improvement, the same centre is kept ($x_{k+1}=x_k$) and the mesh is shrunk to $\Delta_{k+1}<\Delta_k$.
- This process repeats until it meets some convergence criteria.
- We now explore various ways in which GPS improves coordinate search.
- Coordinate search moves along coordinate directions, but this is not necessarily desirable.
- Instead, the GPS search directions only need to form a positive spanning set.

5. Generalized Pattern Search

- Given a set of directions $\mathcal{D}=\{d_1, d_1, \ldots, d_{nd}\}$, the set $\mathcal{D}$ is a positive spanning set if the vectors are linearly independent, and a non-negative linear combination of these vectors spans the n-dimensional space.
- Coordinate vectors fulfil this requirement, but there is an infinite number of options.
- Vectors d are referred to as positive spanning directions.
- Only linear combinations with positive multipliers are considered, so in two dimensions, the unit coordinate vectors $\hat{e}_1$ and $\hat{e}_2$ are not sufficient to span two-dimensional space; however, $\hat{e}_1$, $\hat{e}_2$, $-\hat{e}_1$ and $-\hat{e}_2$ are sufficient.
- For a given dimension n, the largest number of vectors that could be used while remaining linearly independent (known as the maximal set) is $2n$.

5. Generalized Pattern Search

- Conversely, the minimum number of possible vectors needed to span the space (known as the minimal set) is $n+1$.
- These sizes are necessary but not sufficient conditions.
- Some algorithms randomly generate a positive spanning set, whereas other algorithms require the user to specify a set based on knowledge of the problem.
- The positive spanning set need not be fixed throughout the optimization.
- A common default for a maximal set is the set of coordinate directions $\pm\hat{e}_i$.

5. Generalized Pattern Search

- In three dimensions, this would be:

$$\mathcal{D} = \{d_1, \ldots., d_6\} = \begin{cases} d_1 = [1,0,0] \\ d_2 = [0,1,0] \\ d_3 = [0,0,1] \\ d_4 = [-1,0,0] \\ d_5 = [0,-1,0] \\ d_6 = [0,0,-1] \end{cases} \quad (5.08)$$

- A potential default minimal set is the positive coordinate directions $+\hat{e}_i$ and a vector filled with -1 (or more generally, the negative sum of the other vectors).
- As an example in three dimensions:

5. Generalized Pattern Search

$$\mathcal{D} = \{d_1, \ldots., d_4\} = \begin{cases} d_1 = [1,0,0] \\ d_2 = [0,1,0] \\ d_3 = [0,0,1] \\ d_4 = [-1,-1,-1] \end{cases} \quad (5.09)$$

- Figure 5.07 shows an example maximal set (four vectors) and minimal set (three vectors) for a two-dimensional problem.

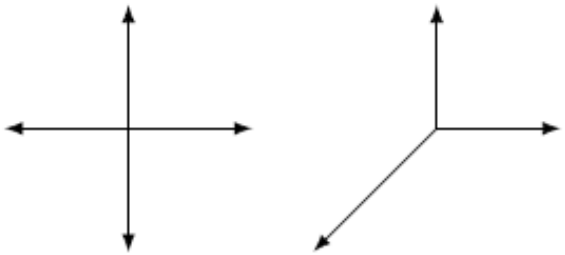

Figure 5.07 A maximal set of positive spanning vectors in two dimensions (left) and a minimal set (right).

Source: Martins et Ning, 2021

5. Generalized Pattern Search

- These direction vectors are then used to create a mesh.
- Given a current centre point x_k, which is the best point found so far, and a mesh size Δ_k, the mesh is created as follows:

$$x_k = \Delta_k d \quad for\ all \quad d \in \mathcal{D} \tag{5.10}$$

- For example, in two dimensions, if the current point is x_k=[1,1], the mesh size is Δ_k =0.5, and we use the coordinate directions for *d*, then the mesh points would be {[1,1.5],[1,0.5],[0.5,1],[1.5,1],}.
- The evaluation of points in the mesh is called polling or a poll.
- In the coordinate search example, every point in the mesh is evaluated, which is usually inefficient.
- More typically, opportunistic polling is used, which terminates polling at the first point that offers an improvement.

5. Generalized Pattern Search

- Figure 5.08 shows a two-dimensional example where the order of evaluation is d_1=[1,0], d_2=[0,1], d_3=[-1,0], d_4=[0,-1].
- Because we found an improvement at d_2, we do not continue evaluating d_3 and d_4.
- Opportunistic polling may not yield the best point in the mesh, but the improvement in efficiency is usually worth the trade-off.

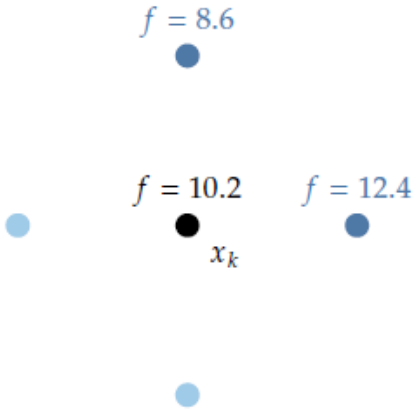

Figure 5.08 A two-dimensional example of opportunistic polling with d_1=[1,0], d_2=[0,1], d_3=[-1,0], d_4=[0,-1]. An improvement in f was found at d_2, so we do not evaluate d_3 and d_4 (shown with a faded colour).

Source: Martins et Ning, 2021

5. Generalized Pattern Search

- Some algorithms add a user option for utilizing a full poll, in which case all points in the mesh are evaluated.
- If more than one point offers a reduction, the best one is accepted.
- Another approach that is sometimes used is called dynamic polling.
- In this approach, a successful poll reorders the direction vectors so that the direction that was successful last time is checked first in the next poll.
- A key feature of GPS is its use of two phases: a search phase and a poll phase.
- The search phase is intended to be global, whereas the poll phase is local, as discussed previously.

5. Generalized Pattern Search

- The search phase is intended to be flexible and is not specified by the GPS algorithm.
- Common options for the search phase include the following:
  - No search phase
  - A mesh search, similar to polling but with large spacing across the domain
  - An alternative solver, such as Nelder–Mead or a genetic algorithm
  - A surrogate model, which could then use any number of solvers that include gradient-based methods. This approach is often used when the function is expensive, and a lower-fidelity surrogate can guide the optimizer to promising regions of the larger design space.
  - Random evaluation using a space-filling method

5. Generalized Pattern Search

- The type of search can even change throughout the optimization.
- Like the polling phase, the goal of the search phase is to find a better point (i.e., $f(x_{k+1})<f(x_k)$) but within a broader domain.
- We begin with a search at every iteration.
- If the search fails to produce a better point, we continue with a poll.
- If a better point is identified in either phase, the iteration ends, and we begin a new search.
- In some variants of this algorithm, a successful poll is followed by another poll.
- Thus, at each iteration, a search and a poll, just a search, or just a poll might be performed.

5. Generalized Pattern Search

- We describe one option for a search procedure based on the same mesh ideas as the polling step.
- The concept is to extend the mesh throughout the entire domain, as shown in Fig. 5.09.

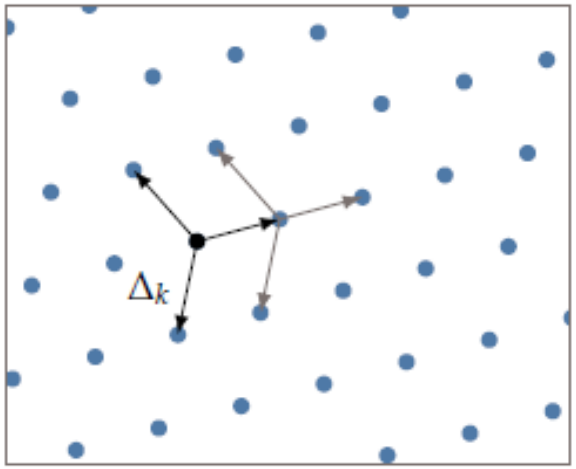

Figure 5.09 Meshing strategy extended across the domain. The same directions (and potentially spacing) are repeated at each mesh point, as indicated by the lighter arrows throughout the entire domain.

Source: Martins et Ning, 2021

5. Generalized Pattern Search

- In this example, the mesh size Δ_k is shared between the search and poll phases.
- However, it is usually more effective if these phases are independent.
- Mathematically, the global mesh can be defined as the set

$$G = \{x_k + \Delta_k D z \quad for\ all \quad z_i \in \mathcal{Z}^+\} \tag{5.11}$$

where D is a matrix, whose columns contain the basis vectors d.

- Vector z consists of nonnegative integers, and we consider all possible combinations of integers that fall within the bounds of the domain.
- A fixed number of search evaluation points are chosen and points are randomly selected from the global mesh for the search strategy.

5. Generalized Pattern Search

- If improvement is found among that set, then we recentre x_{k+1} at this improved point, grow the mesh ($\Delta_{k+1} > \Delta_k$), and end the iteration (and then restart the search).
- A simple search phase along these lines is described in Algorithm 5.2.
- Termination is often triggered simply by a user-specified maximum number of iterations.
- However, other convergence criteria are sometimes used, such as a threshold on mesh size or a threshold on the improvement in the function value across multiple iterations.
- The method can handle linear and nonlinear constraints.
- For linear constraints, one effective strategy is to change the positive spanning directions so that they align with any linear constraints that are nearby (Fig. 5.10).

5. Generalized Pattern Search

- For nonlinear constraints, penalty approaches (Section 10) are applicable, although the filter method is another effective approach.
- An overview of a generalized pattern-search algorithm is shown in Algorithm 5.3.

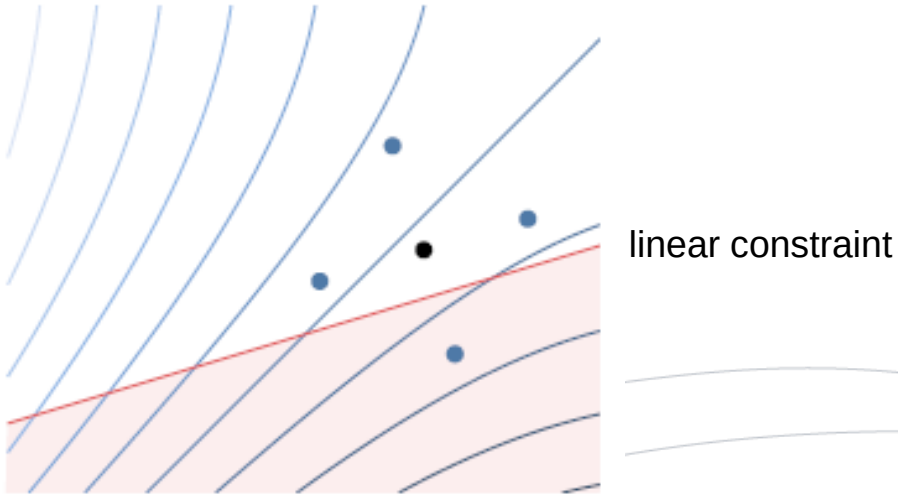

Figure 5.10 Mesh direction changed during optimization to align with linear constraints when close to the constraint.

Source: Martins et Ning, 2021

5. Generalized Pattern Search

Algorithm 5.2: An example search phase for GPS

Inputs:

x_k: Center point

Δ_k: Mesh size

$\underline{x}$, $\overline{x}$: Lower and upper bounds

D: Column vectors representing positive spanning set

n_s: Number of search points

f_k: The function previously evaluated at $f(x_k)$

Outputs:

success: True if successful in finding improved point

x_{k+1}: New center point

f_{k+1}: Corresponding function value

---

success = false
$x_{k+1} = x_k$
$f_{k+1} = f_k$
Construct global mesh G, using directions D, mesh size Δ_k, and bounds $\underline{x}$, $\overline{x}$
for $i = 1$ **to** n_s **do**
  Randomly select $s \in G$
  Evaluate $f_s = f(s)$
  if $f_s < f_k$ **then**
    $x_{k+1} = s$
    $f_{k+1} = f_s$
    success = true
    break
  end if
end for

5. Generalized Pattern Search

Algorithm 5.3: Generalized Pattern Search

Inputs:

x_0: Starting point

$\underline{x}, \overline{x}$: Lower and upper bounds

Δ_0: Starting mesh size

n_s: Number of search points

$k_{\max}$: Maximum number of iterations

Outputs:

x^*: Best point

f^*: Corresponding function value

$D = [I, -I]$ where I is $(n \times n)$ — A coordinate aligned maximal positive spanning set (for example)

$k = 0$

$x_k = x_0$

Evaluate $f_k = f(x_k)$

while $k < k_{\max}$ **do** — Or other termination criteria

  search_success, x_{k+1}, f_{k+1} = search(x_k, Δ_k, f_k) — Any search strategy

  if search_success **then**

    $\Delta_{k+1} = \min(2\Delta_k, \Delta_{\max})$ — Or some other growth rate

    $k = k + 1$

    continue — Move on to next iteration

  else — Poll

    poll_success = false

    for $j = 1$ to n_d **do**

      $s = x_k + \Delta_k d_j$ — Where d_j is a column of D

      Evaluate $f_s = f(s)$

      if $f_s < f_k$ **then**

        $x_{k+1} = s$

        $f_{k+1} = f_s$

        $\Delta_{k+1} = \Delta_k$

        k = k + 1

        poll_success = true

        break

      end if

    end for

  end if

  if not poll_success **then**

    $x_{k+1} = x_k$

    $f_{k+1} = f_k$

    $\Delta_{k+1} = 0.5\Delta_k$ — Shrink

  end if

  k = k + 1

end while

5. Generalized Pattern Search

Example 5.2: Minimization of a multimodal function with GPS.

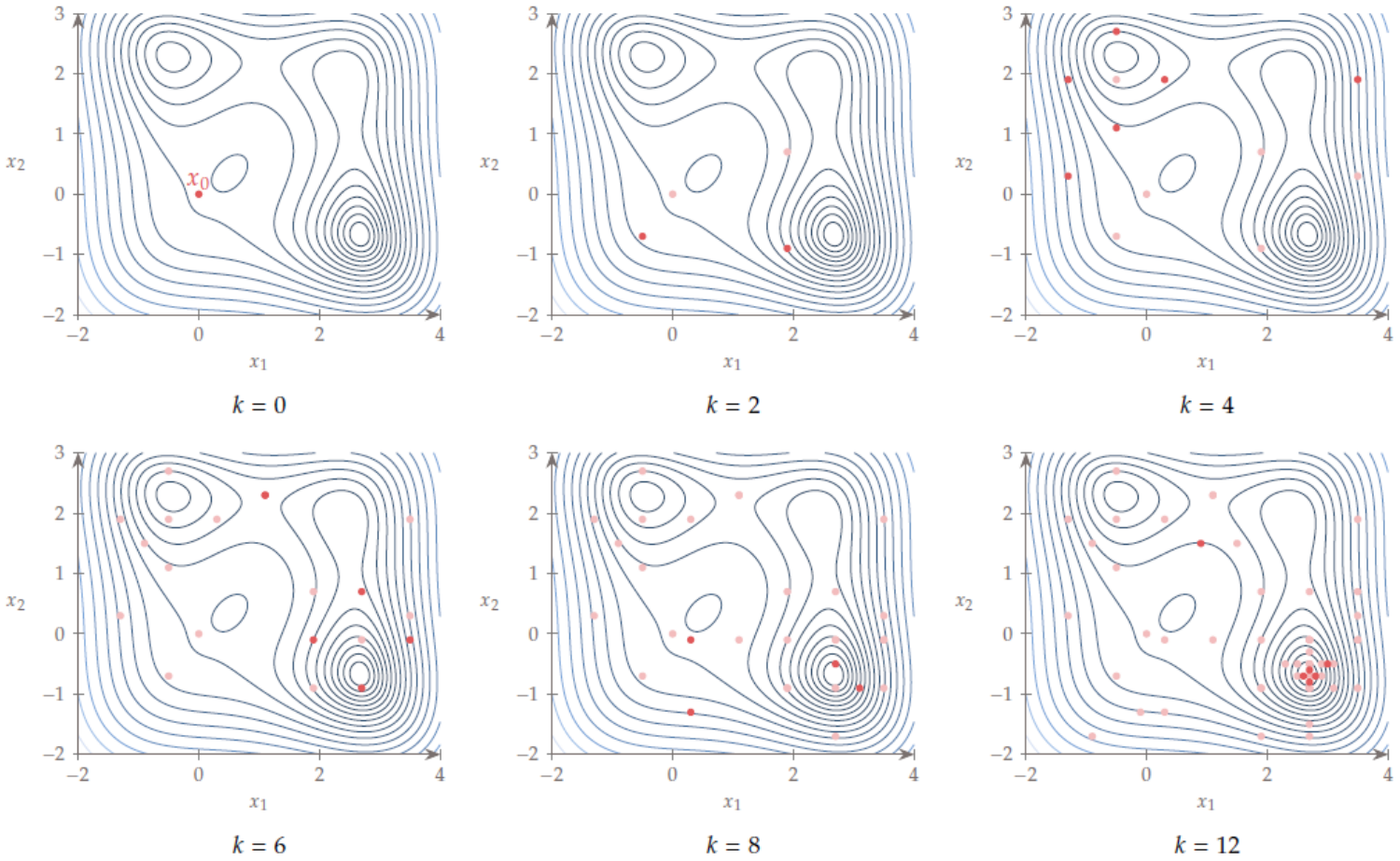

Figure 5.11 Convergence history of a GPS algorithm on the multimodal Jones function.

Source: Martins et Ning, 2021

6. DIRECT Algorithm

- The DIRECT algorithm is different from the other gradient-free optimization algorithms in this chapter in that it is based on mathematical arguments.
- This is a deterministic method guaranteed to converge to the global optimum under conditions that are not too restrictive (although it might require a prohibitive number of function evaluations).
- One way to ensure that we find the global optimum within a finite design space is by dividing this space into a regular rectangular grid and evaluating every point in this grid.
- This is called an exhaustive search, and the precision of the minimum depends on how fine the grid is.
- The cost of this brute-force strategy is high and goes up exponentially with the number of design variables.

6. DIRECT Algorithm

- The DIRECT method relies on a grid, but it uses an adaptive meshing scheme that dramatically reduces the cost.
- It starts with a single *n*-dimensional hypercube that spans the whole design space - like many other gradient-free methods, DIRECT requires upper and lower bounds on all the design variables.
- Each iteration divides this hypercube into smaller ones and evaluates the objective function at the centre of each of these.
- At each iteration, the algorithm only divides rectangles determined to be potentially optimal.
- The fundamental strategy in the DIRECT method is how it determines this subset of potentially optimal rectangles, which is based on the mathematical concept of Lipschitz continuity.

6. DIRECT Algorithm

- We start by explaining Lipschitz continuity and then describe an algorithm for finding the global minimum of a one-dimensional function using this concept - Shubert's algorithm.
- Although Shubert's algorithm is not practical in general, it will help us understand the mathematical rationale for the DIRECT algorithm.
- Then we explain the DIRECT algorithm for one-dimensional functions before generalizing it for *n* dimensions.

6.1. Lipschitz constant

- Consider the single-variable function $f(x)$ shown in Fig. 5.12.
- For a trial point x^*, a cone is drawn with slope L by drawing the lines to the left and right, respectively:

$$f_+(x) = f(x^*) + L(x - x^*) \quad (5.12)$$

$$f_-(x) = f(x^*) - L(x - x^*) \quad (5.13)$$

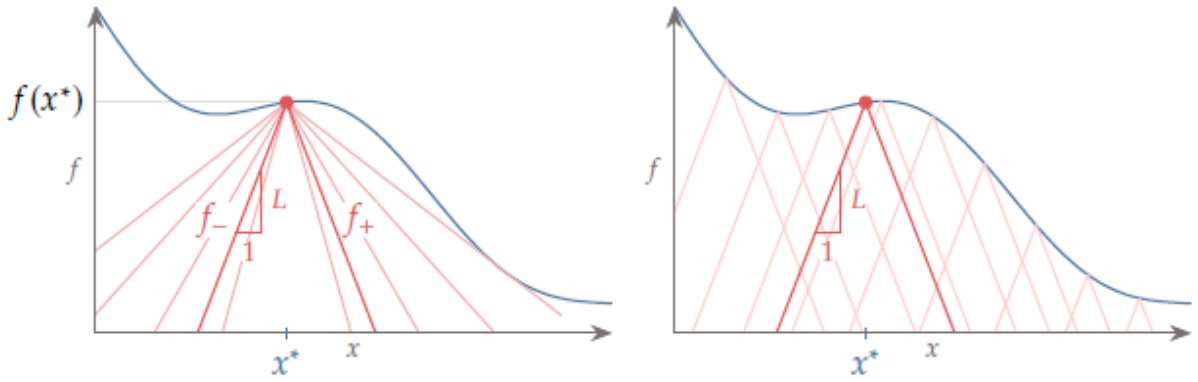

Figure 5.12 From a given trial point x^* we can draw a cone with slope L (left). For a function to be Lipschitz continuous, we need all cones with slope L to lie under the function for all points in the domain (right).

Source: Martins et Ning, 2021

6.1. Lipschitz constant

- This cone is shown in Fig. 5.12 (left), as well as cones corresponding to other values of L.
- A function f is said to be Lipschitz continuous if there is a cone slope L such that the cones for all possible trial points in the domain remain under the function.
- This means that there is a positive constant k such that

$$|f(x) - f(x^*)| \leq L|x - x^*| \quad for\ all \quad x, x^* \in D \qquad (5.14)$$

where D is the function domain.

- Graphically, this condition means a cone with slope L can be drawn from any trial point evaluation $f(x^*)$ such that the function is always bounded by the cone, as shown in Fig. 5.12 (right).

6.1. Lipschitz constant

- Any L that satisfies Eq. 5.14 is a Lipschitz constant for the corresponding function.

6.2. Shubert's algorithm

- If a Lipschitz constant for a single-variable function is known, Shubert's algorithm can find the global minimum of that function.
- Because the Lipschitz constant is not available in the general case, the DIRECT algorithm is designed to not require this constant.
- However, we explain Shubert's algorithm first because it provides some of the basic concepts used in the DIRECT algorithm.
- Shubert's algorithm starts with a domain within which we want to find the global minimum - [a,b] in Fig. 5.13.
- Using the property of the Lipschitz constant L defined in Eq. 5.14, we know that the function is always above a cone of slope L evaluated at any point in the domain.

6.2. Shubert's algorithm

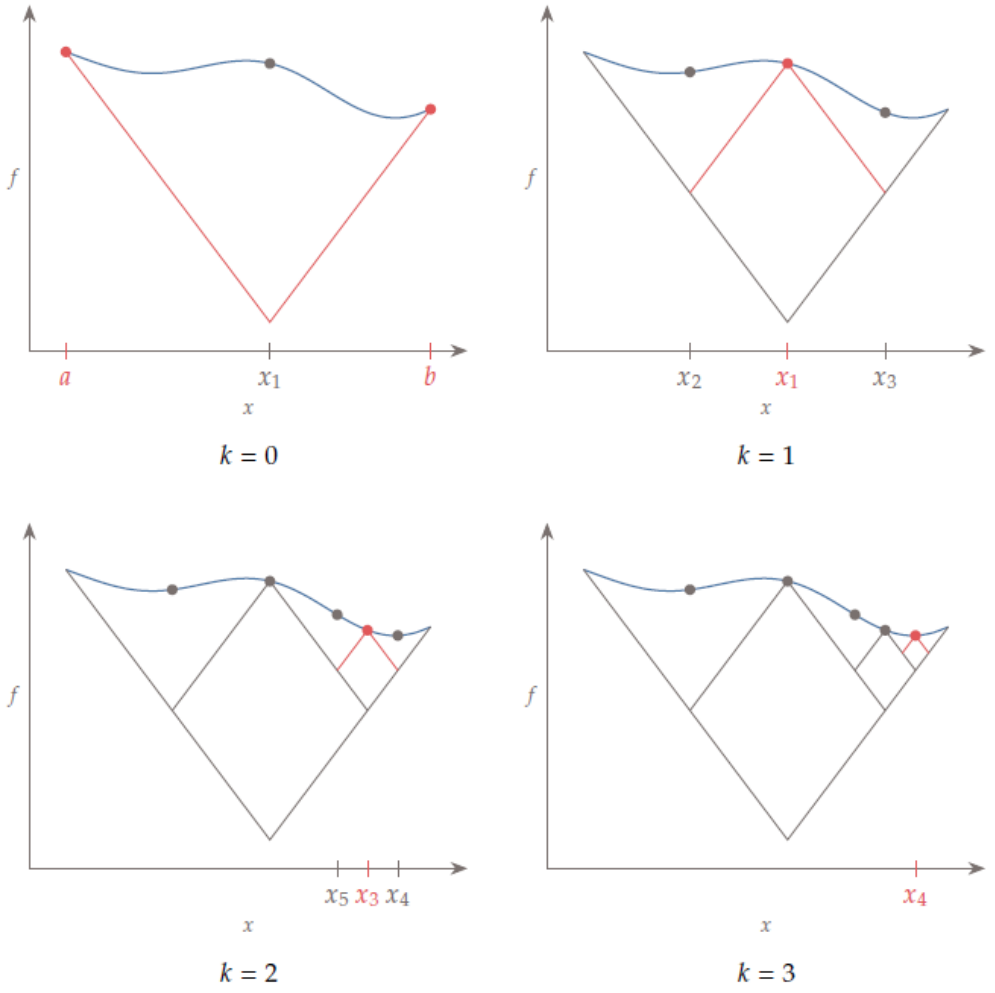

Figure 5.13 Shubert's algorithm requires an initial domain and a valid Lipschitz constant and then increases the lower bound of the global minimum with each successive iteration.

Source: Martins et Ning, 2021

6.2. Shubert's algorithm

- Shubert's algorithm starts by sampling the endpoints of the interval within which we want to find the global minimum ([a,b] in Fig. 5.13).
- A first lower bound on the global minimum is established by finding the cone's intersection (x_1 in Fig. 5.13, k=0) for the extremes of the domain.
- The function is evaluated at x_1 and a cone can be drawn about this point to find two more intersections (x_2 and x_3).
- Because these two points always intersect at the same objective lower bound value, they both need to be evaluated.
- Each subsequent iteration of Shubert's algorithm adds two new points to either side of the current point.
- These two points are evaluated, and the lower bounding function is updated with the resulting new cones.

6.2. Shubert's algorithm

- We then iterate by finding the two points that minimize the new lower bounding function, evaluating the function at these points, updating the lower bounding function, and so on.
- The lowest bound on the function increases at each iteration and ultimately converges to the *global* minimum.
- At the same time, the segments in x decrease in size.
- The lower bound can switch from distinct regions as the lower bound in one region increases beyond the lower bound in another region.
- The two significant shortcomings of Shubert's algorithm are that
  1. a Lipschitz constant is usually not available for a general function
  2. it is not easily extended to n dimensions
- The DIRECT algorithm addresses these two shortcomings.

6.3. One-dimensional DIRECT

- Before explaining the n-dimensional DIRECT algorithm, we introduce the one-dimensional version based on principles similar to those of the Shubert algorithm.
- Like Shubert's method, DIRECT starts with the domain $[a,b]$
- However, instead of sampling the endpoints a and b, it samples the midpoint.
- Consider the closed domain $[a,b]$ shown in Fig. 5.14 (left).
- For each segment, the objective function is evaluated at the segment's midpoint.
- In the first segment, which spans the whole domain, the midpoint is $c_0=(a+b)/2$.
- Assuming some value of L, which is not known and will not be needed, the lower bound on the minimum would be $f(c)-L(b-a)/2$.

6.3. One-dimensional DIRECT

- We want to increase this lower bound on the function minimum by dividing this segment further.
- To do this in a regular way that reuses previously evaluated points and can be repeated indefinitely, we divide it into three segments, as shown in Fig. 5.14 (right).

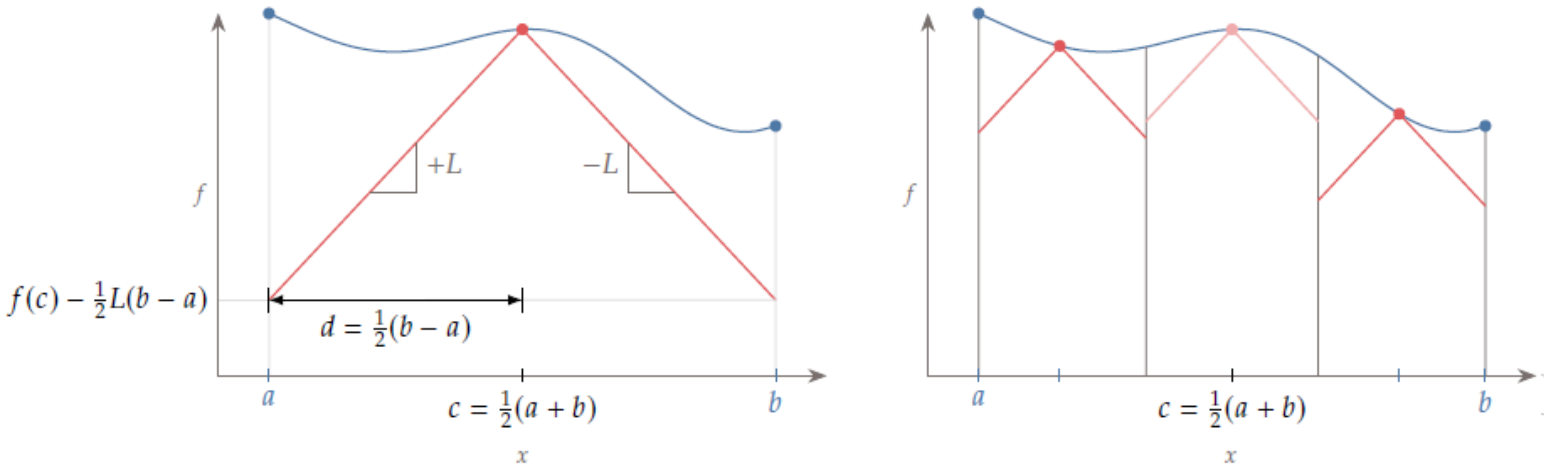

Figure 5.14 The DIRECT algorithm evaluates the middle point (left), and each successive iteration trisects the segments that have the greatest potential (right).

Source: Martins et Ning, 2021

6.3. One-dimensional DIRECT

- Now we have increased the lower bound on the minimum.
- Unlike the Shubert algorithm, the lower bound is a discontinuous function across the segments, as shown in Fig. 5.14 (right).
- Instead of continuing to divide every segment into three other segments, we only divide segments selected according to a potentially optimal criterion.
- To better understand this criterion, consider a set of segments $[a_i,b_i]$ at a given DIRECT iteration, where segment i has a half-length $d_i=(b_i-a_i)/2$ and a function value $f(c_i)$ evaluated at the segment centre $c_i=(a_i+b_i)/2$.
- If we plot $f(c_i)$ versus d_i for a set of segments, we get the pattern shown in Fig. 5.15.

6.3. One-dimensional DIRECT

- The overall rationale for the potentially optimal criterion is that two metrics quantify this potential: the size of the segment and the function value at the centre of the segment.
- The larger the segment is, the greater the potential for that segment to contain the global minimum.
- The lower the function value, the greater that potential is as well.

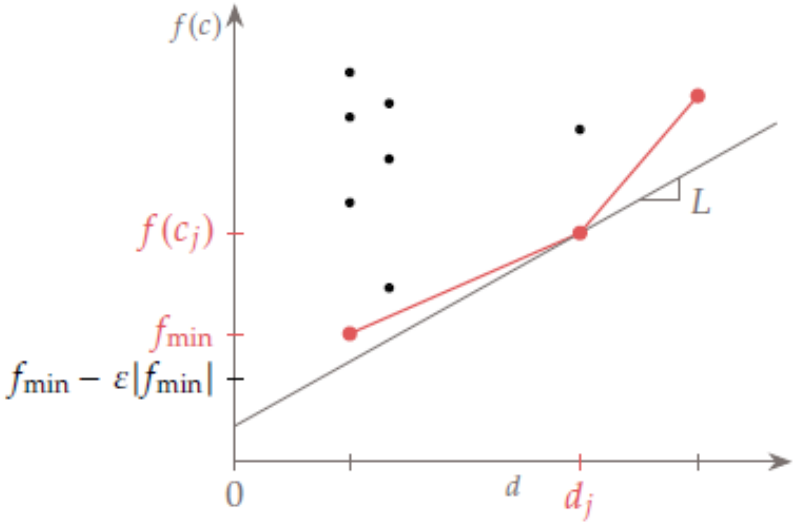

Figure 5.15 Potentially optimal segments in the DIRECT algorithm are identified by the lower convex hull of this plot.

Source: Martins et Ning, 2021

6.3. One-dimensional DIRECT

- For a set of segments of the same size, we know that the one with the lowest function value has the best potential and should be selected.
- If two segments have the same function value and different sizes, we should select the one with the largest size.
- For a general set of segments with various sizes and value combinations, there might be multiple segments that can be considered potentially optimal.
- Potentially optimal segments are identified as follows.
- If we draw a line with a slope corresponding to a Lipschitz constant L from any point in Fig. 5.15, the intersection of this line with the vertical axis is a bound on the objective function for the corresponding segment.

6.3. One-dimensional DIRECT

- Therefore, the lowest bound for a given L can be found by drawing a line through the point that achieves the lowest intersection.
- However, L is not known, and we do not want to assume a value because we do not want to bias the search.
- If L were high, it would favour dividing the larger segments.
- Low values of L would result in dividing the smaller segments.
- The DIRECT method hinges on considering all possible values of L, effectively eliminating the need for this constant.
- To eliminate the dependence on L, we select all the points for which there is a line with slope L that does not go above any other point.

6.3. One-dimensional DIRECT

- This corresponds to selecting the points that form a lower convex hull, as shown by the piecewise linear function in Fig. 5.15.
- This establishes a lower bound on the function for each segment size.
- Mathematically, a segment j in the set of current segments S is said to be potentially optimal if there is a $L \geq 0$ such that

$$f(c_j) - Ld_j \leq f(c_i) - Ld_j \quad for\ all \quad i \in S \tag{5.15}$$

$$f(c_j) - Ld_j \leq f_{min} - \varepsilon |f_{min}| \tag{5.16}$$

where f_{min} is the best current objective function value, and ε is a small positive parameter.

- The first condition corresponds to finding the points in the lower convex hull mentioned previously.

6.3. One-dimensional DIRECT

- The second condition in Eq. 5.16 ensures that the potential minimum is better than the lowest function value found so far by at least a small amount.
- This prevents the algorithm from becoming too local, wasting function evaluations in search of smaller function improvements.
- The parameter ε balances the search between local and global.
- A typical value is $\varepsilon = 10^{-4}$, and its range is usually such that $10^{-7} \leq \varepsilon \leq 10^{-2}$.
- There are efficient algorithms for finding the convex hull of an arbitrary set of points in two dimensions, such as the Jarvis march.

6.3. One-dimensional DIRECT

- These algorithms are more than we need because we only require the lower part of the convex hull, so the algorithms can be simplified for our purposes.
- As in the Shubert algorithm, the division might switch from one part of the domain to another, depending on the new function values.
- Compared with the Shubert algorithm, the DIRECT algorithm produces a discontinuous lower bound on the function values, as shown in Fig. 5.16.

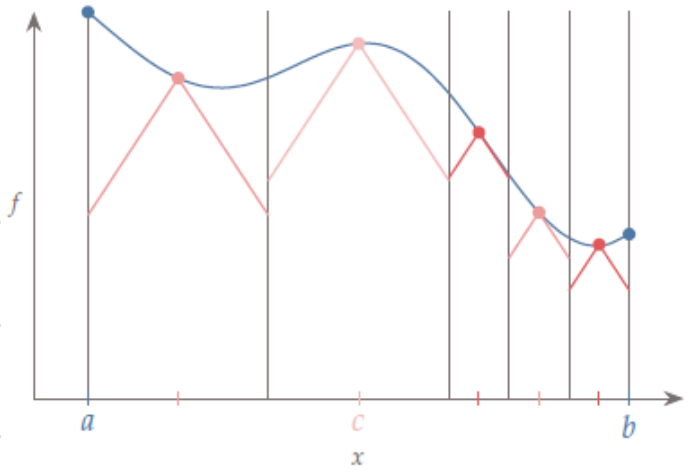

Figure 5.16 The lower bound for the DIRECT method is discontinuous at the segment boundaries.

Source: Martins et Ning, 2021

6.4. DIRECT in *n* dimensions

- The *n*-dimensional DIRECT algorithm is similar to the one-dimensional version but becomes more complex.
- The main difference is that we deal with hyperrectangles instead of segments.
- A hyperrectangle can be defined by its centre-point position *c* in *n*-dimensional space and a half-length in each direction *i*, δ_{ei}, as shown in Fig. 5.17.
- The DIRECT algorithm assumes that the initial dimensions are normalized so that we start with a hypercube.

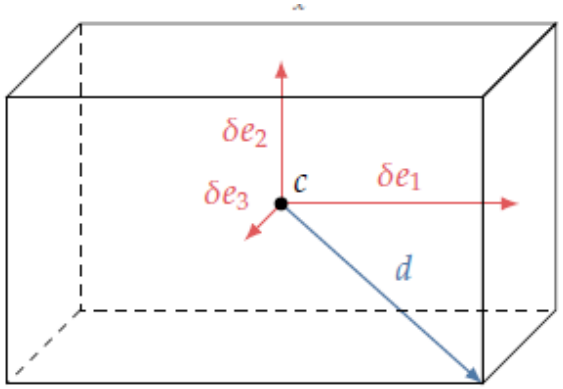

Figure 5.17 Hyperrectangle in three dimensions, where *d* is the maximum distance between the centre and the vertices, and δ_{ei} is the half-length in each direction *i*.

Source: Martins et Ning, 2021

6.4. DIRECT in *n* dimensions

- To identify the potentially optimal rectangles at a given iteration, we use exactly the same conditions in Eqs. 5.15 and 5.16, but c_i is now the centre of the hyperrectangle, and d_i is the maximum distance from the centre to a vertex.
- The explanation illustrated in Fig. 5.15 still applies in the *n*-dimensional case and still involves simply finding the lower convex hull of a set of points with different combinations of *f* and *d*.
- The main complication introduced in the *n*-dimensional case is the division (trisection) of a selected hyperrectangle.
- The question is which directions should be divided first.

6.4. DIRECT in *n* dimensions

- The logic to handle this in the DIRECT algorithm is to prioritize reducing the dimensions with the maximum length, ensuring that hyperrectangles do not deviate too much from the proportions of a hypercube.
- First, we select the set of the longest dimensions for division (there are often multiple dimensions with the same length).
- Among this set of the longest dimensions, we select the direction that has been divided the least over the whole history of the search.
- If there are still multiple dimensions in the selection, we simply select the one with the lowest index.
- Algorithm 5.4 details the full algorithm.

6.4. DIRECT in *n* dimensions

- Figure 5.18 shows the first three iterations for a two-dimensional example and the corresponding visualization of the conditions expressed in Eqs. 5.15 and 5.16.
- We start with a square that contains the whole domain and evaluate the centre point.
- The value of this point is plotted on the *f*–*d* plot on the far right.
- The first iteration trisects the starting square along the first dimension and evaluates the two new points.
- The values for these three points are plotted in the second column from the right in the *f*–*d* plot, where the centre point is reused, as indicated by the arrow and the matching colour.
- At this iteration, we have two points that define the convex hull.

6.4. DIRECT in *n* dimensions

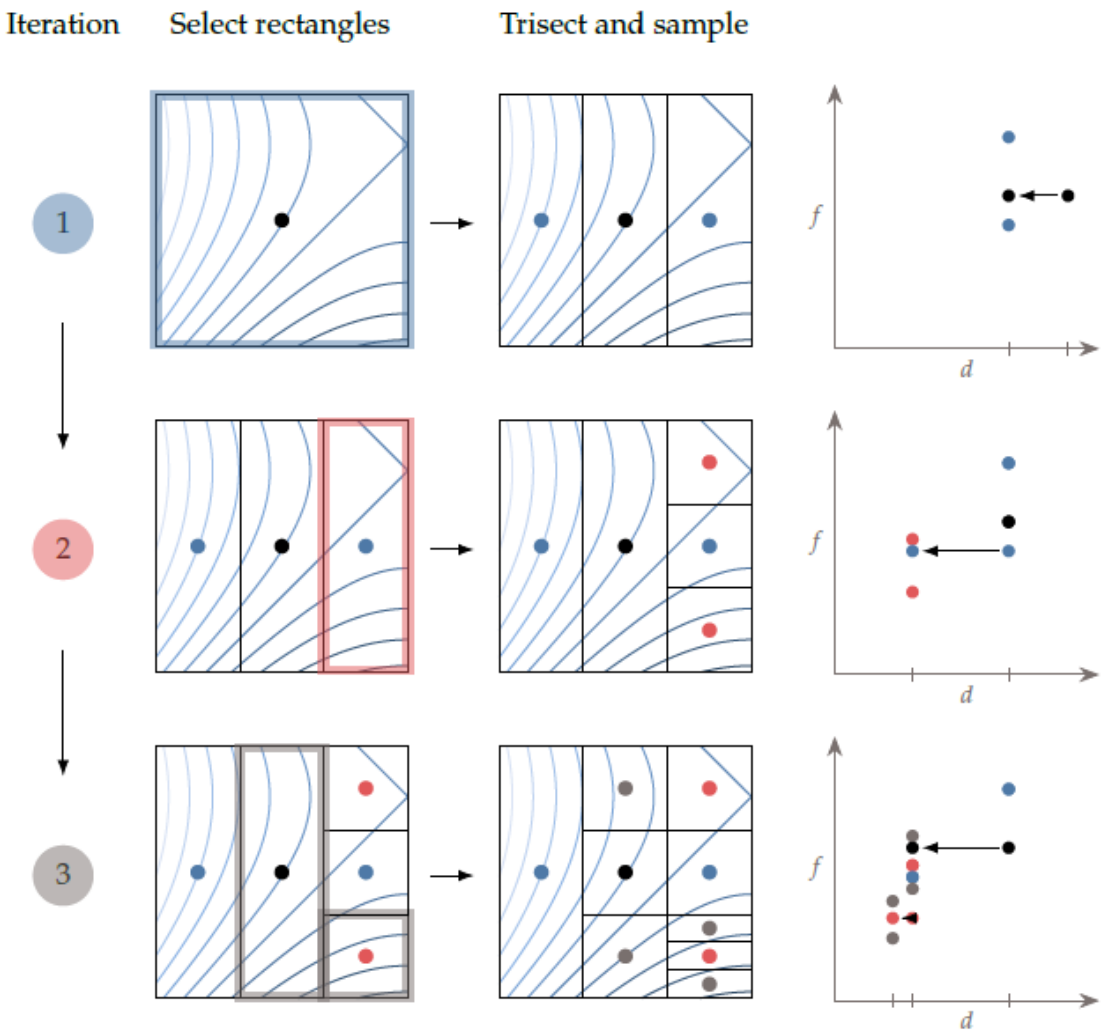

Figure 5.18 DIRECT iterations for two-dimensional case (left) and corresponding identification of potentially optimal rectangles (right).

Source: Martins et Ning, 2021

6.4. DIRECT in *n* dimensions

- In the second iteration, we have three rectangles of the same size, so we divide the one with the lowest value and evaluate the centres of the two new rectangles (which are squares in this case).
- We now have another column of points in the *f*–*d* plot corresponding to a smaller *d* and an additional point that defines the lower convex hull.
- Because the convex hull now has two points, we trisect two different rectangles in the third iteration.

6.4. DIRECT in n dimensions

Algorithm 5.4: DIRECT in n-dimensions

Inputs:

$\underline{x}, \overline{x}$: Lower and upper bounds

Outputs:

x^*: Optimal point

$k = 0$ — Initialize iteration counter

Normalize bounded space to hypercube and evaluate its center, c_0

$f_{\min} = f(c_0)$ — Stores the minimum function value so far

Initialize $t(i) = 0$ for $i = 1, \ldots, n$ — Counts the times dimension i has been trisected

while not converged **do**

  Find set S of potentially optimal hyperrectangles

  for each hyperrectangle in S **do**

    Find the set I of dimensions with maximum side length

    Select i in I with the lowest $t(i)$, breaking ties in favor of lower i

    Divide the rectangle into thirds along dimension i

    $t(i) = t(i) + 1$

    Evaluate the center points of the outer two hyperrectangles

    Update $f_{\min}$ based on these evaluations

  end for

  $k = k + 1$ — Increment iteration counter

end while

6.4. DIRECT in *n* dimensions

Example 5.3: Minimization of a multimodal function (Jones function) with DIRECT.

$$f(x_1, x_2) = x_1^4 + x_2^4 - 4x_1^3 - 3x_2^3 + 2x_1^2 + 2x_1x_2$$

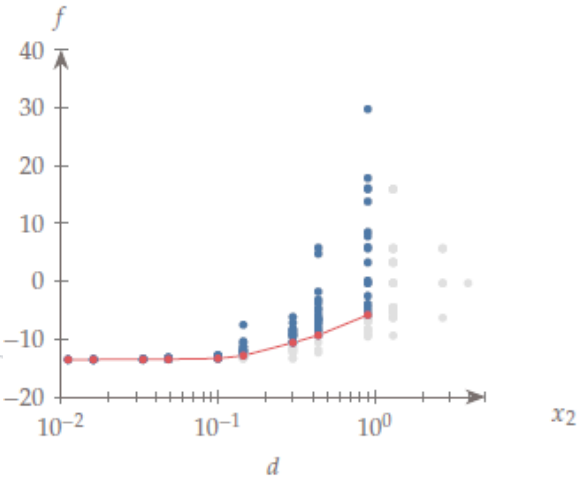

Figure 5.19 Potentially optimal rectangles for the DIRECT iterations shown in Fig. 5.20.

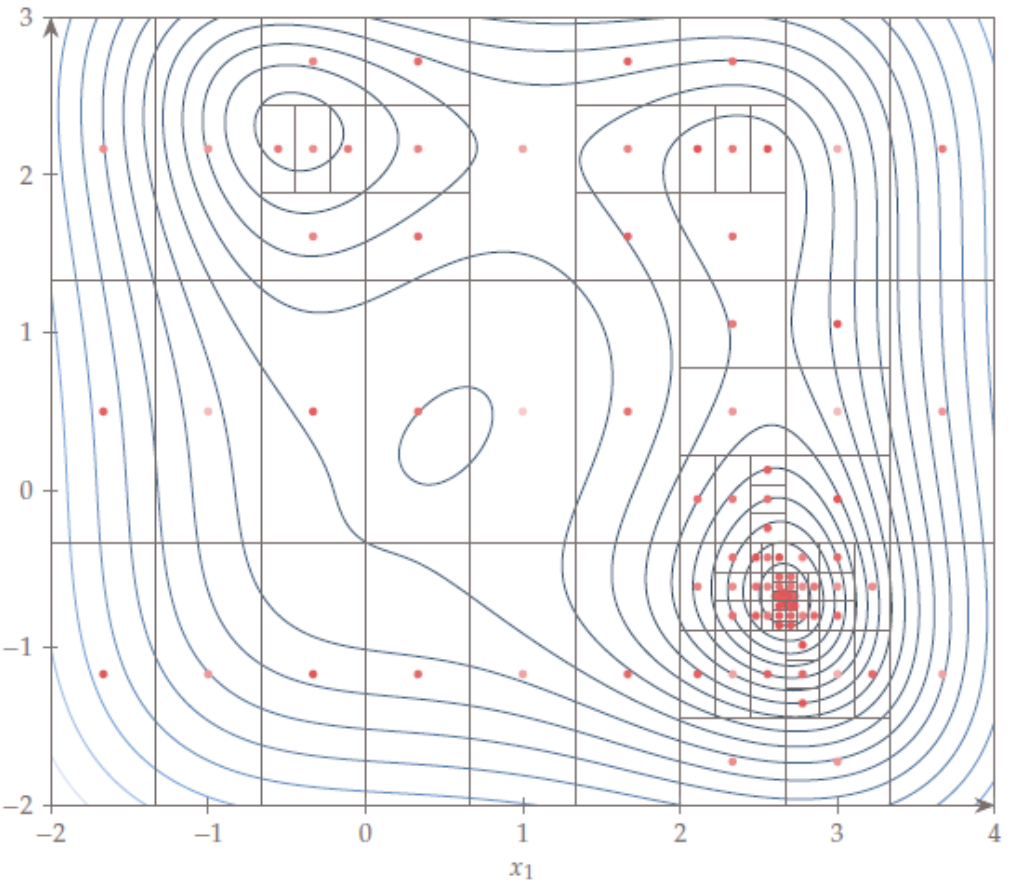

Figure 5.20 The DIRECT method quickly determines the region with the global minimum of the Jones function after briefly exploring the regions with other minima.

Source: Martins et Ning, 2021

7. Genetic algorithms

- Genetic algorithms (GAs) are the most well-known and widely used type of evolutionary algorithm.
- They were also among the earliest to have been developed.
- Like many evolutionary algorithms, GAs are population based.
- The optimization starts with a set of design points (the population) rather than a single starting point, and each optimization iteration updates this set in some way.
- Each iteration in the GA is called a generation, and each generation has a population with n_p points.
- A chromosome is used to represent each point and contains the values for all the design variables, as shown in Fig. 5.21.
- Each design variable is represented by a gene.
- As we will see later, there are different ways for genes to represent the design variables.

7. Genetic algorithms

- GAs evolve the population using an algorithm inspired by biological reproduction and evolution using three main steps:
  1. selection
  2. crossover
  3. mutation
- Selection is based on natural selection, where members of the population that acquire favourable adaptations are more likely to survive longer and contribute more to the population gene pool.

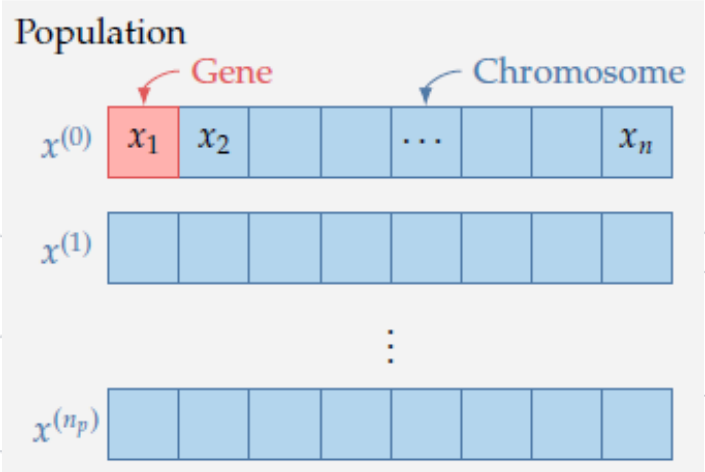

Figure 5.21 Each GA iteration involves a population of design points, where each design is represented by a chromosome, and each design variable is represented by a gene.

Source: Martins et Ning, 2021

7. Genetic algorithms

- Crossover is inspired by chromosomal crossover, which is the exchange of genetic material between chromosomes during sexual reproduction.
- Mutation mimics genetic mutation, which is a permanent change in the gene sequence that occurs naturally.
- Algorithm 5.5 and Fig. 5.22 show how these three steps perform optimization.
- Although most GAs follow this general procedure, there is a great degree of flexibility in how the steps are performed, leading to many variations in GAs.

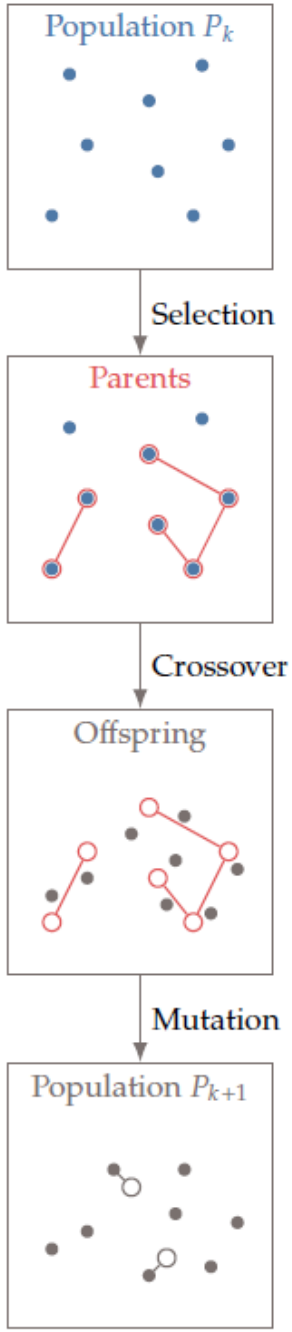

Figure 5.22 At each GA iteration, pairs of parents are selected from the population to generate the offspring through crossover, which becomes the new population.

Source: Martins et Ning, 2021

7. Genetic algorithms

- For example, there is no single method specified for the generation of the initial population, and the size of that population varies.
- Similarly, there are many possible methods for selecting the parents, generating the offspring, and selecting the survivors.
- Here, the new population (P_{k+1}) is formed exclusively by the offspring generated from the crossover.
- However, some GAs add an extra selection process that selects a surviving population of size n_p among the population of parents and offspring.

7. Genetic algorithms

Algorithm 5.5: Genetic algorithm.

Inputs:

$\underline{x}, \overline{x}$: Lower and upper bounds

Outputs:

x^*: Best point

f^*: Corresponding function value

$k = 0$

$P_k = \left\{x^{(1)}, x^{(2)}, \ldots, x^{(n_p)}\right\}$ — Generate initial population

while $k < k_{\max}$ **do**

  Compute $f(x)$ for all $x \in P_k$ — Evaluate objective function

  Select $n_p/2$ parent pairs from P_k for crossover — Selection

  Generate a new population of n_p offspring (P_{k+1}) — Crossover

  Randomly mutate some points in the population — Mutation

  $k = k + 1$

end while

7. Genetic algorithms

- In addition to the flexibility in the various operations, GAs use different methods for representing the design variables.
- The design variable representation can be used to classify genetic algorithms into two broad categories: binary-encoded and real-encoded genetic algorithms.
- Binary-encoded algorithms use bits to represent the design variables, whereas the real-encoded algorithms keep the same real value representation used in most other algorithms.
- The details of the operations in Algorithm 5.5 depend on whether we are using one or the other representation, but the principles remain the same.
- In the rest of this section, we describe a particular way of performing these operations for each of the possible design variable representations.

7.1. Binary-encoded Genetic Algorithms

- The original GAs were based on binary encoding because they more naturally mimic chromosome encoding.
- Binary-coded GAs are applicable to discrete or mixed-integer problems.
- When using binary encoding, we represent each variable as a binary number with m bits.
- Each bit in the binary representation has a location, i, and a value, b_i (which is either 0 or 1).
- To represent a real-valued variable, a finite interval $x \in [\underline{x}, \overline{x}]$ needs to be considered first, which can then be divided into 2^m-1 intervals.
- The size of the interval is given by

$$\Delta x = \frac{\overline{x} - \underline{x}}{2^m - 1} \tag{5.17}$$

7.1. Binary-encoded Genetic Algorithms

- To have a more precise representation, more bits must be used.
- When using binary-encoded GAs, we do not need to encode the design variables because they are generated and manipulated directly in the binary representation.
- Still, it is necessary to decode them before providing them to the evaluation function.
- To decode a binary representation, we use

$$x = \underline{x} + \sum_{i=0}^{m-1} b_i 2^i \Delta x \qquad (5.18)$$

7.1. Binary-encoded Genetic Algorithms

Example 5.4: Binary representation of a real number.

- Suppose we have a continuous design variable x that we want to represent in the interval [-20,80] using 12 bits. Then, we have 2^{12}-1=4095 intervals, and using Eq. 5.17, we get Δx=0.0244. This interval is the error in this finite-precision representation.
- For the following sample binary representation:

i	1	2	3	4	5	6	7	8	9	10	11	12
b_i	0	0	0	1	0	1	1	0	0	0	0	1

we can use Eq. 5.18 to compute the equivalent real number, which turns out to be $x \approx 32.55$.

7.1.1. Initial population

- The first step in a genetic algorithm is to generate an initial set (population) of points.
- As a rule of thumb, the population size should be approximately one order of magnitude larger than the number of design variables, and this size should be tuned.
- One popular way to choose the initial population is to do it at random.
- Using binary encoding, we can assign each bit in the representation of the design variables a 50 percent chance of being either 1 or 0.
- This can be done by generating a random number $0 \leq r \leq 1$ and setting the bit to 0 if $r \leq 0.5$ and 1 if $r > 0.5$.

7.1.1. Initial population

- For a population of size n_p, with n design variables, where each variable is encoded using m bits, the total number of bits that needs to be generated is $n_p \times n \times m$.
- To achieve better spread in a larger dimensional space, the sampling methods described in Chapter 6.? are generally more effective than random populations.
- Although we then need to evaluate the function across many points (a population), these evaluations can be performed in parallel.

7.1.2. Selection

- In this step, we choose points from the population for reproduction in a subsequent step (called a *mating pool*).
- On average, it is desirable to choose a mating pool that improves in fitness (thus mimicking the concept of natural selection), but it is also essential to maintain diversity.
- In total, we need to generate $n_p/2$ pairs.
- The simplest selection method is to randomly select two points from the population until the requisite number of pairs is complete.
- This approach is not particularly effective because there is no mechanism to move the population toward points with better objective functions.

7.1.2. Selection

- Tournament selection is a better method that randomly pairs up n_p points and selects the best point from each pair to join the mating pool.
- The same pairing and selection process is repeated to create $n_p/2$ more points to complete a mating pool of n_p points.

7.1.2. Selection

Example 5.5: Tournament selection process.

- Figure 5.23 illustrates the process with a small population. Each member of the population ends up in the mating pool zero, one, or two times, with better points more likely to appear in the pool. The best point in the population will always end up in the pool twice, whereas the worst point in the population is always eliminated.

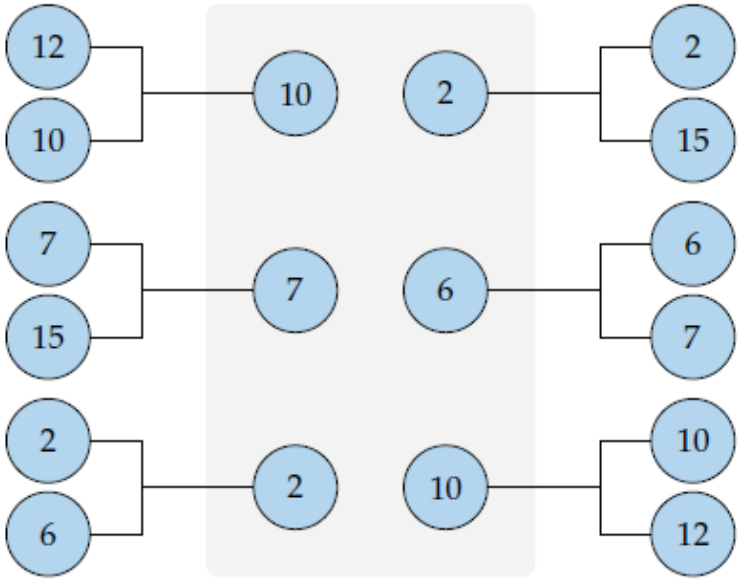

Figure 5.23 Tournament selection example.

Source: Martins et Ning, 2021

7.1.2. Selection

- Another standard method is roulette wheel selection.
- This concept is patterned after a roulette wheel used in a casino.
- It assigns better points to a larger sector on the roulette wheel to have a higher probability of being selected.
- First, the objective function for all the points in the population must be converted to a fitness value because the roulette wheel needs all positive values and is based on maximizing rather than minimizing.
- To achieve that, the following conversion to fitness is first performed:

$$F = \frac{-f_i + \Delta F}{\max(1, \Delta F - f_{low})} \tag{5.19}$$

7.1.2. Selection

where $\Delta F = 1.1f_{high} - 0.1f_{low}$ is based on the highest and lowest function values in the population, and the denominator is introduced to scale the fitness.

- Then, to find the sizes of the sectors in the roulette wheel selection, we take the normalized cumulative sum of the scaled fitness values to compute an interval for each member in the population j as

$$S_j = \frac{\sum_{i=1}^{j} F_i}{\sum_{i=1}^{n_p} F_i} \tag{5.20}$$

7.1.2. Selection

- We can now create a mating pool of n_p points by turning the roulette wheel n_p times.
- We do this by generating a random number $0 \leq r \leq 1$ at each turn.
- The jth member is copied to the mating pool if

$$S_{j-1} \leq r \leq S_j \tag{5.21}$$

- This ensures that the probability of a member being selected for reproduction is proportional to its scaled fitness value.

7.1.2. Selection

Example 5.6: Tournament selection process.

- Assume that F=[5,10,20,45]. Then S=[0.25,0.3125,0.875,1], which divides the "wheel" into four segments, shown graphically in Fig. 5.24. We would then draw four random numbers (say, 0.6, 0.2, 0.9, 0.7), which would correspond to the following $n_p/2$ pairs: (x_3 and x_1), (x_4 and x_3).

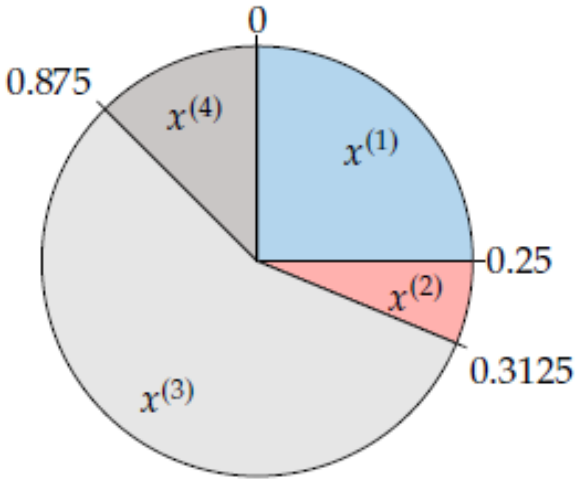

Figure 5.24 Roulette wheel selection example.

Source: Martins et Ning, 2021

7.1.3. Crossover

- In the reproduction operation, two points (offspring) are generated from a pair of points (parents).
- Various strategies are possible in genetic algorithms.
- Single-point crossover usually involves generating a random integer $1 \leq k \leq m-1$ that defines the crossover point.
- This is illustrated in Fig. 5.25.
- For one of the offspring, the first k bits are taken from parent 1 and the remaining bits from parent 2.

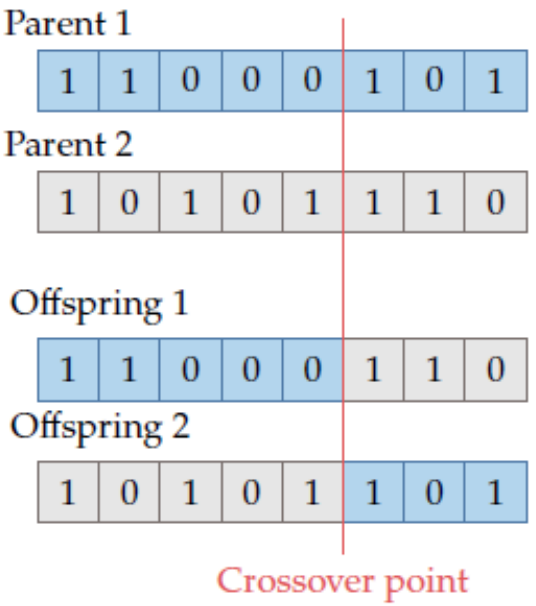

Figure 5.25 The crossover point determines which parts of the chromosome from each parent get inherited by each offspring.

Source: Martins et Ning, 2021

7.1.3. Crossover

- For the second offspring, the first k bits are taken from parent 2 and the remaining ones from parent 1.
- Various extensions exist, such as two-point crossover or n-point crossover.

7.1.4. Mutation

- Mutation is a random operation performed to change the genetic information and is needed because even though selection and reproduction effectively recombine existing information, occasionally some useful genetic information might be lost.
- The mutation operation protects against such irrecoverable loss and introduces additional diversity into the population.
- When using bit representation, every bit is assigned a small permutation probability, say p=0.005~0.05.
- This is done by generating a random number $0 \leq r \leq 1$ for each bit, which is changed if $r < p$.

7.1.4. Mutation

- An example is illustrated in Fig. 5.26.

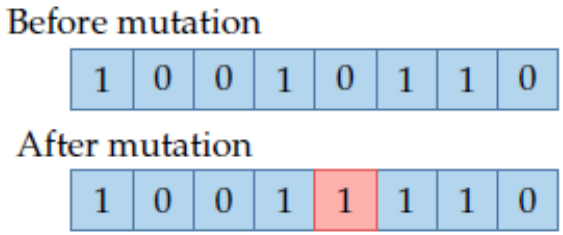

Figure 5.26 Mutation randomly switches some of the bits with low probability.

Source: Martins et Ning, 2021

7.2. Real-encoded Genetic Algorithms

- As the name implies, real-encoded GAs represent the design variables in their original representation as real numbers.
- This has several advantages over the binary-encoded approach.
- First, real encoding represents numbers up to machine precision rather than being limited by the initial choice of string length required in binary encoding.
- Second, it avoids the "Hamming cliff" issue of binary encoding, which is caused by the fact that many bits must change to move between adjacent real numbers (e.g., 0111 to 1000).
- Third, some real-encoded GAs can generate points outside the design variable bounds used to create the initial population; in many problems, the design variables are not bounded.
- Finally, it avoids the burden of binary coding and decoding.

7.2. Real-encoded Genetic Algorithms

- The main disadvantage is that integer or discrete variables cannot be handled straightforwardly.
- For continuous problems, a real-encoded GA is generally more efficient than a binary-encoded GA.
- We now describe the required changes to the GA operations in the real-encoded approach.

7.2.1. Initial population

- The most common approach is to pick the n_p points using random sampling within the provided design bounds.
- Each member is often chosen at random within some initial bounds.
- For each design variable x_i, with bounds such that $\underline{x}_i \leq x_i \leq \overline{x}_i$, we could use

$$x_i = \underline{x}_i + r(\overline{x}_i - \underline{x}_i) \qquad (5.22)$$

where r is a random number such that $0 \leq r \leq 1$.

- Again, the sampling methods described in Chapter 6.? are more effective for higher dimensional spaces.

7.2.2. Selection

- The selection operation does not depend on the design variable encoding.
- Therefore, we can use one of the selection approaches described for the binary-encoded GA: tournament or roulette wheel selection.

7.2.3. Crossover

- When using real encoding, the term crossover does not accurately describe the process of creating the two offspring from a pair of points.
- Instead, the approaches are more accurately described as a blending, although the name crossover is still often used.
- There are various options for the reproduction of two points encoded using real numbers.
- A standard method is linear crossover, which generates two or more points in the line defined by the two parent points.
- One option for linear crossover is to generate the following two
- points:

$$x_{c_1} = 0.5x_{p_1} + 0.5x_{p_2}$$
$$x_{c_2} = 2x_{p_2} - 0.5x_{p_1} \qquad (5.23)$$

7.2.3. Crossover

where parent 2 is more fit than parent 1 ($f(x_{p2})<f(x_{p1})$).

- An example of this linear crossover approach is shown in Fig. 5.27, where we can see that child 1 is the average of the two parent points, whereas child 2 is obtained by extrapolating in the direction of the "fitter" parent.
- Another option is a simple crossover like the binary case where a random integer is generated to split the vectors - for example, with a split after the first index:

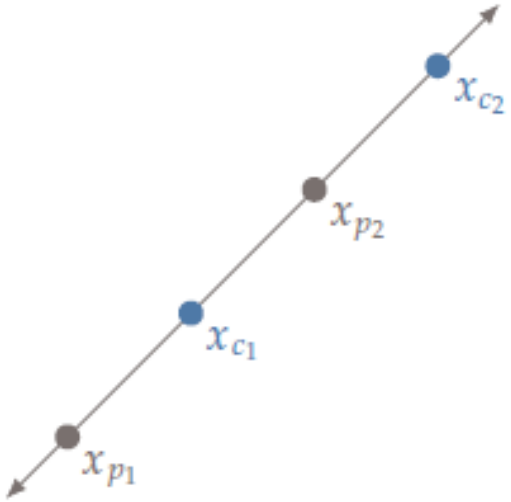

Figure 5.27 Linear crossover produces two new points along the line defined by the two parent points.

Source: Martins et Ning, 2021

7.2.3. Crossover

$$
\begin{aligned}
x_{p_1} &= [x_1, x_2, x_3, x_4] \\
x_{p_2} &= [x_5, x_6, x_7, x_8] \\
&\Downarrow \\
x_{c_1} &= [x_1, x_6, x_7, x_8] \\
x_{c_2} &= [x_5, x_2, x_3, x_4]
\end{aligned}
\tag{5.24}
$$

- This simple crossover does not generate as much diversity as the binary case and relies more heavily on effective mutation.
- Many other strategies have been devised for real-encoded GAs.

7.2.4. Mutation

- As with a binary-encoded GA, mutation should only occur with a small probability (e.g., p=0.0005~0.05).
- However, rather than changing each bit with probability p, we now change each design variable with probability p.
- Many mutation methods rely on random variations around an existing member, such as a uniform random operator:

$$x_{new_i} = x_i + (r_i - 0.5)\Delta_i, \qquad for \quad i = 1, \ldots n \qquad (5.25)$$

where r_i is a random number between 0 and 1, and Δ_i is a preselected maximum perturbation in the ith direction.

- Many nonuniform methods exist as well.

7.2.4. Mutation

- For example, we can use a normal probability distribution

$$x_{new_i} = x_i + \mathcal{N}(0, \sigma_i), \qquad for \quad i = 1, \dots n \tag{5.26}$$

where σ_i is a preselected standard deviation, and random samples are drawn from the normal distribution.

- During the mutation operations, bound checking is necessary to ensure the mutations stay within the lower and upper limits.

7. Genetic algorithms

Example 5.7: Genetic algorithm applied to the bean function.

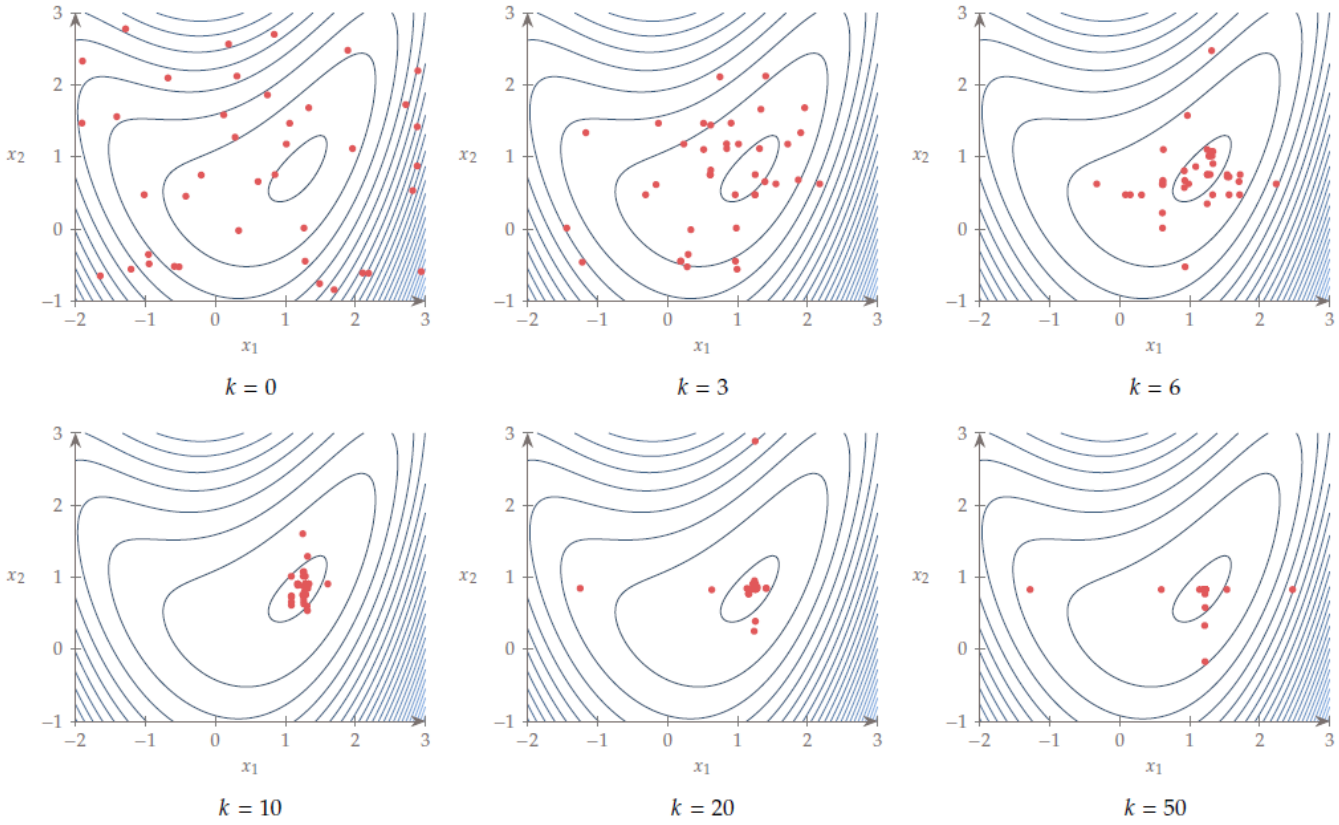

Figure 5.28 Evolution of the population using a bit-encoded GA to minimize the bean function, where k is the generation number.

Source: Martins et Ning, 2021

7. Genetic algorithms

Example 5.7: Genetic algorithm applied to the bean function.

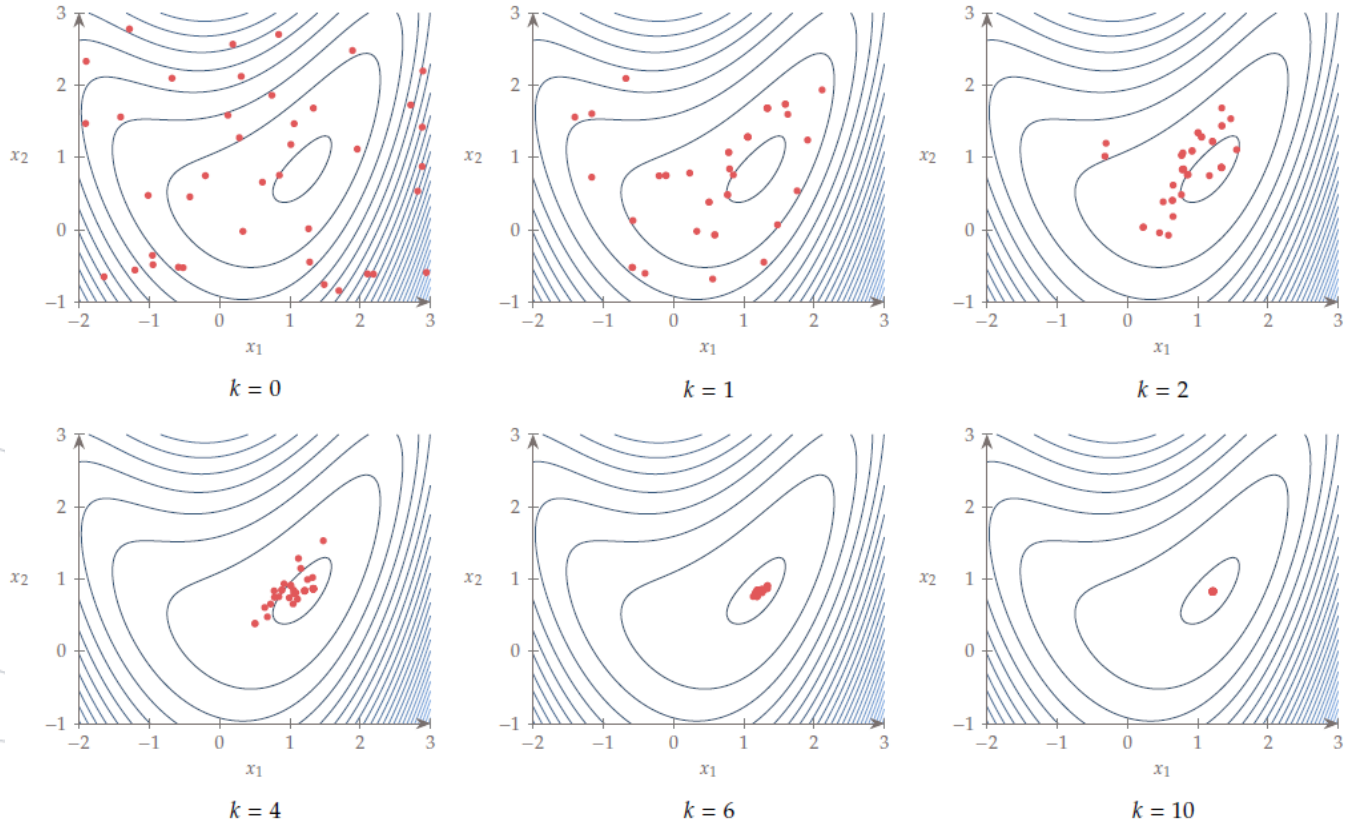

Figure 5.29 Evolution of the population using a real-encoded GA to minimize the bean function, where k is the generation number.

Source: Martins et Ning, 2021

7.3. Constraint handling

- Various approaches exist for handling constraints.
- Like the Nelder–Mead method, we can use a penalty method (e.g., augmented Lagrangian, linear penalty).
- However, there are additional options for GAs.
- In the tournament selection, we can use other selection criteria that do not depend on penalty parameters.
- One such approach for choosing the best selection among two competitors is as follows:
  1. Prefer a feasible solution.
  2. Among two feasible solutions, choose the one with a better objective.
  3. Among two infeasible solutions, choose the one with a smaller constraint violation.
- This concept is a lot like the filter methods.

7.4. Convergence

- Rigorous mathematical convergence criteria, like those used in gradient-based optimization, do not apply to GAs.
- The most common way to terminate a GA is to specify a maximum number of iterations, which corresponds to a computational budget.
- Another similar approach is to let the algorithm run indefinitely until the user manually terminates the algorithm, usually by monitoring the trends in population fitness.
- A more automated approach is to track a running average of the population's fitness.
- However, it can be challenging to decide what tolerance to apply to this criterion because we generally are not interested in the average performance.

7.4. Convergence

- A more direct metric of interest is the fitness of the best member in the population.
- However, this can be a problematic criterion because the best member can disappear as a result of crossover or mutation.
- To avoid this and to improve convergence, many GAs employ elitism.
- This means that the fittest population member is retained to guarantee that the population does not regress.
- Even without this behaviour, the best member often changes slowly, so the user should not terminate the algorithm unless the best member has not improved for several generations.

8. Particle Swarm Optimization

- Like a GA, particle swarm optimization (PSO) is a stochastic population-based optimization algorithm based on the concept of “swarm intelligence”.
- Swarm intelligence is the property of a system whereby the collective behaviours of unsophisticated agents interacting locally with their environment cause coherent global patterns.
- In other words: dumb agents, properly connected into a swarm, can yield smart results.
- The “swarm” in PSO is a set of design points (agents or particles) that move in *n*-dimensional space, looking for the best solution.
- Although these are just design points, the history for each point is relevant to the PSO algorithm, so the term particle is adopted.

8. Particle Swarm Optimization

- Each particle moves according to a velocity.
- This velocity changes according to the past objective function values of that particle and the current objective values of the rest of the particles.
- Each particle remembers the location where it found its best result so far, and it exchanges information with the swarm about the location where the swarm has found the best result so far.
- The position of particle i for iteration k+1 is updated according to

$$x_{k+1}^{(i)} = x_k^{(i)} + v_{k+1}^{(i)} \Delta t \qquad (5.27)$$

where Δt is a constant artificial time step.

8. Particle Swarm Optimization

- The velocity for each particle is updated as follows:

$$v_{k+1}^{(i)} = \alpha v_k^{(i)} + \beta \frac{x_{best}^{(i)} - x_k^{(i)}}{\Delta t} + \gamma \frac{x_{best} - x_k^{(i)}}{\Delta t} \qquad (5.28)$$

- The first component in this update is the “inertia”, which determines how similar the new velocity is to the velocity in the previous iteration through the parameter α.
- Typical values for the inertia parameter are in the interval [0.8,1.2].
- A lower value of α reduces the particle’s inertia and tends toward faster convergence to a minimum.
- A higher value of α increases the particle’s inertia and tends toward increased exploration to potentially help discover multiple minima.

8. Particle Swarm Optimization

- Some methods are adaptive, choosing the value of α based on the optimizer's progress.
- The second term represents "memory" and is a vector pointing toward the best position particle i has seen in all its iterations so far, $x_{best}^{(i)}$.
- The weight in this term consists of a random number β in the interval $[0,\beta_{max}]$ that introduces a stochastic component to the algorithm.
- Thus, β controls how much influence the best point found by the particle so far has on the next direction.
- The third term represents "social" influence.

8. Particle Swarm Optimization

- It behaves similarly to the memory component, except that x_{best} is the best point the entire swarm has found so far, and γ is a random number between $[0,\gamma_{max}]$ that controls how much of an influence this best point has in the next direction.
- The relative values of β and γ thus control the tendency toward local versus global search, respectively.
- Both β_{max} and γ_{max} are in the interval [0,2] and are typically closer to 2.
- Sometimes, rather than using the best point in the entire swarm, the best point is chosen within a neighbourhood.
- Because the time step is artificial, we can eliminate it by multiplying Eq. 5.28 by Δt to yield a step:

$$\Delta x_{k+1}^{(i)} = \alpha \Delta x_k^{(i)} + \beta \left(x_{best}^{(i)} - x_k^{(i)} \right) + \gamma \left(x_{best} - x_k^{(i)} \right) \qquad (5.29)$$

8. Particle Swarm Optimization

- We then use this step to update the particle position for the next iteration:

$$x_{k+1}^{(i)} = x_k^{(i)} + \Delta x_{k+1}^{(i)} \tag{5.30}$$

- The three components of the update in Eq. 5.29 are shown in Fig. 5.30 for a two-dimensional case.

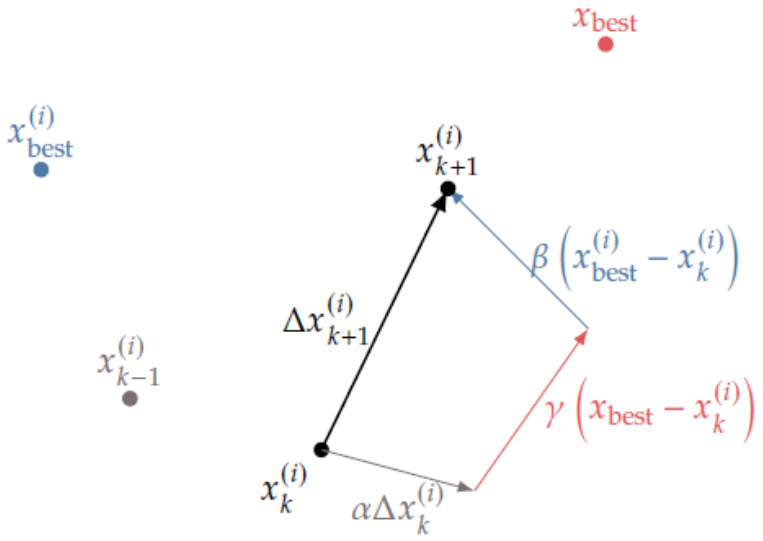

Figure 5.30 Components of the PSO update.

Source: Martins et Ning, 2021

8. Particle Swarm Optimization

- The first step in the PSO algorithm is to initialize the set of particles (Algorithm 5.6).
- As with a GA, the initial set of points can be determined randomly or can use a more sophisticated sampling strategy (see Chapter 6.?).
- The velocities are also randomly initialized, generally using some fraction of the domain size ($\overline{x} - \underline{x}$).
- The main loop in the algorithm computes the steps to be added to each particle and updates their positions.
- Particles must be prevented from going beyond the bounds.
- If a particle reaches a boundary and has a velocity pointing out of bounds, it is helpful to reset the velocity to zero or reorient it toward the interior for the next iteration.

8. Particle Swarm Optimization

Algorithm 5.6: Particle swarm optimization algorithm.

Inputs:

- $\overline{x}$: Variable upper bounds
- $\underline{x}$: Variable lower bounds
- α: Inertia parameter
- $\beta_{\max}$: Self influence parameter
- $\gamma_{\max}$: Social influence parameter
- $\Delta x_{\max}$: Maximum velocity

Outputs:

- x^*: Best point
- f^*: Corresponding function value

$k = 0$
for $i = 1$ **to** n **do** — Loop to initialize all particles
  Generate position $x_0^{(i)}$ within specified bounds.
  Initialize "velocity" $\Delta x_0^{(i)}$
end for
while not converged **do** — Main iteration loop
  for $i = 1$ **to** n **do**
    if $f\left(x^{(i)}\right) < f\left(x_{\text{best}}^{(i)}\right)$ **then** — Best individual points
      $x_{\text{best}}^{(i)} = x^{(i)}$
    end if
    if $f(x^{(i)}) < f(x_{\text{best}})$ **then** — Best swarm point
      $x_{\text{best}} = x^{(i)}$
    end if
  end for
  for $i = 1$ **to** n **do**
    $\Delta x_{k+1}^{(i)} = \alpha \Delta x_k^{(i)} + \beta\left(x_{\text{best}}^{(i)} - x_k^{(i)}\right) + \gamma\left(x_{\text{best}} - x_k^{(i)}\right)$
    $\Delta x_{k+1}^{(i)} = \max\left(\min\left(\Delta x_{k+1}^{(i)}, \Delta x_{\max}\right), -\Delta x_{\max}\right)$ — Limit velocity
    $x_{k+1}^{(i)} = x_k^{(i)} + \Delta x_{k+1}^{(i)}$ — Update the particle position
    $x_{k+1}^{(i)} = \max\left(\min\left(x_{k+1}^{(i)}, \overline{x}\right), \underline{x}\right)$ — Enforce bounds
  end for
  $k = k + 1$
end while

8. Particle Swarm Optimization

- It is also helpful to impose a maximum velocity.
- If the velocity is too large, the updated positions are unrelated to their previous positions, and the search is more random.
- The maximum velocity might also decrease across iterations to shift from exploration toward exploitation.
- Several convergence criteria are possible, some of which are similar to the Nelder–Mead algorithm and GAs.
- Examples of convergence criteria include the distance (sum or norm) between each particle and the best particle, the best particle's objective function value changes for the last few generations, and the difference between the best and worst member.

8. Particle Swarm Optimization

- For PSO, another alternative is to check whether the velocities for all particles (as measured by a metric such as norm or mean) are below some tolerance.
- Some of these criteria that assume all the particles congregate (distance, velocities) do not work well for multimodal problems.
- In those cases, tracking only the best particle's objective function value may be more appropriate.

8. Particle Swarm Optimization

Example 5.8: PSO algorithm applied to the bean function.

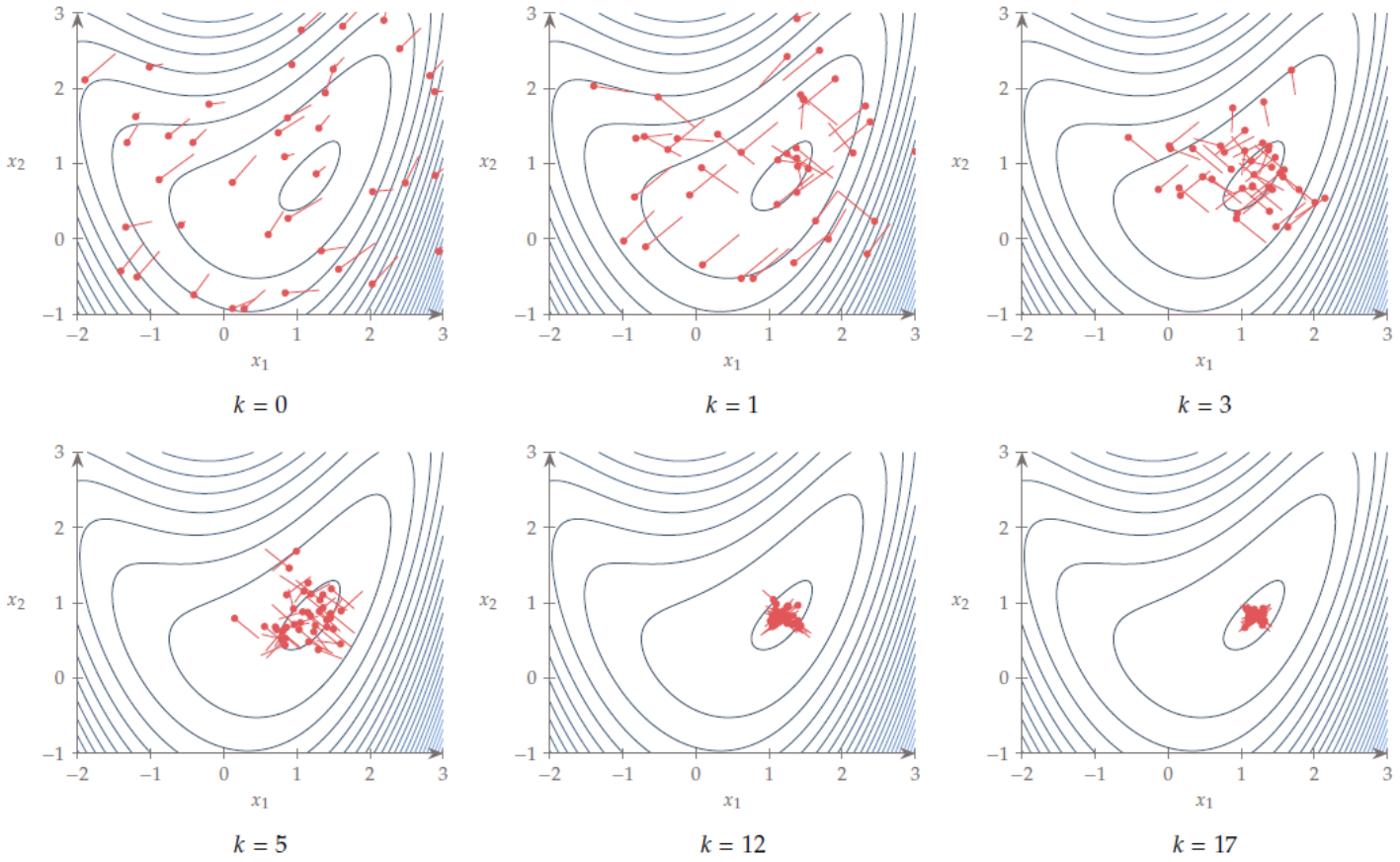

Figure 5.31 Sequence of particles at iterations : that minimizes the bean function.

Source: Martins et Ning, 2021

9. Hybrid Metaheuristic Algorithms

- The hybridization of evolutionary algorithms (EA) is popular, partly due to its better performance in handling noise, uncertainty, vagueness, and imprecision.
- There are two prominent issues of EAs in solving global and highly nonconvex optimization problems:
  1. Premature convergence: The problem of premature convergence results in the lack of accuracy of the final solution. The final solution is a feasible solution close to the global optimal, often regarded as satisfactory or close-to-optimal solution.
  2. Slow convergence: Slow convergence means the solution quality does not improve sufficiently quickly. It shows stagnation or almost flatness on a convergence graph (either a single iteration or the average of multiple iterations).

9. Hybrid Metaheuristic Algorithms

- Hybrid algorithms are two or more algorithms that run together and complement each other to produce a profitable synergy from their integration.
- These algorithms are commonly known as hybrid metaheuristics (HMs) or hybrid algorithms (HA) for simplicity.
- Hybrid algorithms play a prominent role in improving the search capability of algorithms.
- Hybridization aims to combine the advantages of each algorithm to form a hybrid algorithm, while simultaneously trying to minimize any substantial disadvantage.
- In general, the outcome of hybridization can usually make some improvements in terms of either computational speed or accuracy.

9. Hybrid Metaheuristic Algorithms

- See book chapter for further details:
  - T. O. Ting, Xin-She Yang, Shi Cheng, Kaizhu Huang, “Hybrid Metaheuristic Algorithms: Past, Present, and Future”, in book: Recent Advances in Swarm Intelligence and Evolutionary Computation, December 2015, DOI: 10.1007/978-3-319-13826-8_4.

10. Penalty Methods

- The concept behind penalty methods is intuitive: to transform a constrained problem into an unconstrained one by adding a penalty to the objective function when constraints are violated.
- Penalty methods are no longer used directly in gradient-based optimization algorithms because they have difficulty converging to the true solution.
- However, these methods are still valuable because:
  1. they are simple and thus ease the transition into understanding constrained optimization
  2. although not effective for gradient-based optimization, they are still useful in some constrained gradient-free methods
  3. they can be useful as merit functions in line search algorithms

10. Penalty Methods

- The penalized function can be written as

$$\hat{f}(x) = f(x) + \mu\pi(x) \tag{5.31}$$

where $\pi(x)$ is a penalty function, and the scalar μ is a penalty parameter.

- This is similar in form to the Lagrangian, but one difference is that a value for μ is fixed in advance instead of solved for.
- We can use the unconstrained optimization techniques to minimize $\hat{f}(x)$.
- However, instead of just solving a single optimization problem, penalty methods usually solve a sequence of problems with different values of μ to get closer to the actual constrained minimum.

10. Penalty Methods

- We will see shortly why we need to solve a sequence of problems rather than just one problem.
- Various forms for $\pi(x)$ can be used, leading to different penalty methods.
- There are two main types of penalty functions:
  - Exterior penalties, which impose a penalty only when constraints are violated
  - Interior penalty functions, which impose a penalty that increases as a constraint is approached
- Figure 5.32 shows both interior and exterior penalties for a two-dimensional function.
- The exterior penalty leads to slightly infeasible solutions, whereas an interior penalty leads to a feasible solution but underpredicts the objective.

10. Penalty Methods

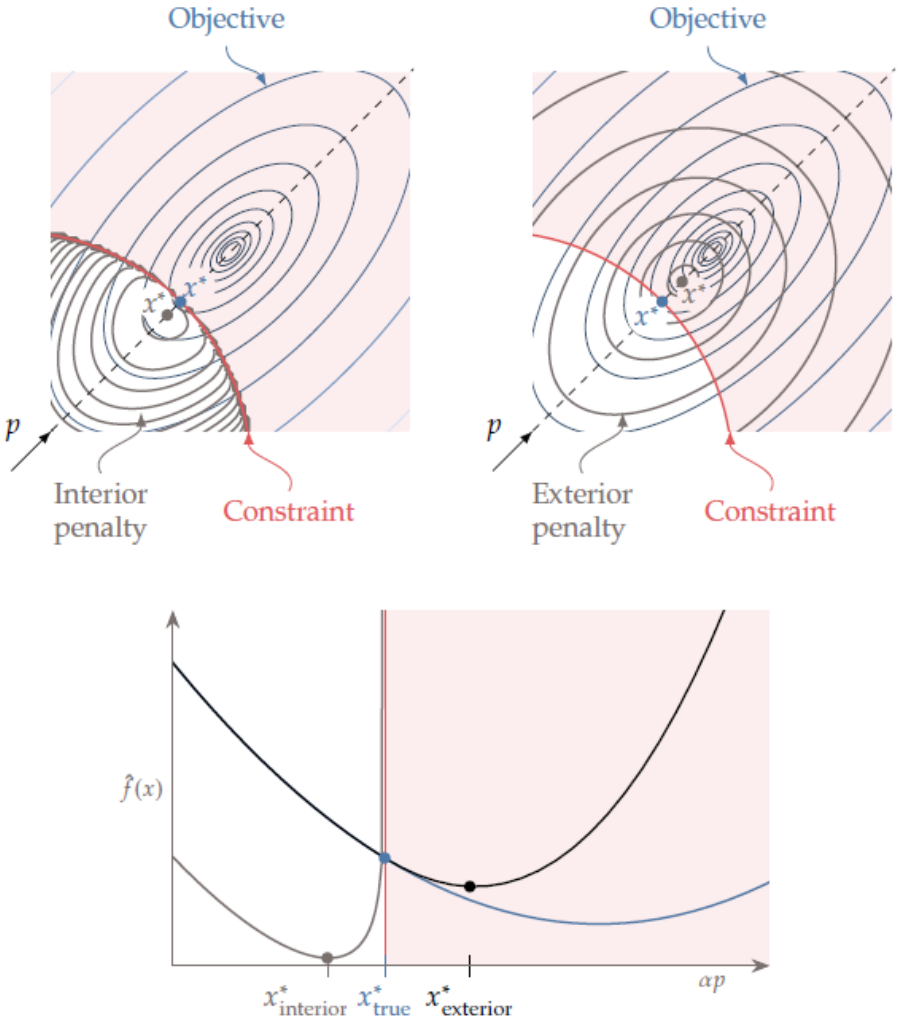

Figure 5.32 Interior penalties tend to infinity as the constraint is approached from the feasible side of the constraint (left), whereas exterior penalty functions activate when the points are not feasible (right). The minimum for both approaches is different from the true constrained minimum.

Source: Martins et Ning, 2021

10.1. Exterior penalty method

- Of the many possible exterior penalty methods, we focus on two of the most popular ones:
  - quadratic penalties
  - augmented Lagrangian method
- Quadratic penalties are continuously differentiable and straightforward to implement, but they suffer from numerical ill-conditioning.
- The augmented Lagrangian method is more sophisticated; it is based on the quadratic penalty but adds terms that improve the numerical properties.
- Many other penalties are possible, such as 1-norms, which are often used when continuous differentiability is unnecessary.

10.1.1. Quadratic penalty method

- For equality constrained problems, the quadratic penalty method takes the form

$$\hat{f}(x;\mu) = f(x) + \frac{\mu}{2}\sum_{i}^{n_h} h_i(x)^2 \qquad (5.32)$$

where the semicolon denotes that μ is a fixed parameter.

- The motivation for a quadratic penalty is that it is simple and results in a function that is continuously differentiable.
- The factor of one half is unnecessary but is included by convention because it eliminates the extra factor of two when taking derivatives.
- The penalty is nonzero unless the constraints are satisfied (h_i=0), as desired.

10.1.1. Quadratic penalty method

- The value of the penalty parameter μ must be chosen carefully.
- Mathematically, we recover the exact solution to the constrained problem only as μ tends to infinity (see Fig. 5.27).
- However, starting with a large value for μ is not practical.

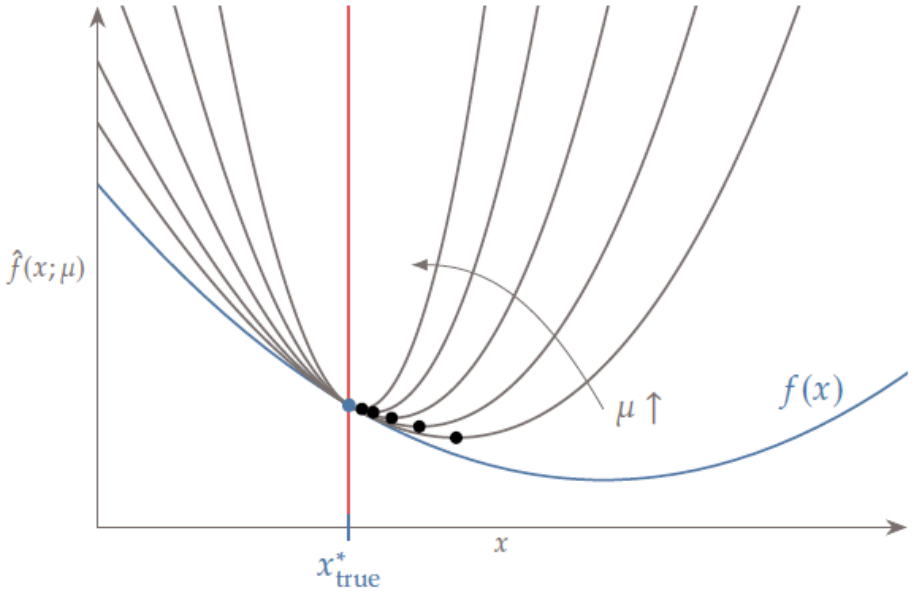

Figure 5.33 Quadratic penalty for an equality constrained one-dimensional problem. The minimum of the penalized function (black dots) approaches the true constrained minimum (blue circle) as the penalty parameter μ increases.

Source: Martins et Ning, 2021

10.1.1. Quadratic penalty method

- This is because the larger the value of μ, the larger the Hessian condition number, which corresponds to the curvature varying greatly with direction.
- This behaviour makes the problem difficult to solve numerically.
- To solve the problem more effectively, we begin with a small value of μ and solve the unconstrained problem.
- We then increase μ and solve the new unconstrained problem, using the previous solution as the starting point.
- We repeat this process until the optimality conditions are satisfied (or some other approximate convergence criteria are satisfied), as outlined in Algoritm 5.7.

10.1.1. Quadratic penalty method

Algorithm 5.7: Exterior penalty method.

Inputs:

x_0: Starting point

$\mu_0 > 0$: Initial penalty parameter

$\rho > 1$: Penalty increase factor ($\rho \sim 1.2$ is conservative, $\rho \sim 10$ is aggressive)

Outputs:

x^*: Optimal point

$f(x^*)$: Corresponding function value

$k = 0$

while not converged **do**

$x_k^* \leftarrow \underset{x_k}{\text{minimize}}\ \hat{f}(x_k; \mu_k)$

$\mu_{k+1} = \rho_l \mu_k$ — Increase penalty

$x_{k+1} = x_k^*$ — Update starting point for next optimization

$k = k + 1$

end while

10.1.1. Quadratic penalty method

- By gradually increasing μ and reusing the solution from the previous problem, we avoid some of the ill-conditioning issues.
- Thus, the original constrained problem is transformed into a sequence of unconstrained optimization problems.
- There are three potential issues with the approach outlined in Algorithm 5.7.
- Suppose the starting value for μ is too low.
- In that case, the penalty might not be enough to overcome a function that is unbounded from below, and the penalized function has no minimum.
- The second issue is that we cannot practically approach) $\mu \to \infty$.
- Hence, the solution to the problem is always slightly infeasible.

10.1.1. Quadratic penalty method

- The third issue has to do with the curvature of the penalized function, which is directly proportional to μ.
- The extra curvature is added in a direction perpendicular to the constraints, making the Hessian of the penalized function increasingly ill-conditioned as μ increases.
- Thus, the need to increase μ to improve accuracy directly leads to a function space that is increasingly challenging to solve.

10.1.1. Quadratic penalty method

Example 5.9: Quadratic penalty for equality constrained problem.

$$\hat{f}(x;\mu) = x_1 + 2x_2 + \frac{\mu}{2}\left(\frac{1}{2}x_1^2 + x_2^2 - 1\right)^2$$

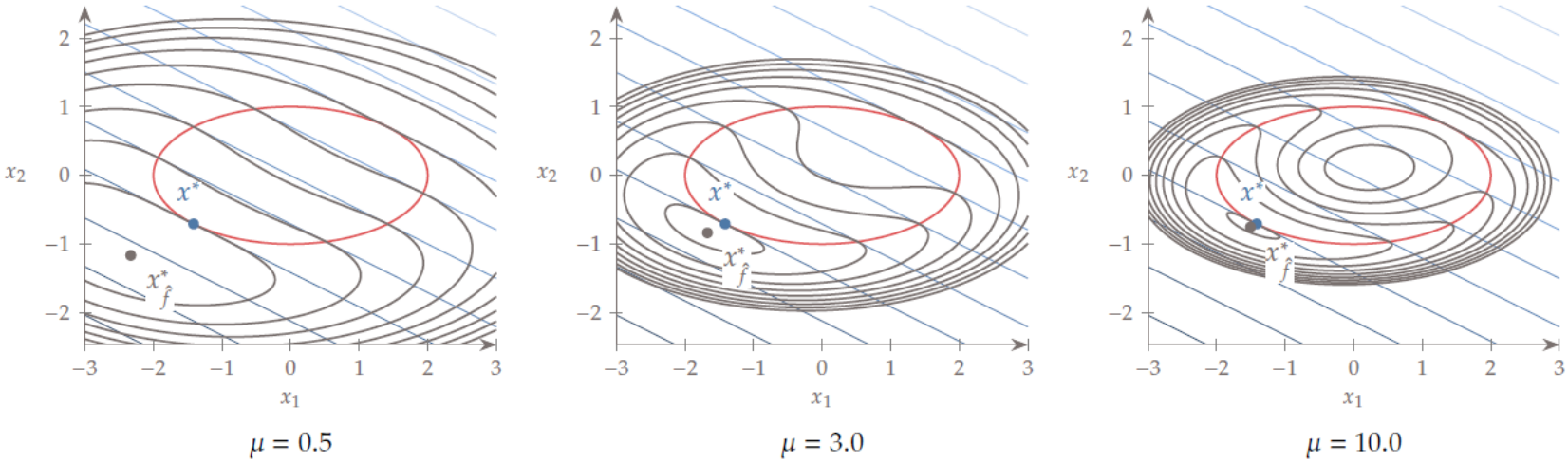

Figure 5.34 Quadratic penalty for one equality constraint. The minimum of the penalized function approaches the constrained minimum as the penalty parameter increases..

Source: Martins et Ning, 2021

10.1.1. Quadratic penalty method

Example 5.9: Quadratic penalty for equality constrained problem (continued).

$$\hat{f}(x;\mu) = x_1 + 2x_2 + \frac{\mu}{2}\left(\frac{1}{2}x_1^2 + x_2^2 - 1\right)^2$$

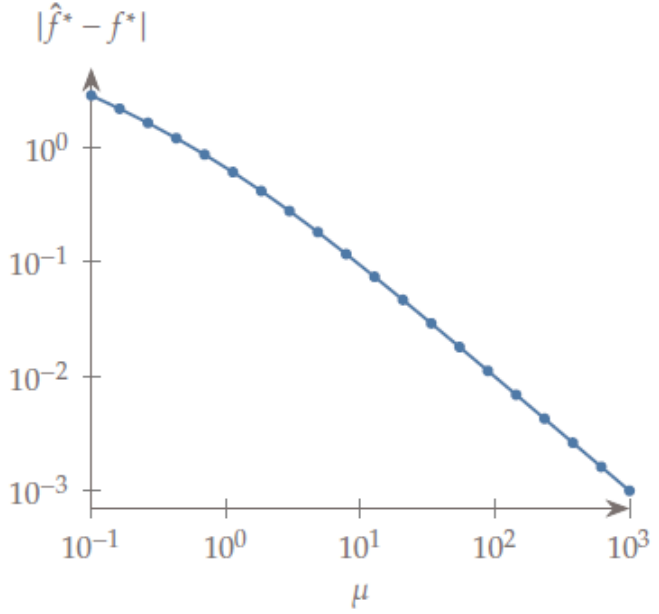

Figure 5.35 Error in optimal solution as compared to true solution as a function of an increasing penalty parameter.

Source: Martins et Ning, 2021

10.1.1. Quadratic penalty method

- The approach discussed so far handles only equality constraints, but it can be extended to handle inequality constraints.
- Instead of adding a penalty to both sides of the constraints, we add the penalty when the inequality constraint is violated (i.e., when $g_j(x)>0$).
- This behaviour can be achieved by defining a new penalty function as

$$\hat{f}(x;\mu) = f(x) + \frac{\mu}{2}\sum_{j}^{n_g} max\left[0, g_j(x)\right]^2 \qquad (5.33)$$

10.1.1. Quadratic penalty method

- The only difference relative to the equality constraint penalty shown in Fig. 5.33 is that the penalty is removed on the feasible side of the inequality constraint, as shown in Fig. 5.36.

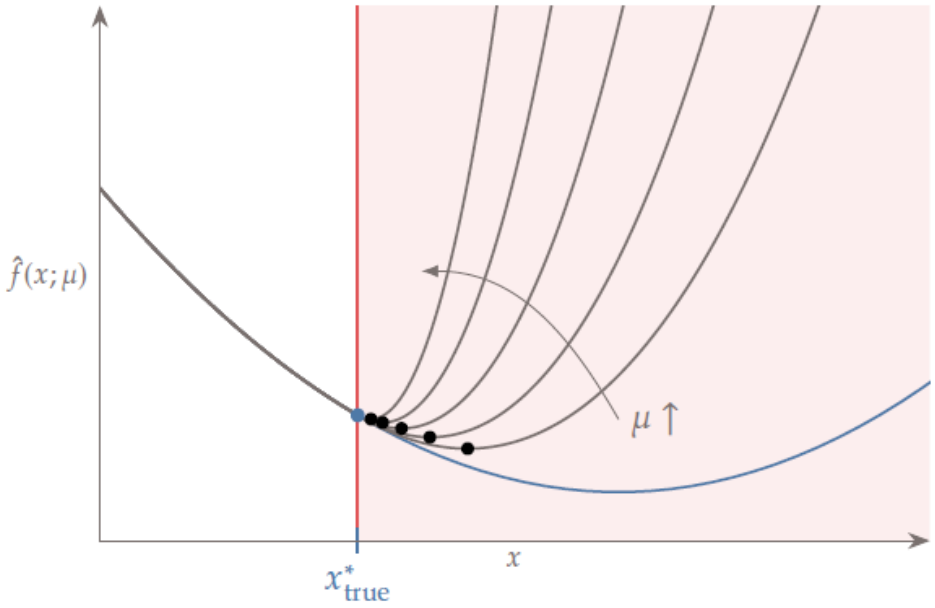

Figure 5.36 Quadratic penalty for an inequality constrained one-dimensional problem. The minimum of the penalized function approaches the constrained minimum from the infeasible side.

Source: Martins et Ning, 2021

10.1.1. Quadratic penalty method

Example 5.10: Quadratic penalty for inequality constrained problem.

$$\hat{f}(x;\mu) = x_1 + 2x_2 + \frac{\mu}{2} max\left[0, \frac{1}{2}x_1^2 + x_2^2 - 1\right]^2$$

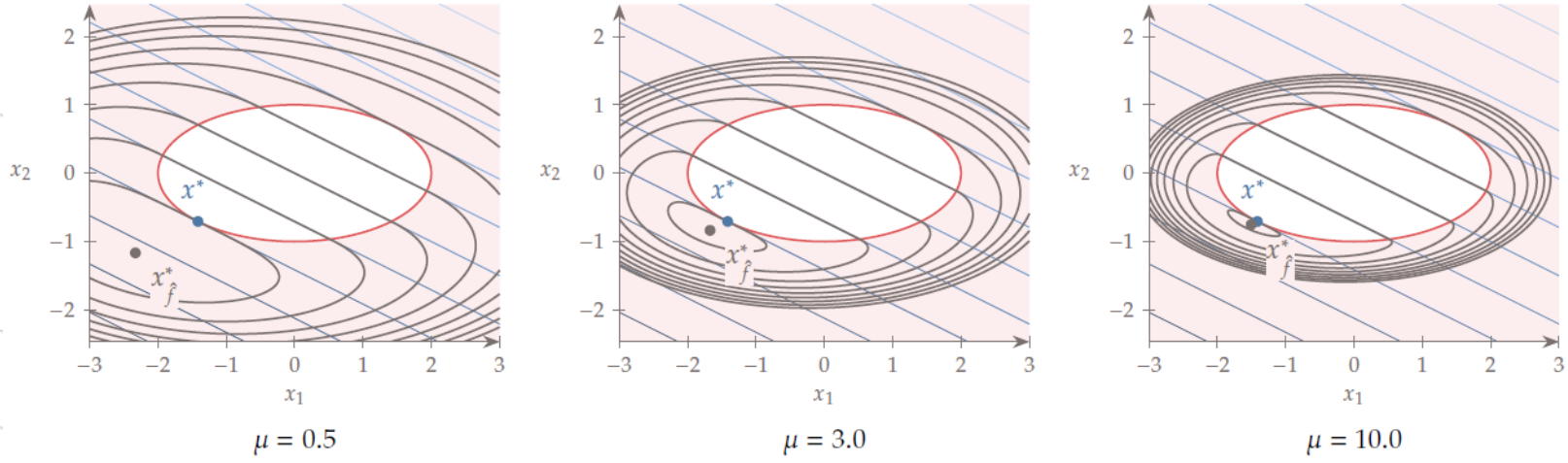

Figure 5.37 Quadratic penalty for one inequality constraint. The minimum of the penalized function approaches the constrained minimum from the infeasible side.

Source: Martins et Ning, 2021

10.1.1. Quadratic penalty method

- The inequality quadratic penalty can be used together with the quadratic penalty for equality constraints if both types of constraints need to be handled

$$\hat{f}(x;\mu) = f(x) + \frac{\mu_h}{2}\sum_{i}^{n_h} h_i(x)^2 + \frac{\mu_g}{2}\sum_{j}^{n_g} max\left[0, g_j(x)\right]^2 \quad (5.34)$$

- The two penalty parameters can be incremented in lockstep or independently.

10.1.1. Quadratic penalty method

- Scaling is also important for constrained problems.
- Similar to scaling the objective function, a good scaling rule of thumb is to normalize such that each constraint function is of order 1.
- For constraints, a natural scale is typically already defined by the limits we provide.
- For example, instead of

$$g_j(x) - g_{max_j} \leq 0 \tag{5.35}$$

a scaled version can be expressed as

$$\frac{g_j(x)}{g_{max_j}} - 1 \leq 0 \tag{5.36}$$

10.1.2. Augmented Lagrangian

- As explained previously, the quadratic penalty method requires a large value of μ for constraint satisfaction, but the large μ degrades the numerical conditioning.
- The augmented Lagrangian method helps alleviate this dilemma by adding the quadratic penalty to the Lagrangian instead of just adding it to the function.
- The augmented Lagrangian function for equality constraints is

$$\hat{f}(x;\lambda,\mu) = f(x) + \sum_{j=1}^{n_h} \lambda_i h_i(x) + \frac{\mu_h}{2} \sum_{j=1}^{n_h} h_i(x)^2 \qquad (5.37)$$

- Unfortunately, the Lagrange multipliers cannot be solved for in a penalty approach, so we need some way to estimate them.
- In other words, they are a parameter in this case, not a variable.

10.1.2. Augmented Lagrangian

- To obtain an estimate of the Lagrange multipliers, we can compare the optimality conditions for the augmented Lagrangian,

$$\nabla_x \hat{f}(x;\lambda,\mu) = \nabla f(x) + \sum_{j=1}^{n_h} \left[\lambda_j + \mu h_j(x)\right] \lambda_j \nabla h_j = 0 \qquad (5.38)$$

to those of the actual Lagrangian,

$$\nabla_x \mathcal{L}(x^*,\lambda^*) = \nabla f(x^*) + \sum_{j=1}^{n_h} \lambda_j^* \nabla h_j(x^*) = 0 \qquad (5.39)$$

which suggests the approximation

$$\lambda_j^* \approx \lambda_j \ + \mu h_j \qquad (5.40)$$

10.1.2. Augmented Lagrangian

- Therefore, we update the vector of Lagrange multipliers based on the current estimate of the Lagrange multipliers and constraint values using

$$\lambda_{k+1} = \lambda_k + \mu_j h(x_k) \tag{5.41}$$

- The complete algorithm is shown in Algorithm 5.8.

10.1.2. Augmented Lagrangian

Algorithm 5.8: Augmented Lagrangian penalty method

Inputs:

x_0: Starting point
$\lambda_0 = 0$: Initial Lagrange multiplier
$\mu_0 > 0$: Initial penalty parameter
$\rho > 1$: Penalty increase factor

Outputs:

x^*: Optimal point
$f(x^*)$: Corresponding function value

$k = 0$
while not converged do
  $x_k^* \leftarrow \underset{x_k}{\text{minimize}}\, \hat{f}(x_k; \lambda_k, \mu_k)$
  $\lambda_{k+1} = \lambda_k + \mu_k h(x_k)$ — Update Lagrange multipliers
  $\mu_{k+1} = \rho \mu_k$ — Increase penalty parameter
  $x_{k+1} = x_k^*$ — Update starting point for next optimization
  $k = k + 1$
end while

10.1.2. Augmented Lagrangian

- This approach is an improvement on the plain quadratic penalty because updating the Lagrange multiplier estimates at each iteration allows for more accurate solutions without increasing μ as much.
- Comparing the augmented Lagrangian approximation to the constraints obtained from Eq. 5.40,

$$h_j \approx \frac{1}{\mu}\left(\lambda_j^* - \lambda_j\right) \tag{5.42}$$

with the corresponding approximation in the quadratic penalty method

$$h_j \approx \frac{\lambda_j^*}{\mu} \tag{5.43}$$

10.1.2. Augmented Lagrangian

- The quadratic penalty relies solely on increasing μ in the denominator to drive the constraints to zero.
- However, with the augmented Lagrangian, we can also control the numerator through the Lagrange multiplier estimate.
- If the estimate is reasonably close to the true Lagrange multiplier, then the numerator becomes small for modest values of μ.
- Thus, the augmented Lagrangian can provide a good solution for x^* while avoiding the ill-conditioning issues of the quadratic penalty.

10.1.2. Augmented Lagrangian

Example 5.11: Augmented Lagrangian for inequality constrained problem.

$$\hat{f}(x;\mu) = x_1 + 2x_2 + \lambda\left(\frac{1}{4}x_1^2 + x_2^2 - 1\right)^2 + \frac{\mu}{2}\left(\frac{1}{4}x_1^2 + x_2^2 - 1\right)^2$$

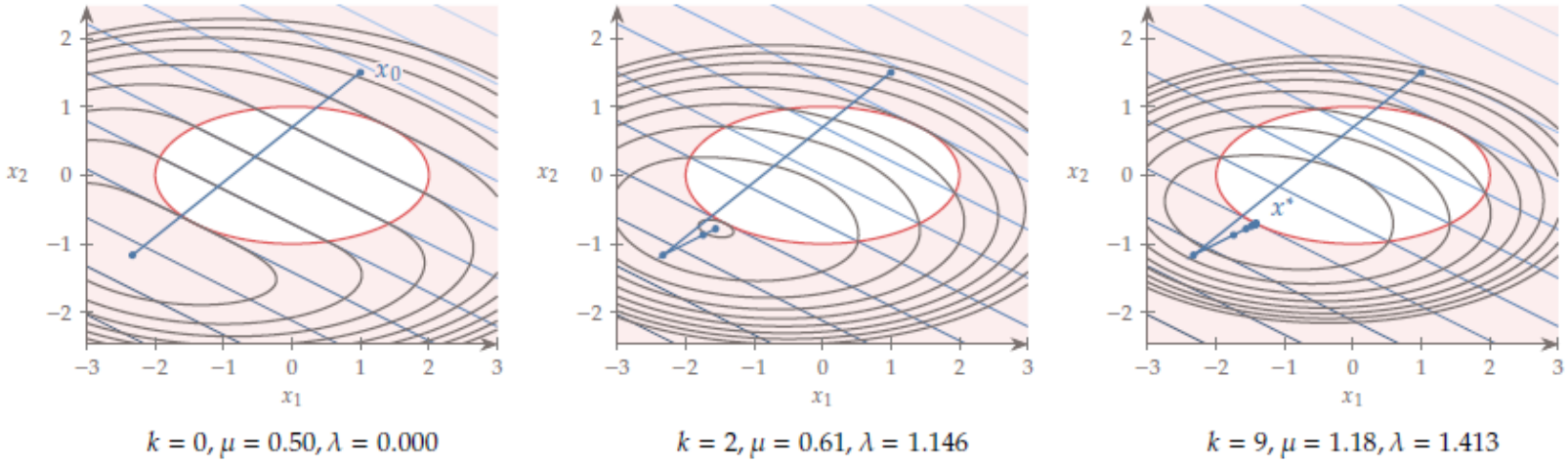

Figure 5.38 Augmented Lagrangian applied to inequality constrained problem..

Source: Martins et Ning, 2021

10.1.2. Augmented Lagrangian

Example 5.11: Augmented Lagrangian for inequality constrained problem (continued).

$$\hat{f}(x;\mu) = x_1 + 2x_2 + \lambda\left(\frac{1}{4}x_1^2 + x_2^2 - 1\right)^2 + \frac{\mu}{2}\left(\frac{1}{4}x_1^2 + x_2^2 - 1\right)^2$$

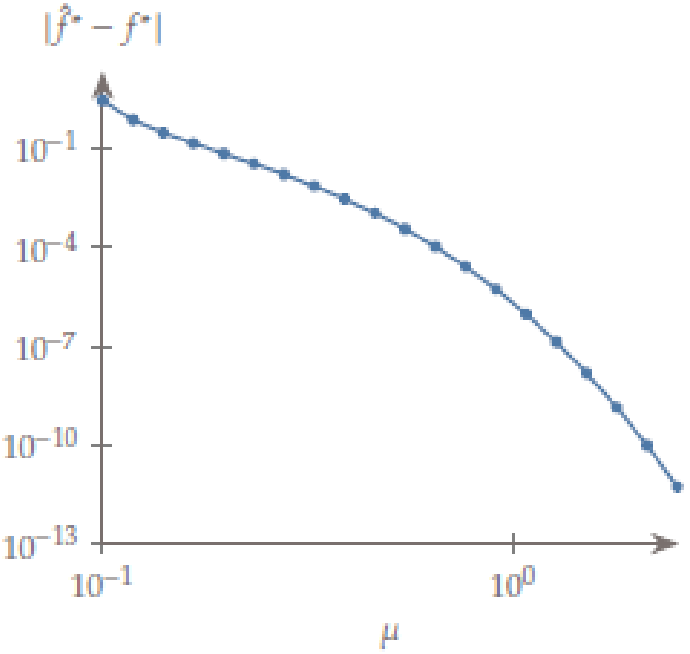

Figure 5.39 Error in optimal solution as compared with true solution as a function of an increasing penalty parameter.

Source: Martins et Ning, 2021

10.1.2. Augmented Lagrangian

- So far we have only discussed equality constraints where the definition for the augmented Lagrangian is universal.
- The above example included an inequality constraint by assuming it was active and treating it like an equality, but this is not an approach that can be used in general.
- An augmented Lagrangian can be used with inequality constraints, though many alternative formulations exist.
- One well-known approach is given by

$$\hat{f}(x;\mu) = f(x) + \lambda^T \overline{g}(x) + \frac{1}{2}\mu \|\overline{g}(x)\|_2^2 \qquad (5.44)$$

where

$$\overline{g}_j(x) = \begin{cases} h_j(x) & for\ equality\ constraints \\ g_j(x) & if\ g_j \geq -\lambda_j/\mu \\ \geq -\lambda_j/\mu & otherwise \end{cases} \qquad (5.45)$$

10.2. Interior penalty method

- Interior penalty methods work the same way as exterior penalty methods - they transform the constrained problem into a series of unconstrained problems.
- The main difference with interior penalty methods is that they always seek to maintain feasibility.
- Instead of adding a penalty only when constraints are violated, they add a penalty as the constraint is approached from the feasible region.
- This type of penalty is particularly desirable if the objective function is ill-defined outside the feasible region.
- These methods are called interior because the iteration points remain on the interior of the feasible region.

10.2. Interior penalty method

- They are also referred to as barrier methods because the penalty function acts as a barrier preventing iterates from leaving the feasible region.
- One possible interior penalty function to enforce $g(x) \leq 0$ is the inverse function (top of Fig. 5.40),

$$\pi(x) = \sum_{j=1}^{n_g} -\frac{1}{g_j(x)} \qquad (5.46)$$

where $\pi(x) \to \infty$ as $g_j(x) \to 0^-$.

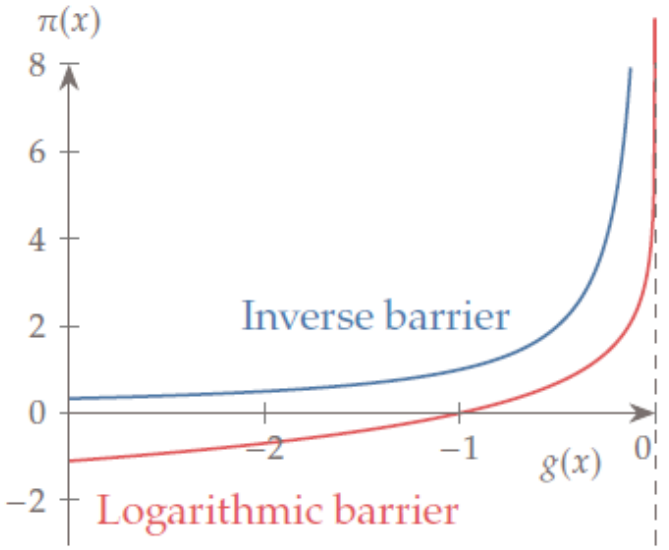

Figure 5.40 Two different interior barrier functions.

Source: Martins et Ning, 2021

10.2. Interior penalty method

- A more popular interior penalty function is the logarithmic barrier (bottom of Fig. 5.40),

$$\pi(x) = \sum_{j=1}^{n_g} -ln\big(-g_j(x)\big) \tag{5.47}$$

which also approaches infinity as the constraint tends to zero from the feasible side.

- The penalty function is then

$$\hat{f}(x;\mu) = f(x) - \mu \sum_{j=1}^{n_g} ln\big(-g_j(x)\big) \tag{5.48}$$

- Neither of these penalty functions applies when $g>0$ because they are designed to be evaluated only within the feasible space.

10.2. Interior penalty method

- Algorithms based on these penalties must be prevented from evaluating infeasible points.
- Like exterior penalty methods, interior penalty methods must also solve a sequence of unconstrained problems but with $\mu \to 0$ (see Algorithm 5.8).
- As the penalty parameter decreases, the region across which the penalty acts decreases, as shown in Fig. 5.41.

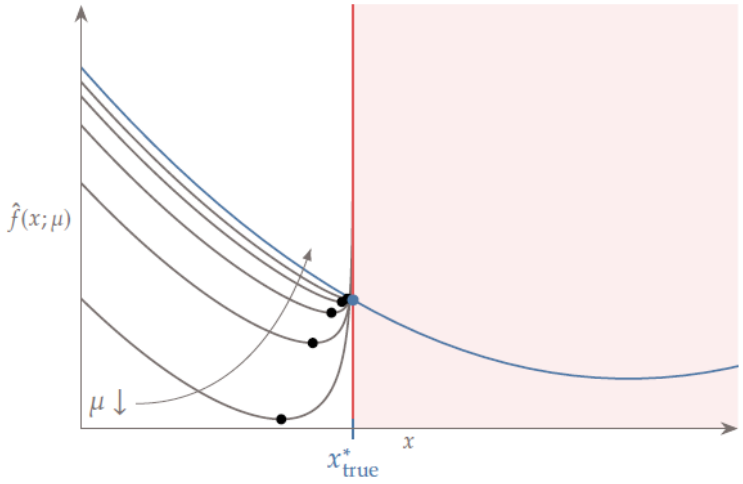

Figure 5.41 Logarithmic barrier penalty for an inequality constrained one-dimensional problem. The minimum of the penalized function (black circles) approaches the true constrained minimum (blue circle) as the penalty parameter μ decreases.

Source: Martins et Ning, 2021

10.2. Interior penalty method

Algorithm 5.8: Interior penalty method.

Inputs:

x_0: Starting point

$\mu_0 > 0$: Initial penalty parameter

$\rho < 1$: Penalty decrease factor

Outputs:

x^*: Optimal point

$f(x^*)$: Corresponding function value

$k = 0$

while not converged **do**

$x_k^* \leftarrow \underset{x_k}{\text{minimize}}\ \hat{f}(x_k; \mu_k)$

$\mu_{k+1} = \rho \mu_k$ Decrease penalty parameter

$x_{k+1} = x_k^*$ Update starting point for next optimization

$k = k + 1$

end while

10.2. Interior penalty method

- The methodology is the same as is described in Algorithm 5.7 but with a decreasing penalty parameter.
- One major weakness of the method is that the penalty function is not defined for infeasible points, so a feasible starting point must be provided.
- For some problems, providing a feasible starting point may be difficult or practically impossible.
- The line search must be safeguarded to prevent the algorithm from becoming infeasible when starting from a feasible point.
- This can be achieved by checking the values of the constraints and backtracking if any of them is greater than or equal to zero.
- Multiple backtracking iterations might be required.

10.2. Interior penalty method

- Like exterior penalty methods, the Hessian for interior penalty methods becomes increasingly ill-conditioned as the penalty parameter tends to zero.
- There are augmented and modified barrier approaches that can avoid the ill-conditioning issue (and other methods that remain ill-conditioned but can still be solved reliably, albeit inefficiently).
- However, these methods have been superseded by the modern interior point methods.

10.2. Interior penalty method

Example 5.12: Logarithmic penalty for inequality constrained problem.

$$\hat{f}(x;\mu) = x_1 + 2x_2 - \mu ln\left(-\frac{1}{4}x_1^2 - x_2^2 + 1\right)$$

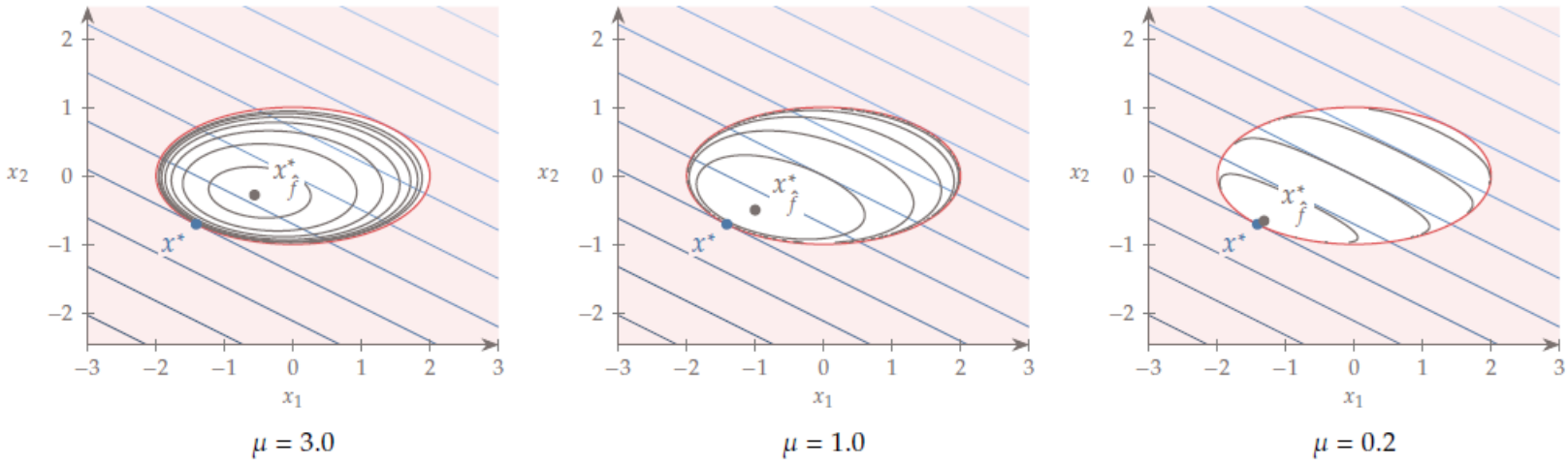

Figure 5.42 Logarithmic penalty for one inequality constraint. The minimum of the penalized function approaches the constrained minimum from the feasible side.

Source: Martins et Ning, 2021