Chapter 6

Gradient-Free Optimization

6.1 Introduction

Using optimization in the solution of practical applications we often encounter one or more of the following challenges:

- non-differentiable functions and/or constraints
- disconnected and/or non-convex feasible space
- discrete feasible space
- mixed variables (discrete, continuous, permutation)
- large dimensionality
- multiple local minima (multi-modal)
- multiple objectives

Gradient-based optimizers are efficient at finding local minima for high-dimensional, nonlinearly-constrained, convex problems; however, most gradient-based optimizers have problems dealing with noisy and discontinuous functions, and they are not designed to handle multi-modal problems or discrete and mixed discrete-continuous design variables.

Consider, for example, the Griewank function:

$$\begin{aligned} f(x) &= \sum_{i=1}^{n} \left(\frac{x_i^2}{4000}\right) - \prod_{\mathrm{i}=1}^{\mathrm{n}} \cos\left(\frac{\mathrm{x_i}}{\sqrt{\mathrm{i}}}\right) + 1 \\ -600 &\leq \mathrm{x_i} \leq 600 \end{aligned} \tag{6.1}$$

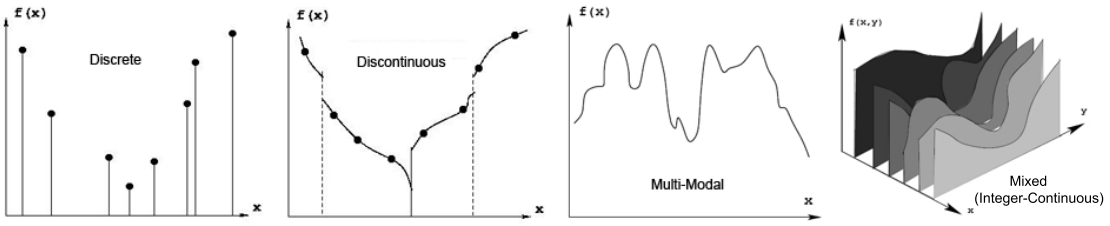


Figure 6.1: Graphs illustrating the various types of functions that are problematic for gradient-based optimization algorithms

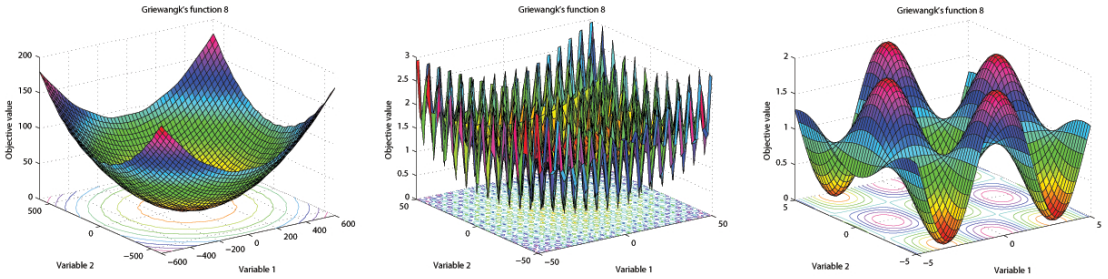


Figure 6.2: The Griewank function looks deceptively smooth when plotted in a large domain (left), but when you zoom in, you can see that the design space has multiple local minima (center) although the function is still smooth (right)

How we could find *the best* solution for this example?

- Multiple point restarts of gradient (local) based optimizer
- Systematically search the design space
- Use gradient-free optimizers

Many gradient-free methods mimic mechanisms observed in nature or use heuristics. Unlike gradient-based methods in a convex search space, gradient-free methods are not necessarily guaranteed to find the true global optimal solutions, but they are able to find many good solutions (the mathematician's answer vs. the engineer's answer).

The key strength of gradient-free methods is their ability to solve problems that are difficult to solve using gradient-based methods. Furthermore, many of them are designed as global optimizers and thus are able to find multiple local optima while searching for the global optimum.

Various gradient-free methods have been developed. We are going to look at some of the most commonly used algorithms:

- Nelder–Mead Simplex (Nonlinear Simplex)
- Simulated Annealing
- Divided Rectangles Method
- Genetic Algorithms
- Particle Swarm Optimization

6.2 Nelder–Mead Simplex

The simplex method of Nelder and Mead performs a search in n-dimensional space using heuristic ideas. It is also known as the *nonlinear simplex* and is not to be confused with the linear simplex, with which it has nothing in common.

Its main strengths are that it requires no derivatives to be computed and that it does not require the objective function to be smooth. The weakness of this method is that it is not very efficient, particularly for problems with more than about 10 design variables; above this number of variables convergence becomes increasingly difficult.

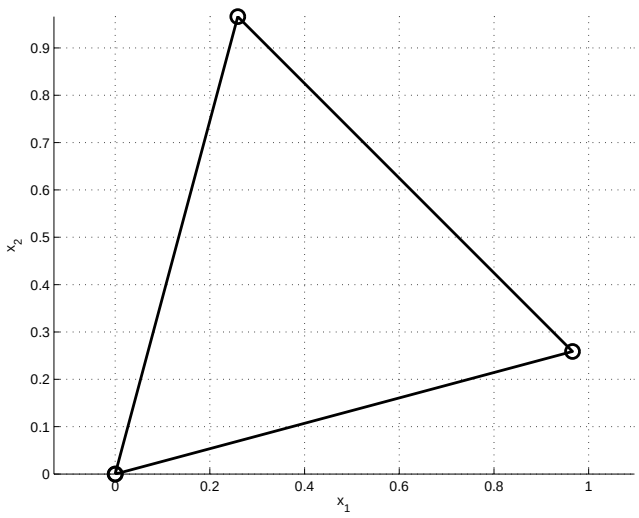


Figure 6.3: Starting simplex for $n = 2$, $x_0 = [0, 0]^T$ and $c = 1$. The two other vertices are $[a, b]$ and $[b, a]$.

A *simplex* is a structure in n-dimensional space formed by $n+1$ points that are not in the same plane. Hence a line segment is a 1-dimensional simplex, a triangle is a 2-dimensional simplex and a tetrahedron forms a simplex in 3-dimensional space. The simplex is also called a hypertetrahedron.

The Nelder–Mead algorithm starts with a simplex ($n + 1$ sets of design variables x) and then modifies the simplex at each iteration using four 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.

The first step of the simplex algorithm is to find the $n+1$ points of the simplex given an initial guess x_0. This can be easily done by simply adding a step to each component of x_0 to generate n new points.

However, generating a simplex with equal length edges is preferable. Suppose the length of all sides is required to be c and that the initial guess, x_0 is the $(n + 1)^{\text{th}}$ point. The remaining points of the simplex, $i = 1, \ldots, n$ can be computed by adding a vector to x_0 whose components are all b except for the i^{th} component which is set to a, where

$$b = \frac{c}{n\sqrt{2}}\left(\sqrt{n+1} - 1\right) \tag{6.2}$$

$$a = b + \frac{c}{\sqrt{2}}. \tag{6.3}$$

After generating the initial simplex, we have to evaluate the objective function at each of its vertices in order to identify three key points: the points associated with the highest ("worst") the second highest ("lousy") and the lowest ("best") values of the objective.

The Nelder–Mead algorithm performs four main operations to the simplex: *reflection*, *expansion*, *outside contraction*, *inside contraction* and *shrinking*. Each of these operations generates a new point and the sequence of operations performed in one iteration depends on the value of the objective at the new point relative to the other key points. Let x_w, x_l and x_b denote the worst, the second worst (or "lousy") and best points, respectively, among all $n + 1$ points of the simplex. We first

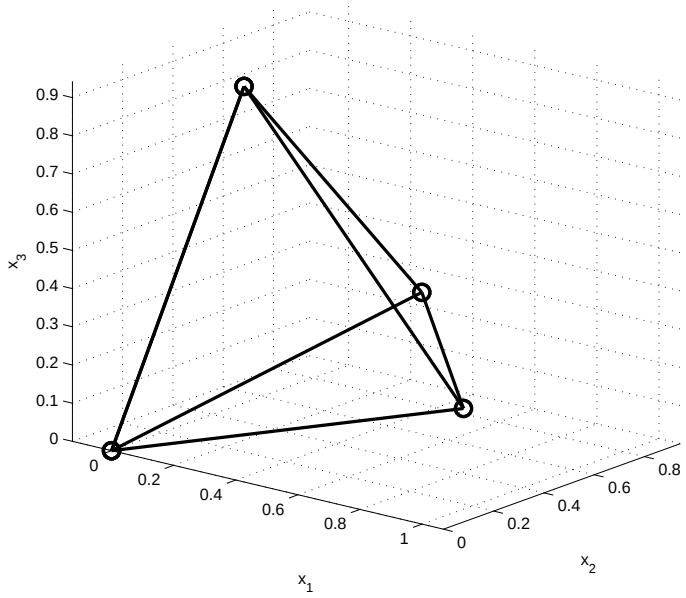


Figure 6.4: Starting simplex for $n = 3$, $x_0 = [0, 0, 0]^T$ and $c = 1$. The three other vertices are $[a, b, b]$, $[b, a, b]$ and $[b, b, a]$.

calculate the average of the n points that exclude exclude the worst,

$$x_a = \frac{1}{n} \sum_{i=1, i \neq w}^{n+1} x_i. \tag{6.4}$$

After computing x_a, we know that the line from x_w to x_a is a descent direction. A new point is found on this line by *reflection*, given by

$$x_r = x_a + \alpha \left(x_a - x_w \right) \tag{6.5}$$

If the value of the function at this reflected point is better than the best point, then the reflection has been especially successful and we step further in the same direction by performing an *expansion*,

$$x_e = x_r + \gamma \left(x_r - x_a \right), \tag{6.6}$$

where the expansion parameter γ is usually set to 1.

If, however the reflected point is worse than the worst point, we assume that a better point exists between x_w and x_a and perform an *inside contraction*

$$x_c = x_a - \beta \left(x_a - x_w \right), \tag{6.7}$$

where the contraction factor is usually set to $\beta = 0.5$. If the reflected point is not worse than the worst but is still worse than the lousy point, then an *outside contraction* is performed, that is,

$$x_o = x_a + \beta \left(x_a - x_w \right). \tag{6.8}$$

After the expansion, we accept the new point if the value of the objective function is better than the best point. Otherwise, we just accept the previous reflected point.

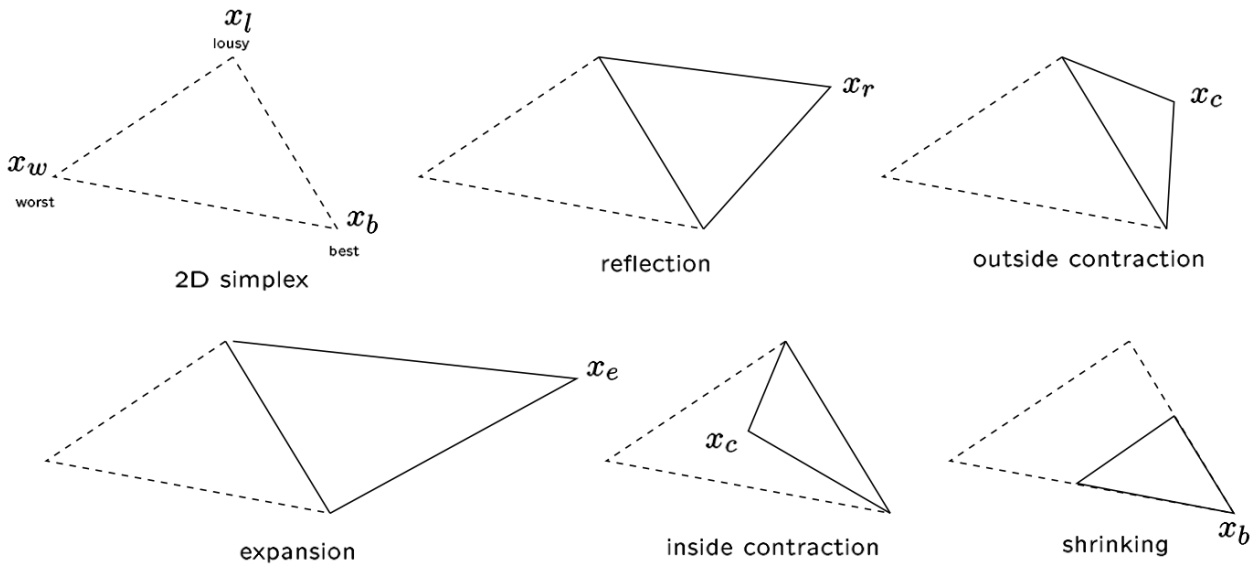


Figure 6.5: Operations performed on the simplex in Nelder–Mead's algorithm for $n = 2$.

If reflection, expansion, and contraction fail, we resort to a *shrinking* operation. This operation retains the best point and shrinks the simplex, that is, for all point of the simplex except the best one, we compute a new position

$$x_i = x_b + \rho\,(x_i - x_b)\,, \tag{6.9}$$

where the scaling parameter is usually set to $\rho = 0.5$.

The simplex operations are shown in Figure 6.5 for the $n = 2$ case. A flow chart of the algorithm is shown in Figure 6.6 and the algorithm itself in Algorithm 12.

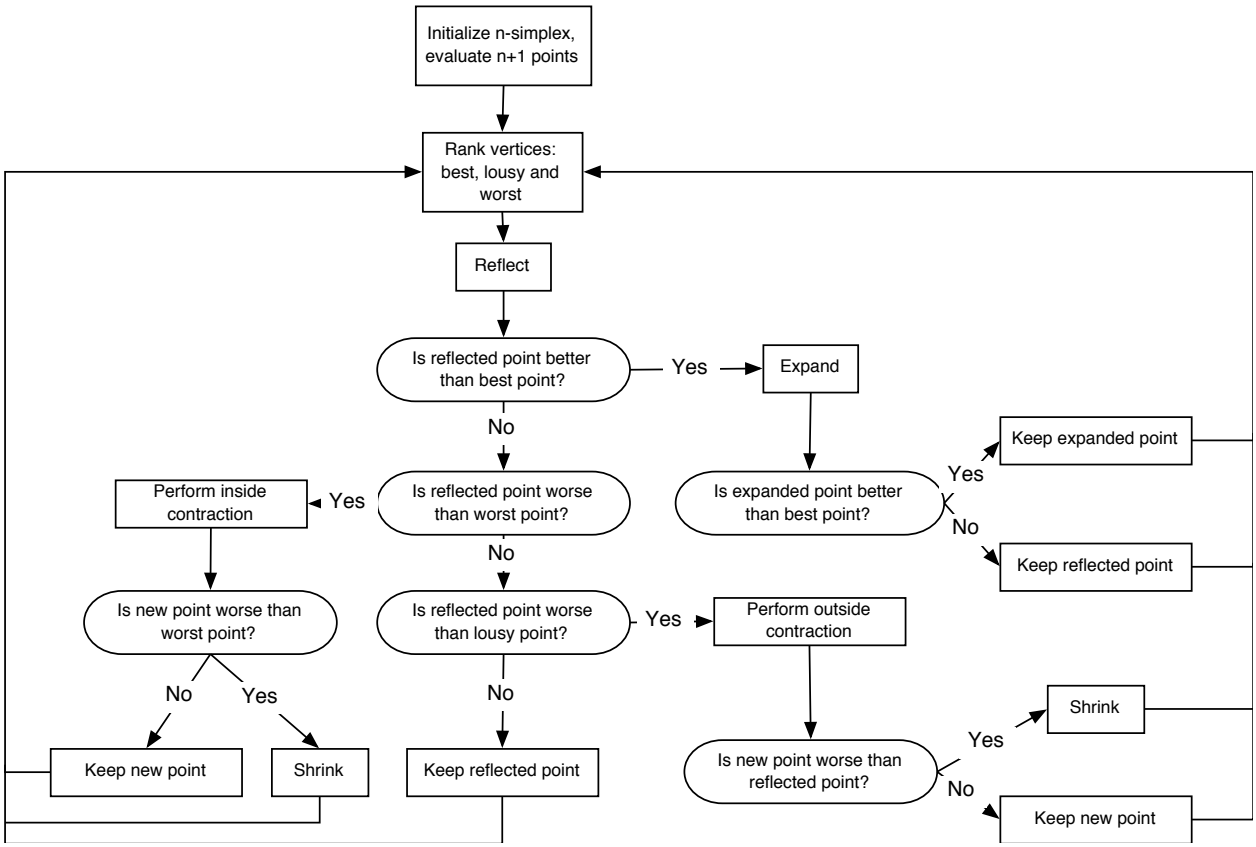


Figure 6.6: Flow chart for the Nelder–Mead algorithm. The best point is that with the lowest value of the objective function, the worst point is that with the highest value and the lousy point has the second highest value.

Algorithm 12 Nelder–Mead Algorithm

Input: Initial guess, x_0
Output: Optimum, x^*

```
k ← 0
Create a simplex with edge length c
repeat
  Identify the highest (x_w: worst), second highest (x_l, lousy) and lowest (x_b: best) value points
  with function values f_w, f_l, and f_b, respectively
  Evaluate x_a, the average of the point in simplex excluding x_w
  Perform reflection to obtain x_r, evaluate f_r
  if f_r < f_b then
    Perform expansion to obtain x_e, evaluate f_e.
    if f_e < f_b then
      x_w ← x_e, f_w ← f_e (accept expansion)
    else
      x_w ← x_r, f_w ← f_r (accept reflection)
    end if
  else if f_r ≤ f_l then
    x_w ← x_r, f_w ← f_r (accept reflected point)
  else
    if f_r > f_w then
      Perform an inside contraction and evaluate f_c
      if f_c < f_w then
        x_w ← x_c (accept contraction)
      else
        Shrink the simplex
      end if
    else
      Perform an outside contraction and evaluate f_c
      if f_c ≤ f_r then
        x_w ← x_c (accept contraction)
      else
        Shrink the simplex
      end if
    end if
  end if
  k ← k + 1
until (f_w − f_b) < (ε_1 + ε_2|f_b|)
```

The convergence criterion can also be that the size of simplex, i.e.,

$$s = \sum_{i=1}^{n} |x_i - x_{n+1}| \tag{6.10}$$

must be less than a certain tolerance. Another measure of convergence that can be used is the standard deviation,

$$\sigma = \sqrt{\frac{\sum_{i=1}^{n+1} \left(f_i - \bar{f}\right)^2}{n+1}}$$

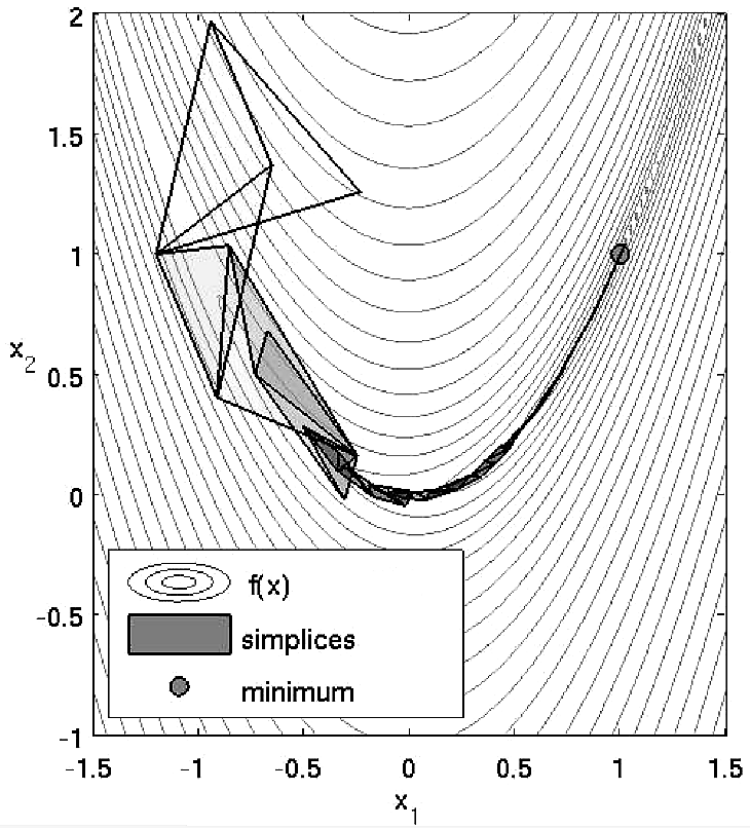


Figure 6.7: Sequence of simplices that minimize the Rosenbrock function

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

Variations of the simplex algorithm described above have been tried, with interesting results. For example, note that if $f_e < f_b$ but f_r is even better, that is $f_r < f_e$, the algorithm still accepts the expanded point x_e. Now, it is standard practice to accept the best of f_r and f_e

Example 6.28. Minimization of the Rosenbrock Function Using Nelder–Meade

Figure 6.7 shows the sequence of simplices that results when minimizing the Rosenbrock function. The initial simplex on the upper left is equilateral. The first iteration is an inside contraction, followed by a reflection, another inside contraction and then an expansion. The simplices then reduce dramatically in size and follow the Rosenbrock valley, slowly converging to the minimum.

6.3 DIvided RECTangles (DIRECT) Method

As we saw before, one of the strategies to do global optimization is to perform a systematic search of the design space. The DIRECT method uses a hyperdimensional adaptive meshing scheme to search all the design space to find the optimum. The overall idea behind DIRECT is as follows.

1. The algorithm begins by scaling the design box to a n-dimensional unit hypercube and evaluating the objective function at the center point of the hypercube

2. The method then divides the potentially optimal hyper-rectangles by sampling the longest coordinate directions of the hyper-rectangle and trisecting based on the directions with the

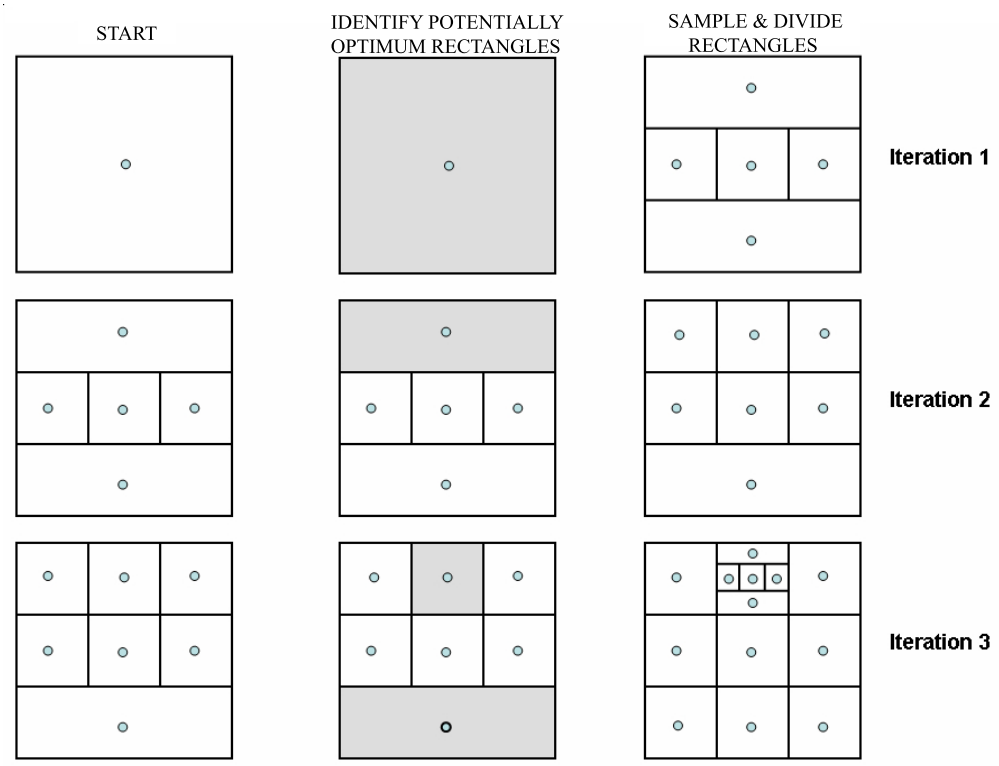


Figure 6.8: Example of DIRECT Iterations

smallest function value until the global minimum is found

3. Sampling of the maximum length directions prevents boxes from becoming overly skewed and trisecting in the direction of the best function value allows the biggest rectangles contain the best function value. This strategy increases the attractiveness of searching near points with good function values

4. Iterating the above procedure allow to identify and zoom into the most promising design space regions

To identify the *potentially optimal rectangles* we do the following. Assuming that the unit hypercube with center c_i is divided into m hyper-rectangles, a hyper-rectangle j is said to be potentially optimal if there exists rate-of-change constant $\bar{K} > 0$ such that

$$\begin{aligned} &f(c_j) - \bar{K} d_j \leq f(c_i) - \bar{K} d_i \quad \text{for all} \quad i = 1, ..., m \\ &f(c_j) - \bar{K} d_j \leq f_{\min} - \varepsilon |f_{min}|, \end{aligned} \tag{6.11}$$

where d is the distance between c and the vertices of the hyper-rectangle, $f_{\min}$ is the best current value of the objective function, and ε is positive parameter used so that $f(c_j)$ exceeds the current best solution by a non-trivial amount.

To visualize these equations, we plot the f versus the d for a given group of points in Figure 6.9. The line connecting the points with the lowest f for a given d (or greatest d for a given f) represent the points with the most potential. The first equation forces the selection of the rectangles on this line, and the second equation insists that the obtained function value exceeds the current best function value by an amount that is not insignificant. This prevents the algorithm from becoming

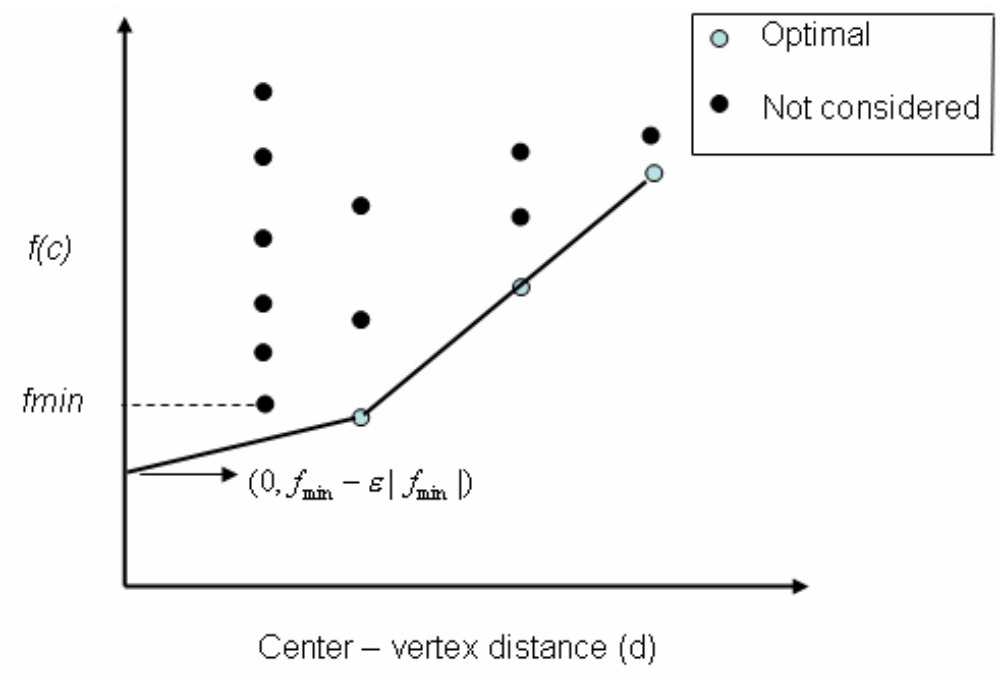


Figure 6.9: Identification of potentially optimal rectangles

too local, wasting precious 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 the range is usually such that $10^{-2} \leq \varepsilon \leq 10^{-7}$.

Algorithm 13 DIRECT Algorithm

Input: Initial guess, x_0
Output: Optimum, x^*

```
k ← 0
repeat
  Normalize the search space to be the unit hypercube. Let c_1 be the center point of this
  hypercube and evaluate f(c_1).
  Identify the set S of potentially optimal rectangles/cubes, that is all those rectangles defining
  the bottom of the convex hull of a scatter plot of rectangle diameter versus f(c_i) for all rectangle
  centers c_i
  for all Rectangles r ∈ S do
    Identify the set I of dimensions with the maximum side length
    Set δ equal one third of this maximum side length
    for all i ∈ I do
      Evaluate the rectangle/cube at the point c_r ± δe_i for all i ∈ I, where c_r is the center of
      the rectangle r, and e_i is the i^th unit vector
    end for
    Divide the rectangle r into thirds along the dimensions in I, starting with the dimension with
    the lowest value of f(c ± δe_i) and continuing to the dimension with the highest f(c ± δe_i).
  end for
until Converged
```

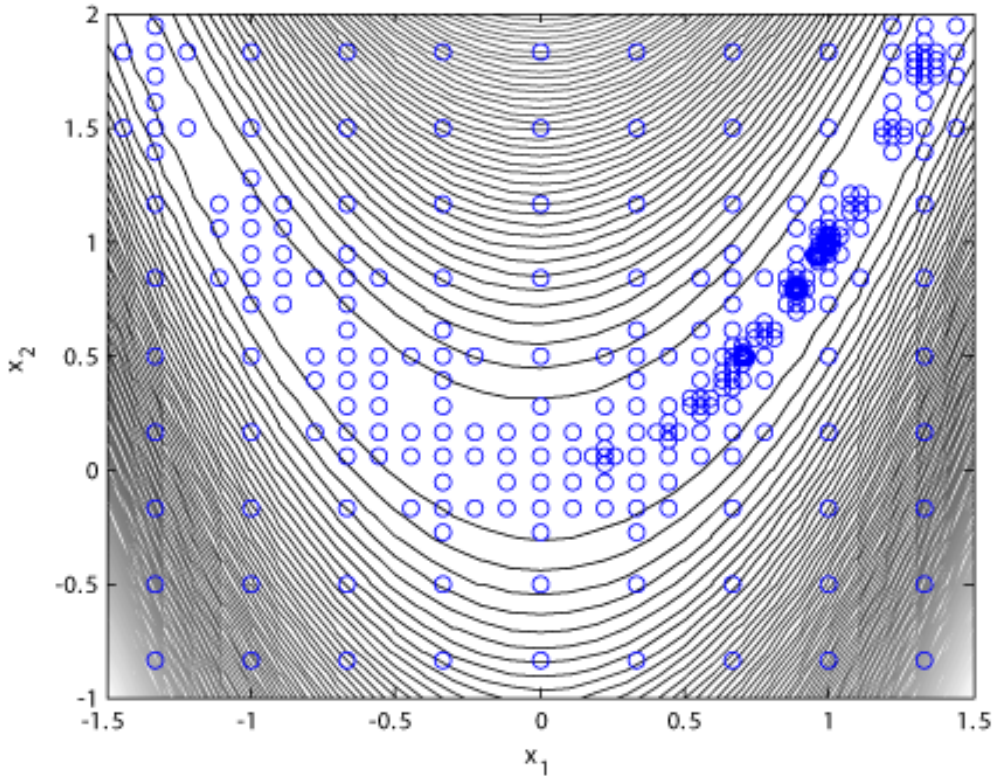


Figure 6.10: Minimization of the Rosenbrock function using DIRECT

Example 6.29. Minimization of the Rosenbrock Function Using DIvided RECTangles Method

6.4 Genetic Algorithms

Genetic algorithms for optimization were inspired by the process of natural evolution of organisms. This type of algorithm was first developed by John Holland in the mid 1960's. Holland was motivated by a desire to better understand the evolution of life by simulating it in a computer and the use of this process in optimization.

Genetic algorithms are based on three essential components:

- Survival of the fittest (Selection)
- Reproduction processes where genetic traits are propagated (Crossover)
- Variation (Mutation)

We will use the term "genetic algorithms" generically, which is the most common term used when expressing a large family of optimization approaches that use the three components above. Depending on the approach they have different names, for example: genetic algorithms, evolutionary computation, genetic programming, evolutionary programming, evolutionary strategies.

We will start by posing the unconstrained optimization problem with design variable bounds,

$$\begin{aligned} \text{minimize} \quad & f(x) \\ \text{subject to} \quad & x_l \leq x \leq x_u \end{aligned}$$

where x_l and x_u are the vectors of lower and upper bounds on x, respectively.

In the context of genetic algorithms we will call each design variable vector x a *population member*. The value of the objective function, $f(x)$ is termed the *fitness*.

Genetic algorithms are radically different from the gradient based methods we have covered so far. Instead of looking at *one* point at a time and stepping to a new point for each iteration, a whole *population* of solutions is iterated towards the optimum at the same time. Using a population lets us explore multiple "buckets" (local minima) simultaneously, increasing the likelihood of finding the global optimum.

The main advantages of genetic algorithms are:

- The basic algorithm uses a coding of the parameter set, not the parameter themselves; consequently, the algorithm can handle mixed continuous, integer and discrete design variables.
- The population can cover a large range of the design space and is less likely than gradient based methods to "get stuck" in local minima.
- As with other gradient-free methods like Nelder–Mead's algorithm, it can handle noisy objective functions.
- The implementation is straightforward and easily parallelized.
- Can be used for *multiobjective optimization*.

There is "no free lunch", of course, and these methods have some disadvantages. The main one is that genetic algorithms are expensive when compared to gradient-based methods, especially for problems with a large number of design variables.

However, there are cases where it is difficult to make gradient-based methods work or where they do not work at all. In some of these problems genetic algorithms work very well with little effort.

Although genetic algorithms are much better than completely random methods, they are still "brute force" methods that require a large number of function evaluations.

Single-Objective Optimization

The procedure of a genetic algorithm can be described as follows:

1. **Initialize a Population**

   Each member of the population represents a design point, x and has a value of the objective (fitness), and information about its constraint violations associated with it.

2. **Determine Mating Pool**

   Each population member is paired for reproduction by using one of the following methods:

   - Random selection
   - Based on fitness: make the better members to reproduce more often than the others.

3. **Generate Offspring**

   To generate offspring we need a scheme for the crossover operation. There are various schemes that one can use. When the design variables are continuous, for example, one offspring can be found by interpolating between the two parents and the other one can be extrapolated in the direction of the fitter parent.

4. **Mutation**

   Add some randomness in the offspring's variables to maintain diversity.

5. **Compute Offspring's Fitness**

   Evaluate the value of the objective function and constraint violations for each offspring.

6. **Tournament**

   Again, there are different schemes that can be used in this step. One method involves replacing the worst parent from each "family" with the best offspring.

7. **Identify the Best Member**

   Convergence is difficult to determine because the best solution so far may be maintained for many generations. As a rule of thumb, if the best solution among the current population hasn't changed (much) for about 10 generations, it can be assumed as the optimum for the problem.

8. **Return to Step 2**

Note that since GAs are probabilistic methods (due to the random initial population and mutation), it is crucial to run the problem multiple times when studying its characteristics.

Multi-Objective Optimization

What if we want to investigate the trade-off between two (or more) conflicting objectives? In the design of a supersonic aircraft, for example, we might want to simultaneously minimize aerodynamic drag and sonic boom and we do not know what the trade-off is. How much would the drag increase for a given reduction in sonic boom?

In this situation there is no one "best design". There is a set (a population) of designs that are the best possible for that combination of the two objectives. In other words, for these optimal solutions, the only way to improve one objective is to worsen the other.

This is perfect application for genetic algorithms. We already have to evaluate a whole population, so we can use this to our advantage. This is much better than using composite weighted functions in conjunction with gradient-based methods since there is no need to solve the problem for various weights.

The concept of *dominance* is the key to the use of GA's in multi-objective optimization.

As an example, assume we have a population of 3 members, A, B and C, and that we want to minimize two objective functions, f_1 and f_2.

Comparing members A and B, we can see that A has a higher (worse) f_1 than B, but has a lower (better) f_2. Hence we cannot determine whether A is better than B or vice versa.

On the other hand, B is clearly a fitter member than C since both of B's objectives are lower. We say that B *dominates* C.

Comparing A and C, once again we are unable to say that one is better than the other.

In summary:

Member	f_1	f_2
A	10	12
B	8	13
C	9	14

- A is non-dominated by either B or C
- B is non-dominated by either A or C
- C is dominated by B but not by A

The *rank* of a member is the number of members that dominate it plus one. In this case the ranks of the three members are:

$$\text{rank(A)} = 1$$
$$\text{rank(B)} = 1$$
$$\text{rank(C)} = 2$$

In multi-objective optimization the rank is crucial in determining which population member are the fittest. A solution of rank one is said to be *Pareto optimal* and the set of rank one points for a given generation is called the *Pareto set*. As the number of generations increases, and the fitness of the population improves, the size of the Pareto set grows.

In the case above, the Pareto set includes A and B. The graphical representation of a Pareto set is called a *Pareto front*.

The procedure of a two-objective genetic algorithm is similar to the single-objective one, with the following modifications.

1. **Initialize a Population**

   Compute all objectives and rank the population according to the definition of dominance

2. **Determine Mating Pool**

3. **Generate Offspring**

4. **Mutation**

5. **Compute Offspring's Fitness**

   Evaluate their objectives and rank offspring population.

6. **Tournament**

   Now this is based on rank rather than objective function.

7. **Rank New Population**

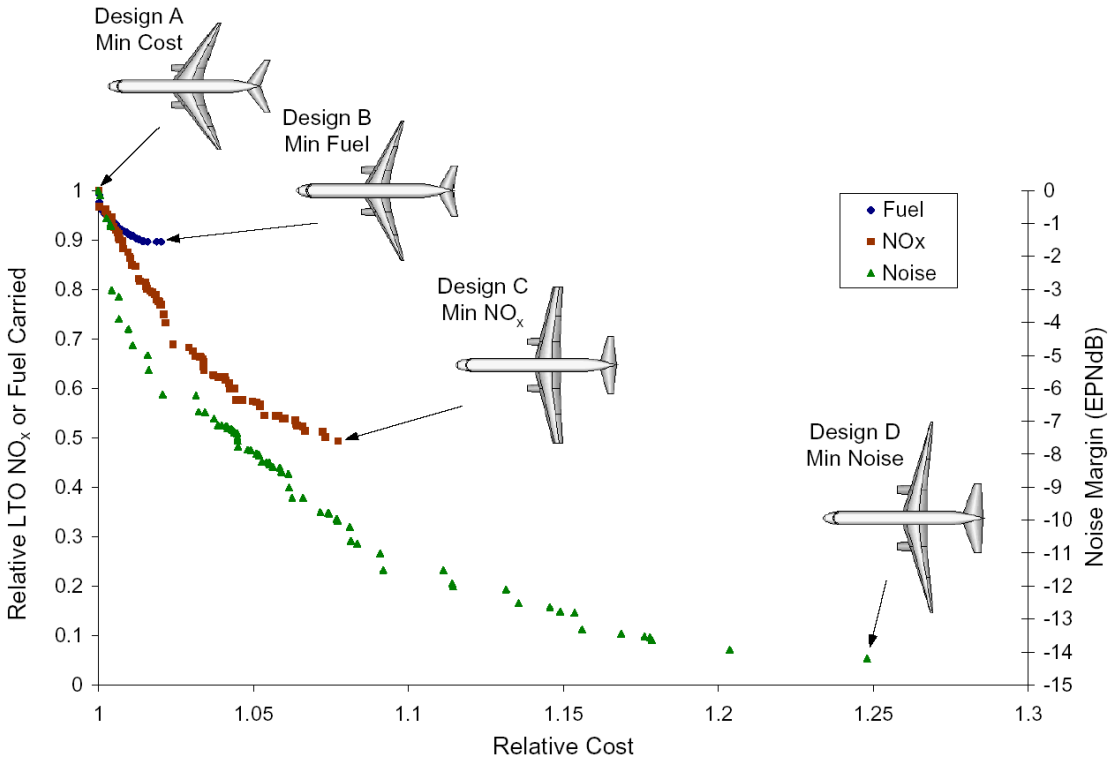
   Display rank one population, that is the Pareto set.

8. **Repeat From Step 2**

   We iterate until the Pareto front is no longer "moving".

One of the problems with this method is that there is no mechanism "pushing" the Pareto front to a better one. If one new member dominates every other member, the rank of the other members grows by one and the process continues. Some work has been invested in applying gradients to the front, with mixed results.

Example 6.30. Pareto Front in Aircraft Design [1]



6.4.1 Coding and Decoding of Variables

In a few variants of genetic algorithms, the design variables are stored as a vector of real numbers, just as one would do for the other optimization methods we have covered in this course.

However, genetic algorithms more commonly represent each variable as a binary number of say m bits. If we want to represent a real-valued variable, we have to divide the feasible interval of x_i into $2^m - 1$ intervals. Then each possibility for x_i can be represented by any combination of m bits. For $m = 5$, for example, the number of intervals would be 31 and a possible representation for x_i would be 10101, which can be decoded as

$$x_i = x_l + s_i \left(1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0\right) = x_l + 21 s_i, \tag{6.12}$$

where s_i is the size of interval for x_i given by

$$s_i = \frac{x_{u_i} - x_{l_i}}{31}. \tag{6.13}$$

Creation of the Initial Population

As a rule of thumb, the population size should be of 15 to 20 times the number of design variables. Experiment with different population sizes.

Using bit encoding, each bit is assigned a 50% chance of being either 1 or 0. One way of doing this is to generate a random number $0 \leq r \leq 1$ and setting the bit to 0 if $r \leq 0.5$ and 1 if $r > 0.5$.

Each member is chosen at random. For a problem with real design variables and a given variable x such that $x_l \leq x \leq x_u$, we could use,

$$x = x_l + r(x_u - x_l) \tag{6.14}$$

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

Often genetic algorithms represent design variables as a binary number of m bits. The coding and decoding of the design variables then depends on the nature of the design variables themselves.

6.4.2 Selection: Determining the Mating Pool

Let's assume we want to *maximize* $f(x)$. Consider the highest (best) and the lowest (worst) values, f_h and f_l respectively.

It is convenient to scale the function values such that they all become positive. This can be done by assigning a rank as the fitness value for each member of the population.

The function values can be converted to a positive quantity by adding,

$$C = 0.1f_h - 1.1f_l \tag{6.15}$$

to each function value. Thus the new highest value will be $1.1(f_h - f_l)$ and the new lowest value $0.1(f_h - f_l)$. The values are then normalized as follows,

$$f_i' = \frac{f_i + C}{D} \tag{6.16}$$

where

$$D = \max(1, f_h + C). \tag{6.17}$$

After the fitness values are scaled, they are summed,

$$S = \sum_{i=1}^{N} f_i' \tag{6.18}$$

where N is the number of members in the population.

We now use *roulette wheel selection* to make copies of the fittest members for reproduction. A mating pool of N members is created by turning the roulette wheel N times. A random number $0 \leq r \leq 1$ is generated at each turn. The j^{th} member is copied to the mating pool if

$$f_1' + \ldots + f_{j-1}' \leq rS \leq f_1' + \ldots + f_j' \tag{6.19}$$

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

Crossover Operation (Generating Offspring)

Various crossover strategies are possible in genetic algorithms.

The following crossover strategy is one devised specifically for optimization problems with real-valued design variables. In this case each member of the population corresponds to a point in n-space, that is, a vector x

Let two members of the population that have been mated (parents) be ordered such that $f_{p1} < f_{p2}$. Two offspring are to be generated.

The first offspring is interpolated using

$$x_{c1} = \frac{1}{2}(x_{p1} + x_{p2}) \tag{6.20}$$

which yields the midpoint between the two parents.

The second offspring is extrapolated in the direction defined by the two parents using

$$x_{c2} = 2x_{p1} + x_{p2} \tag{6.21}$$

which gives a point located beyond the better parent.

Using this strategy the tournament, or selection, is performed by selecting the best parent (x_{p1}) and either the second parent or the best offspring, whichever is the best one of the two.

When the information is stored as bits, the crossover operation usually involves generating a random integer $1 \leq k \leq m-1$ that defines the *crossover point*. For one of the offspring, the first k bits are taken from say parent 1 and the remaining bits from parent 2. For the second offspring, the first k bits are taken from parent 2 and the remaining ones from parent 1.

Before Crossover	After Crossover
11 111	11 000
00 000	00 111

6.4.3 Mutation

Mutation is a random operation performed to change the genetic information. Mutation is needed because even though reproduction and crossover effectively recombine existing information, occasionally some useful genetic information might be lost. The mutation operation protects against such irrecoverable loss. It also introduces additional diversity into the population.

When using bit representation, every bit is assigned a small permutation probability, say $p = 0.005 \sim 0.1$. This is done by generating a random number $0 \leq r \leq 1$ for each bit, which is changed if $r < p$.

Before Mutation	After Mutation
11111	11010

6.4.4 Why do genetic algorithms work?

A fundamental question which is still being researched is how the three main operations (Selection,Crossover and Mutation) are able to find better solutions. Two main mechanism allow the algorithm to progress towards better solutions:

Selection + Mutation = Improvement: Mutation makes local changes while selection accepts better changes, this can be seen as a resilient and general form of reducing the objective function.

Selection + Crossover = Innovation: When the information of the best population members is exchanged, there is a greater chance a new better combination will emerge.

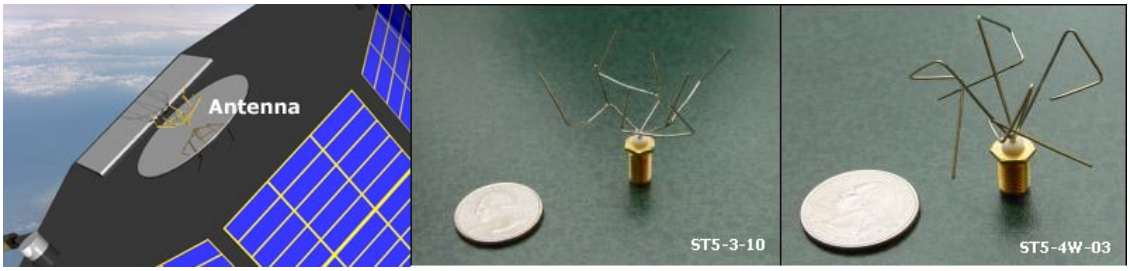
Example 6.31. Jet Engine Design at General Electric [12]

- Genetic algorithm combined with expert system
- Find the most efficient shape for the fan blades in the GE90 jet engines
- 100 design variables
- Found 2% increase in efficiency as compared to previous engines
- Allowed the elimination of one stage of the engine's compressor reducing engine weight and manufacturing costs without any sacrifice in performance

Example 6.32. ST5 Antenna The antenna for the ST5 satellite system presented a challenging design problem, requiring both a wide beam width for a circularly-polarized wave and a wide bandwidth.

The GA found an antenna configuration (ST5-3-10) that was slightly more difficult to manufacture, but:

- Used less power
- Removed two steps in design and fabrication
- Had more uniform coverage and wider range of operational elevation angle relative to the ground changes
- Took 3 person-months to design and fabricate the first prototype as compared to 5 person-months for the conventionally designed antenna.



6.5 Particle Swarm Optimization

PSO is a stochastic, population-based computer algorithm developed in 1995 by James Kennedy (social-psychologist) and Russell Eberhart (electrical engineer) it applies the concept of swarm intelligence (SI) to problem solving [5].

What is Swarm Intelligence?

SI is the property of a system whereby the collective behaviors of (unsophisticated) agents interacting locally with their environment cause coherent functional global patterns to emerge (e.g. self-organization, emergent behavior).

PSO Formulation

Basic idea:

- Each agent (or particle) represents a design point and moves in n-dimensional space looking for the best solution.
- Each agent adjusts its movement according to the effects of cognitivism (self experience) and sociocognition (social interaction).

The update of particle i's position is given by:

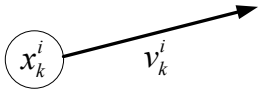
$$x_{k+1}^i = x_k^i + v_{k+1}^i \Delta t \tag{6.22}$$

where the velocity of the particle is given by

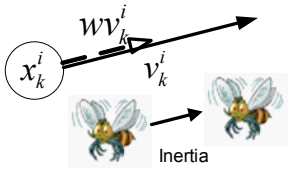
$$v_{k+1}^i = w v_k^i + c_1 r_1 \frac{\left(p_k^i - x_k^i\right)}{\Delta t} + c_2 r_2 \frac{\left(p_k^g - x_k^i\right)}{\Delta t} \tag{6.23}$$

- r_1 and r_2 are random numbers in the interval $[0, 1]$
- p_k^i is particle i's best position so far, p_k^g is the swarm's best particle position at iteration k
- c_1 is the cognitive parameter (confidence in itself), c_2 is the social parameter (confidence in the swarm)
- w is the inertia

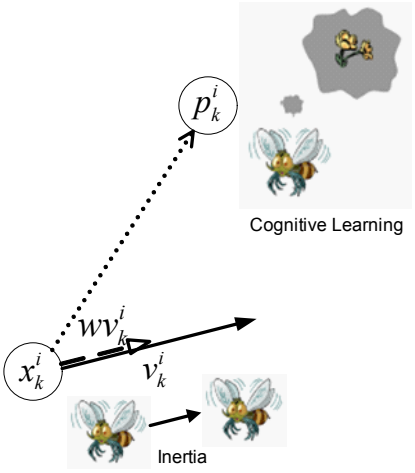
How the swarm is updated:



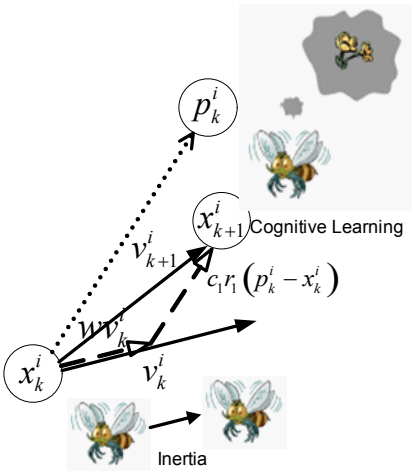
How the swarm is updated:



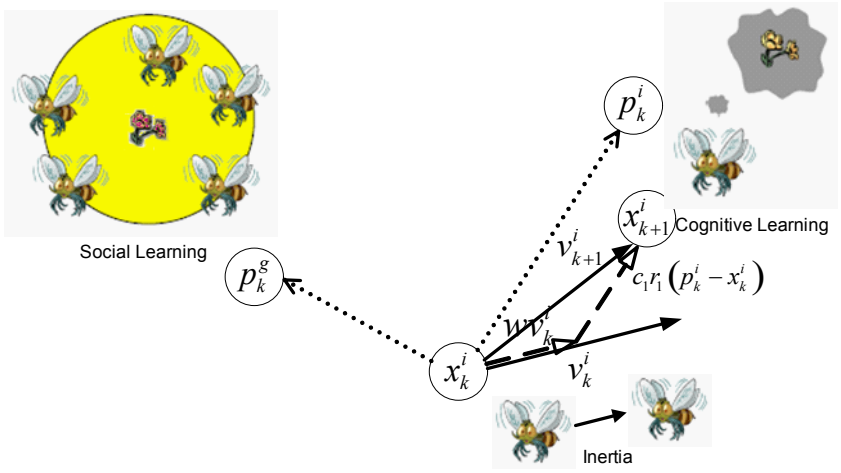
How the swarm is updated:



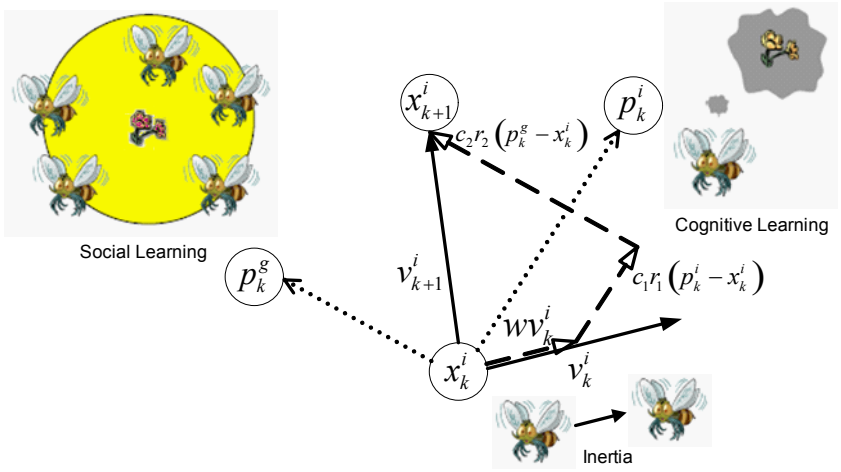
How the swarm is updated:



How the swarm is updated:



How the swarm is updated:



PSO Algorithm

1. Initialize a set of particles positions x_o^i and velocities v_o^i randomly distributed throughout the design space bounded by specified limits
2. Evaluate the objective function values $f\left(x_k^i\right)$ using the design space positions x_k^i
3. Update the optimum particle position p_k^i at current iteration (k) and global optimum particle position p_k^g
4. Update the position of each particle using its previous position and updated velocity vector.
5. Repeat steps 2-4 until the stopping criteria is met.

PSO Characteristics Compare to other global optimization approaches:

- Simple algorithm, extremely easy to implement.
- Still a population based algorithm, however it works well with few particles (10 to 40 are usual) and there is not such thing as "generations"
- Unlike evolutionary approaches, design variables are directly updated, there are no chromosomes, survival of the fittest, selection or crossover operations.
- Global and local search behavior can be directly "adjusted" as desired using the cognitive c_1 and social c_2 parameters.
- Convergence "balance" is achieved through the inertial weight factor w

Algorithm Analysis

If we replace the velocity update equation into the position update the following expression is obtained:

$$x_{k+1}^i = x_k^i + \left(w v_k^i + c_1 r_1 \frac{\left(p_k^i - x_k^i\right)}{\Delta t} + c_2 r_2 \frac{\left(p_k^g - x_k^i\right)}{\Delta t} \right) \Delta t \tag{6.24}$$

Factorizing the cognitive and social terms:

$$x_{k+1}^i = \underbrace{x_k^i + w v_k^i \Delta t}_{\hat{x}_k^i} + \underbrace{\left(c_1 r_1 + c_2 r_2\right)}_{\alpha_k} \underbrace{\left(\frac{c_1 r_1 p_k^i + c_2 r_2 p_k^g}{c_1 r_1 + c_2 r_2} - x_k^i \right)}_{\hat{p}_k} \tag{6.25}$$

So the behavior of each particle can be viewed as a line-search dependent on a stochastic step size and search direction.

Re-arranging the position and velocity term in the above equation we have:

$$\begin{aligned} x_{k+1}^i &= x_k^i \left(1 - c_1 r_1 - c_2 r_2\right) + w V_k^i \Delta t + c_1 r_1 p_k^i + c_2 r_2 p_k^g \\ v_{k+1}^i &= -x_k^i \tfrac{\left(c_1 r_1 + c_2 r_2\right)}{\Delta t} + w V_k^i + c_1 r_1 \tfrac{p_k^i}{\Delta t} + c_2 r_2 \tfrac{p_k^g}{\Delta t} \end{aligned} \tag{6.26}$$

which can be combined and written in a matrix form as:

$$\begin{bmatrix} x_{k+1}^i \\ V_{k+1}^i \end{bmatrix} = \begin{bmatrix} 1 - c_1 r_1 - c_2 r_2 & w \Delta t \\ -\frac{\left(c_1 r_1 + c_2 r_2\right)}{\Delta t} & w \end{bmatrix} \begin{bmatrix} x_k^i \\ V_k^i \end{bmatrix} + \begin{bmatrix} c_1 r_1 & c_2 r_2 \\ \frac{c_1 r_1}{\Delta t} & \frac{c_2 r_2}{\Delta t} \end{bmatrix} \begin{bmatrix} p_k^i \\ p_k^g \end{bmatrix} \tag{6.27}$$

where the above representation can be seen as a representation of a discrete-dynamic system from which we can find stability criteria [10].

Assuming constant external inputs, the system reduces to:

$$\begin{bmatrix} 0 \\ 0 \end{bmatrix} = \begin{bmatrix} -(c_1 r_1 + c_2 r_2) & w\Delta t \\ -\frac{(c_1 r_1 + c_2 r_2)}{\Delta t} & w - 1 \end{bmatrix} \begin{bmatrix} x_k^i \\ V_k^i \end{bmatrix} + \begin{bmatrix} c_1 r_1 & c_2 r_2 \\ \frac{c_1 r_1}{\Delta t} & \frac{c_2 r_2}{\Delta t} \end{bmatrix} \begin{bmatrix} p_k^i \\ p_k^g \end{bmatrix} \tag{6.28}$$

where the above is true only when $V_k^i = 0$ and $x_k^i = p_k^i = p_k^g$ (equilibrium point).

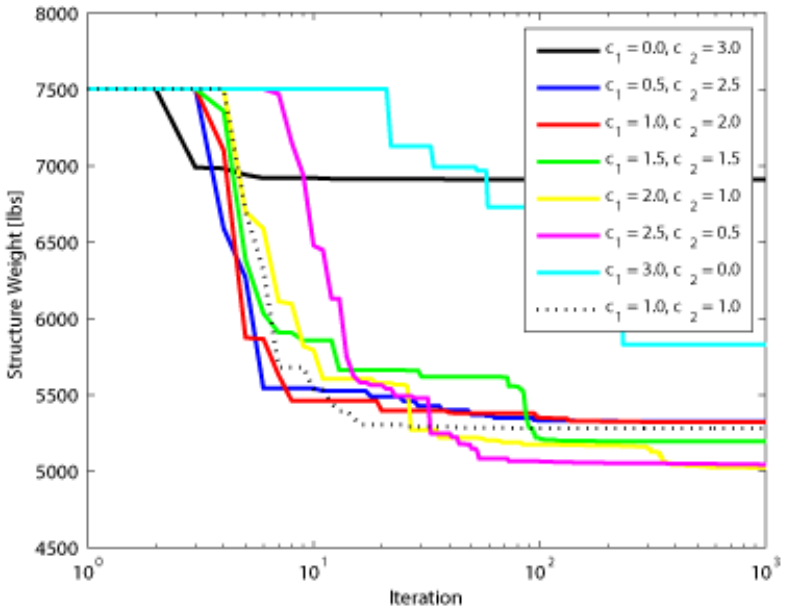
The eigenvalues of the dynamic system are:

$$\lambda^2 - (w - c_1 r_1 - c_2 r_2 + 1)\,\lambda + w = 0 \tag{6.29}$$

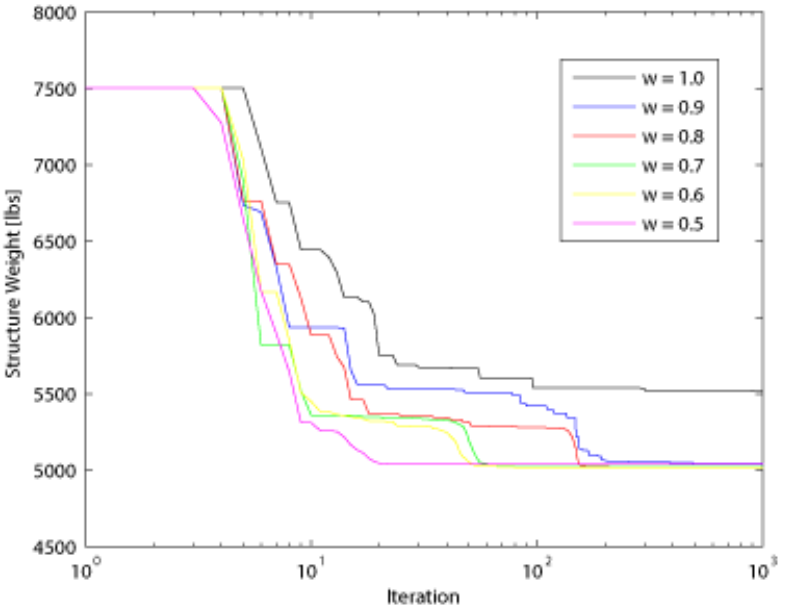
Hence, the stability in the PSO dynamic system is guaranteed if $|\lambda_{i=1,\ldots,n}| < 1$, which leads to:

$$\begin{aligned} 0 &< (c_1 + c_2) < 4 \\ \tfrac{(c_1+c_2)}{2} - 1 &< w < 1 \end{aligned} \tag{6.30}$$

Effect of varying c_1 and c_2:



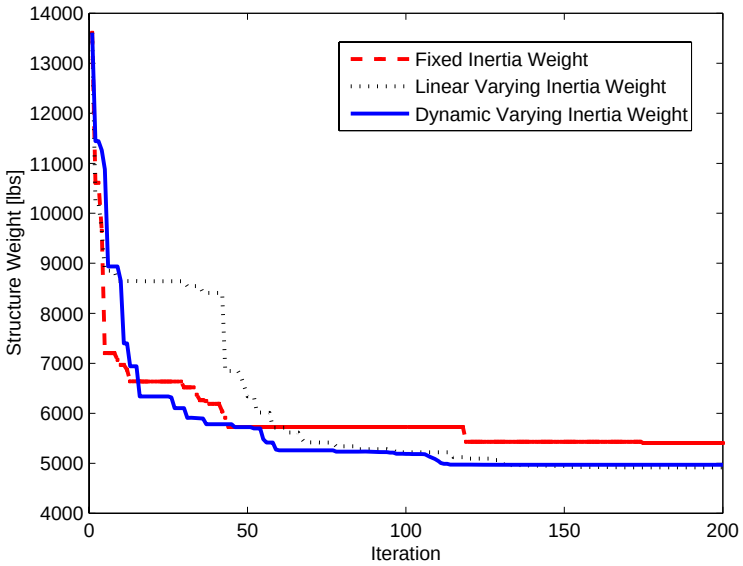
Effect of varying the inertia:



Algorithm Issues and Improvements

Updating the Inertia Weight:

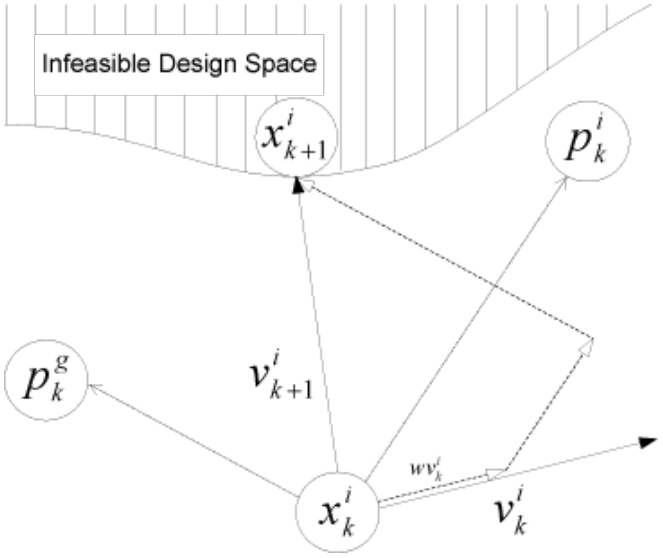
- As $k \to \infty$ particles "cluster" towards the "global" optimum.
- Fixed inertia makes the particles to overshoot the best regions (too much momentum).
- A better way of controlling the global search is to dynamically update the inertia weight.



Violated Design Points Redirection:

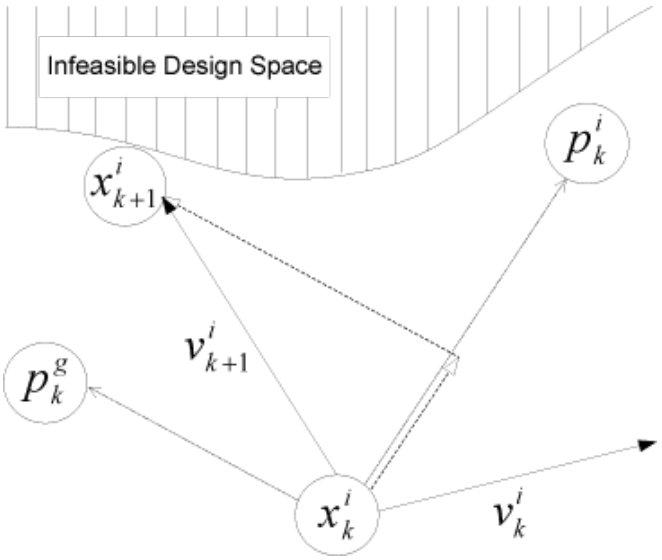
We can restrict the velocity vector of a constraint violated particle to a usable feasible direction:

$$v_{k+1}^i = c_1 r_1 \frac{\left(p_k^i - x_k^i\right)}{\Delta t} + c_2 r_2 \frac{\left(p_k^g - x_k^i\right)}{\Delta t} \tag{6.31}$$



Violated Design Points Redirection:
We can restrict the velocity vector of a constraint violated particle to a usable feasible direction:

$$v_{k+1}^i = c_1 r_1 \frac{\left(p_k^i - x_k^i\right)}{\Delta t} + c_2 r_2 \frac{\left(p_k^g - x_k^i\right)}{\Delta t} \tag{6.32}$$



Constraint Handling:
The basic PSO algorithm is an unconstrained optimizer, to include constraints we can use:

- Penalty Methods
- Augmented Lagrangian function

Recall the Lagrangian function:

$$\mathcal{L}_i\left(x_k^i, \lambda^i\right) = f\left(x_k^i\right) + \sum_{j=1}^{m} \lambda_j^i g_j\left(x_k^i\right) \tag{6.33}$$

Augmented Lagrangian function:

$$\mathcal{L}_i\left(x_k^i, \lambda^i, r_p^i\right) = f\left(x_k^i\right) + \sum_{j=1}^{m} \lambda_j^i \theta_j\left(x_k^i\right) + \sum_{j=1}^{m} r_{p,j} \theta_j^2\left(x_k^i\right) \tag{6.34}$$

where:

$$\theta_j\left(x_k^i\right) = \max\left[g_j\left(x_k^i\right), \frac{-\lambda_j}{2r_{p,i}}\right] \tag{6.35}$$

- Multipliers and penalty factors that lead to the optimum are unknown and problem dependent.
- A sequence of unconstrained minimizations of the Augmented Lagrange function are required to obtain a solution.

Multiplier Update

$$\lambda_j^i\big|_{v+1} = \lambda_j^i\big|_v + 2\, r_{p,j}|_v\, \theta_j\left(x_k^i\right) \tag{6.36}$$

Penalty Factor Update (Penalizes infeasible movements):

$$r_{p,j}|_{v+1} = \begin{cases} 2\, r_{p,j}|_v & \text{if} \quad g_j\left(x_v^i\right) > g_j\left(x_{v-1}^i\right) \wedge g_j\left(x_v^i\right) > \varepsilon_g \\ \frac{1}{2} r_{p,j}\big|_v & \text{if} \quad g_j\left(x_v^i\right) \leq \varepsilon_g \\ r_{p,j}\big|_v & \text{otherwise} \end{cases} \tag{6.37}$$

Augmented Lagrangian PSO Algorithm [7]:

1. Initialize a set of particles positions x_o^i and velocities v_o^i randomly distributed throughout the design space bounded by specified limits. Also initialize the Lagrange multipliers and penalty factors, e.g. $\lambda_j^i\Big|_0 = 0$, $r_{p,j}|_0 = r_0$. Evaluate the objective function values using the initial design space positions.
2. Solve the unconstrained optimization problem described in the Augmented Lagrange Multiplier Equation using the basic PSO algorithm for $k_{\max}$ iterations.
3. Update the Lagrange multipliers and penalty factors.
4. Repeat steps 2–4 until a stopping criterion is met.

6.6 Some Examples

So how do the different gradient-free methods compare?

A simple numerical example:

The Griewank Function for $n = 100$

Optimizer	F-Evals	Global Opt?	F-Val	t-CPU (s)
PSO (pop 40)	12,001	Yes	6.33e-07	15.9
GA (pop 250)	51,000	No	86.84	86.8438
DIRECT	649,522	Yes	1.47271e-011	321.57

Table 6.1: Comparison of gradient-free methods applied the minimization of the Griewank function. The results of PSO and GA are the mean of 50 trials.

Bibliography

[1] Nicolas E. Antoine and Ilan M. Kroo. Aircraft optimization for minimal environmental impact. *Journal of Aircraft*, 41(4):790–797, 2004.

[2] Ashok D. Belegundu and Tirupathi R. Chandrupatla. *Optimization Concepts and Applications in Engineering*, chapter 7. Prentice Hall, 1999.

[3] Andrew R. Conn, Katya Scheinberg, and Luis N. Vicente. *Introduction to Derivative-Free Optimization*. SIAM, 2008.

[4] J. E. Dennis and Virginia J. Torczon. Derivative-free pattern search methods for multidisciplinary design problems. *AIAA Paper* 94-4349, 1994.

[5] R.C. Eberhart and J.A. Kennedy. New optimizer using particle swarm theory. In *Sixth International Symposium on Micro Machine and Human Science*, pages 39–43, Nagoya, Japan, 1995.

[6] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*, chapter 1. Addison-Wesley Longman Publishing, Boston, MA, 1989.

[7] Peter W. Jansen and Ruben E. Perez. Constrained structural design optimization via a parallel augmented Lagrangian particle swarm optimization approach. *Computers & Structures*, 89: 1352–1366, 2011. doi: 10.1016/j.compstruc.2011.03.011.

[8] J. A. Nelder and R. Mead. Simplex method for function minimization. *Computer Journal*, 7: 308–313, 1965.

[9] Chinyere Onwubiko. *Introduction to Engineering Design Optimization*, chapter 4. Prentice Hall, 2000.

[10] R. Perez and K. Behdinan. Particle swarm approach for structural design optimization. *International Journal of Computer and Structures*, 85(19-20):1579–1588, October 2007.

[11] C.D. Perttunen, D.R. Jones, and B.E. Stuckman. Lipschitzian optimization without the lipschitz constant. *Journal of Optimization Theory and Application*, 79(1):157–181, October 1993.

[12] David J. Powell, Siu Shing Tong, and Michael M. Skolnick. Engeneous domain independent, machine learning for design optimization. In *Proceedings of the third international conference on Genetic algorithms*, pages 151–159, San Francisco, CA, USA, 1989. Morgan Kaufmann Publishers Inc.

[13] Peter Sturdza. An aerodynamic design method for supersonic natural laminar flow aircraft. PhD thesis 96–98, Stanford University, Stanford, California, 2003.