9 Population Methods

Previous chapters have focused on methods where a single design point is moved incrementally toward a minimum. This chapter presents a variety of population methods that involve optimization using a collection of design points, called individuals. Having a large number of individuals distributed throughout the design space can help the algorithm avoid becoming stuck in a local minimum. Information at different points in the design space can be shared between individuals to globally optimize the objective function. Most population methods are stochastic in nature, and it is generally easy to parallelize the computation.

9.1 Initialization

Population methods begin with an initial population, just as descent methods require an initial design point. The initial population should be spread over the design space to increase the chances that the samples are close to the best regions. This section presents several initialization methods, but more advanced sampling methods are discussed in detail in chapter 13.

We can often constrain the design variables to a region of interest consisting of a hyperrectangle defined by lower and upper bounds \(a\) and \(b\). Initial populations can be sampled from a uniform distribution for each coordinate:

\[
x^{(j)}_i \sim U(a_i, b_i)
\]

where \(x^{(j)}_i\) is the \(j\)th individual in the population as seen in algorithm 9.1.

Another common approach is to use a multivariate normal distribution centered over a region of interest. The covariance matrix is typically diagonal, with diagonal entries scaled to adequately cover the search space. Algorithm 9.2 provides an implementation.

\(^1\)Some population methods require additional information to be associated with the individual, such as velocity in the case of particle swarm optimization, discussed later. Velocity is often initialized according to a uniform or normal distribution.
function rand_population_uniform(m, a, b)
    d = length(a)
    return [a+rand(d).* (b-a) for i in 1:m]
end

using Distributions
function rand_population_normal(m, μ, Σ)
    D = MvNormal(μ,Σ)
    return [rand(D) for i in 1:m]
end

Algorithm 9.1. A method for sampling an initial population of \( m \) design points over a uniform hyper-rectangle with lower-bound vector \( a \) and upper-bound vector \( b \).

Algorithm 9.2. A method for sampling an initial population of \( m \) design points using a multivariate normal distribution with mean \( \mu \) and covariance \( \Sigma \).

Uniform and normal distributions limit the covered design space to a concentrated region. The Cauchy distribution (figure 9.1) has an unbounded variance and can cover a much broader space. Algorithm 9.3 provides an implementation. Figure 9.2, on the next page, compares example initial populations generated using different methods.

![Cauchy and Normal Distributions](image)

Figure 9.1. A comparison of the normal distribution with standard deviation 1 and the Cauchy distribution with scale 1. Although \( \sigma \) is sometimes used for the scale parameter in the Cauchy distribution, this should not be confused with the standard deviation since the standard deviation of the Cauchy distribution is undefined. The Cauchy distribution is heavy-tailed, allowing it to cover the design space more broadly.

9.2 Genetic Algorithms

Genetic algorithms (algorithm 9.4) borrow inspiration from biological evolution, where fitter individuals are more likely to pass on their genes to the next generation.\(^2\) An individual’s fitness for reproduction is inversely related to the value of the objective function at that point. The design point associated with an individual is represented as a chromosome. At each generation, the chromosomes of the fitter individuals are passed on to the next generation after undergoing the genetic operations of crossover and mutation.

9.2. Genetic Algorithms

Algorithm 9.3. A method for sampling an initial population of \( m \) design points using a Cauchy distribution with location \( \mu \) and scale \( \sigma \) for each dimension. The location and scale are analogous to the mean and standard deviation used in a normal distribution.

Figure 9.2. Initial populations of size 1,000 sampled using a uniform hyperrectangle with \( a = [-2,-2] \), \( b = [2,2] \), a zero-mean normal distribution with diagonal covariance \( \Sigma = I \), and Cauchy distributions centered at the origin with scale \( \sigma = 1 \).

Algorithm 9.4. The genetic algorithm, which takes an objective function \( f \), an initial population, number of iterations \( k_{\text{max}} \), a SelectionMethod \( S \), a CrossoverMethod \( C \), and a MutationMethod \( M \).
9.2.1 Chromosomes

There are several ways to represent chromosomes. The simplest is the binary string chromosome, a representation that is similar to the way DNA is encoded. A random binary string of length $d$ can be generated using `bitrand(d)`. A binary string chromosome is depicted in figure 9.3.

Binary strings are often used due to the ease of expressing crossover and mutation. Unfortunately, the process of decoding a binary string and producing a design point is not always straightforward. Sometimes the binary string might not represent a valid point in the design space. It is often more natural to represent a chromosome using a list of real values. Such real-valued chromosomes are vectors in $\mathbb{R}^d$ that directly correspond to points in the design space.

9.2.2 Initialization

Genetic algorithms start with a random initial population. Binary string chromosomes are typically initialized using random bit strings as seen in algorithm 9.5. Real-valued chromosomes are typically initialized using the methods from the previous section.

```python
rand_population_binary(m, n) = [bitrand(n) for i in 1:m]
```

Algorithm 9.5. A method for sampling random starting populations of $m$ bit-string chromosomes of length $n$.

9.2.3 Selection

Selection is the process of choosing chromosomes to use as parents for the next generation. For a population with $m$ chromosomes, a selection method will produce a list of $m$ parental pairs for the $m$ children of the next generation. The selected pairs may contain duplicates.

3 Instead of a binary representation, DNA contains four nucleobases: adenine, thymine, cytosine, and guanine, which are often abbreviated A, T, C, and G.

4 In some cases, one might use groups, should one wish to combine more than two parents to form a child.
There are several approaches for biasing the selection toward the fittest (algorithm 9.6). In truncation selection (figure 9.4), we sample parents from among the best $k$ chromosomes in the population. In tournament selection (figure 9.5), each parent is the fittest out of $k$ randomly chosen chromosomes of the population. In roulette wheel selection (figure 9.6), also known as fitness proportionate selection, each parent is chosen with a probability proportional to its performance relative to the population. Since we are interested in minimizing an objective function $f$, the fitness of the $i$th individual $x^{(i)}$ is inversely related to $y^{(i)} = f(x^{(i)})$. There are different ways to transform a collection $y^{(1)}, \ldots, y^{(m)}$ into fitnesses. A simple approach is to assign the fitness of individual $i$ according to $\max\{y^{(1)}, \ldots, y^{(m)}\} - y^{(i)}$.

Figure 9.4. Truncation selection with a population size $m = 7$ and sample size $k = 3$. The height of a bar indicates its objective function value whereas its color indicates what individual it corresponds to.

Figure 9.5. Tournament selection with a population size $m = 7$ and a sample size $k = 3$, which is run separately for each parent. The height of a bar indicates its objective function value whereas its color indicates what individual it corresponds to.

Figure 9.6. Roulette wheel selection with a population size $m = 7$, which is run separately for each parent. The approach used causes the individual with the worst objective function value to have a zero likelihood of being selected. The height of a bar indicates its objective function value (left), or its likelihood (right), whereas its color indicates what individual it corresponds to.

© 2019 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.

2022-05-22 00:25:57-07:00, revision 47fd495, comments to bugs@algorithmsbook.com
Algorithm 9.6. Several selection methods for genetic algorithms. Calling `select` with a `SelectionMethod` and a list of objective function values `y` will produce a list of parental pairs.
9.2.4 Crossover

Crossover combines the chromosomes of parents to form children. As with selection, there are several crossover schemes (algorithm 9.7).

- In **single-point crossover** (figure 9.7), the first portion of parent A’s chromosome forms the first portion of the child chromosome, and the latter portion of parent B’s chromosome forms the latter part of the child chromosome. The *crossover point* where the transition occurs is determined uniformly at random.

![Figure 9.7. Single-point crossover.](image)

- In **two-point crossover** (figure 9.8), we use two random crossover points.

![Figure 9.8. Two-point crossover.](image)

- In **uniform crossover** (figure 9.9), each bit has a fifty percent chance of coming from either one of the two parents. This scheme is equivalent to each point having a fifty percent chance of being a crossover point.

![Figure 9.9. Uniform crossover.](image)

The previous crossover methods also work for real-valued chromosomes. We can, however, define an additional crossover routine that interpolates between real values (algorithm 9.8). Here, the real values are linearly interpolated between the parents’ values $x_a$ and $x_b$:

$$
x \leftarrow (1 - \lambda)x_a + \lambda x_b$$  \hspace{1cm} (9.2)

where $\lambda$ is a scalar parameter typically set to one-half.
Algorithm 9.7. Several crossover methods for genetic algorithms. Calling crossover with a `CrossoverMethod` and two parents `a` and `b` will produce a child chromosome that contains a mixture of the parents’ genetic codes. These methods work for both binary string and real-valued chromosomes.

Algorithm 9.8. A crossover method for real-valued chromosomes which performs linear interpolation between the parents.
9.2.5 Mutations

If new chromosomes were produced only through crossover, many traits that were not present in the initial random population could never occur, and the most-fit genes could saturate the population. Mutation allows new traits to spontaneously appear, allowing the genetic algorithm to explore more of the state space. Child chromosomes undergo mutation after crossover.

Each bit in a binary-valued chromosome typically has a small probability of being flipped (Figure 9.10). For a chromosome with \( m \) bits, this mutation rate is typically set to \( 1/m \), yielding an average of one mutation per child chromosome. Mutation for real-valued chromosomes can be implemented using bitwise flips, but it is more common to add zero-mean Gaussian noise. Algorithm 9.9 provides implementations.

![Mutation Example](image)

**Figure 9.10.** Mutation for binary string chromosomes gives each bit a small probability of flipping.

```
abstract type MutationMethod end
struct BitwiseMutation <: MutationMethod
    λ
end
function mutate(M::BitwiseMutation, child)
    return [rand() < M.λ ? !v : v for v in child]
end

struct GaussianMutation <: MutationMethod
    σ
end
function mutate(M::GaussianMutation, child)
    return child + randn(length(child))*M.σ
end
```

Algorithm 9.9. The bitwise mutation method for binary string chromosomes and the Gaussian mutation method for real-valued chromosomes. Here, \( λ \) is the mutation rate, and \( σ \) is the standard deviation.

Figure 9.11 illustrates several generations of a genetic algorithm. Example 9.1 shows how to combine selection, crossover, and mutation strategies discussed in this section.
We will demonstrate using genetic algorithms to optimize a simple function.

```python
import Random: seed!
import LinearAlgebra: norm
seed!(0) # set random seed for reproducible results
f = x->norm(x)
m = 100 # population size
k_max = 10 # number of iterations
population = rand_population_uniform(m, [-3, 3], [3,3])
S = TruncationSelection(10) # select top 10
C = SinglePointCrossover()
M = GaussianMutation(0.5) # small mutation rate
x = genetic_algorithm(f, population, k_max, S, C, M)
@show x

x = [-0.00994906141228906, -0.05198433759424115]
```

Example 9.1. Demonstration of using a genetic algorithm for optimizing a simple function.
9.3 Differential Evolution

Differential evolution (algorithm 9.10) attempts to improve each individual in the population by recombining other individuals in the population according to a simple formula. It is parameterized by a crossover probability \( p \) and a differential weight \( w \). Typically, \( w \) is between 0.4 and 1. For each individual \( x \):

1. Choose three random distinct individuals \( a, b, \) and \( c \).
2. Construct an interim design \( z = a + w \cdot (b - c) \) as shown in figure 9.12.
3. Choose a random dimension \( j \in [1, \ldots, n] \) for optimization in \( n \) dimensions.
4. Construct the candidate individual \( x' \) using binary crossover.

\[
x'_i = \begin{cases} 
  z_i & \text{if } i = j \text{ or with probability } p \\
  x_i & \text{otherwise}
\end{cases} \quad (9.3)
\]

5. Insert the better design between \( x \) and \( x' \) into the next generation.

The algorithm is demonstrated in figure 9.13.

Algorithm 9.10. Differential evolution, which takes an objective function \( f \), a population \( \text{population} \), a number of iterations \( k_{\text{max}} \), a crossover probability \( p \), and a differential weight \( w \). The best individual is returned.

Figure 9.12. Differential evolution takes three individuals \( a, b, \) and \( c \) and combines them to form the candidate individual \( z \).
9.4 Particle Swarm Optimization

Particle swarm optimization introduces momentum to accelerate convergence toward minima. Each individual, or particle, in the population keeps track of its current position, velocity, and the best position it has seen so far (algorithm 9.11). Momentum allows an individual to accumulate speed in a favorable direction, independent of local perturbations.

\begin{verbatim}
mutable struct Particle
    x
    v
    x_best
end
\end{verbatim}

At each iteration, each individual is accelerated toward both the best position it has seen and the best position found thus far by any individual. The acceleration is weighted by a random term, with separate random numbers being generated.

Figure 9.13. Differential evolution with $p = 0.5$ and $w = 0.2$ applied to Ackley’s function, defined in appendix B.1.


Algorithm 9.11. Each particle in particle swarm optimization has a position $x$ and velocity $v$ in design space and keeps track of the best design point found so far, $x_{\text{best}}$. 
for each acceleration. The update equations are:

\[ x^{(i)} \leftarrow x^{(i)} + v^{(i)} \]  
\[ v^{(i)} \leftarrow wv^{(i)} + c_1 r_1 (x_{\text{best}}^{(i)} - x^{(i)}) + c_2 r_2 (x_{\text{best}} - x^{(i)}) \]  

where \( x_{\text{best}} \) is the best location found so far over all particles; \( w, c_1, \) and \( c_2 \) are parameters; and \( r_1 \) and \( r_2 \) are random numbers drawn from \( U(0, 1) \). Algorithm 9.12 provides an implementation. Figure 9.14 shows several iterations of the algorithm.

Algorithm 9.12. Particle swarm optimization, which takes an objective function \( f \), a list of particles \( \text{population} \), a number of iterations \( k_{\text{max}} \), an inertia \( w \), and momentum coefficients \( c_1 \) and \( c_2 \). The default values are those used by R. Eberhart and J. Kennedy, “A New Optimizer Using Particle Swarm Theory,” in *International Symposium on Micro Machine and Human Science*, 1995.

### 9.5 Firefly Algorithm

The firefly algorithm (algorithm 9.13) was inspired by the manner in which fireflies flash their lights to attract mates. In the firefly algorithm, each individual in the population is a firefly and can flash to attract other fireflies. At each iteration, all fireflies are moved toward all more attractive fireflies. A firefly \( a \) is moved toward a firefly \( b \) with greater attraction according to

\[ a \leftarrow a + \beta I(||b - a||) (b - a) + \alpha \epsilon \]  

A common strategy is to allow the inertia \( w \) to decay over time.

Algorithm 9.13. Particle swarm optimization, which takes an objective function \( f \), a list of particles \( \text{population} \), a number of iterations \( k_{\text{max}} \), an inertia \( w \), and momentum coefficients \( c_1 \) and \( c_2 \). The default values are those used by R. Eberhart and J. Kennedy, “A New Optimizer Using Particle Swarm Theory,” in *International Symposium on Micro Machine and Human Science*, 1995.

9.5 Firefly Algorithm

The firefly algorithm (algorithm 9.13) was inspired by the manner in which fireflies flash their lights to attract mates. In the firefly algorithm, each individual in the population is a firefly and can flash to attract other fireflies. At each iteration, all fireflies are moved toward all more attractive fireflies. A firefly \( a \) is moved toward a firefly \( b \) with greater attraction according to

\[ a \leftarrow a + \beta I(||b - a||) (b - a) + \alpha \epsilon \]
where \( I \) is the intensity of the attraction and \( \beta \) is the source intensity. A random walk component is included as well, where \( \epsilon \) is drawn from a zero-mean, unit covariance multivariate Gaussian, and \( \alpha \) scales the step size. The resulting update is a random walk biased toward brighter fireflies.\(^9\)

The intensity \( I \) decreases as the distance \( r \) between the two fireflies increases and is defined to be 1 when \( r = 0 \). One approach is to model the intensity as a point source radiating into space, in which case the intensity decreases according to the inverse square law

\[
I(r) = \frac{1}{r^2} \quad (9.7)
\]

Alternatively, if a source is suspended in a medium that absorbs light, the intensity will decrease according to an exponential decay

\[
I(r) = e^{-\gamma r} \quad (9.8)
\]

where \( \gamma \) is a the light absorption coefficient.\(^10\)

We generally want to avoid equation (9.7) in practice due to the singularity at \( r = 0 \). A combination of the inverse square law and absorption can be approximated with a Gaussian brightness drop-off:

\[
I(r) = e^{-\gamma r^2} \quad (9.9)
\]
A firefly’s attraction is proportional to its performance. Attraction affects only whether one fly is attracted to another fly, whereas intensity affects how much the less attractive fly moves. Figure 9.15 shows a few iterations of the algorithm.

Algorithm 9.13. The firefly algorithm, which takes an objective function \( f \), a population of flies consisting of design points, a number of iterations \( k_{\text{max}} \), a source intensity \( \beta \), a random walk step size \( \alpha \), and an intensity function \( I \). The best design point is returned.

\[
\begin{align*}
\text{using Distributions} \\
\text{function firefly}(f, \text{population}, k_{\text{max}}; \\
\quad \beta=1, \alpha=0.1, \text{brightness}=r\to\exp(-r^2)) \\
\quad m = \text{length}(\text{population}[1]) \\
\quad N = \text{MvNormal}(\text{Matrix}(1.0I, m, m)) \\
\quad \text{for } k \text{ in } 1:k_{\text{max}} \\
\quad \quad \text{for } a \text{ in } \text{population}, b \text{ in } \text{population} \\
\quad \quad \quad \text{if } f(b) < f(a) \\
\quad \quad \quad \quad r = \text{norm}(b-a) \\
\quad \quad \quad \quad a[:\ ] += \beta*\text{brightness}(r)*(b-a) + \alpha*\text{rand}(N) \\
\quad \quad \text{end} \\
\quad \text{end} \\
\quad \text{return } \text{population}[\text{argmin}([f(x) \text{ for } x \text{ in } \text{population}])] \\
\end{align*}
\]

Figure 9.15. Firefly search with \( \alpha = 0.5, \beta = 1 \), and \( \gamma = 0.1 \) applied to the Branin function (appendix B.3).

9.6 Cuckoo Search

Cuckoo search (algorithm 9.14) is another nature-inspired algorithm named after the cuckoo bird, which engages in a form of brood parasitism.\(^{11}\) Cuckoos lay their eggs in the nests of other birds, often birds of other species. When this occurs, the host bird may detect the invasive egg and then destroy it or establish a new

nest somewhere else. However, there is also a chance that the egg is accepted and raised by the host bird.  

In cuckoo search, each nest represents a design point. New design points can be produced using Lévy flights from nests, which are random walks with step-lengths from a heavy-tailed distribution. A new design point can replace a nest if it has a better objective function value, which is analogous to cuckoo eggs replacing the eggs of birds of other species.

The core rules are:

1. A cuckoo will lay an egg in a randomly chosen nest.
2. The best nests with the best eggs will survive to the next generation.
3. Cuckoo eggs have a chance of being discovered by the host bird, in which case the eggs are destroyed.

Cuckoo search relies on random flights to establish new nest locations. These flights start from an existing nest and then move randomly to a new location. While we might be tempted to use a uniform or Gaussian distribution for the walk, these restrict the search to a relatively concentrated region. Instead, cuckoo search uses a Cauchy distribution, which has a heavier tail. In addition, the Cauchy distribution has been shown to be more representative of the movements of other animals in the wild. Figure 9.16 shows a few iterations of cuckoo search.

Other nature-inspired algorithms include the artificial bee colony, the gray wolf optimizer, the bat algorithm, glowworm swarm optimization, intelligent water drops, and harmony search. There has been some criticism of the proliferation of methods that make analogies to nature without fundamentally contributing novel methods and understanding.

9.7 Hybrid Methods

Many population methods perform well in global search, being able to avoid local minima and finding the best regions of the design space. Unfortunately, these methods do not perform as well in local search in comparison to descent methods. Several hybrid methods have been developed to extend population methods with descent-based features to improve their performance in local search. There are two general approaches to combining population methods with local search techniques:
Algorithm 9.14. Cuckoo search, which takes an objective function $f$, an initial set of nests $population$, a number of iterations $k_{max}$, percent of nests to abandon $p_{a}$, and flight distribution $C$. The flight distribution is typically a centered Cauchy distribution.

```
using Distributions
mutable struct Nest
    x # position
    y # value, f(x)
end

function cuckoo_search(f, population, k_max;
    p_a=0.1, C=Cauchy(0,1))
    m, n = length(population), length(population[1].x)
    a = round(Int, m*p_a)
    for k in 1:k_max
        i, j = rand(1:m), rand(1:m)
        x = population[j].x + [rand(C) for k in 1:n]
        y = f(x)
        if y < population[i].y
            population[i].x[:] = x
            population[i].y = y
        end
    end
    p = sortperm(population, by=nest->nest.y, rev=true)
    for i in 1:a
        j = rand(1:m-a)+a
        population[p[i]] = Nest(population[p[j]].x +
            [rand(C) for k in 1:n],
            f(population[p[j]].x))
    end
end
```

Figure 9.16. Cuckoo search applied to the Branin function (appendix B.3).
In **Lamarckian learning**, the population method is extended with a local search method that locally improves each individual. The original individual and its objective function value are replaced by the individual’s optimized counterpart and its objective function value.

In **Baldwinian learning**, the same local search method is applied to each individual, but the results are used only to update the individual’s perceived objective function value. Individuals are not replaced but are merely associated with optimized objective function values, which are not the same as their actual objective function value. Baldwinian learning can help prevent premature convergence.

The difference between these approaches is illustrated in example 9.2.

Consider optimizing $f(x) = -e^{-x^2} - 2e^{-(x-3)^2}$ using a population of individuals initialized near $x = 0$.

A Lamarckian local search update applied to this population would move the individuals toward the local minimum, reducing the chance that future individuals escape and find the global optimum near $x = 3$. A Baldwinian approach will compute the same update but leaves the original designs unchanged. The selection step will value each design according to its value from a local search.
9.8 Summary

- Population methods use a collection of individuals in the design space to guide progression toward an optimum.

- Genetic algorithms leverage selection, crossover, and mutations to produce better subsequent generations.

- Differential evolution, particle swarm optimization, the firefly algorithm, and cuckoo search include rules and mechanisms for attracting design points to the best individuals in the population while maintaining suitable state space exploration.

- Population methods can be extended with local search approaches to improve convergence.

9.9 Exercises

Exercise 9.1. What is the motivation behind the selection operation in genetic algorithms?

Exercise 9.2. Why does mutation play such a fundamental role in genetic algorithms? How would we choose the mutation rate if we suspect there is a better optimal solution?

Exercise 9.3. If we observe that particle swarm optimization results in fast convergence to a nonglobal minimum, how might we change the parameters of the algorithm?