Week #13: Stochastic Optimization Methods

Notes by: Francisco Richter

May 15, 2025

Overview

Stochastic optimization encompasses a family of algorithms designed to find optimal solutions in the presence of randomness or uncertainty. Unlike deterministic optimization methods that follow fixed trajectories through the solution space, stochastic optimization methods incorporate randomness to explore the search space more effectively, particularly when dealing with complex, non-convex, or noisy objective functions.

These methods have gained significant importance across various domains, from engineering and finance to machine learning and artificial intelligence. Their ability to escape local optima and navigate rugged fitness landscapes makes them particularly valuable for problems where traditional gradient-based methods struggle. Additionally, many stochastic optimization techniques draw inspiration from natural phenomena, such as evolutionary processes, physical annealing, or swarm intelligence, providing intuitive frameworks for algorithm design.

In this lecture, we will explore several key stochastic optimization methods: Simulated Annealing, Genetic Algorithms, Differential Evolution, Particle Swarm Optimization, and Greedy Algorithms. For each method, we will examine the underlying principles, mathematical formulations, implementation considerations, and practical applications. We will also discuss the strengths and limitations of each approach, providing insights into when and how to apply these methods effectively.

---

1 Introduction to Stochastic Optimization

Optimization problems are ubiquitous across science, engineering, and economics. They involve finding the values of variables that minimize or maximize an objective function, often subject to constraints. Formally, an optimization problem can be expressed as:

$$\begin{aligned} \text{Minimize (or Maximize)} \quad & f(x) && (1)\\ \text{subject to} \quad & g_i(x) \le 0, \quad i = 1, 2, \ldots, m && (2)\\ & h_j(x) = 0, \quad j = 1, 2, \ldots, p && (3)\\ & x \in \mathcal{X} \subseteq \mathbb{R}^n && (4) \end{aligned}$$

where $f : \mathbb{R}^n \to \mathbb{R}$ is the objective function, g_i and h_j are inequality and equality constraint functions, respectively, and $\mathcal{X}$ is the feasible set.

Definition. Stochastic Optimization

Stochastic optimization refers to the minimization or maximization of an objective function when randomness is present, either in the objective function evaluation, the constraints, or the optimization algorithm itself. These methods typically use probabilistic elements to guide the search process.

Intuition: Why use randomness?

Imagine trying to find the highest peak in a mountain range on a foggy day. Deterministic methods are like always walking uphill—you'll reach a peak, but it might just be a local maximum. Stochastic methods are like occasionally taking random directions, even downhill, which might lead you to discover higher peaks elsewhere. The randomness helps escape local optima and explore the search space more thoroughly.

Stochastic optimization methods are particularly valuable when the objective function is non-convex with multiple local optima, when derivatives are unavailable or expensive to compute, when variables are discrete or combinatorial, when evaluations are noisy, or when the search space is extremely large or high-dimensional.

1.1 Classification of Stochastic Optimization Methods

Stochastic optimization methods can be classified along four axes: whether they maintain a single solution (as in Simulated Annealing) or a population of solutions (as in Genetic Algorithms, PSO, DE); whether they draw inspiration from natural processes (evolution, annealing, swarms) or from pure mathematical constructs; whether they use memory of past states (Tabu Search) or are essentially memoryless; and whether they emphasize global exploration or local exploitation.

Example 1.1 (Real-world applications)**.** *Stochastic optimization methods have been successfully applied to:*

- *Portfolio optimization in finance*
- *Neural network training in machine learning*
- *Vehicle routing and scheduling in logistics*
- *Protein folding in computational biology*
- *Circuit design in electrical engineering*
- *Resource allocation in operations research*

---

2 Simulated Annealing

Simulated Annealing (SA) is a probabilistic technique inspired by the physical process of annealing in metallurgy, where a material is heated and then slowly cooled to reduce defects and increase the size of crystals in the material.

2.1 Principles and Mathematical Formulation

The key insight of SA is to occasionally accept worse solutions during the optimization process, with the probability of accepting such moves decreasing over time. This strategy allows the algorithm to escape local optima, especially in the early stages of the search.

Definition. Simulated Annealing

Simulated Annealing is a stochastic optimization method that iteratively modifies a current solution by applying a neighborhood operator and accepting or rejecting the new solution based on both its quality and a temperature parameter that decreases over time.

Given a current solution x and a candidate neighbor solution x', the probability of accepting x' is:

$$P(\text{accept } x') = \begin{cases} 1, & \text{if } f(x') \leq f(x) \\ \exp\left(-\frac{f(x')-f(x)}{T}\right), & \text{if } f(x') > f(x) \end{cases} \tag{5}$$

where T is the temperature parameter that decreases according to a cooling schedule.

Intuition: Temperature and acceptance probability

The temperature T controls the exploration-exploitation trade-off. At high temperatures, the algorithm freely explores the search space by frequently accepting worse solutions. As the temperature decreases, the algorithm becomes more selective, gradually focusing on exploitation (improvement) rather than exploration. This mimics the physical annealing process where atoms initially move freely at high temperatures but gradually settle into low-energy configurations as the material cools.

2.2 Algorithm and Implementation

Algorithm 1 Simulated Annealing

```
Require: Initial solution $x_0$, initial temperature $T_0$, cooling rate $\alpha$, minimum temperature $T_{min}$, iterations
    per temperature $n_{iter}$
Ensure: Approximately optimal solution $x^*$
1:  $x \leftarrow x_0$                                  ▷ Current solution
2:  $x^* \leftarrow x_0$                                ▷ Best solution found
3:  $T \leftarrow T_0$                                  ▷ Current temperature
4:  while $T > T_{min}$ do
5:      for $i = 1$ to $n_{iter}$ do
6:          Generate neighbor $x'$ from $x$
7:          $\Delta E \leftarrow f(x') - f(x)$
8:          if $\Delta E \leq 0$ then                   ▷ Better solution
9:              $x \leftarrow x'$
10:             if $f(x') < f(x^*)$ then
11:                 $x^* \leftarrow x'$
12:             end if
13:         else                                        ▷ Worse solution
14:             Generate random number $r \in [0, 1]$
15:             if $r < \exp(-\Delta E/T)$ then
16:                 $x \leftarrow x'$
17:             end if
18:         end if
19:     end for
20:     $T \leftarrow \alpha \cdot T$                   ▷ Reduce temperature
21: end while
22: return $x^*$
```

2.3 Cooling Schedules

The cooling schedule determines how the temperature decreases over time and significantly impacts the algorithm's performance:

- **Geometric cooling**: $T_{k+1} = \alpha \cdot T_k$ where $\alpha \in (0, 1)$ is the cooling rate (typically 0.8 to 0.99)
- **Linear cooling**: $T_k = T_0 - k \cdot \beta$ where $\beta > 0$ is the cooling step
- **Logarithmic cooling**: $T_k = \frac{T_0}{\log(k+1)}$ (theoretically guarantees convergence to global optimum but is impractically slow)
- **Adaptive cooling**: Adjusts the cooling rate based on the algorithm's progress

2.4 Theoretical Properties

Simulated Annealing has several important theoretical properties:

Theorem 2.1 (Convergence of Simulated Annealing)**.** *If the temperature decreases logarithmically, i.e.,* $T_k = \frac{c}{\log(k+1)}$ *where c is a problem-dependent constant, then Simulated Annealing converges in probability to the global optimum as $k \to \infty$.*

Proof Sketch. The proof relies on modeling SA as a non-homogeneous Markov chain and showing that with the logarithmic cooling schedule, the stationary distribution of the chain concentrates on the global optima as $k \to \infty$. The key insight is that the logarithmic cooling is slow enough to allow the system to reach equilibrium at each temperature. □

2.5 Practical Considerations

When implementing Simulated Annealing, several practical considerations should be taken into account:

- **Initial temperature**: Should be high enough to allow most transitions to be accepted initially (typically set so that the initial acceptance probability is around 0.8)
- **Neighborhood structure**: Defines how new candidate solutions are generated and significantly impacts the algorithm's performance
- **Cooling schedule**: Faster cooling leads to quicker convergence but increases the risk of getting trapped in local optima
- **Stopping criteria**: Can be based on minimum temperature, maximum iterations, or lack of improvement
- **Reheating strategies**: Occasionally increasing the temperature can help escape local optima

Example 2.1 (Traveling Salesman Problem with Simulated Annealing)**.** *For the Traveling Salesman Problem (TSP), we can implement SA as follows:*

- ***Solution representation****: Permutation of cities*
- ***Objective function****: Total distance of the tour*
- ***Neighborhood operator****: 2-opt move (swap two edges)*
- ***Initial temperature****: Set so that a 5% worse solution has a 50% chance of being accepted*
- ***Cooling schedule****: Geometric with* $\alpha = 0.95$

SA has been shown to find near-optimal solutions for TSP instances with thousands of cities, significantly outperforming greedy heuristics.

Practice Problems

1. Implement Simulated Annealing for minimizing the function $f(x) = x^2 \sin(x)$ over the interval $[-10, 10]$. Experiment with different cooling schedules and analyze their impact on the solution quality.
2. Consider a binary optimization problem where solutions are bit strings of length n. Define an appropriate neighborhood structure and apply Simulated Annealing to maximize the number of alternating bits (1-0-1-0...).
3. Prove that if the temperature in Simulated Annealing is kept constant (i.e., $T_k = T$ for all k), the algorithm reduces to a random walk for $T \to \infty$ and to a greedy hill-climbing algorithm for $T \to 0$.

Solution:

For problem 3: When $T \to \infty$, the acceptance probability $\exp(-\Delta E/T) \to \exp(0) = 1$ for any ΔE. This means all moves are accepted regardless of whether they improve the solution, resulting in a random walk.

When $T \to 0$, for $\Delta E > 0$ (worse solutions), the acceptance probability $\exp(-\Delta E/T) \to \exp(-\infty) = 0$, meaning worse solutions are never accepted. For $\Delta E \leq 0$ (better or equal solutions), the acceptance probability is 1. This is exactly the behavior of a greedy hill-climbing algorithm that only accepts improving or equal-quality moves.

---

3 Genetic Algorithms

Genetic Algorithms (GAs) are population-based optimization methods inspired by the process of natural selection. They operate by evolving a population of candidate solutions through mechanisms analogous to biological evolution: selection, crossover, and mutation.

3.1 Principles and Mathematical Formulation

The fundamental principle of GAs is that better solutions (individuals) have a higher chance of survival and reproduction, passing their beneficial characteristics to future generations.

Definition. Genetic Algorithm

A Genetic Algorithm is a search heuristic that mimics the process of natural selection by maintaining a population of candidate solutions that evolve over generations through selection, crossover, and mutation operators.

Let $P^{(t)} = \{x_1^{(t)}, x_2^{(t)}, \ldots, x_N^{(t)}\}$ represent the population at generation t, where each $x_i^{(t)}$ is an individual (candidate solution). The fitness function $f(x_i)$ evaluates the quality of each individual.

Intuition: Biological analogy

In GAs, candidate solutions are like organisms in a population, with their "DNA" encoding potential solutions to the problem. The fitness function acts like natural selection, favoring better solutions. Crossover is analogous to genetic recombination during reproduction, while mutation introduces random variations, similar to genetic mutations in nature. Over generations, the population evolves toward better solutions, just as species adapt to their environments.

3.2 Algorithm and Implementation

Algorithm 2 Genetic Algorithm

```
Require: Population size $N$, maximum generations $G_{max}$, crossover probability $p_c$, mutation probability $p_m$
Ensure: Best solution found
 1: Initialize population $P^{(0)} = \{x_1^{(0)}, x_2^{(0)}, \dots, x_N^{(0)}\}$ randomly
 2: Evaluate fitness $f(x_i^{(0)})$ for each individual $i = 1, 2, \dots, N$
 3: for $t = 0$ to $G_{max} - 1$ do
 4:     $P' \leftarrow \emptyset$                                        ▷ New population
 5:     while $|P'| < N$ do
 6:         Select parents $x_a^{(t)}$ and $x_b^{(t)}$ from $P^{(t)}$ based on fitness
 7:         Generate random number $r \in [0, 1]$
 8:         if $r < p_c$ then                                         ▷ Apply crossover
 9:             $(y_1, y_2) \leftarrow$ Crossover$(x_a^{(t)}, x_b^{(t)})$
10:         else                                                    ▷ No crossover
11:             $(y_1, y_2) \leftarrow (x_a^{(t)}, x_b^{(t)})$
12:         end if
13:         Apply mutation to $y_1$ and $y_2$ with probability $p_m$
14:         Add $y_1$ and $y_2$ to $P'$
15:     end while
16:     $P^{(t+1)} \leftarrow P'$
17:     Evaluate fitness $f(x_i^{(t+1)})$ for each individual $i = 1, 2, \dots, N$
18: end for
19: return best individual in $P^{(G_{max})}$
```

3.3 Key Components of Genetic Algorithms

3.3.1 Solution Representation

The choice of representation (encoding) significantly impacts the GA's performance:

- **Binary representation**: Solutions encoded as bit strings (0s and 1s)
- **Integer representation**: Solutions encoded as sequences of integers
- **Real-valued representation**: Solutions encoded as vectors of real numbers
- **Permutation representation**: Solutions encoded as permutations (e.g., for TSP)
- **Tree representation**: Solutions encoded as tree structures (e.g., for genetic programming)

3.3.2 Selection Methods

Selection methods determine which individuals will be chosen as parents for reproduction:

- **Roulette wheel selection**: Probability of selection proportional to fitness

$$P(x_i) = \frac{f(x_i)}{\sum_{j=1}^{N} f(x_j)}$$

- **Tournament selection**: Randomly select k individuals and choose the best one

- **Rank selection**: Probability of selection based on the rank of individuals
- **Elitism**: Automatically carry the best individuals to the next generation

3.3.3 Crossover Operators

Crossover combines genetic material from two parents to create offspring:

- **Single-point crossover**: Choose a random point and swap the segments
- **Two-point crossover**: Choose two random points and swap the middle segment
- **Uniform crossover**: Each bit/gene has a fixed probability of being swapped
- **Arithmetic crossover**: For real-valued representations,

  $$y_1 = \alpha x_a + (1-\alpha)x_b, \quad y_2 = (1-\alpha)x_a + \alpha x_b,$$

  where $\alpha \in [0, 1]$.

3.3.4 Mutation Operators

Mutation introduces random changes to maintain diversity:

- **Bit flip**: For binary representations, flip bits with probability p_m
- **Gaussian mutation**: For real-valued representations, add Gaussian noise,

  $$y_i = x_i + \mathcal{N}(0, \sigma^2)$$

- **Swap mutation**: For permutation representations, swap two elements
- **Inversion mutation**: Reverse the order of a randomly chosen subsequence

3.4 Theoretical Foundations

Theorem 3.1 (Schema Theorem (Holland, 1975))**.** *Short, low-order schemata (patterns) with above-average fitness increase exponentially in successive generations. Specifically, the expected number of instances of schema H in generation $t+1$ is:*

$$E[m(H, t+1)] \geq m(H, t)\,\frac{f(H)}{\bar{f}}\,\Big(1 - p_c\,\frac{\delta(H)}{l-1} - p_m\,o(H)\Big),$$

where $m(H, t)$ is the count of H in generation t, $f(H)$ its average fitness, $\bar{f}$ the population average, $\delta(H)$ the defining length, l the chromosome length, and $o(H)$ the schema order.

Intuition: Building Block Hypothesis

The Schema Theorem supports the Building Block Hypothesis, which suggests that GAs work by identifying and recombining "building blocks" (short, high-fitness schemata) to form increasingly better solutions. This explains why GAs can effectively explore complex search spaces by simultaneously evaluating many schemata in parallel.

3.5 Advanced GA Variants

Several advanced variants of GAs have been developed to address specific challenges:

- **Adaptive GAs**: Dynamically adjust parameters like crossover and mutation rates
- **Parallel GAs**: Maintain multiple subpopulations that occasionally exchange individuals
- **Hybrid GAs**: Combine GAs with local search methods (memetic algorithms)
- **Multi-objective GAs**: Handle problems with multiple competing objectives (e.g., NSGA-II)
- **Genetic Programming**: Evolve computer programs rather than parameter values

Example 3.1 (Function Optimization with GA)**.** *Consider optimizing the function* $f(x, y) = \sin(x)\,\cos(y)\,e^{-x^2-y^2}$ *over* $[-2, 2] \times [-2, 2]$.

We can implement a GA with:

- ***Representation**: Real-valued vectors* (x, y)
- ***Population size**: 50 individuals*
- ***Selection**: Tournament selection, size 3*
- ***Crossover**: Arithmetic crossover,* $\alpha = 0.5$
- ***Mutation**: Gaussian,* $\sigma = 0.1$
- ***Termination**: 100 generations or no-improve for 20 generations*

The GA consistently finds solutions very close to the global optimum at $(0, 0)$ *with value 1.0.*

Practice Problems

1. Implement a GA to solve the 0-1 Knapsack Problem with n items, each with weight w_i and value v_i, and a knapsack capacity W. Use binary representation where each bit indicates whether an item is included.
2. Compare the performance of different selection methods (roulette wheel, tournament, rank) on a test function of your choice. Analyze how each method affects population diversity and convergence speed.
3. Derive the expected number of function evaluations required for a GA to find the global optimum of the OneMax problem (maximizing the number of 1s in a bit string) as a function of the string length n.

---

4 Differential Evolution

Differential Evolution (DE) is a population-based optimization algorithm particularly effective for continuous optimization problems. Developed by Storn and Price in 1997, DE uses vector differences for perturbing the population vectors, making it self-adapting to the fitness landscape.

4.1 Principles and Mathematical Formulation

The key idea in DE is to use vector differences between randomly selected population members to generate new candidate solutions. This approach automatically adapts the step size and direction of the search based on the population distribution.

Definition. Differential Evolution

Differential Evolution is a population-based optimization algorithm that creates new candidate solutions by adding weighted differences between population vectors to another vector, followed by crossover with the current solution and selection based on fitness.

Let $P^{(t)} = \{\mathbf{x}_1^{(t)}, \ldots, \mathbf{x}_N^{(t)}\}$ be the population at generation t, each $\mathbf{x}_i^{(t)} \in \mathbb{R}^D$.

Intuition: Vector differences

The brilliance of DE lies in using vector differences to guide the search. When the population is widely spread, the differences are large, allowing for exploration. As the population converges toward an optimum, the differences naturally become smaller, transitioning to exploitation. This self-adapting behavior makes DE particularly effective for a wide range of problems without requiring extensive parameter tuning.

4.2 Algorithm and Implementation

Algorithm 3 Differential Evolution

```
Require: Population size $N$, scaling factor $F$, crossover rate $CR$, maximum generations $G_{max}$
Ensure: Best solution found
 1: Initialize $P^{(0)} = \{\mathbf{x}_1^{(0)}, \ldots, \mathbf{x}_N^{(0)}\}$ randomly
 2: Evaluate $f(\mathbf{x}_i^{(0)})$ for $i = 1, \ldots, N$
 3: for $t = 0$ to $G_{max} - 1$ do
 4:     for $i = 1$ to $N$ do
 5:         Select distinct $r_1, r_2, r_3 \neq i$
 6:         $\mathbf{v}_i = \mathbf{x}_{r_1} + F(\mathbf{x}_{r_2} - \mathbf{x}_{r_3})$
 7:         Select random $j_{rand} \in \{1, \ldots, D\}$
 8:         for $j = 1$ to $D$ do
 9:             $u_{i,j} = \begin{cases} v_{i,j}, & \text{if } r \leq CR \text{ or } j = j_{rand}, \\ x_{i,j}, & \text{otherwise.} \end{cases}$
10:         end for
11:         if $f(\mathbf{u}_i) \leq f(\mathbf{x}_i)$ then
12:             $\mathbf{x}_i \leftarrow \mathbf{u}_i$
13:         end if
14:     end for
15: end for
16: return best in $P^{(G_{max})}$
```

4.3 DE Variants and Strategies

DE variants use the notation DE/x/y/z:

- DE/rand/1/bin: $\mathbf{v} = \mathbf{x}_{r_1} + F(\mathbf{x}_{r_2} - \mathbf{x}_{r_3})$

- DE/best/1/bin: $\mathbf{v} = \mathbf{x}_{best} + F(\mathbf{x}_{r_1} - \mathbf{x}_{r_2})$
- DE/rand/2/bin: two difference vectors
- DE/current-to-best/1/bin: comb. current and best

4.4 Parameter Settings and Control

Typical: $N \approx 5$–$10\,D$, $F \in [0.4, 1.0]$, $CR \in [0.1, 0.9]$. Adaptive schemes (jDE, JADE, SaDE, SHADE) evolve or adjust F, CR on-line.

4.5 Theoretical Analysis

Theorem 4.1 (Convergence of DE)**.** *Under mild conditions, DE converges in probability to the global optimum as generations* $\to \infty$*.*

Theorem 4.2 (Invariance Properties)**.** *DE is invariant to rotation, translation, and scaling of the search space.*

4.6 Comparison with Other Evolutionary Algorithms

DE is simple, self-adaptive, efficient (few evals), robust across problems, and naturally parallelizable.

Example 4.1 (Constrained Optimization with DE)**.** *Minimize* $(x-2)^2 + (y-1)^2$ *subject to* $x - 2y + 1 \le 0$, $x^2 + y^2 \le 25$ *via penalty:*

$$F(x, y) = f(x, y) + \lambda \max(0, g_1)^2 + \lambda \max(0, g_2)^2,$$

with DE/rand/1/bin, $N = 30$, $F = 0.8$, $CR = 0.9$, $\lambda = 100$*. Finds* $(0.82, 0.91)$, $f = 2.59$.

Practice Problems

1. Implement DE/rand/1/bin vs. DE/best/1/bin on the Rosenbrock function; compare convergence.
2. Analyze impact of (F, CR) via contour plot of solution quality.
3. Derive expected variance change in one-D DE/rand/1/bin.

---

5 Particle Swarm Optimization

Particle Swarm Optimization (PSO) is a population-based stochastic optimization technique inspired by the social behavior of bird flocking or fish schooling. Developed by Kennedy and Eberhart in 1995, PSO simulates a swarm of particles moving through the search space, with each particle adjusting its trajectory based on its own experience and the experience of its neighbors.

5.1 Principles and Mathematical Formulation

Each particle i has position $\mathbf{x}_i$ and velocity $\mathbf{v}_i$. They update by

$$\mathbf{v}_i \leftarrow w\,\mathbf{v}_i + c_1\,\mathbf{r}_1 \odot (\mathbf{p}_i - \mathbf{x}_i) + c_2\,\mathbf{r}_2 \odot (\mathbf{g} - \mathbf{x}_i), \quad \mathbf{x}_i \leftarrow \mathbf{x}_i + \mathbf{v}_i,$$

where $\mathbf{p}_i$ is its personal best and $\mathbf{g}$ the global (or neighborhood) best.

Definition. Particle Swarm Optimization

PSO optimizes by iteratively improving a swarm of candidate solutions via velocity and position updates combining personal and social learning.

Intuition: Social learning

Particles mimic birds in a flock: each adjusts based on its own success and that of the group, balancing exploration and exploitation.

5.2 Algorithm and Implementation

Algorithm 4 Particle Swarm Optimization

```
Require: Swarm size $N$, inertia $w$, cognitive $c_1$, social $c_2$, iterations $T_{max}$
Ensure: Best solution $\mathbf{g}$
 1: Initialize $\mathbf{x}_i, \mathbf{v}_i$ randomly for $i = 1..N$
 2: Evaluate $f(\mathbf{x}_i)$, set $\mathbf{p}_i = \mathbf{x}_i$, $\mathbf{g} = \arg\min f(\mathbf{p}_i)$
 3: for $t = 1$ to $T_{max}$ do
 4:     for $i = 1$ to $N$ do
 5:         Draw $\mathbf{r}_1, \mathbf{r}_2 \in [0,1]^D$
 6:         Update $\mathbf{v}_i, \mathbf{x}_i$ as above
 7:         Evaluate $f(\mathbf{x}_i)$
 8:         if $f(\mathbf{x}_i) < f(\mathbf{p}_i)$ then
 9:             $\mathbf{p}_i \leftarrow \mathbf{x}_i$
10:             if $f(\mathbf{p}_i) < f(\mathbf{g})$ then
11:                 $\mathbf{g} \leftarrow \mathbf{p}_i$
12:             end if
13:         end if
14:     end for
15: end for
16: return $\mathbf{g}$
```

5.3 PSO Variants and Topologies

Variants: global-best, local-best (ring), unified, comprehensive learning, bare-bones. Topologies: star, ring, von Neumann, random, adaptive.

5.4 Parameter Settings and Control

Typical: $w \in [0.4, 0.9]$, $c_1 = c_2 \approx 2.0$, $N = 20$–50, velocity clamping. Controls: linear decreasing w, constriction factor, adaptive/self-adaptive.

5.5 Theoretical Analysis

Theorem 5.1 (Convergence of PSO)**.** *If* $0 < w < 1$ *and* $0 < c_1 + c_2 < 4(1 + w)$*, the swarm converges.*

Theorem 5.2 (Order-1 Stability)**.** *Order-1 stability requires* $w^2 - 1 + c < 0$ *and* $w - 1 + c > 0$*, with* $c = (c_1 + c_2)/2$.

5.6 Comparison with Other Swarm Intelligence Algorithms

PSO is simple, efficient (few evals), memory-keeping (personal/global best), flexible, and parallelizable.

Example 5.1 (Multi-modal Function Optimization with PSO)**.** *Optimize the Rastrigin function*

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

global min at $\mathbf{0}$ *with* $f = 0$*. A 40-particle von Neumann PSO with* w *decaying 0.9→0.4 and* $c_1 = c_2 = 2.0$ *finds near-zero in 1000 iterations.*

Practice Problems

1. Implement global-best vs. local-best PSO on Ackley; compare escape from local minima.
2. Study effect of topology (star, ring, von Neumann) on exploration/exploitation.
3. Derive order-2 stability conditions in w, c_1, c_2.

---

6 Greedy Algorithms in Stochastic Optimization

Greedy algorithms make locally optimal choices at each step with the hope of finding a global optimum. While traditional greedy algorithms are deterministic, stochastic variants incorporate randomness to improve performance on complex problems.

6.1 Principles and Mathematical Formulation

Stochastic greedy algorithms combine the efficiency of greedy approaches with the exploration capabilities of randomized methods.

Definition. Stochastic Greedy Algorithm

A stochastic greedy algorithm is an optimization method that makes locally optimal choices with some probability distribution, rather than deterministically selecting the best option at each step.

Intuition: Why add randomness to greedy algorithms?

Traditional greedy algorithms can get trapped in local optima because they always make the seemingly best choice at each step. Adding randomness allows occasional exploration of seemingly suboptimal paths that might lead to better solutions in the long run. It's like occasionally taking a detour while driving—it might seem inefficient initially but could help you avoid traffic jams and reach your destination faster.

6.2 Types of Stochastic Greedy Algorithms

6.2.1 Randomized Greedy Algorithms

These algorithms introduce randomness in the selection process:

- **ϵ-greedy**: With probability ϵ, make a random choice; otherwise, make the greedy choice

- **Softmax selection**: Select options with probability proportional to their quality,

$$P(i) = \frac{e^{Q(i)/\tau}}{\sum_j e^{Q(j)/\tau}},$$

  where τ is a temperature.

- **Upper Confidence Bound (UCB)**: Select based on quality plus uncertainty,

$$\text{UCB}(i) = Q(i) + c\sqrt{\frac{\ln n}{n_i}}.$$

6.2.2 Stochastic Local Search

These algorithms use greedy principles but allow occasional non-improving moves:

- **Stochastic Hill Climbing**: Accept improving moves with probability 1 and lateral moves with probability p.
- **Greedy Randomized Adaptive Search Procedure (GRASP)**: Combine greedy construction with local search.
- **Iterated Local Search**: Apply perturbations to escape local optima, then greedy local search.

6.2.3 Stochastic Beam Search

Maintain multiple candidates and apply stochastic selection:

- **Stochastic Beam Search**: Keep k solutions, probabilistically select successors.
- **Genetic Beam Search**: Combine beam search with genetic operators.

6.3 Algorithm and Implementation

Algorithm 5 Stochastic Greedy Algorithm (Generic Template)

```
Require: Problem instance, randomization parameter $\epsilon$
Ensure: Approximately optimal solution
 1: Initialize solution $S \leftarrow \emptyset$
 2: while $S$ is incomplete do
 3:     Generate candidate set $C$
 4:     Evaluate quality $Q(c)$ for $c \in C$
 5:     Draw $r \sim U(0,1)$
 6:     if $r < \epsilon$ then                          ▷ Explore
 7:         Select random $e \in C$
 8:     else                                            ▷ Exploit
 9:         $e \leftarrow \arg\max_{c \in C} Q(c)$
10:     end if
11:     $S \leftarrow S \cup \{e\}$
12: end while
13: return $S$
```

6.4 Applications of Stochastic Greedy Algorithms

Stochastic greedy algorithms have been successfully applied to:

- Combinatorial optimization: set covering, max-SAT, TSP
- Machine learning: feature selection, decision-tree construction, pruning
- Reinforcement learning: multi-armed bandits
- Scheduling: job assignment, resource allocation
- Network design: facility location, routing, sensor placement

6.5 Theoretical Analysis

Theorem 6.1 (Approximation Guarantee)**.** *For monotone submodular maximization with cardinality k, the stochastic greedy algorithm achieves an expected $(1 - 1/e - \epsilon)$-approximation using $O(n \ln(1/\epsilon))$ evaluations.*

Theorem 6.2 (Convergence of ϵ-greedy)**.** *In a multi-armed bandit, ϵ-greedy with decreasing ϵ converges to the optimal arm with probability 1 as iterations $\to \infty$.*

6.6 Comparison with Deterministic Greedy Algorithms

Stochastic greedy methods add:

- **Exploration** to avoid local traps
- **Robustness** to initial conditions
- **Adaptability** via tunable randomness
- **Efficiency**: fewer evaluations in complex settings
- **Parallelizability**: many variants parallelize naturally

Example 6.1 (Maximum Coverage)**.** *Given sets $S_1, \ldots, S_m$ over a universe of n elements, choose k sets to maximize coverage. An ϵ-greedy approach (choose best with $1 - \epsilon$, random with ϵ) attains within 1–2*

Practice Problems

1. Implement deterministic vs. ϵ-greedy knapsack; compare on correlated vs. random instances.
2. Analyze softmax temperature τ on a combinatorial test problem.
3. Prove the $O(n \ln(1/\epsilon))$-evaluation bound for stochastic greedy maximization.

7 Comparative Analysis and Practical Guidelines

Method	**Pop.**	**Memory**	**Exploration**	**Exploitation**	**Parallel**
SA	single	low	high→low	low→high	low
GA	multi	medium	medium	medium	high
DE	multi	low	medium	high	high
PSO	multi	high	medium	high	high
Stoch. Greedy	mixed	low	low→med	high	med.

Different problems call for different tools: SA excels on combinatorial, GA/PSO on non-linear interactions, DE on continuous correlated spaces, and stochastic greedy when incremental construction is natural. Hybrid "memetic" and surrogate-assisted schemes further boost performance.

Recent advances: surrogate-models, multi-objective extensions, large-scale methods, adaptive/self-adaptive schemes, quantum-inspired heuristics, and hyper-heuristics.