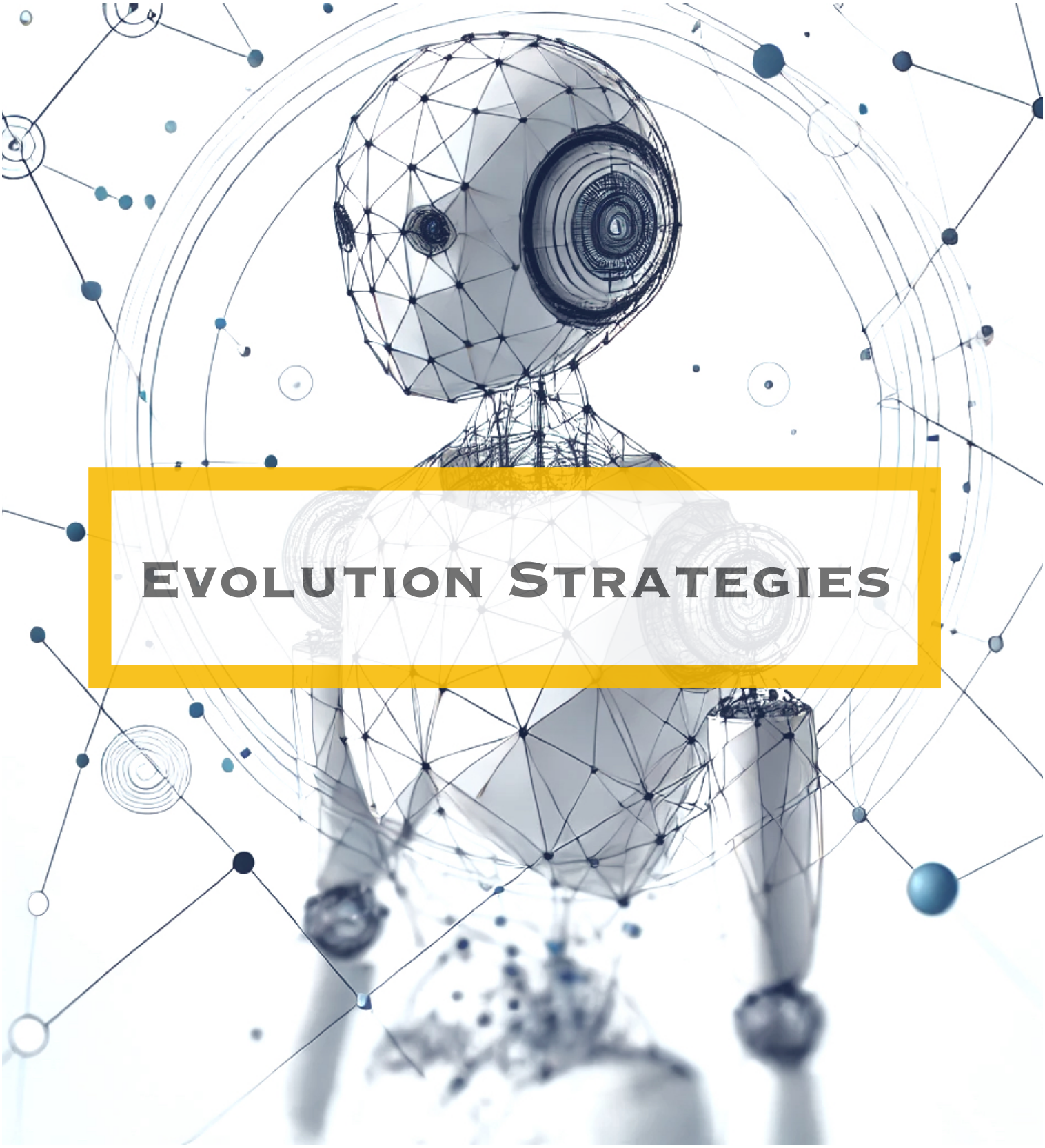

Evolution Strategies

Preface

This book provides a structured introduction to Evolution Strategies (ES) as methods for black-box optimization. It begins with the fundamentals of optimization and evolutionary algorithms, including historical background and practical guidance for experimentation. The core of the book develops ES step by step: starting with the classical $(1 + 1)$-ES, moving to Rechenberg's step-size control and the $1/5$ success rule, and then extending to population-based variants such as the $(\mu + \lambda)$-ES and plus/comma selection schemes. Self-adaptation mechanisms and cumulative step-size control via evolution paths are introduced as principled ways of controlling mutation strength. Building on these foundations, the book presents correlated mutations and the Covariance Matrix Adaptation Evolution Strategy (CMA-ES), including simple implementations and practical usage. A dedicated chapter on progress rate analysis provides theoretical insight into convergence behavior and parameter choice. Benchmark functions and experimental methodology support empirical evaluation. Throughout the book, theoretical concepts are combined with hands-on examples, Python implementations, and exercises to encourage experimentation and deeper understanding.

Oliver Kramer
Oldenburg, February 23, 2026

Contents

Chapter 1

Introduction

1.1 What is Optimization?

Before diving into evolutionary algorithms, we first clarify the central concept of this book: *optimization*. Optimization means finding the best possible solution among many possibilities. In everyday life, we constantly solve optimization problems. When choosing the fastest route, designing an efficient engine, or training a neural network to reduce prediction error, we aim to improve a system according to a defined objective.

In mathematics, optimization is typically formulated as

$$\min_{\mathbf{x}\in\mathbb{R}^n} f(\mathbf{x}) \tag{1}$$

or

$$\max_{\mathbf{x}\in\mathbb{R}^n} f(\mathbf{x}),$$

where $\mathbf{x}$ denotes a candidate solution vector and $f(\mathbf{x})$ is the objective (or fitness) function. The search space consists of all admissible vectors $\mathbf{x}$, and the goal is to find the vector that yields the best objective value. A vector $\mathbf{x}^*$ is called a *global minimum* if

$$f(\mathbf{x}^*) \leq f(\mathbf{x}) \quad \text{for all } \mathbf{x} \in \mathbb{R}^n. \tag{2}$$

As a simple example, consider the one-dimensional function

$$f(x) = (x-3)^2.$$

This function has its global minimum at $x = 3$, where $f(x) = 0$. An optimization algorithm does not know this solution beforehand. It must explore different candidate vectors $\mathbf{x}$, evaluate their objective values, and iteratively improve them in order to approach the optimum.

Real-world problems are usually much more complicated. Many objective functions resemble mountain landscapes with hills, valleys, ridges, and plateaus. A *local minimum* is a solution that is better than its direct neighbors, while a *global minimum* is the best solution overall. An algorithm may get trapped in a local minimum and fail to reach the global one. Avoiding such traps is one of the central challenges in optimization.

Gradient Descent

Gradient Descent is an iterative optimization algorithm for minimizing differentiable objective functions. It follows the direction of steepest descent, given by the negative gradient.
Starting from an initial point $\mathbf{x}_0$, the update rule is

$$\mathbf{x}_{t+1} = \mathbf{x}_t - \eta \nabla f(\mathbf{x}_t),$$

where $\eta > 0$ is the learning rate and $\nabla f(\mathbf{x}_t)$ is the gradient at iteration t.
The gradient points in the direction of steepest increase. By subtracting it, the algorithm moves toward smaller function values. The learning rate controls the step size: if too small, convergence is slow; if too large, the method may overshoot or diverge.
Gradient Descent is efficient for smooth problems but requires gradient information and may converge to local minima.

In many practical applications, we do not know the mathematical structure of the objective function. We may not know its formula, we may not have access to derivatives, and sometimes we can only evaluate it through expensive simulations or real-world experiments. Such problems are called *black-box optimization problems*. For instance, evaluating a new airplane design may require a complex simulation, testing a drug candidate may require laboratory experiments, and measuring the performance of a machine learning model requires full training and validation. In these situations, classical gradient-based methods are often not suitable.

This is where evolutionary algorithms become particularly useful. They are well suited for problems in which the objective function is noisy, highly nonlinear, non-differentiable, high-dimensional, or only accessible through evaluations. Instead of following exact gradient information, they explore the search space using populations of candidate solutions and stochastic variation. They trade precise mathematical direction for robustness and broad exploration. In one simple sentence: Optimization means trying many possi-

ble solutions, measuring how good they are, and gradually improving them. Evolution Strategies are a powerful and elegant way to accomplish exactly this.

1.2 Evolutionary Algorithms

Evolutionary Algorithms (EAs) are a family of population-based optimization methods inspired by the principles of natural evolution. Each individual in the population represents a candidate solution to the optimization problem. This solution is typically encoded as a *genotype*, such as a vector of real numbers, a binary string, or a symbolic expression, which maps to a concrete solution in the problem space, called the *phenotype*.

A fitness function evaluates the quality of each solution, guiding the search process toward better candidates. Over successive generations, the population evolves through the iterative application of variation operators, such as mutation and recombination, and selection based on fitness. This evolutionary cycle, illustrated in Figure 1.1, gradually steers the population toward optimal or near-optimal solutions.

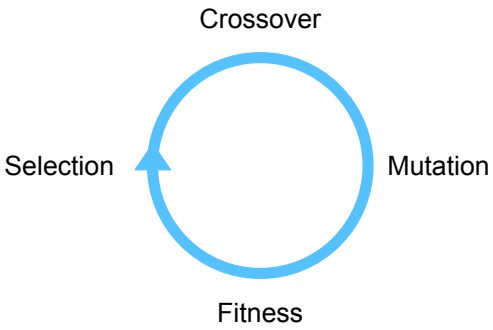

Figure 1.1: Evolutionary cycle: variation, evaluation, and selection

This book focuses on *Evolution Strategies* (ES), a prominent class of evolutionary algorithms designed for black-box optimization, especially in continuous domains. Unlike Genetic Algorithms, which often use symbolic or combinatorial representations, ES operate directly on real-valued vectors and adapt their mutation distributions based on feedback from the optimization process.

1.3 History of Evolutionary Algorithms

The history of EAs spans several decades and is marked by significant milestones that have shaped the development of the field. Below is an overview of key events and achievements that have contributed to the evolution of EA:

- **1950s–1960s: Foundational Ideas** — Early concepts of evolution-inspired optimization appear, including work by Nils Aall Barricelli (1954) on self-replicating automata and Alan Turing's thoughts on machine learning via mutation and selection.
- **1964: Evolution Strategies (ES)** — Ingo Rechenberg and Hans-Paul Schwefel develop evolution strategies in Germany for solving engineering design problems [6], focusing on mutation and step-size adaptation.
- **1975: Genetic Algorithms (GA)** — John Holland publishes *Adaptation in Natural and Artificial Systems* [4], formalizing the concept of genetic algorithms and introducing the schema theorem and the idea of building blocks.
- **1970s–1980s: Genetic Programming (GP)** — John Koza and others extend the GA framework to evolve computer programs, laying the groundwork for genetic programming.
- **1990s: Unification and Growth** — The term *evolutionary algorithms* emerges as a unifying umbrella for GAs, ES, GP, and evolutionary programming (EP). Practical applications expand into machine learning, control, scheduling, and design.
- **1995: Differential Evolution (DE)** — Storn and Price introduce DE, a simple yet powerful real-valued optimization method using vector differences for mutation.
- **1999: NSGA-II** — Deb et al. propose NSGA-II, a breakthrough in multi-objective evolutionary optimization that combines non-dominated sorting and diversity preservation.
- **2000s–2010s: Theory and Hybridization** — Runtime analysis, surrogate modeling, and hybrid methods integrating EAs with local search, gradient methods, and machine learning emerge.
- **2010s–Present: Modern Integration** — EAs are integrated with deep learning, neural architecture search, and reinforcement learning. Applications include drug discovery, robotics, and creative design. New paradigms like quality diversity, novelty search, and open-ended evolution gain attention.

1.4 How to Use This Book

This book is designed to blend theory with practice and guide you through Hands on exploration of AI methods. Important equations are numbered:

$$\mathbf{x}' = \mathbf{x} + \mathcal{N}(\mathbf{0}, \mathbf{I}) \tag{3}$$

Along the way, you'll find useful visual cues:

- 🕮 This icon indicates links to recommended reading or further literature, helping you dive deeper into the topic.
- 💻 Each chapter concludes with a set of practical exercises to reinforce your understanding and encourage experimentation.

Sketch Boxes

Sketch Boxes offer concise yet complete explanations of key methods, concepts, or tools. They are crafted to enrich understanding at a glance, providing clarity without breaking the rhythm of the chapter.

Action Boxes

Action Boxes highlight related tasks and suggestions that go beyond algorithms and code. They are designed to encourage broader engagement with the field and support personal and professional development alongside technical learning.

1.5 Further Reading: GA and ES Textbooks

Below is a selection of recommended textbooks specifically focused on GAs and ES):

- *Genetic Algorithms in Search, Optimization, and Machine Learning* by David E. Goldberg [2]: A classic introduction to the principles and applications of genetic algorithms, including schema theory and GA-based optimization.
- *Evolutionary Computation: A Unified Approach* by Kenneth A. De Jong [5]: Offers a broad view of evolutionary computation, with a solid theoretical foundation for genetic algorithms and evolution strategies.
- *Introduction to Evolutionary Computing* by A. E. Eiben and J. E. Smith [1]: A widely used textbook that covers GAs, ES, evolutionary programming, and genetic programming in a unified framework.

- *The CMA Evolution Strategy: A Tutorial* by Nikolaus Hansen [3]: A comprehensive tutorial covering the Covariance Matrix Adaptation Evolution Strategy (CMA-ES), one of the most powerful ES variants for continuous optimization.

There is a wide range of additional books and literature available for further reading.

Chapter 2

$(1+1)$-ES

2.1 $(1+1)$-ES

Evolution Strategies (ES) are a family of black-box optimization algorithms designed for numerical optimization problems. The simplest member of this family is the $(1 + 1)$-ES, which operates on a single solution vector and generates exactly one offspring per generation.

The main idea is to iteratively improve a single solution $\mathbf{x} \in \mathbb{R}^n$ by applying Gaussian mutation. The mutation step creates a new candidate solution $\mathbf{x}'$ as follows:

$$\mathbf{x}' = \mathbf{x} + \sigma \cdot \mathcal{N}(\mathbf{0}, \mathbf{1}), \tag{4}$$

where $\sigma > 0$ is the mutation strength, also known as step size, and $\mathcal{N}(\mathbf{0}, \mathbf{1})$ is a vector of independent standard normally distributed random variables.

After mutation, the better of the two solutions, either the parent $\mathbf{x}$ or the offspring $\mathbf{x}'$, is selected for the next generation:

$$\mathbf{x} \leftarrow \begin{cases} \mathbf{x}' & \text{if } f(\mathbf{x}') \leq f(\mathbf{x}), \\ \mathbf{x} & \text{otherwise.} \end{cases}$$

The algorithm proceeds for a fixed number of generations or until a predefined termination criterion is met, such as reaching a target fitness value or detecting stagnation in progress. *Stagnation* refers to the situation where the fitness does not improve over a predefined number of consecutive generations, indicating that the search has stopped making meaningful progress.

Algorithm 1 $(1+1)$-Evolution Strategy

1: Initialize solution $\mathbf{x} \in \mathbb{R}^n$, step size $\sigma > 0$
2: **repeat**
3: Sample $\mathbf{z} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
4: Create offspring $\mathbf{x}' = \mathbf{x} + \sigma \cdot \mathbf{z}$
5: **if** $f(\mathbf{x}') \leq f(\mathbf{x})$ **then** $\mathbf{x} \leftarrow \mathbf{x}'$
6: **until** termination condition is met

Probability Density of Isotropic Gaussian

The probability density function of an n-dimensional isotropic Gaussian distribution is given by:

$$p(\mathbf{z}) = \frac{1}{(2\pi)^{n/2}} \exp\left(-\frac{1}{2}\|\mathbf{z}\|^2\right),$$

where $\mathbf{z} \in \mathbb{R}^n$ and $\|\mathbf{z}\|$ is the Euclidean norm. Each component of $\mathbf{z}$ is drawn independently from a standard normal distribution with mean 0 and variance 1.

2.2 Hands on $(1+1)$-ES

Consider the optimization of a simple quadratic function

$$f(\mathbf{x}) = \|\mathbf{x}\|^2$$

with the initial solution

$$\mathbf{x}_0 = \begin{bmatrix} 3 \\ 4 \end{bmatrix} \quad \text{and} \quad \sigma_0 = 1.0$$

We apply a $(1+1)$-ES with Gaussian mutation

$$\mathbf{x}' = \mathbf{x} + \sigma \cdot \mathcal{N}(0, \mathbf{I})$$

and evaluate the new point. If the offspring is better (i.e., has a lower objective value), it replaces the parent. Assume the first sampled mutation vector is

$$\mathcal{N}(0, \mathbf{I}) = \begin{bmatrix} -0.5 \\ 0.6 \end{bmatrix} \Rightarrow \mathbf{x}_1 = \mathbf{x}_0 + \sigma_0 \cdot \begin{bmatrix} -0.5 \\ 0.6 \end{bmatrix} = \begin{bmatrix} 2.5 \\ 4.6 \end{bmatrix}$$

Compute objective values:

$$f(\mathbf{x}_0) = 3^2 + 4^2 = 25, \quad f(\mathbf{x}_1) = 2.5^2 + 4.6^2 = 6.25 + 21.16 = 27.41$$

Since $f(\mathbf{x}_1) > f(\mathbf{x}_0)$, the new point is rejected, and $\mathbf{x}_0$ is retained.

Properties of Mutation Operators

Mutation operators in evolutionary algorithms must satisfy key properties to ensure effective search performance. Gaussian mutation fulfills the following:

1. **Reachability:** Any point in the search space must be reachable through a sequence of mutations, allowing the algorithm to explore the entire domain. The Gaussian density is positive everywhere in $\mathbb{R}^N$, hence any region can be reached with non-zero probability.
2. **Unbiasedness:** The mutation operator should not favor any particular direction on average, preserving isotropic search behavior. For $\mathcal{N}(\mathbf{0}, \mathbf{I})$, we have $\mathbb{E}[\mathbf{z}] = \mathbf{0}$ and isotropic covariance, so no direction is preferred.
3. **Scalability:** The mutation strength should be scalable. The step size σ directly scales the mutation $\mathbf{x}' = \mathbf{x} + \sigma\mathbf{z}$, allowing the search radius to be adjusted to different problem scales.

2.3 $(1+1)$-ES in Python

The algorithm continues for a predefined number of generations or until a desired fitness value is reached.

```python
import numpy as np

def sphere(x):
    return np.sum(x**2)

fitness_function = sphere

def one_plus_one_es_classic(fitness_function, N, sigma,
   max_generations):
    x = np.random.uniform(-5, 5, N)

    for t in range(1, max_generations + 1):
        x_prime = x + sigma * np.random.normal(0, 1, N)
        if fitness_function(x_prime) <= fitness_function(x):
            x = x_prime

return x, sigma
```

This Python implementation captures the core steps of the $(1+1)$-ES: random initialization, Gaussian mutation, and selection. Despite its simplicity, the $(1+1)$-ES is effective for smooth, unimodal functions and forms the foundation for more advanced evolutionary strategies.

2.4 Exercises

1. Concepts:
   (a) Sketch the $(1+1)$-ES and explain how it fits into the evolutionary cycle.
   (b) State the equation of Gaussian mutation and explain the variables.
   (c) Explain the role of the step size σ in controlling exploration and exploitation.
   (d) State the key conditions for mutation operators and explain why Gaussian mutation satisfies them.
   (e) Explain the role of selection in the $(1+1)$-ES.
2. Manual Run:
   (a) Simulate three iterations of the $(1+1)$-ES on $f(\mathbf{x}) = x_1^2 + x_2^2$, starting from $(3, 4)$ with $\sigma = 1.0$.
   (b) Sample Gaussian mutation vectors, compute offspring, evaluate fitness values, and apply the selection rule step by step.
3. Coding:
   (a) Implement the $(1+1)$-ES.
   (b) Test it on the Sphere function with $n = 2$ and $n = 10$ dimensions for 1000 generations.
   (c) Compare runs with $\sigma \in \{0.01, 1.0, 10.0\}$ and discuss how step sizes influences convergence behavior.

Explore the Origins of Evolution

Research Charles Darwin, *On the Origin of Species*, and the Galápagos Islands. Summarize how biological evolution inspired evolutionary algorithms.

Chapter 3

Rechenberg

3.1 Step Size Adaptation

In Evolution Strategies, the step size σ plays a central role in controlling the scale of variation applied to candidate solutions. An appropriate step size allows the algorithm to explore the search space efficiently, large steps encourage global exploration, while small steps focus the search locally. Step size adaptation refers to techniques that adjust σ dynamically during the evolutionary process to balance exploration and exploitation. Effective adaptation strategies, such as the 1/5th success rule, help ES algorithms avoid premature convergence and improve convergence speed.

Parameter Control

Parameter control adapts internal strategy parameters, such as mutation step size σ, population size, or recombination weights, during the run of an Evolution Strategy. This improves performance by allowing the algorithm to adjust to the problem landscape over time. Common types of parameter control include:

- **Deterministic control:** Parameters follow a predefined schedule (e.g., decreasing step size over time).
- **Adaptive control:** Parameters are adjusted based on feedback, such as the success rate of mutations.
- **Self-adaptation:** Parameters are encoded into individuals and evolved along with the solution vector.

Self-adaptive control of σ is a key innovation in modern ES, allowing autonomous and problem-specific tuning of the mutation strength.

3.2 Rechenberg's 1/5 Success Rule

Parameter control is a crucial aspect of ES. One of the most influential self-adaptation methods is the Rechenberg 1/5 Success Rule, introduced by Ingo Rechenberg in the context of the $(1+1)$-Evolution Strategy. This adaptive mechanism adjusts the mutation step size σ based on the observed success rate of mutations. The idea is to maintain a success rate of approximately $\frac{1}{5}$, which empirically leads to efficient progress in many continuous optimization problems. If the mutation success rate is high, increase the step size, if the success rate is low, reduce it. This adaptive adjustment helps the algorithm converge efficiently.

Let $s \in [0, 1]$ denote the success rate, i.e., the fraction of successful mutations within a recent window of generations (typically 5 to 20 iterations). The mutation strength σ is updated according to

$$\sigma \leftarrow \begin{cases} \sigma \cdot \tau & \text{if } s > \frac{1}{5}, \\ \sigma / \tau & \text{if } s < \frac{1}{5}, \end{cases} \tag{5}$$

where $\tau > 1$ is a scaling factor (e.g., $\tau = 1.5$). If the success rate exceeds 1/5, the step size is increased to encourage larger exploratory moves. If the success rate falls below 1/5, the step size is reduced to refine the search.

Figure 3.1 illustrates Rechenberg's 1/5 rule. Mutations are sampled within circles representing the current step size. During the first five generations, three successful mutations (blue squares) and two unsuccessful ones (gray squares) occur. Because the success rate exceeds 1/5, the step size is increased, leading to the larger blue circle.

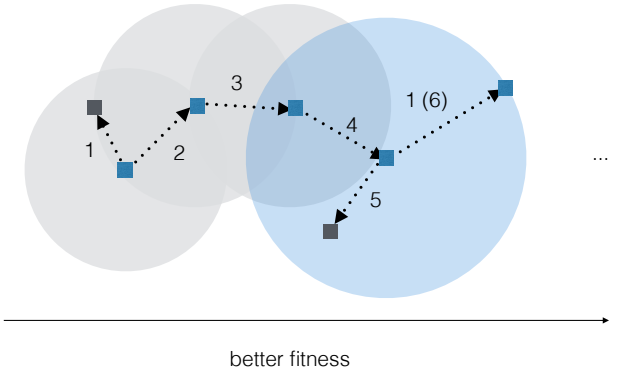

Figure 3.1: Illustration of Rechenberg's 1/5 step-size control with a generation window of five. Three out of five mutations are successful, leading to an increase in the mutation strength.

Per-Generation Rechenberg Rule

A per-generation variant of the 1/5 success rule increases step size σ by multiplication with $\exp(4/5)$ in case of a successful mutation, and decreases σ by $\exp(-1/5)$ in case of an unsuccessful mutation:

$$\sigma' = \sigma \cdot \exp^{1/d}(\mathcal{I}_{f(\mathbf{x}')\leq f(\mathbf{x})} - 1/5) \tag{6}$$

with scaling parameter d[a]. Indicator function $\mathcal{I}_{f(\mathbf{x}')\leq f(\mathbf{x})}$ delivers 1 in case of improvement, i.e. $f(\mathbf{x}') \leq f(\mathbf{x})$ is true, and 0 otherwise. Below is a Python implementation:

[a] As convenient notation we use $\exp^{1/d}(x) = (e^{1/d})^x = e^{x/d}$.

3.3 Hands-On Rechenberg Rule

We minimize the Sphere function

$$f(\mathbf{x}) = x_1^2 + x_2^2$$

with initial point

$$\mathbf{x}_0 = \begin{bmatrix} 3 \\ 4 \end{bmatrix}, \qquad f(\mathbf{x}_0) = 25, \qquad \sigma = 1.0.$$

Using a generation window $k = 5$ and scaling factor $\tau = 1.5$, we obtain five offspring:

$$\begin{aligned}
\mathbf{x}_1' &= (2.8,\ 4.1), \quad f = 24.65 \\
\mathbf{x}_2' &= (3.5,\ 4.3), \quad f = 30.74 \\
\mathbf{x}_3' &= (2.6,\ 3.7), \quad f = 20.45 \\
\mathbf{x}_4' &= (3.2,\ 4.4), \quad f = 29.60 \\
\mathbf{x}_5' &= (3.1,\ 4.2), \quad f = 27.25
\end{aligned}$$

Two of five mutations improve the fitness, hence

$$s = \frac{2}{5} = 0.4.$$

Since $s > \frac{1}{5}$, the mutation strength is increased:

$$\sigma \leftarrow \sigma\tau = 1.5.$$

The rule increases the step size when improvements are frequent and decreases it otherwise.

Restarts

Restarts are used to escape local optima by periodically reinitializing the search process. The basic idea is to monitor the step size σ: if it drops below a threshold ϵ, the search is likely converging. Since we cannot guarantee convergence to the global optimum, a restart increases the chances of finding better solutions.

$$\text{if } \sigma < \epsilon \text{ then restart}$$

When restarting, the solution is reinitialized randomly in the search space, and the step size is reset. This improves exploration and reduces the risk of premature convergence.

3.4 Rechenberg in Python

The algorithm continues for a predefined number of generations or until a desired fitness value is reached.

```
import numpy as np

def one_plus_one_es_success_rulefitness_function, N, sigma,
    max_generations):
     x = np.random.uniform(-5, 5, N)
     tau = 1.5
     k = 20# evaluation window
     success_count = 0

     for t in range(1, max_generations + 1):
         x_prime = x + sigma * np.random.normal(0, 1, N)
         if fitness_function(x_prime) <= fitness_function(x):
             x = x_prime
             success_count += 1

         if t % k == 0:
             success_rate = success_count / k
             if success_rate > 1/5:
                 sigma *= tau
             else:
                 sigma /= tau
             success_count = 0

return x, sigma
```

This Python implementation captures the core steps of the $(1 + 1)$-ES: random initialization, Gaussian mutation, and selection. Despite its simplicity, the $(1 + 1)$-ES is effective for smooth, unimodal functions and forms the foundation for more advanced evolutionary strategies.

3.5 Exercises

1. Concepts:
   (a) What is the difference between deterministic and adaptive parameter control?
   (b) What are the benefits of adaptive step sizes?
   (c) Describe the intuition behind the 1/5 success rule.
   (d) Why does the 1-step Rechenberg have a similar effect. What happens if in 5 generations 1 is successful and 4 not.
2. Manual Simulation:
   (a) Simulate 3 steps of (1+1)-ES on $f(\mathbf{x}) = \sum x_i^2$ from $\mathbf{x} = (3, 4)$, $\sigma = 1.0$.
   (b) Apply the 1/5 rule with $\tau = 1.5$ every the three steps, track updates to $\mathbf{x}$, σ, and fitness.
3. Coding:
   (a) Implement Rechenberg's per-generation update rule in Python.
   (b) Run it on the Sphere function with $n = 2$ dimensions and compare different settings for τ.
   (c) Plot fitness and σ development over time, and compare to a run with fixed $\sigma = 0.5$.
   (d) Implement the restart mechanism and compare the results.

Explore EA Conferences

Find out about major conferences in Evolutionary Computation such as GECCO (Genetic and Evolutionary Computation Conference), CEC (IEEE Congress on Evolutionary Computation), and PPSN (Parallel Problem Solving from Nature).

Chapter 4

$(\mu + \lambda)$-ES

4.1 Populations

Populations are a central component of evolutionary algorithms, representing a diverse set of candidate solutions that evolve over time. Unlike single-solution methods, population-based algorithms maintain and update multiple individuals in parallel, enabling richer exploration of the search space. The structure and size of the population influence the algorithm's ability to balance exploration and exploitation. While small populations may converge quickly, they risk getting trapped in *local optima*: larger populations provide greater diversity but increase computational cost. Different variants of Evolution Strategies, such as $(1+1)$-ES, $(\mu+\lambda)$-ES, and (μ, λ)-ES, use different population models to guide selection, recombination, and replacement.

Local Optima

A local optimum is a solution $\mathbf{x}^*$ such that

$$f(\mathbf{x}^*) \leq f(\mathbf{x})$$

for all $\mathbf{x}$ in a neighborhood of $\mathbf{x}^*$. Unlike a global optimum, it is only optimal within a limited region of the search space. In multimodal functions, evolutionary algorithms may converge to local optima if mutation strength is too small or diversity is lost. Strategies such as larger step sizes, restarts, or population-based search help to escape local optima.

4.2 $(\mu + \lambda)$-ES

The $(\mu + \lambda)$-ES employs a population of μ parent solutions to generate λ offspring in each generation. The use of a population increases the robustness and exploration capabilities of the algorithm. Each generation applies recombination (crossover) and mutation to generate offspring.

Recombination combines two or more randomly selected parents to form an intermediate offspring. In the case of intermediate recombination, the arithmetic mean of ρ parents is computed:

$$\mathbf{x} = \frac{1}{\rho} \sum_{i=1}^{\rho} \mathbf{x}_i. \tag{7}$$

In multi-recombination, i.e., when $\rho = \mu$, also written as a $(\mu/\mu + \lambda)$-ES, the arithmetic mean is taken over the entire parent population $\mathcal{P}$. Here, solutions are not sampled randomly but all parents contribute equally.

Algorithm 2 $(\mu + \lambda)$-ES

1: Initialize a parent population $\{\mathbf{x}_1, \ldots, \mathbf{x}_\mu\} \subset \mathbb{R}^n$
2: **repeat**
3: **for** $k = 1$ **to** λ **do**
4: Select two parents $\mathbf{x}_i, \mathbf{x}_j$ uniformly at random
5: Recombine: $\mathbf{y}_k = \frac{1}{2}(\mathbf{x}_i + \mathbf{x}_j)$
6: Mutate: $\mathbf{x}_k = \mathbf{y}_k + \sigma\mathcal{N}(\mathbf{0}, \mathbf{I})$
7: Evaluate: $f(\mathbf{x}_k)$
8: **end for**
9: Select the best μ individuals from the union of parents and offspring
10: **until** termination condition is met

An alternative to intermediate recombination is dominant recombination, where each component of the offspring is randomly selected from one of the ρ parents:

$$(\mathbf{x})_i = (\mathbf{x}_j)_i, \quad \text{with } j \sim \text{uniform}(\{1, \ldots, \rho\}). \tag{8}$$

Experimental Evaluation

Due to the stochastic nature of ES, reliable performance analysis requires multiple independent runs. Typical experimental protocols involve:

- Benchmark functions such as Sphere, Rosenbrock, Griewank, Rastrigin.
- 30 to 100 repetitions per setup to estimate statistical variation.
- Fitness metrics: mean, standard deviation, median, best, and worst.
- Performance measures: fitness after a fixed evaluation budget, or evaluations needed to reach a given target.

Fitness development is often shown on a log scale. When comparing algorithms with different population sizes, evaluation budgets must be normalized by total fitness function calls.

4.3 Plus and Comma Selection

In plus selection, i.e., the $(\mu + \lambda)$-ES, the next parent population consists of the best μ individuals selected from the combined set of μ parents and λ offspring. This guarantees that the best solutions are never lost between generations. In contrast, comma selection, denoted as (μ, λ)-ES, selects the new generation only from the λ offspring. This introduces the possibility of losing the current best solution but improves the algorithm's ability to escape local optima. A further generalization is the $(\mu/\rho, \lambda, \kappa)$-ES, in which each solution is assigned a lifetime κ. A solution can no longer participate in selection after surviving for κ generations. This introduces age-based diversity control and helps to avoid premature convergence.

Statistical Significance Testing

To determine if observed performance differences are statistically meaningful, hypothesis testing is applied. As the distribution of ES outcomes is often non-normal, non-parametric tests are preferred. The Wilcoxon signed-rank test is commonly used for comparing two algorithms over multiple runs on the same problem. It tests whether the median difference between paired observations is zero. The result includes a test statistic and a *p-value*, which quantifies the probability that the observed difference is due to chance. A small p-value (e.g., $p < 0.05$) indicates a statistically significant difference.

4.4 Hands-On $(\mu + \lambda)$-ES

To illustrate the $(\mu+\lambda)$-Evolution Strategy, consider optimizing the function

$$f(\mathbf{x}) = \|\mathbf{x}\|^2$$

in $\mathbb{R}^2$, using a population of $\mu = 2$ parents and $\lambda = 3$ offspring. The mutation strength is fixed at $\sigma = 0.5$.

Let's assume the parental population is:

$$\mathbf{x}_1 = \begin{bmatrix} 2 \\ 3 \end{bmatrix}, \quad \mathbf{x}_2 = \begin{bmatrix} 4 \\ 1 \end{bmatrix}$$

Fitness:

$$f(\mathbf{x}_1) = 13, \quad f(\mathbf{x}_2) = 17$$

As first step, recombination is performed. We use intermediate recombination, i.e., the arithmetic mean of both parents:

$$\mathbf{y} = \frac{1}{2}(\mathbf{x}_1 + \mathbf{x}_2) = \begin{bmatrix} 3 \\ 2 \end{bmatrix}$$

Then, mutation is applied. We generate $\lambda = 3$ offspring by adding Gaussian noise:

$$\mathbf{x}_1' = \begin{bmatrix} 3.2 \\ 1.6 \end{bmatrix}, \quad \mathbf{x}_2' = \begin{bmatrix} 2.7 \\ 2.3 \end{bmatrix}, \quad \mathbf{x}_3' = \begin{bmatrix} 3.4 \\ 1.9 \end{bmatrix}$$

For selection, we evaluate the offspring individuals:

$$\begin{aligned} f(\mathbf{x}_1') &= 3.2^2 + 1.6^2 = 12.80, \\ f(\mathbf{x}_2') &= 2.7^2 + 2.3^2 = 12.58, \\ f(\mathbf{x}_3') &= 3.4^2 + 1.9^2 = 15.37. \end{aligned}$$

Combine with original parents:

$$f(\mathbf{x}_1) = 13, \quad f(\mathbf{x}_2) = 17$$

We now select the best $\mu = 2$ individuals among all:

$$\mathbf{x}_{\text{new},1} = \mathbf{x}_2', \quad \mathbf{x}_{\text{new},2} = \mathbf{x}_1'$$

The new parent population becomes:

$$\mathbf{x}_{\text{new},1} = \begin{bmatrix} 2.7 \\ 2.3 \end{bmatrix}, \quad \mathbf{x}_{\text{new},2} = \begin{bmatrix} 3.2 \\ 1.6 \end{bmatrix}$$

This example illustrates how recombination pulls offspring toward a central point, while mutation adds variability. The selection step ensures that only the best solutions survive. Over generations, the population is expected to converge toward the global minimum at $\mathbf{0}$.

4.5 $(\mu + \lambda)$-ES in Python

Below is a minimal Python implementation of $(\mu + \lambda)$-ES.

```
import numpy as np

mu = 5
lmbda = 20
dim = 10
generations = 100
sigma = 0.3# Mutation strength

# Initialize population
population = np.random.randn(mu, dim)

for gen in range(generations):
# Generate lambda offspring with Gaussian mutation
offspring = []

for _ in range(lmbda):
    parent = population[np.random.randint(mu)]
    child = parent + sigma * np.random.randn(dim)
    offspring.append(child)
    offspring = np.array(offspring)

    # Combine parents and offspring
    combined = np.vstack([population, offspring])
    fitness = np.array([sphere(ind) for ind in combined])

    # Select the best mu individuals for the next generation
    best_indices = np.argsort(fitness)[:mu]
    population = combined[best_indices]

    # Optional: track progress
    print(f"Generation {gen}: Best fitness =
       {fitness[best_indices[0]]:.4f}")
```

This simple implementation demonstrates how the $(\mu + \lambda)$-ES maintains diversity through offspring generation and ensures quality by selecting only the top-performing individuals. It can be extended with recombination, step-size adaptation, and constraint handling to tackle more complex optimization problems.

4.6 Exercises

1. Concepts:
   (a) What are the advantages of populations?
   (b) What is a $(\mu + \lambda)$-ES?

(c) Why is recombination used in ES and what types exist?
(d) Compare plus and comma selection.

2. Manual Simulation:
   (a) Simulate 2 generations of a $(3 + 5)$-ES on the Sphere function with $n = 2$.
   (b) Use intermediate recombination and track parents, intermediate solutions, and offspring,.

3. Coding:
   (a) Implement $(\mu + \lambda)$-ES with $\sigma = 0.3$ and intermediate recombination.
   (b) Run the $(\mu + \lambda)$-ES on the benchmark set with $n = 10$ dimensions
   (c) Compare intermediate vs. dominant recombination as well as plus vs. comma selection.

Meta-ES

Implement a Meta-ES that optimizes the parameters of another ES. Use an *outer* ES to tune the population sizes and hyperparameters like τ of an *inner* ES solving the Sphere function.

Chapter 5

Self-Adaptation

5.1 Introduction

Self-adaptation introduces a mechanism by which strategy parameters, typically the mutation step size σ, are encoded into the individual and evolved alongside the solution vector. This allows the algorithm to adapt its search behavior over time without manual tuning of hyperparameters.

The core idea is that successful individuals are likely to carry not only good solution vectors $\mathbf{x}$, but also good strategy parameters σ. Therefore, the step size is inherited by offspring and mutated together with the solution. If a certain step size consistently leads to better solutions, it will be propagated to future generations.

5.2 σ-Self-Adaptation

Each individual is extended to include both object variables and strategy parameters:

$$(\mathbf{x}, \sigma)$$

Offspring are generated by mutating both components:

$$\begin{aligned} \sigma' &= \sigma \cdot \exp(\tau\, \mathcal{N}(0,1))\,, \\ \mathbf{x}' &= \mathbf{x} + \sigma'\, \mathbf{z}, \qquad \mathbf{z} \sim \mathcal{N}(\mathbf{0}, \mathbf{I}) \end{aligned} \tag{9}$$

Here, τ is a learning rate, often set to $\tau = \frac{1}{\sqrt{n}}$, where n is the dimension of the search space.

The mutation of the step size is multiplicative and log-normal, which ensures positivity and scales changes appropriately. This leads to a form of

meta-evolution: not only are solutions evolved, but also the way the search is performed.

Multivariate Self-Adaptation

In multi-dimensional search, the self-adaptive scheme can be extended to use a separate step size for each coordinate:

$$\sigma_i' = \sigma_i \cdot \exp\left(\tau' \cdot \mathcal{N}(0,1) + \tau \cdot \mathcal{N}_i(0,1)\right)$$

$$x_i' = x_i + \sigma_i' \cdot \mathcal{N}_i(0,1)$$

with global learning rate $\tau' = \frac{1}{\sqrt{2n}}$ and local learning rate $\tau = \frac{1}{\sqrt{2\sqrt{n}}}$. This allows the algorithm to adaptively shape the mutation distribution and exploit problem structure more efficiently.

5.3 Hands-On σ-Self-Adaptation

In self-adaptive Evolution Strategies, each individual carries not only a solution $\mathbf{x}$, but also its own mutation strength σ. Both components evolve together over time.

We consider the optimization of the function

$$f(\mathbf{x}) = \|\mathbf{x}\|^2$$

in dimension $n = 2$, with learning rate $\tau = \frac{1}{\sqrt{2}} \approx 0.707$. The initial individual is

$$\mathbf{x} = \begin{bmatrix} 3.0 \\ 4.0 \end{bmatrix}, \quad \sigma = 1.0.$$

To begin, the mutation strength is updated via a log-normal rule. A sample from a standard normal distribution yields

$$\mathcal{N}(0,1) = 0.5 \quad \Rightarrow \quad \sigma' = 1.0 \cdot \exp(0.707 \cdot 0.5) \approx 1.424.$$

Next, the solution vector is mutated using the new step size. Sampling a 2-dimensional standard normal vector gives

$$\mathcal{N}(\mathbf{0}, \mathbf{I}) = \begin{bmatrix} -0.3 \\ 0.7 \end{bmatrix},$$

which leads to

$$\mathbf{x}' = \mathbf{x} + \sigma' \cdot \begin{bmatrix} -0.3 \\ 0.7 \end{bmatrix} = \begin{bmatrix} 3.0 \\ 4.0 \end{bmatrix} + 1.424 \cdot \begin{bmatrix} -0.3 \\ 0.7 \end{bmatrix} = \begin{bmatrix} 2.573 \\ 5.0 \end{bmatrix}.$$

We evaluate the objective function at both the parent and offspring:

$$f(\mathbf{x}) = 3^2 + 4^2 = 25, \quad f(\mathbf{x}') = 2.573^2 + 5.0^2 \approx 6.62 + 25 = 31.62.$$

Since the offspring has a worse fitness value, it would be rejected in a $(1 + 1)$-ES. If the offspring were better, both the solution $\mathbf{x}'$ and its mutation strength σ' would be inherited. This illustrates how the algorithm co-evolves both the candidate solution and its search behavior, enabling automatic adaptation of the mutation strength during optimization.

Derandomized Self-Adaptation

Classic self-adaptation mutates the step size σ using independent noise, which introduces *selection noise*: a successful offspring may result from an unfavorable realization of σ, and vice versa. Derandomized self-adaptation removes this source of noise by reusing the same mutation vector $\mathbf{z} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ that generates the offspring:

$$\mathbf{x}' = \mathbf{x} + \sigma\mathbf{z}, \qquad \sigma_i' = \sigma_i \exp(d_i z_i).$$

Thus, the step sizes are adapted consistently with the actual mutation direction. If a component z_i contributes to a successful step, the corresponding σ_i is reinforced; otherwise it is reduced. The learning rates $d_i \ll 1$ ensure stable adaptation and play a role similar to recombination in larger populations.

5.4 σ-Self-Adaptation in Python

Below is a minimal Python implementation of σ-Self-Adaptation.

```
import numpy as np

# Parameters
mu = 5
lmbda = 20
dim = 10
generations = 100
tau = 1 / np.sqrt(dim)

# Initialize population: each individual is (x, sigma)
population = [(np.random.randn(dim), 0.5) for _ in range(mu)]

for gen in range(generations):
offspring = []
```

```
for _ in range(lmbda):
    parent_x, parent_sigma =
        population[np.random.randint(mu)]
    # Mutate sigma
    sigma_prime = parent_sigma * np.exp(tau *
        np.random.randn())
    sigma_prime = max(sigma_prime, 1e-8)# Prevent too small
        sigma
    # Mutate x
    x_prime = parent_x + sigma_prime * np.random.randn(dim)
    offspring.append((x_prime, sigma_prime))

    # Combine parent and offspring
    combined = population + offspring
    fitness = [sphere(ind[0]) for ind in combined]

    # Select best mu individuals
    best_indices = np.argsort(fitness)[:mu]
    population = [combined[i] for i in best_indices]

    # Optional: track best
    best_fit = fitness[best_indices[0]]
    print(f"Gen {gen}: Best fitness = {best_fit:.4f}")
```

This approach allows both the solutions and their mutation strengths to co-evolve, making the algorithm more robust to different problem landscapes. Self-adaptation is particularly useful in high-dimensional or non-separable problems, where fixed or globally adapted step sizes may fail.

5.5 Exercises

1. Concepts:
   (a) What is the idea of self-adaptation in ES? How does it compare to adaptive step size control?
   (b) Why use a log-normal distribution for mutating σ?
   (c) What are the roles of τ' and τ in multivariate adaptation?
   (d) What is selection noise and derandomized self-adaptation?
2. Manual Simulation:
   (a) Simulate one generation of a $(\mu + \lambda)$-ES with $\mu = 2$, $\lambda = 3$, $n = 2$, using $f(\mathbf{x}) = \sum x_i^2$.
   (b) Report original/mutated σ, offspring, and selected solutions.
3. Coding:
   (a) Implement a $(\mu + \lambda)$-ES with multivariate self-adaptive σ.
   (b) Run it on the Sphere function with $n = 2, 10, 100$ dimensions.

(c) Plot fitness and σ curves.

Differential Evolution (DE)

Find out what Differential Evolution (DE) is and how its mutation and crossover operators differ from those in ES. Implement DE and compare its performance to an ES on benchmark functions such as Sphere and Rastrigin.

Chapter 6

Evolution Path

6.1 Evolution Path

The evolution path $\mathbf{p}_\sigma \in \mathbb{R}^n$ is a cumulative vector that summarizes recent search directions in ES. It tracks the direction and consistency of normalized steps across generations. At each generation, the path is updated using the standardized mutation vector from the current parent $\mathbf{x}$ to the offspring $\mathbf{x}'$:

$$\mathbf{p}_\sigma \leftarrow (1 - c_\sigma)\mathbf{p}_\sigma + \sqrt{c_\sigma(2 - c_\sigma)} \cdot \frac{\mathbf{x}' - \mathbf{x}}{\sigma}, \tag{10}$$

where $c_\sigma \in (0, 1)$ is the cumulation rate and σ is the mutation step size. The vector $\frac{\mathbf{x}'-\mathbf{x}}{\sigma}$ represents the standardized mutation.

If steps align over time, the path grows in length. If directions fluctuate, the path shortens. This accumulated signal is used to adjust the step size.

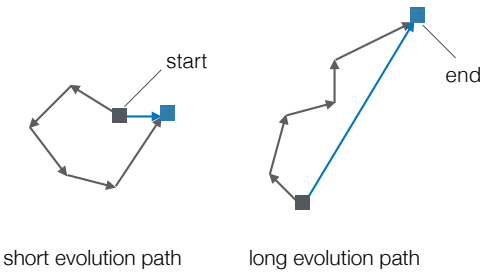

Figure 6.1: Illustration of an evolution path with aligned and alternating steps.

6.2 Step Size Adaptation

The mutation strength σ is adapted using the norm of the evolution path $\mathbf{p}_\sigma \in \mathbb{R}^n$. The update rule is:

$$\sigma \leftarrow \sigma \cdot \exp\left(\frac{c_\sigma}{d_\sigma}\left(\frac{\|\mathbf{p}_\sigma\|}{\mathbb{E}[\|\mathcal{N}(\mathbf{0},\mathbf{I})\|]} - 1\right)\right), \tag{11}$$

where $d_\sigma > 0$ is a damping parameter. The normalization factor is the expected norm of a standard normal vector in N dimensions:

$$\mathbb{E}[\|\mathcal{N}(\mathbf{0},\mathbf{I})\|] \approx \sqrt{n}\left(1 - \frac{1}{4n} + \frac{1}{21N^2}\right).$$

The step size increases only if the accumulated path length exceeds the expected length of a random sequence of independent steps. Thus, both directional consistency and sufficient magnitude are required. If the steps are small, even if consistently improving, the path remains shorter than the random expectation and the step size decreases.

Intuitively, a long evolution path indicates sustained and coherent movement in a particular direction, suggesting that the optimizer is confidently progressing and can afford larger steps. A short or fluctuating path, in contrast, indicates weak correlation or cancellation of steps, suggesting cautious refinement with smaller step sizes.

6.3 Hands-On Cumulative Path Step Size Control

We minimize the two-dimensional Sphere function

$$f(\mathbf{x}) = x_1^2 + x_2^2$$

with

$$\mathbf{x} = \begin{bmatrix}2\\0\end{bmatrix}, \quad \sigma = 0.5, \quad \mathbf{p}_\sigma = \begin{bmatrix}0\\0\end{bmatrix}, \quad c_\sigma = 0.3, \quad d_\sigma = 1.0.$$

For $n = 2$,

$$\mathbb{E}[\|\mathcal{N}(\mathbf{0},\mathbf{I})\|] \approx 1.25, \qquad \sqrt{c_\sigma(2-c_\sigma)} \approx 0.714.$$

First mutation:

$$\mathbf{z}_1 = (-0.4,\, 0.1), \quad \mathbf{x}' = (1.8,\, 0.05),$$

accepted. The path becomes

$$\mathbf{p}_\sigma = 0.714\,\mathbf{z}_1 = (-0.286,\ 0.071), \quad \|\mathbf{p}_\sigma\| \approx 0.295.$$

Since $0.295 < 1.25$, the step size decreases to

$$\sigma \approx 0.406.$$

Second mutation:

$$\mathbf{z}_2 = (-0.3,\ -0.2), \quad \mathbf{x}' = (1.678,\ -0.031),$$

accepted. The updated path is

$$\mathbf{p}_\sigma = 0.7(-0.286,\ 0.071) + 0.714(-0.3,\ -0.2) = (-0.414,\ -0.093),$$

with

$$\|\mathbf{p}_\sigma\| \approx 0.425.$$

Again $0.425 < 1.25$, hence the step size decreases further. Although both steps improved the objective, the accumulated path length remained below the expected random length, and therefore the mutation strength was reduced.

6.4 Evolution Path in Python

The following code implements the univariate $(1 + 1)$-ES with cumulative path length control.

```
import numpy as np
import matplotlib.pyplot as plt

# Parameters
generations = 20
x = 0.0
sigma = 0.5
p_sigma = 0.0
c_sigma = 0.3
d_sigma = 1.0
expected_norm = np.sqrt(2 / np.pi)

x_vals = []
sigma_vals = []
p_vals = []

for _ in range(generations):
    z = np.random.randn()
    x_candidate = x + sigma * z
```

```
    if sphere(x_candidate) < sphere(x):
        x = x_candidate
        p_sigma = (1 - c_sigma) * p_sigma + np.sqrt(c_sigma
            * (2 - c_sigma)) * z
        sigma *= np.exp((c_sigma / d_sigma) * (abs(p_sigma)
            / expected_norm - 1))

    x_vals.append(x)
    sigma_vals.append(sigma)
    p_vals.append(p_sigma)
```

Constraint Handling with Penalty Functions

A constrained optimization problem has the form

$$\min_{\mathbf{x}\in\mathbb{R}^n} f(\mathbf{x}) \quad \text{subject to} \quad g(\mathbf{x}) \leq 0.$$

Penalty methods transform the constrained problem into an unconstrained one by adding a penalty term to the objective. Given a constraint $g(\mathbf{x}) \leq 0$, a quadratic penalty function is defined as

$$P(\mathbf{x}) = \max\{0, g(\mathbf{x})\}^2,$$

and the penalized objective becomes

$$f'(\mathbf{x}) = f(\mathbf{x}) + \alpha P(\mathbf{x}),$$

where $\alpha > 0$ controls the strength of constraint violations.

6.5 Exercises

1. Concepts:
   (a) What does the evolution path $\mathbf{p}_\sigma$ accumulate over time, and why is this useful?
   (b) Explain the cumulation path-bsaed step size mechanism with the main equation.
   (c) What does it mean if the norm $\|\mathbf{p}_\sigma\|$ is small or large compared to the expected length?
   (d) What happens if you set $c_\sigma = 1.0$? What if $c_\sigma = 0.0$?
2. Manual Simulation:

(a) Simulate one step of a $(1+1)$-ES on $f(x) = x^2$, starting at $x = 2.0$, $\sigma = 0.5$, and $p_\sigma = 0$. Use $c_\sigma = 0.3$, $d_\sigma = 1.0$, and $\mathbb{E}[|N(0,1)|] \approx 0.7979$. Sample $z = -0.4$ and $z = -0.2$.

(b) For each accepted step, compute the updated p_σ and σ.

(c) Interpret the results: why did the step size change the way it did?

3. Coding: Evolution Path Control

(a) Implement a $(1+1)$-ES with cumulative step size adaptation.

(b) Track and plot the evolution of x, σ, and p_σ over 200 generations on the Sphere with $n = 10$.

4. Coding: Penalty Functions

(a) Implement a $(1+1)$-ES for the constrained problem

$$\min_{\mathbf{x}\in\mathbb{R}^2} f(\mathbf{x}) = \|\mathbf{x}\|^2 \quad \text{subject to} \quad x_1 \geq 1.$$

(b) Rewrite the constraint as $g(\mathbf{x}) = 1 - x_1 \leq 0$ and implement the penalized objective

$$f'(\mathbf{x}) = \|\mathbf{x}\|^2 + \alpha \max\{0, 1 - x_1\}^2.$$

(c) Experiment with different penalty factors α and analyze how the solution behavior changes.

(d) Plot the search trajectory in the plane and visualize the feasible region.

Investigate AlphaEvolve

Find out about AlphaEvolve and related AI systems that discover new algorithms. Analyze how evolutionary ideas are used to improve code or mathematical expressions.

Chapter 7

CMA-ES

7.1 Correlated Mutations

In standard Evolution Strategies, the mutation operator is typically *isotropic*, i.e., based on $\mathbf{z} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$, corresponding to a spherical Gaussian distribution with identical variance in all directions. This approach works well when the coordinate axes are aligned with the structure of the objective function, see Figure 7.1 (left for isotropic and middle for axis-parallel mutation ellipsoids). However, for ill-conditioned or rotated landscapes, isotropic mutation is inefficient.

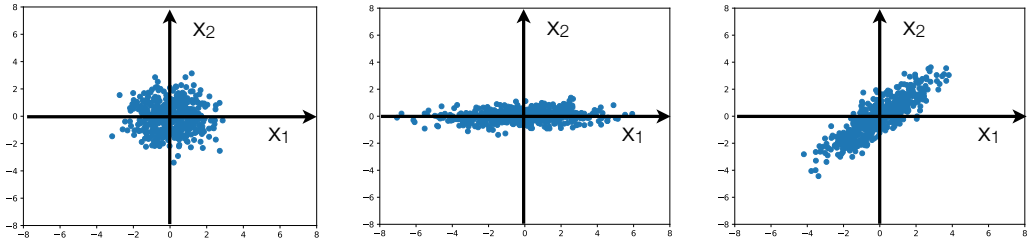

Figure 7.1: Gaussian mutation variants: (left) isotropic, (middle) axis-aligned, and (right) correlated.

To improve search performance, the mutation distribution can be adapted to the shape of the fitness landscape, see Figure 7.1 (right). This is achieved by introducing a full covariance matrix $\mathbf{C}$, leading to correlated mutation:

$$\mathbf{x}' = \mathbf{x} + \sigma \cdot \mathcal{N}(\mathbf{0}, \mathbf{C}). \tag{12}$$

The matrix $\mathbf{C}$ encodes directions and scales of successful search steps. The goal is to learn and adapt $\mathbf{C}$ from experience.

7.2 Simple CMA-ES.

The simple Covariance Matrix Adaptation Evolution Strategy (Simple CMA-ES) maintains a mean $\mathbf{x}$, a global step size σ, and a covariance matrix $\mathbf{C}$. In each generation, λ offspring are sampled from a multivariate normal distribution centered at $\mathbf{x}$. The best μ individuals are selected, and their distribution is used to update $\mathbf{C}$ using a rank-μ empirical estimate.

Let $\mathbf{x}_{(1)}, \ldots, \mathbf{x}_{(\mu)}$ be the best μ individuals (with lowest objective values), and let their mean be:

$$\bar{\mathbf{x}} = \frac{1}{\mu} \sum_{i=1}^{\mu} \mathbf{x}_{(i)}.$$

Then, the covariance matrix is updated as:

$$\mathbf{C} \leftarrow (1 - c_\mu)\mathbf{C} + c_\mu \cdot \frac{1}{\mu} \sum_{i=1}^{\mu} (\mathbf{x}_{(i)} - \bar{\mathbf{x}})(\mathbf{x}_{(i)} - \bar{\mathbf{x}})^\top,$$

where $c_\mu \in (0, 1]$ is a learning rate. This update accumulates the shape of the distribution of the selected individuals.

The step size σ is adapted using a simple success rule:

$$\sigma \leftarrow \sigma \cdot \exp^{1/d}(\mathcal{I}_{f(\bar{\mathbf{x}}) \leq f(\mathbf{x})} - 1/5),$$

where $d \approx \sqrt{N+1}$, and success is defined as improvement over the previous mean.

Algorithm 3 Simple CMA-ES

Initialize $\mathbf{x} \in \mathbb{R}^n$, $\sigma > 0$, $\mathbf{C} = \mathbf{I}_N$
repeat
  Sample λ offspring: $\mathbf{x}_k = \mathbf{x} + \sigma \cdot \mathbf{A} \cdot \mathbf{z}_k$, $\mathbf{z}_k \sim \mathcal{N}(0, \mathbf{I})$
  Evaluate $f(\mathbf{x}_k)$, select best μ individuals
  Compute mean $\bar{\mathbf{x}} = \frac{1}{\mu} \sum_{i=1}^{\mu} \mathbf{x}_{(i)}$
  Update covariance: $\mathbf{C} \leftarrow (1 - c_\mu)\mathbf{C} + c_\mu \cdot \frac{1}{\mu} \sum (\mathbf{x}_{(i)} - \bar{\mathbf{x}})(\cdot)^\top$
  Adapt step size: $\sigma \leftarrow \sigma \cdot \exp^{1/d}(\mathcal{I}_{f(\bar{\mathbf{x}}) \leq f(\mathbf{x})} - 1/5)$
  if $f(\bar{\mathbf{x}}) \leq f(\mathbf{x})$ **then**
    $\mathbf{x} \leftarrow \bar{\mathbf{x}}$
  end if
until termination condition

Rank-1 Covariance Update

The rank-1 update in CMA-ES adjusts the covariance matrix based on a single evolution path, which captures the overall search direction over several generations. It allows the strategy to accumulate information about the most promising direction. The evolution path $\mathbf{p}_c$ is updated according to

$$\mathbf{p}_c \leftarrow (1 - c_p) \cdot \mathbf{p}_c + \sqrt{c_p(2 - c_p) \cdot \mu_{\text{eff}}} \cdot \frac{\bar{\mathbf{x}} - \mathbf{x}}{\sigma},$$

where c_1 is the learning rate for the rank-1 update, c_p is the learning rate for the evolution path, and μ_{eff} is the variance-effective selection mass.

7.3 Simple CMA-ES in Python

The following Python code implements a simple version of the Covariance Matrix Adaptation Evolution Strategy (CMA-ES) to optimize the Sphere function in $\mathbb{R}^n$. The algorithm uses Cholesky decomposition for sampling correlated mutations and adapts both the covariance matrix and the global step size σ.

```
def simple_cma_es(fitness_fn, N=10, sigma=0.5,
                  lambda_=10, mu=5, generations=200,
                     seed=42):
    np.random.seed(seed)
    x = np.random.randn(N)
    C = np.eye(N)
    fitness_history = []

    for gen in range(generations):
        A = np.linalg.cholesky(C)
        Z = np.random.randn(lambda_, N)
        X = x + sigma * Z @ A.T
        fitnesses = np.array([fitness_fn(xi) for xi in X])
        indices = np.argsort(fitnesses)
        selected = X[indices[:mu]]
        x_mean = np.mean(selected, axis=0)

        C_update = np.zeros((N, N))
        for xi in selected:
            diff = xi - x_mean
            C_update += np.outer(diff, diff)
        C = (1 - 0.2) * C + 0.2 * (C_update / mu)

        success = fitness_fn(x_mean) <= fitness_fn(x)
        sigma *= np.exp((1 / N) * (success - 0.2))
```

```
        if success:
            x = x_mean

        fitness_history.append(fitness_fn(x))

    return fitness_history

fitness_history = simple_cma_es(sphere)
```

This implementation demonstrates how the simple CMA-ES adapts both the search distribution and step size over time. The algorithm successfully reduces the fitness on the convex Sphere function and illustrates the self-adaptive behavior of the method.

Progress Rate

The progress rate measures the expected improvement of an evolutionary algorithm in one iteration. For a minimization problem with current solution $\mathbf{x}_t$, the progress rate in objective space is

$$\varphi(\mathbf{x}_t) = \mathbb{E}\big[f(\mathbf{x}_t) - f(\mathbf{x}_{t+1}) \mid \mathbf{x}_t\big].$$

It quantifies the expected decrease in fitness per generation. Often it is more intuitive to measure progress geometrically. Let $\mathbf{x}^*$ denote the global optimum and define the distance

$$r_t = \|\mathbf{x}_t - \mathbf{x}^*\|.$$

The radial progress rate is

$$\varphi_r(r_t) = \mathbb{E}[r_t - r_{t+1} \mid r_t].$$

A positive value indicates expected improvement. The dependence of the progress rate on the mutation strength explains why intermediate step sizes yield maximal progress.

7.4 CMA-ES in `pycma`

The CMA-ES adapts the covariance matrix of its search distribution to learn the shape of the objective function landscape, enabling efficient search in non-separable and ill-conditioned problems. The rank-one and rank-μ updates combine information from the best individuals across generations to estimate favorable search directions and scale. Below is an example using the

pycma library to optimize a 20-dimensional SPHERE function. The parameters CMA_rankone and CMA_rankmu control the learning rate for the update rules:

```
import cma
import numpy as np

n = 20
initial_mean = np.random.randn(N)
initial_sigma = 0.5

es = cma.CMAEvolutionStrategy(
initial_mean, initial_sigma, {
'popsize': 20,
'CMA_rankone': 0.0,
'CMA_rankmu': 0.1
}
)

def sphere_function(x):
return np.sum(x**2)

es.optimize(sphere_function, maxfun=10000)
```

7.5 Exercises

1. Conceptual:
   (a) Why are isotropic mutations inefficient in some solution spaces?
   (b) What does the covariance matrix $\mathbf{C}$ encode?
   (c) Explain the rank-μ update of the CMA-ES.
2. Manual Simulation:
   (a) Given $\mathbf{x} = [2, 1]$, $\mathbf{C} = \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}$, $\sigma = 0.5$, compute one mutation using $\mathbf{z} \sim \mathcal{N}(0, \mathbf{I})$.
   (b) From offspring $\mathbf{x}_1 = [1, 2]$, $\mathbf{x}_2 = [1.5, 2.5]$, $\mathbf{x}_3 = [0.5, 1.5]$, compute $\bar{\mathbf{x}}$ and the empirical covariance.
3. Coding:
   (a) Implement the simple CMA-ES with rank-μ update.
   (b) Compare it to the CMA-ES in pycma on the benchmark problems with $n = 2, 5, 10, 100$ dimensions.
4. Experimental Comparison:

(a) Your two favorite ES on the benchmark set. Define relevant experimental settings (problem dimension, evaluation budget, population sizes, and algorithm parameters). Conduct 100 independent runs per algorithm and report best, worst, mean, median, and standard deviation of the final fitness values. Summarize the results in tabular form and assess statistical significance using the Wilcoxon rank-sum test at a predefined significance level.

Natural Evolution Strategies (NES)

Natural Evolution Strategies optimize the parameters of a search distribution $p_\theta(\mathbf{x})$, typically a multivariate Gaussian, instead of optimizing individual solutions directly. A population is sampled from the current distribution, evaluated, and the distribution parameters are updated toward regions of higher expected fitness. The objective is to maximize the expected fitness

$$J(\theta) = \mathbb{E}[f(\mathbf{x})].$$

Using the log-likelihood identity, the gradient can be written as

$$\nabla_\theta J(\theta) = \mathbb{E}[f(\mathbf{x}) \nabla_\theta \log p_\theta(\mathbf{x})] \,,$$

which allows gradient estimation without requiring derivatives of the fitness function itself. To obtain stable updates, NES uses the natural gradient,

$$\theta \leftarrow \theta + \eta \, \mathbf{F}^{-1} \nabla_\theta J(\theta),$$

where $\mathbf{F}$ is the Fisher information matrix. This adjustment accounts for the geometry of the distribution and leads to invariant and well-scaled updates.

Build an ES Agent

Design and implement an ES agent using OpenClaw. The agent should be able to run different types of ES and adapt their parameters automatically (optionally learning good parameter combinations). Integrate a messaging interface (e.g., Telegram) to issue commands and monitor progress, and enable the agent to publish summarized results to a social media platform.

Appendix A

Benchmarks

To evaluate and compare optimization algorithms, several standard benchmark functions are used:
Sphere

$$f(\mathbf{x}) = \sum_{i=1}^{n} x_i^2, \quad f(\mathbf{0}) = 0$$

Convex and smooth; baseline for convergence speed.
Rosenbrock

$$f(\mathbf{x}) = \sum_{i=1}^{N-1} \left[100(x_{i+1} - x_i^2)^2 + (x_i - 1)^2\right], \quad f(\mathbf{1}) = 0$$

Narrow curved valley, hard for local search.
Griewank

$$f(\mathbf{x}) = 1 + \tfrac{1}{4000} \sum_{i=1}^{n} x_i^2 - \prod_{i=1}^{n} \cos\left(\tfrac{x_i}{\sqrt{i}}\right), \quad f(\mathbf{0}) = 0$$

Many local optima; tests robustness.
Rastrigin

$$f(\mathbf{x}) = 10N + \sum_{i=1}^{n} \left[x_i^2 - 10\cos(2\pi x_i)\right], \quad f(\mathbf{0}) = 0$$

Highly multimodal; difficult global search.

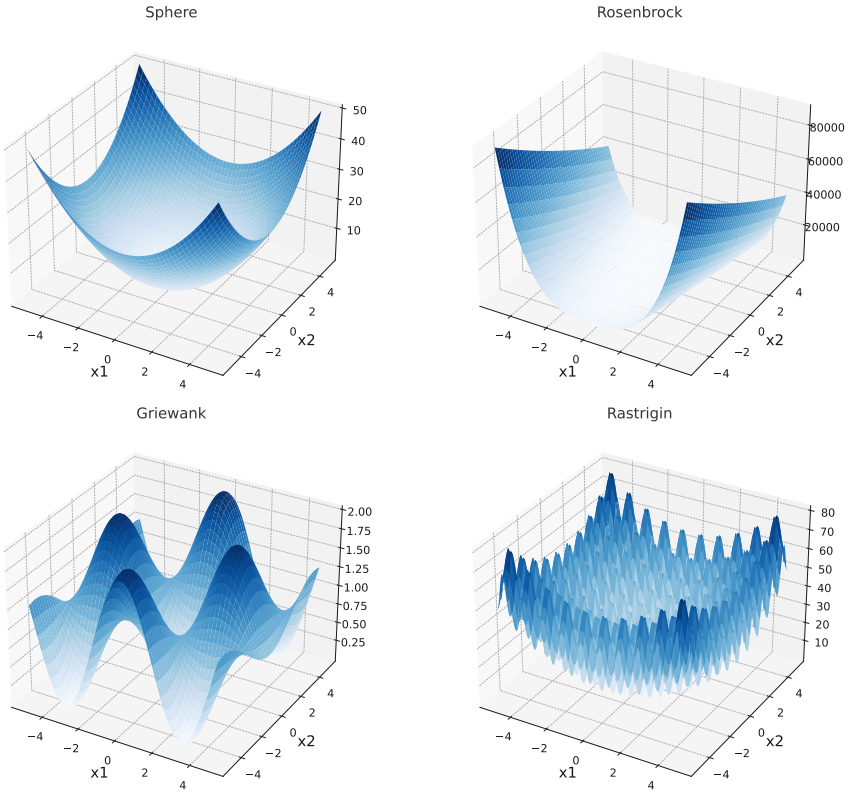

Figure A.1: 2D landscapes of Sphere, Rosenbrock, Griewank, and Rastrigin.

Appendix B

Math Essentials

This appendix summarizes the minimal mathematical tools required for Evolution Strategies.

B.1 Notation

Scalars are written as lowercase letters such as x, r, σ. Vectors are written in bold lowercase letters such as $\mathbf{x}, \mathbf{z} \in \mathbb{R}^n$. Matrices are written in bold uppercase letters such as $\mathbf{C} \in \mathbb{R}^{n \times n}$.

B.2 Vectors and Norms

A vector

$$\mathbf{x} = (x_1, \ldots, x_n) \in \mathbb{R}^n$$

has Euclidean norm

$$\|\mathbf{x}\| = \sqrt{\sum_{i=1}^{n} x_i^2}.$$

The dot product of two vectors $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$ is

$$\mathbf{x} \cdot \mathbf{y} = \sum_{i=1}^{n} x_i y_i, \qquad \|\mathbf{x}\|^2 = \mathbf{x} \cdot \mathbf{x}.$$

These operations are used in mutation, recombination, covariance adaptation, and progress analysis.

B.3 Matrices and Covariance

A matrix $\mathbf{C} \in \mathbb{R}^{n \times n}$ is symmetric if

$$\mathbf{C} = \mathbf{C}^\top.$$

It is positive definite if

$$\mathbf{x}^\top \mathbf{C} \mathbf{x} > 0 \quad \text{for all } \mathbf{x} \neq \mathbf{0}.$$

In CMA-ES, the covariance matrix $\mathbf{C}$ is symmetric and positive definite. Its Cholesky decomposition

$$\mathbf{C} = \mathbf{A}\mathbf{A}^\top$$

is used to generate correlated Gaussian mutations.

B.4 Gaussian Distribution

In Evolution Strategies, mutation vectors are sampled from the standard multivariate normal distribution

$$\mathbf{z} \sim \mathcal{N}(\mathbf{0}, \mathbf{I}),$$

meaning that all components are independent with mean 0 and variance 1. Gaussian mutation is defined as

$$\mathbf{x}' = \mathbf{x} + \sigma \mathbf{z}.$$

B.5 Expectation

The expectation of a random quantity is denoted by $\mathbb{E}[\cdot]$. In cumulative step-size adaptation, the expected norm of a standard Gaussian vector appears:

$$\mathbf{z} \sim \mathcal{N}(\mathbf{0}, \mathbf{I}), \qquad \mathbb{E}[\|\mathbf{z}\|] \approx \sqrt{n} \left(1 - \frac{1}{4n} + \frac{1}{21n^2}\right).$$

Appendix C

Python Basics

C.1 Installation

Install Python (version 3.14 or newer) from `https://python.org`.

To avoid package conflicts, it is recommended to create a separate environment using `conda`. If you do not have conda installed, download Miniconda from `https://docs.conda.io`.

Create and activate a new environment:

```
conda create -n es python=3.14
conda activate es
```

Now install the required packages:

```
pip install numpy matplotlib
```

C.2 Basic Syntax

```
# Variables and data types
x = 42              # integer
pi = 3.14           # float
msg = "Hello"       # string
flag = True         # boolean

# Lists
nums = [1, 2, 3]
nums.append(4)

# Loops
for i in nums:
    print(i)

# Conditionals
```

```
if pi > 3:
    print("pi is large")
else:
    print("pi is small")

# Functions
def square(x):
    return x * x
print(square(5))    # Output: 25
```

C.3 NumPy Essentials

```
import numpy as np

# Arrays and operations
x = np.array([1.0, 2.0, 3.0])
print(np.linalg.norm(x))    # Euclidean norm

# Random numbers (Gaussian)
z = np.random.normal(0, 1, 5)    # 5 samples from N(0,1)
print(z)

# Standard normal vector (used in ES mutation)
dim = 3
z = np.random.randn(dim)    # vector ~ N(0, I)
print(z)

# Elementwise operations
y = np.array([4.0, 5.0, 6.0])
print(x + y)    # [5. 7. 9.]
```

C.4 Example: Evaluating Sphere

```
import numpy as np

def sphere(x):
    return np.sum(x**2)

x = np.random.normal(0, 1, 10)  # 10D vector
print("x:", x)
print("f(x):", sphere(x))
```

This small script shows how benchmark functions can be implemented in Python and evaluated with NumPy arrays.

Bibliography

[1] A. E. Eiben and J. E. Smith. *Introduction to Evolutionary Computing*. Natural Computing Series. Springer, 2003. ISBN: 978-3642072857.

[2] David E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, 1989. ISBN: 978-0201157673.

[3] Nikolaus Hansen. "The CMA Evolution Strategy: A Tutorial". In: *arXiv preprint* arXiv:1604.00772 (2016). URL: `https://arxiv.org/abs/1604.00772`.

[4] John H. Holland. *Adaptation in Natural and Artificial Systems*. MIT Press, 1992.

[5] Kenneth A. De Jong. *Evolutionary Computation: A Unified Approach*. MIT Press, 2006. ISBN: 978-0262041942.

[6] Hans-Paul Schwefel. *Evolution and Optimum Seeking*. John Wiley & Sons, 1995.