Task 2.3: Stochastic Search Methods — CEM, NES, CMA–E

Objective

Implement and compare three population-based, derivative-free stochastic optimization algorithms:

(a) **Cross–Entropy Method (CEM)** — iterative sampling and elite–selection based distribution update.

(b) **Natural Evolution Strategies (NES)** — gradient-based adaptation of the sampling distribution parameters.

(c) **Covariance Matrix Adaptation Evolution Strategy (CMA–ES)** — adaptive evolution strategy with covariance and step-size control.

The aim is to understand sampling-based global optimization, natural-gradient adaptation, and covariance adaptation mechanisms. You will analyze their behaviour on multimodal functions and compare efficiency (objective values, number of function evaluations, runtime).

Algorithms to implement

Translate the provided Julia code into Python. Use clear, well-structured code and ensure deterministic results by fixing the random seed.

Algorithms (Julia reference)

```julia
using Distributions
function cross_entropy_method(f, P, k_max, m=100, m_elite=10)
    for k in 1:k_max
        samples = rand(P, m)  # Generate m samples from current distribution
        order = sortperm([f(samples[:, i]) for i in 1:m])  # Sort by f-value
        P = fit(typeof(P), samples[:, order[1:m_elite]]) # Update distribution
    end
    return P
end
```

Example usage:

```julia
import LinearAlgebra: norm
seed!(0)
f = x -> norm(x)
μ, Σ = [0.5, 1.5], [1.0 0.2; 0.2 2.0]
P = MvNormal(μ, Σ)
k_max = 10
P = cross_entropy_method(f, P, k_max)
```

```julia
using Distributions, LinearAlgebra
function natural_evolution_strategies(f, μ, A, k_max; m=100, α=0.01)
    for k in 1:k_max
        Σ = A' * A                      # Current covariance matrix
        P = MvNormal(μ, Symmetric(Σ))
        samples = [rand(P) for i in 1:m]
        g_μ = sum(f(x) * (Σ \ (x - μ)) for x in samples) / m
        μ -= α * g_μ                    # Update mean
        g_A = zeros(size(A))
        for x in samples
            dx = x - μ
            g_Σ = 0.5 * (Σ \ (dx * dx') * Σ \ I - Σ \ I)
            g_A += f(x) * A * (g_Σ + g_Σ')  # Symmetric gradient
        end
        A -= α * g_A/m                  # Update covariance factor
    end
    return μ, A
end
```

```
function covariance_matrix_adaptation(f, x, k_max;
    σ = 1.5,
    m = 4 + floor(Int, 3*log(length(x))),
    m_elite = div(m,2))
    μ, n = copy(x), length(x)
    ws = log((m+1)/2) .- log.(1:m)
    ws[1:m_elite] ./= sum(ws[1:m_elite])
    μ_eff = 1 / sum(ws[1:m_elite].^2)
    cσ = (μ_eff + 2)/(n + μ_eff + 5)
    dσ = 1 + 2max(0, sqrt((μ_eff-1)/(n+1))-1) + cσ
    cΣ = (4 + μ_eff/n)/(n + 4 + 2μ_eff/n)
    c1 = 2/((n+1.3)^2 + μ_eff)
    cμ = min(1-c1, 2*(μ_eff-2+1/μ_eff)/((n+2)^2 + μ_eff))
    ws[m_elite+1:end] .*= -(1 + c1/cμ)/sum(ws[m_elite+1:end])
    E = n^0.5 * (1 - 1/(4n) + 1/(21n^2))
    pσ, pΣ, Σ = zeros(n), zeros(n), Matrix(1.0I, n, n)
    for k in 1:k_max
        P = MvNormal(μ, σ^2 * Σ)
        xs = [rand(P) for i in 1:m]
        ys = [f(x) for x in xs]
        is = sortperm(ys)
        δs = [(x - μ)/σ for x in xs]
        δw = sum(ws[i] * δs[is[i]] for i in 1:m_elite)
        μ += σ * δw
        C = Σ^(-0.5)
        pσ = (1 - cσ)*pσ + sqrt(cσ*(2 - cσ)*μ_eff) * C * δw
        σ *= exp(cσ/dσ * (norm(pσ)/E - 1))
        hσ = Int(norm(pσ)/sqrt(1 - (1 - cσ)^(2k)) <
                 (1.4 + 2/(n + 1)) * E)
        pΣ = (1 - cΣ)*pΣ + hσ*sqrt(cΣ*(2 - cΣ)*μ_eff) * δw
        w0 = [ws[i] ≥ 0 ? ws[i] :
              n * ws[i] / norm(C * δs[is[i]])^2 for i in 1:m]
        Σ = (1 - c1 - cμ)*Σ +
            c1*(pΣ*pΣ' + (1 - hσ)*cΣ*(2 - cΣ)*Σ) +
            cμ*sum(w0[i]*δs[is[i]]*δs[is[i]]' for i in 1:m)
        Σ = triu(Σ) + triu(Σ,1)'        # enforce symmetry
    end
    return μ
end
```

Cross–Entropy Method (parameters). At the initial iteration, the sampling distribution P is a bivariate normal distribution

$$P_0 = \mathcal{N}(\mu_0, \Sigma_0), \quad \mu_0 \text{ (see the subsection Domains and initialization)}, \quad \Sigma_0 = \begin{pmatrix} 1.0 & 0.2 \\ 0.2 & 2.0 \end{pmatrix}.$$

The remaining parameters are

$$m = 40, \quad m_{\text{elite}} = 10, \quad k_{\max} = 10, \quad \texttt{seed} = 42.$$

Natural Evolution Strategies (parameters). Use μ_0 as defined in the subsection Domains and initialization,

$$A_0 = \begin{pmatrix} 1.0 & 0.0 \\ 0.0 & 1.0 \end{pmatrix}, \quad m = 60, \quad \alpha = 0.005, \quad k_{\max} = 30, \quad \texttt{seed} = 42.$$

CMA–ES (parameters). Use starting points, defined in the subsection Domains and initialization, $k_{\max} = 200$ and $\texttt{seed} = 42$.

Test functions

The following benchmark functions are used to illustrate the behavior of CEM, NES, and CMA–ES. All are defined in two dimensions.

- **Ackley:**
$$f(x,y) = -20e^{-0.2\sqrt{\frac{1}{2}(x^2+y^2)}} - e^{\frac{1}{2}(\cos(2\pi x)+\cos(2\pi y))} + e + 20,$$
with global minimum at $(0,0)$ and $f(0,0) = 0$.

- **Rosenbrock:**
$$f(x,y) = (1-x)^2 + 5(y-x^2)^2,$$
with global minimum at $(1,1)$ and $f(1,1) = 0$.

- **Branin:**
$$f(x,y) = (y - \tfrac{5.1}{4\pi^2}x^2 + \tfrac{5}{\pi}x - 6)^2 + 10(1 - \tfrac{1}{8\pi})\cos(x) + 10,$$
with three equivalent minima at $(-\pi, 12.275)$, $(\pi, 2.275)$, and $(9.424, 2.475)$, where $f(x^*, y^*) \approx 0.398$.

- **Rastrigin:**
$$f(x,y) = 20 + (x^2 - 10\cos(2\pi x)) + (y^2 - 10\cos(2\pi y)),$$
with global minimum at $(0,0)$ and $f(0,0) = 0$.

Domains and initialization

- **Domain:** All functions are evaluated over $[-5,5]^2$.
- **Initialization for CEM and NES:** The initial means are chosen as follows:
$$\begin{aligned} \text{Ackley:} \quad & \mu_0 = (1.0,\, 1.0)^\top, \\ \text{Rosenbrock:} \quad & \mu_0 = (0.0,\, 2.0)^\top, \\ \text{Branin:} \quad & \mu_0 = (2.0,\, 2.0)^\top, \\ \text{Rastrigin:} \quad & \mu_0 = (-1.0,\, 1.0)^\top. \end{aligned}$$
- **Initialization for CMA–ES:** The initial points are chosen as follows: $x^{(0)} = \mu_0$, where meaning for μ_0 defined above.

Experiment protocol and outputs

For each algorithm and test function:

1. Run one deterministic trial (fixed random seed).
2. Record:
`function, method, x_found, f(x_found), iterations, f_evals, time_s.`
3. Produce trajectory / sampling plots:
   - For CEM and NES: scatter of sampled points with mean trajectory.
   - For CMA–ES: contour plot with population samples and mean path.

Comparative check

- Compare convergence speed and stability across CEM, NES, and CMA–ES.
- Discuss which algorithms better locate global minima (multimodal vs smooth functions).
- Comment on efficiency vs. robustness of each method.

Tasks (checklist)

1. Implement CEM, NES, and CMA–ES in Python.
2. Use the same test functions and domain setup as before.
3. Run experiments and produce the comparison table and plots.
4. Write a short (3–5 sentence) comparison discussing:
   - Which method achieved the lowest objective and why.
   - Which method was computationally cheaper.
   - Where each method is preferable (robustness vs efficiency).

Deliverables

- Python implementations of the three algorithms.
- Plots of trajectories or sampled points.
- Console table with results and runtimes.
- Short written discussion.

Additional (advanced) exercises — choose one

1. **Hybrid CEM–CMA–ES:** run CEM for several iterations to obtain a promising distribution, then initialize CMA–ES from its mean and covariance.
2. Comparison with deterministic global optimization (DIRECT): Run CEM, NES, and CMA–ES on the same 2D test functions, then compare convergence speed and final solution quality with the DIRECT algorithm. Analyze strengths and weaknesses of stochastic vs. deterministic global optimization methods.
3. Noise robustness study: Modify the objective functions by adding small Gaussian noise to the function values. Test how each algorithm copes with noisy evaluations and compare the stability of the resulting solutions.

Notes and hints

- Use deterministic seeds (`np.random.seed(42)` in Python).
- Count all function evaluations explicitly.
- Use clear logging and plotting of mean trajectories.
- Document learning rates, sample sizes, and stopping criteria used.

Comments on translating Julia code to Python (CMA–ES)

- **Direct translation is not possible without modifications.** Julia allows certain linear algebra operations (matrix powers, backslash operator, Cholesky-based inversion) to behave stably on small or ill-conditioned matrices. Python/Numpy may produce NaNs or overflow errors in the same steps.
- **Necessary adaptations:**
  1. Replace direct matrix inverses or fractional powers with `np.linalg.svd` or regularized pseudo-inverse to avoid singularity issues.
  2. Add small regularization to covariance matrix Σ at each iteration to ensure it remains positive-definite.
  3. Limit large weighted steps (δw) to prevent floating-point overflow.
  4. Wrap risky operations (e.g., multivariate normal sampling, matrix inversion) with try/except or numerical checks.

- **Impact:** These changes preserve the CMA–ES logic while ensuring Python implementation runs stably on multimodal benchmark functions.
- **Conclusion:** Translating CMA–ES from Julia to Python requires careful attention to numerical stability; a literal line-by-line translation will usually fail for nontrivial problems.