2025-02-13 Progress rate comparison Gaussian vs. Double Geometric

Comparing Mutation Distributions on a Simple Ellipsoid

In this experiment, we compare two integer mutation operators when optimizing a simple ellipsoid function

$$f(x) = \lambda_1 x_1^2 + \lambda_2 x_2^2,$$

with $\lambda_1 = 1.0$ and $\lambda_2 = 2.0$. The optimum is at $(0, 0)$.

We consider two mutation operators:

- **Double Geometric Mutation:**
  Samples integer steps from a symmetric discrete Laplace (double geometric) distribution with

  $$P(k) = \begin{cases} \frac{1-p}{1+p} & k = 0, \\ \frac{1-p}{1+p}\, p^{|k|} & k \neq 0, \end{cases}$$

  where $p = \exp(-1/\sigma)$.
- **Gaussian Mutation:**
  Samples from a bivariate Gaussian with independent coordinates (with standard deviations σ_1 and σ_2) and then rounds the results to the nearest integer.

For both operators, we assume the same domain and use integer truncation.

Tuning and Comparison

For a given parent point

$$x_{\text{parent}} = (d, d),$$

(with $d \geq 0$, so the distance to the optimum is $\sqrt{2}d$), we perform a grid search over σ_1 and σ_2 in $[0.1, 10]$ to determine the optimum expected progress rate

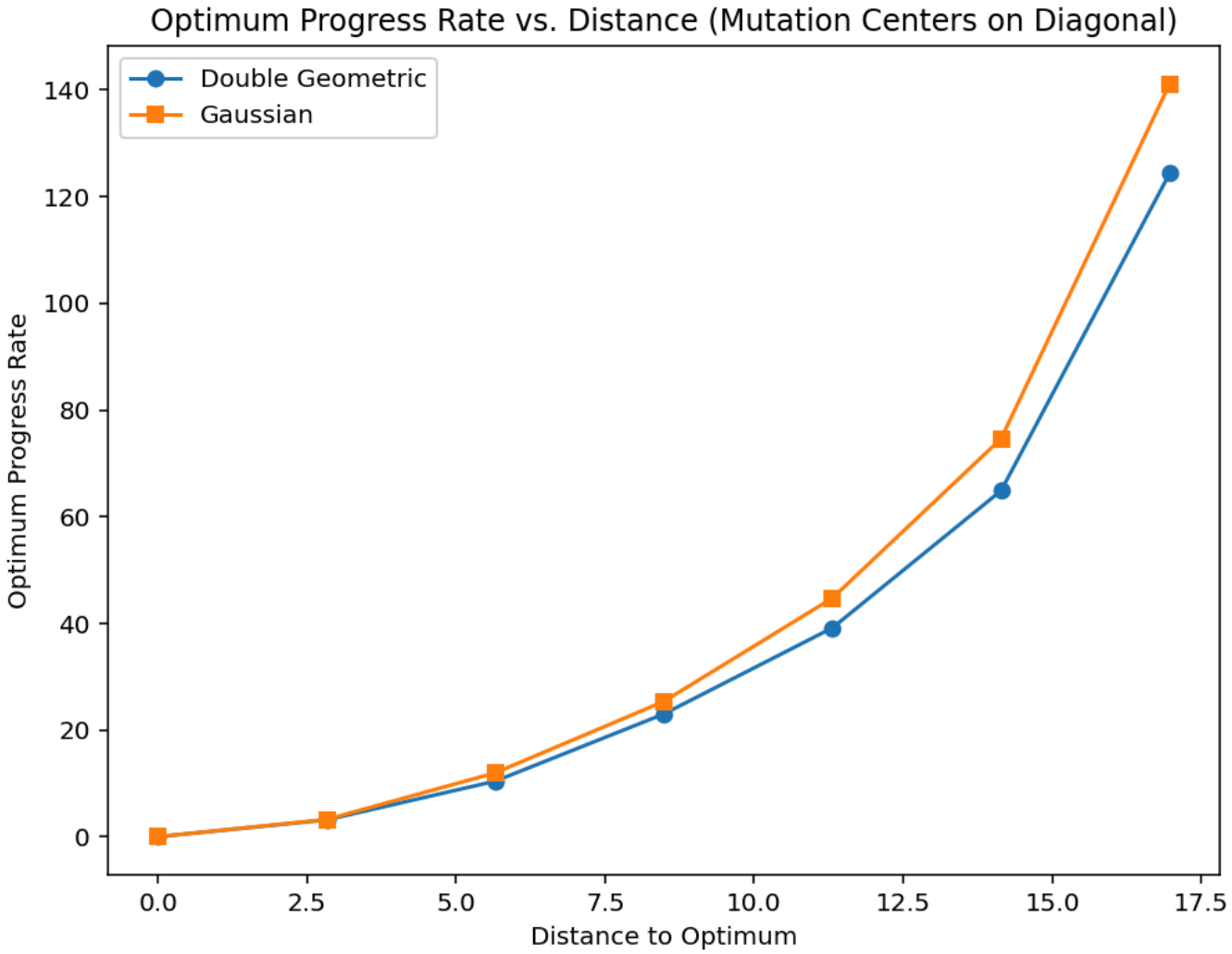
$$\Delta f = \max\{0,\, f(x_{\text{parent}}) - f(x_{\text{offspring}})\}.$$

The progress rate is averaged over many offspring generated by adding a mutation step

$$x_{\text{offspring}} = x_{\text{parent}} + (\delta_1, \delta_2),$$

with each coordinate mutated independently.

We then plot the optimum progress rates achieved by each distribution as a function of the distance from the optimum.



Below is the complete source code.

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

def ellipsoid(x, lambdas=(1.0, 2.0)):
    """
    Computes f(x) = x₁² + 2 x₂² for an integer vector x.
    """
    x = np.array(x, dtype=int)
    return lambdas[0] * x[0]**2 + lambdas[1] * x[1]**2

def sample_double_geometric(sigma, size=None):
```

```
    """
    Samples integer offsets from a symmetric double geometric (discrete
Laplace)
    distribution.

    PMF:
        P(k) = (1-p)/(1+p)        if k = 0,
               (1-p)/(1+p)*p^|k|   if k ≠ 0,
    with p = exp(-1/sigma).
    """
    p = np.exp(-1.0 / sigma)
    if size is None:
        u = np.random.rand()
        threshold = (1 - p) / (1 + p)
        if u < threshold:
            return 0
        else:
            n = np.random.geometric(p=1 - p)
            return np.random.choice([-1, 1]) * n
    else:
        u = np.random.rand(*np.atleast_1d(size))
        threshold = (1 - p) / (1 + p)
        samples = np.zeros(u.shape, dtype=int)
        idx = (u >= threshold)
        if np.any(idx):
            n = np.random.geometric(p=1 - p, size=np.sum(idx))
            samples[idx] = np.random.choice([-1, 1], size=np.sum(idx)) * n
        return samples

def sample_gaussian(sigma, size=None):
    """
    Samples from a Gaussian with mean 0 and standard deviation sigma,
    then rounds to the nearest integer.
    """
    if size is None:
        return int(np.rint(np.random.normal(0, sigma)))
    else:
        return np.rint(np.random.normal(0, sigma, size)).astype(int)

def compute_progress_rate(x, sigma1, sigma2, sample_func, num_samples=5000,
                          lambdas=(1.0, 2.0), lower_bound=-12,
upper_bound=12):
    """
    For a given parent x, estimates the expected progress rate using mutations
    sampled by sample_func with parameters sigma1 and sigma2.
```

```
    Progress is defined as:
        Δf = max{0, f(x_parent) - f(x_offspring)}.

    Mutations resulting in (0,0) are discarded and re-sampled.
    Resampling is attempted in batches and limited to at most 40 iterations.
    """
    x = np.array(x, dtype=int)
    f_parent = ellipsoid(x, lambdas)
    valid_offspring = []
    batch_size = num_samples  # generate this many mutations per iteration
    iterations = 0
    max_iterations = 40  # limit resampling iterations
    while len(valid_offspring) < num_samples and iterations < max_iterations:
        iterations += 1
        delta1 = sample_func(sigma1, size=batch_size)
        delta2 = sample_func(sigma2, size=batch_size)
        # Exclude mutations where both coordinates are zero.
        valid = ~((delta1 == 0) & (delta2 == 0))
        if np.sum(valid) > 0:
            valid_d1 = delta1[valid]
            valid_d2 = delta2[valid]
            new_offspring = np.vstack((x[0] + valid_d1, x[1] + valid_d2)).T
            new_offspring = np.clip(new_offspring, lower_bound, upper_bound)
            valid_offspring.extend(new_offspring.tolist())
    if len(valid_offspring) == 0:
        return 0
    valid_offspring = np.array(valid_offspring[:num_samples])
    f_offspring = np.array([ellipsoid(o, lambdas) for o in valid_offspring])
    improvements = f_parent - f_offspring
    improvements[improvements < 0] = 0
    return np.mean(improvements)

# ------------------------------
# Tuning Procedure with Mutation Centers on the Diagonal
# ------------------------------
# Parent centers are placed on the diagonal: x = (d, d)
d_values = np.arange(0, 13, 2)  # d = 0, 2, 4, ..., 12
sigma_range = np.linspace(0.1, 10.0, 20)  # grid for sigma1 and sigma2

opt_progress_double = []   # optimum progress for double geometric mutation
opt_progress_gaussian = [] # optimum progress for Gaussian mutation
distances = []             # Euclidean distance from the optimum

for d in d_values:
    parent = np.array([d, d], dtype=int)
    distance = np.linalg.norm(parent)
```

```
    distances.append(distance)

    progress_matrix_double = np.zeros((len(sigma_range), len(sigma_range)))
    progress_matrix_gaussian = np.zeros((len(sigma_range), len(sigma_range)))

    for i, s1 in enumerate(sigma_range):
        for j, s2 in enumerate(sigma_range):
            progress_matrix_double[i, j] = compute_progress_rate(parent, s1,
s2, sample_double_geometric, num_samples=5000)
            progress_matrix_gaussian[i, j] = compute_progress_rate(parent, s1,
s2, sample_gaussian, num_samples=5000)

    opt_progress_double.append(np.max(progress_matrix_double))
    opt_progress_gaussian.append(np.max(progress_matrix_gaussian))
    print(f"d = {d}, distance = {distance:.2f} | Double Geometric:
{np.max(progress_matrix_double):.3f}, Gaussian:
{np.max(progress_matrix_gaussian):.3f}")

plt.figure(figsize=(8, 6))
plt.plot(distances, opt_progress_double, marker='o', label='Double Geometric')
plt.plot(distances, opt_progress_gaussian, marker='s', label='Gaussian')
plt.xlabel("Distance to Optimum")
plt.ylabel("Optimum Progress Rate")
plt.title("Optimum Progress Rate vs. Distance (Mutation Centers on Diagonal)")
plt.legend()
plt.show()
```