浙江大学数学科学学院

前沿数学专题讨论 PR05

30 Nov 2022

Submitted To:
张南松老师
22-23 秋冬学期

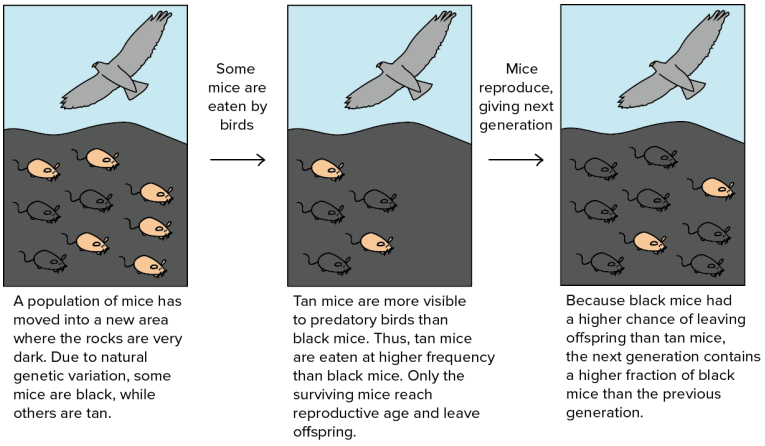
Submitted By :
吴凡
农业工程 + 统计学

Contents

1 Evolution Strategy

Evolution strategies (ES) belong to the big family of evolutionary algorithms. The optimization targets of ES are vectors of real numbers,$x \in R^n$.

Evolutionary algorithms refer to a division of population-based optimization algorithms inspired by natural selection. Natural selection believes that individuals with traits beneficial to their survival can live through generations and pass down the good characteristics to the next generation. Evolution happens by the selection process gradually and the population grows better adapted to the environment.



Evolutionary algorithms can be summarized in the following format as a general optimization solution:

Let's say we want to optimize a function f(x) and we are not able to compute gradients directly. But we still can evaluate f(x) given any x and the result is deterministic. Our belief in the probability distribution over x as a good solution to f(x) optimization is $p_\theta(x)$, parameterized by θ . The goal is to find an optimal configuration of θ .

Starting with an initial value of θ , we can continuously update θ by looping three steps as follows:

1. Generate a population of samples $D = (x_i, f(x_i))$ where $x_i \sim p_\theta(x)$

2. Evaluate the "fitness" of samples in D.
3. Select the best subset of individuals and use them to update θ,generally bases on fitness or rank.

The most basic and canonical version of evolution strategies:Simple Gaussian Evolution Strategies. It models $p_\theta(x)$ as a n-dimensional isotropic Gaussian distribution, in which θ only tracks the mean μ and standard deviation σ.

$$\theta = (\mu, \sigma), p_\theta(x) \sim N(\mu, \sigma^2 I) = \mu + \sigma N(0, I) \tag{1}$$

The process of Simple-Gaussian-ES, given $x \in R^n$:

1. Initialize $\theta = \theta^{(0)}$ and the generation counter t=0
2. Generate the offspring population of size Λ by sampling from the Gaussian distribution:

$$D^{(t+1)} = x_i^{(t+1)} | x_i^{(t+1)} = \mu^{(t)} + \sigma^{(t)} y_i^{(t+1)} where y_i^{(t+1)} \sim N(x|0, I); i = 1, \cdots, \Lambda \tag{2}$$

3. Select a top subset of λ samples with optimal $f(x_i)$ and this subset is called **elite** set. Let's label them as

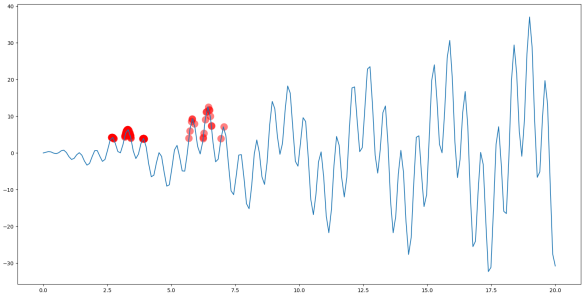
$$D_{elite}^{(t+1)} = x_i^{(t+1)} | x_i^{(t+1)} \in D^{(t+1)}, i = 1, \cdots, \lambda, \lambda \leq \Lambda \tag{3}$$

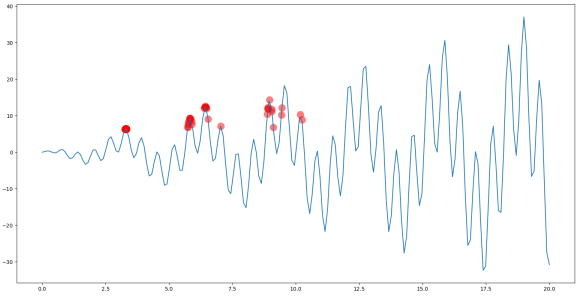
4. Then we estimate the new mean and std for the next generation using the elite set:

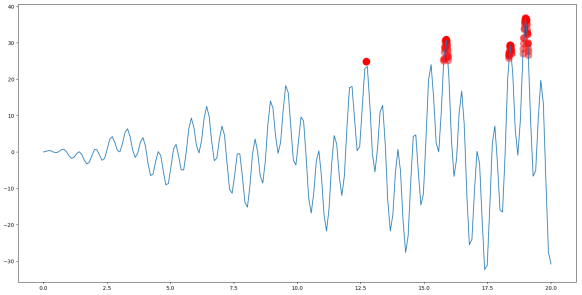
$$\mu^{(t+1)} = avg(D_{elite}^{(t+1)}) = \frac{1}{\lambda} \sum_{i=1}^{\lambda} x_i^{(t+1)} \tag{4}$$

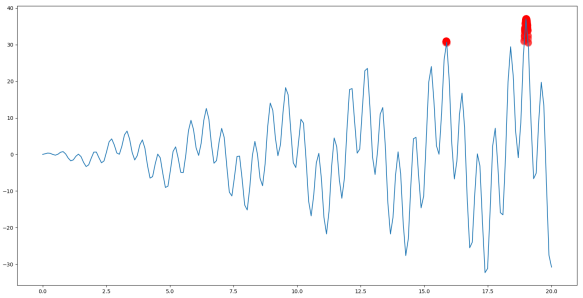
$$\sigma^{(t+1)^2} = var(D_{elite}^{(t+1)}) = \frac{1}{\lambda} \sum_{i=1}^{\lambda} (x_i^{(t+1)} - \mu^{(t)})^2 \tag{5}$$

5. Repeat steps 2-4 until the result is good enough.









2 CMA Evolution Strategy

The standard deviation σ accounts for the level of exploration: the larger σ the bigger search space we can sample our offspring population. In vanilla ES, $\sigma^{(t+1)}$ is highly correlated with σ^t, so the algorithm is not able to rapidly adjust the exploration space when needed (i.e. when the confidence level changes).

CMA-ES, short for "Covariance Matrix Adaptation Evolution Strategy" , fixes the problem by tracking pairwise dependencies between the samples in the distribution with a covariance matrix C. The new distribution parameter becomes:

$$\theta = (\mu, \sigma, C), p_\theta(x) \sim N(\mu, \sigma^2 C) \sim \mu + \sigma N(0, C) \tag{6}$$

where σ controls for the overall scale of the distribution,often knoen as step size.

Compared with ES mentioned above:

$$\theta = (\mu, \sigma), p_\theta(x) \sim N(\mu, \sigma^2 I) = \mu + \sigma N(0, I) \tag{7}$$

2.1 Updating the Mean

$$\mu^{(t+1)} = \mu^{(t)} + \alpha_\mu \frac{1}{\lambda} \sum_{i=1}^{\lambda} (x_i^{(t+1)} - \mu^{(t)}) \tag{8}$$

CMA-ES has a learning rate $\alpha_\mu \leq 1$ to control how fast the mean should be updated.

2.2 Controlling the Step Size

The sampling process can be decoupled from the mean and standard deviation:

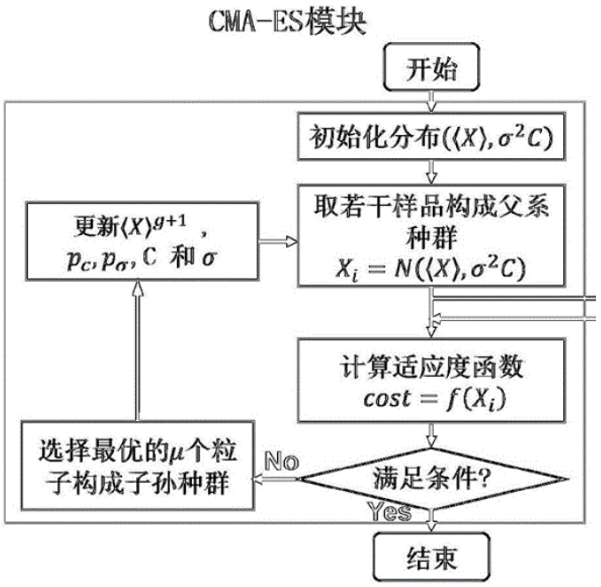
$$x_i^{(t+1)} = \mu^{(t)} + \sigma^{(t)} y_i^{(t+1)}, where y_i^{(t+1)} = \frac{x_i^{(t+1)} - \mu^{(t)}}{\sigma^{(t)}} \sim N(0, C) \tag{9}$$

The parameter σ controls the overall scale of the distribution. It is separated from the covariance matrix so that we can change steps faster than the full covariance. A larger step size leads to faster parameter update. In order to evaluate

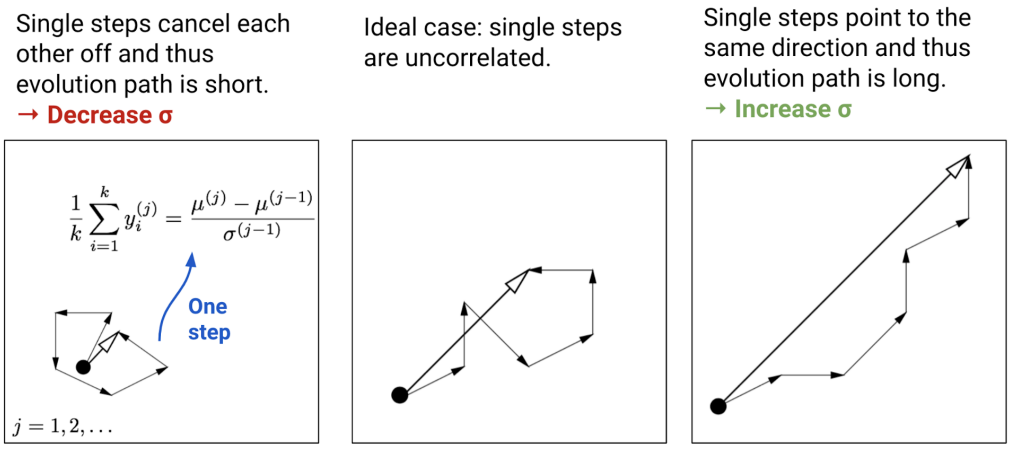
Symbol	Meaning
$x_i^{(t)} \in \mathbb{R}^n$	the i-th samples at the generation (t)
$y_i^{(t)} \in \mathbb{R}^n$	$x_i^{(t)} = \mu^{(t-1)} + \sigma^{(t-1)} y_i^{(t)}$
$\mu^{(t)}$	mean of the generation (t)
$\sigma^{(t)}$	step size
$C^{(t)}$	covariance matrix
$B^{(t)}$	a matrix of C's eigenvectors as row vectors
$D^{(t)}$	a diagonal matrix with C's eigenvalues on the diagnose.
$p_\sigma^{(t)}$	evaluation path for σ at the generation (t)
$p_c^{(t)}$	evaluation path for C at the generation (t)
α_μ	learning rate for μ's update
α_σ	learning rate for p_σ
d_σ	damping factor for σ's update
α_{cp}	learning rate for p_c
$\alpha_{c\lambda}$	learning rate for C's rank-min(λ, n) update
α_{c1}	learning rate for C's rank-1 update

Algorithm 1 CMA-ES: Covariance Matrix Adaptation Evolution Strategies

Require: α_μ, α_σ, α_{cp}, α_{c1}, $\alpha_{c\lambda}$ ▷ Learning rates
Require: d_σ ▷ Damping factor
Require: $t = 0$ ▷ Generation counter
Require: $\mu^{(0)} \in \mathbb{R}^n, \sigma \in \mathbb{E}_+$ ▷ Inputs: initial mean vectors and step size
Require: $C^{(0)} = I$, $p_\sigma^{(0)} = 0$, $p_c^{(0)} = 0$ ▷ Initialize covariance matrix and evolution paths.
repeat
Sample $x_i^{(t+1)} = \mu^{(t)} +^{(t)} y_i$ where $y_i \sim \mathcal{N}(0, C^{(t)})$, $i = 1, \ldots, \Lambda$
Select top λ samples with the best performance $x_i^{(t+1)}, i = 1, \ldots, \lambda$
$\mu^{(t+1)} \leftarrow \mu^{(t)} + \alpha_\mu \frac{1}{\lambda} \sum_{i=1}^{\lambda} (x_i^{(t+1)} - \mu^{(t)})$
$p_\sigma^{(t+1)} \leftarrow (1 - \alpha_\sigma) p_\sigma^{(t)} + \sqrt{\alpha_\sigma (2 - \alpha_\sigma) \lambda}\, C^{(t)^{-\frac{1}{2}}} \frac{\mu^{(t+1)} - \mu^{(t)}}{\sigma^{(t)}}$
$\sigma^{(t+1)} \leftarrow \sigma^{(t)} \exp\left(\frac{\alpha_\sigma}{d_\sigma}\left(\frac{\|p_\sigma^{(t+1)}\|}{\mathbb{E}\|\mathcal{N}(0,I)\|} - 1\right)\right)$
$p_c^{(t+1)} \leftarrow (1 - \alpha_{cp}) p_c^{(t)} + \sqrt{\alpha_{cp} (2 - \alpha_{cp}) \lambda}\, \frac{\mu^{(t+1)} - \mu^{(t)}}{\sigma^{(t)}}$
$C^{(t+1)} \leftarrow (1 - \alpha_{c1}) C^{(t)} + \alpha_{c1}\, p_c^{(t+1)} p_c^{(t+1)^\top}$
$C^{(t+1)} \leftarrow (1 - \alpha_{c\lambda} - \alpha_{c1}) C^{(t)} + \alpha_{c1}\, p_c^{(t+1)} p_c^{(t+1)^\top} + \alpha_{c\lambda} \frac{1}{\lambda} \sum_{i=1}^{\lambda} y_i^{(t+1)} y_i^{(t+1)^\top}$
$t \leftarrow t + 1$
until hit stopping criteria
return $\mu^{(t)}$, $\sigma^{(t)}$, $C^{(t)}$



whether the current step size is proper, CMA-ES constructs an evolution path p_σ by summing up a consecutive sequence of moving steps.



By comparing this path length with its expected length under random selection (meaning single steps are uncorrelated), we are able to adjust σ accordingly.

$$\frac{1}{\lambda}\sum_{i=1}^{\lambda} y_i^{(t+1)} = \frac{\mu^{(t+1)} - \mu^{(t)}}{\sigma^{(t)}} \tag{10}$$

$$\frac{1}{\lambda}\sum_{i=1}^{\lambda} y_i^{(t+1)} \sim \frac{1}{\lambda} N(0, \lambda C^{(t)}) \sim \frac{1}{\sqrt{\lambda}} C^{(t)^{\frac{1}{2}}} N(0, I) \tag{11}$$

$$\Longrightarrow \sqrt{\lambda} C^{(t)^{-\frac{1}{2}}} \frac{\mu^{(t+1)} - \mu^{(t)}}{\sigma^{(t)}} \sim N(0, I) \tag{12}$$

In order to assign higher weights to recent generations, we use polyak averaging to update the evolution path with learning rate α_σ. Meanwhile, the weights are balanced so that p_σ is conjugate,$\sim N(0, I)$ both before and after one update.

$$p_\sigma^{(t+1)} = (1 - \alpha_\sigma) p_\alpha^{(t)} + \sqrt{c_\sigma(2 - \alpha_\sigma)\lambda} C^{(t)^{-\frac{1}{2}}} \frac{\mu^{(t+1)} - \mu^{(t)}}{\sigma^{(t)}} \tag{13}$$

The expected length of p_σ under random selection is $E|N(0, I)|$, that is the expectation of the L2-norm of a N(0,I) random variable.

We adjust the step size according to the ratio of $\frac{||p_\sigma^{(t+1)}||}{E||N(0,I)||}$

$$ln\sigma^{(t+1)} = ln\sigma^{(t)} + \frac{\alpha_\sigma}{d_\sigma}\left(\frac{||p_\sigma^{(t+1)}||}{E||N(0, I)||} - 1\right) \tag{14}$$

$$\sigma^{(t+1)} = \sigma^{(t)} exp\left(\frac{\alpha_\sigma}{d_\sigma}\left(\frac{||p_\sigma^{(t+1)}||}{E||N(0, I)||} - 1\right)\right) \tag{15}$$

where $d_\sigma \approx 1$ is a damping parameter, scaling how fast $ln\sigma$ should be changed.

2.3 Adapting the Covariance Matrix

For the covariance matrix, it can be estimated from scratch using y_i of elite samples(recall that $y_i \sim N(0, C)$)

$$C_\lambda^{(t+1)} = \frac{1}{\lambda}\sum_{i=1}^{\lambda} y_i^{(t+1)} y_i^{(t+1)^T} \tag{16}$$

The above estimation is only reliable when the selected population is large enough. However, we do want to run fast iteration with a small population of samples in each generation. That's why CMA-ES invented a more reliable but also more complicated way to update C.

It involves two independent routes:

- **Rank-min(λ,n) update**:uses the history of C_λ, each estimated from scratch in one generation.
- **Rank-one update**:estimates the moving steps y_i and the sign information from the history.

1. For the first part:
   The first route considers the estimation of C from the entire history of C_λ. For example, if we have experienced a large number of generations, $C^{(t+1)} \approx avg(C_\lambda^{(i)}; i = 1, \cdots, t)$ would be a good estimator. Similar to p_σ, we also use polyak averaging with a learning rate to incorporate the history:

$$C^{(t+1)} = (1 - \alpha_{c\lambda})C^{(t)} + \alpha_{c\lambda}C^{(t+1)} \tag{17}$$

$$C^{(t+1)} = (1 - \alpha_{c\lambda})C^{(t)} + \alpha_{c\lambda}\frac{1}{\lambda}\sum_{i=1}^{\lambda} y_i^{(t+1)} y_i^{(t+1)^T} \tag{18}$$

   A common choice for the learning rate is $\alpha_{c\lambda} \approx min(1, \lambda/n^2)$.

2. For the second part:
   Similar to how we adjust the step size σ, an evolution path p_c is used to track the sign information and it is constructed in a way that p_c is conjugate, $\sim N(0, C)$ both before and after a new generation.

   We may consider p_c as another way to compute $avg_i(y_i)$ (notice that both $\sim N(0, C)$) while the entire history is used and the sign information is maintained.

   In the last section,we know that:

$$\sqrt{\lambda}\frac{\mu^{(t+1)} - \mu^{(t)}}{\sigma^{(t)}} \sim N(0, C) \tag{19}$$

We define:

$$p_c^{(t+1)} = (1-\alpha_{cp})p_c^{(t)} + \sqrt{1-(1-\alpha_{cp})^2}\sqrt{\lambda}\frac{\mu^{(t+1)}-\mu^{(t)}}{\sigma^{(t)}} \tag{20}$$

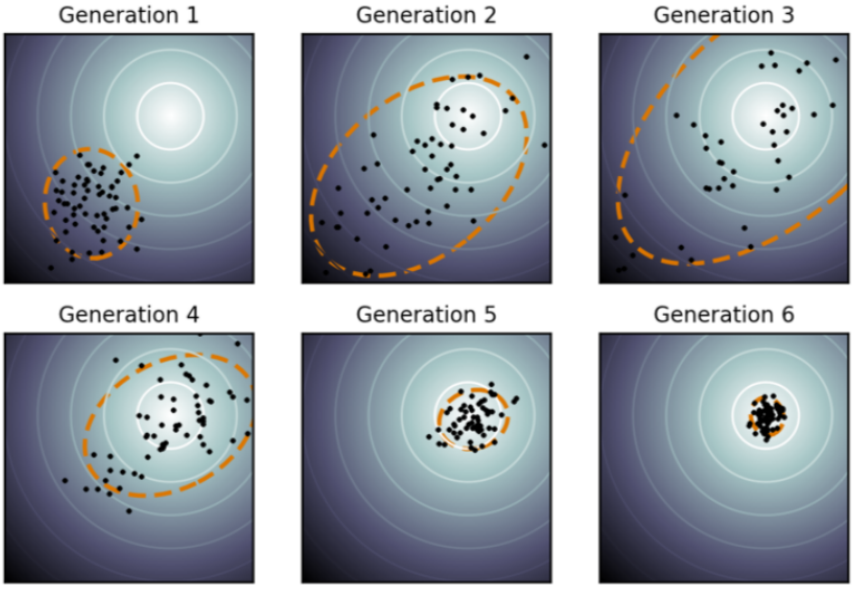
$$p_c^{(t+1)} = (1-\alpha_{cp})p_c^{(t)} + \sqrt{\alpha_{cp}(2-\alpha_{cp})\lambda}\frac{\mu^{(t+1)}-\mu^{(t)}}{\sigma^{(t)}} \tag{21}$$

Then the covariance matrix is updated according to p_c:

$$C^{(t+1)} = (1-\alpha_{c1})C^{(t)} + \alpha_{c1}p_c^{(t+1)}p_c^{(t+1)^T} \tag{22}$$

Eventually we combine two approaches together:

$$C^{(t+1)} = (1-\alpha_{c\lambda}-\alpha_{c1})C^{(t)} + \alpha_{c1}\underbrace{p_c^{(t+1)}p_c^{(t+1)\top}}_{\text{rank-one update}} + \alpha_{c\lambda}\underbrace{\frac{1}{\lambda}\sum_{i=1}^{\lambda} y_i^{(t+1)}y_i^{(t+1)\top}}_{\text{rank-min(lambda, n) update}}$$



Black dots are samples in one generation. The samples are more spread out initially but when the model has higher confidence in finding a good solution in the late stage, the samples become very concentrated over the global optimum.

3 Appendix: Code

3.1 ES

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

DNA_SIZE = 1              # DNA (real number)
DNA_BOUND = [0, 20]        # solution upper and lower bounds
N_GENERATIONS = 200
POP_SIZE = 100            # population size
N_KID = 50                # n kids per generation


def F(x): return np.sin(10*x)*x + np.cos(2*x)*x     # to find the
                                    maximum of this function


# find non-zero fitness for selection
def get_fitness(pred): return pred.flatten()


def make_kid(pop, n_kid):
    # generate empty kid holder
    kids = {'DNA': np.empty((n_kid, DNA_SIZE))}
    kids['mut_strength'] = np.empty_like(kids['DNA'])
    for kv, ks in zip(kids['DNA'], kids['mut_strength']):
        # crossover (roughly half p1 and half p2)
        p1, p2 = np.random.choice(np.arange(POP_SIZE), size=2, replace=
                                            False)
        cp = np.random.randint(0, 2, DNA_SIZE, dtype=np.bool)  #
                                            crossover points
        kv[cp] = pop['DNA'][p1, cp]
        kv[~cp] = pop['DNA'][p2, ~cp]
        ks[cp] = pop['mut_strength'][p1, cp]
        ks[~cp] = pop['mut_strength'][p2, ~cp]

        # mutate (change DNA based on normal distribution)
```

```
        ks[:] = np.maximum(ks + (np.random.rand(*ks.shape)-0.5), 0.)
                                                # must > 0
        kv += ks * np.random.randn(*kv.shape)
        kv[:] = np.clip(kv, *DNA_BOUND)    # clip the mutated value
    return kids


def kill_bad(pop, kids):
    # put pop and kids together
    for key in ['DNA', 'mut_strength']:
        pop[key] = np.vstack((pop[key], kids[key]))

    fitness = get_fitness(F(pop['DNA']))            # calculate global
                                            fitness
    idx = np.arange(pop['DNA'].shape[0])
    good_idx = idx[fitness.argsort()][-POP_SIZE:]   # selected by
                                            fitness ranking (not value)
    for key in ['DNA', 'mut_strength']:
        pop[key] = pop[key][good_idx]
    return pop


pop = dict(DNA=5 * np.random.rand(1, DNA_SIZE).repeat(POP_SIZE, axis=0)
                                        ,   # initialize the pop DNA values
           mut_strength=np.random.rand(POP_SIZE, DNA_SIZE))
                                                                    #
                                                initialize the pop
                                                mutation strength values

plt.ion()       # something about plotting
x = np.linspace(*DNA_BOUND, 200)
plt.plot(x, F(x))

for _ in range(N_GENERATIONS):
    # something about plotting
    if 'sca' in globals(): sca.remove()
    sca = plt.scatter(pop['DNA'], F(pop['DNA']), s=200, lw=0, c='red',
                                            alpha=0.5); plt.pause(0.05)
```

```
    # ES part
    kids = make_kid(pop, N_KID)
    pop = kill_bad(pop, kids)   # keep some good parent for elitism

plt.ioff(); plt.show()
```

3.2 CMA-ES

```
import numpy as np
import numpy.linalg as la


from matplotlib import cm
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

import cma

np.random.seed(0)

"""
The Algorithm

Paramters
Population size  : N
Covariance Matrix: C
mean of best k   : mu

1. Sample N from multivariate normal distribution
2. Calculate fitness and

"""

def rastrigin(X, A=10):
  return A + np.sum((X**2 - A * np.cos(2 * np.pi * X)), -1)
```

```
def rosenbrock(X, a=1, b=100):
  x,y = X[...,0], X[...,1]
  return (a-x)**2 + b*(y-x**2)**2

#(1-x)^2+100(y-x^2)^2


def _plot_points(X, low=-10, high=10, size=1024):
  """ Plots the points on same scale as image """
  tmp = (X - low)/(high-low) * size
  plt.scatter(tmp[:,0], tmp[:,1], color='black', alpha=.5)

if __name__ == '__main__':

  low, high, size = -6, 6, 2048
  spacing = np.linspace(low, high, size)

  grid = np.stack(np.meshgrid(spacing, spacing), -1)
  function = lambda X: rastrigin(X + [3,2])   # Move the min away
                                              from (0,0)
  #function = rosenbrock
  Z = function(grid)



  n = 200      # Population size
  d = 2        # Dimensions
  k = 50       # Size of elite population

  X = np.random.normal(0,1.24, (d, n))

  for i in range(24):
    # Minimize this function
    fitness = function(X.T)
    arg_topk = np.argsort(fitness)[:k]
    topk = X[:,arg_topk]

    #print(f'Iter {i}, score {fitness[arg_topk[0]]}, X = {X[:,
```

```
                                arg_topk[0]]}')
    # Covariance of topk but using mean of entire population
    centered = topk - X.mean(1, keepdims=True)
    C = (centered @ centered.T)/(k-1)
    # Eigenvalue decomposition
    w, E = la.eigh(C)
    # Generate new population
    # Sample from multivariate gaussian with mean of topk
    N = np.random.normal(size=(d,n))
    X = topk.mean(1,keepdims=True) + (E @ np.diag(np.sqrt(w)) @ N)
    if i % 1 == 0:
      print(f'iter {i}, z= {fitness[arg_topk[0]]:.2f}, x= {X[:,
                                arg_topk[0]].round(2)}')
      plt.clf()
      plt.imshow(Z, cmap='Oranges')
      _plot_points(X.T, low, high, size)
      plt.pause(.2)
      plt.draw()
      plt.savefig(f'plots/fig-{i}.jpg')
```