Portfolio optimization

source

Portfolio allocation vector

In this example we show how to do portfolio optimization using CVXPY. We begin with the basic definitions. In portfolio optimization we have some amount of money to invest in any of n different assets. We choose what fraction w_i of our money to invest in each asset i, $i = 1, \ldots, n$.

We call $w \in \mathbf{R}^n$ the *portfolio allocation vector*. We of course have the constraint that $\mathbf{1}^T w = 1$. The allocation $w_i < 0$ means a *short position* in asset i, or that we borrow shares to sell now that we must replace later. The allocation $w \geq 0$ is a *long only* portfolio. The quantity

$$\|w\|_1 = \mathbf{1}^T w_+ + \mathbf{1}^T w_-$$

is known as *leverage*.

Asset returns

We will only model investments held for one period. The initial prices are $p_i > 0$. The end of period prices are $p_i^+ > 0$. The asset (fractional) returns are $r_i = (p_i^+ - p_i)/p_i$. The porfolio (fractional) return is $R = r^T w$.

A common model is that r is a random variable with mean $\mathbf{E}r = \mu$ and covariance $\mathbf{E(r-\mu)(r-\mu)^T} = \Sigma$. It follows that R is a random variable with $\mathbf{E}R = \mu^T w$ and $\mathbf{var}(R) = w^T\Sigma w$. $\mathbf{E}R$ is the (mean) *return* of the portfolio. $\mathbf{var}(R)$ is the *risk* of the portfolio. (Risk is also sometimes given as $\mathbf{std}(R) = \sqrt{\mathbf{var}(R)}$.)

Portfolio optimization has two competing objectives: high return and low risk.

Classical (Markowitz) portfolio optimization

Classical (Markowitz) portfolio optimization solves the optimization problem

$$\begin{array}{ll} \text{maximize} & \mu^T w - \gamma w^T\Sigma w \\ \text{subject to} & \mathbf{1}^T w = 1, \quad w \in \mathcal{W}, \end{array}$$

where $w \in \mathbf{R}^n$ is the optimization variable, $\mathcal{W}$ is a set of allowed portfolios (e.g., $\mathcal{W} = \mathbf{R}^n_+$ for a long only portfolio), and $\gamma > 0$ is the *risk aversion parameter*.

The objective $\mu^T w - \gamma w^T\Sigma w$ is the *risk-adjusted return*. Varying γ gives the optimal *risk-return trade-off*. We can get the same risk-return trade-off by fixing return and minimizing risk.

Example

In the following code we compute and plot the optimal risk-return trade-off for 10 assets, restricting ourselves to a long only portfolio.

In [ ]:

```
# Generate data for long only portfolio optimization.
import numpy as np
np.random.seed(1)
n = 10
mu = np.abs(np.random.randn(n, 1))
Sigma = np.random.randn(n, n)
Sigma = Sigma.T @ Sigma
```

In [ ]:

```
# Long only portfolio optimization.
import cvxpy as cp


w = cp.Variable(n)
gamma = cp.Parameter(nonneg=True)
ret = mu.T @ w
risk = cp.quad_form(w, Sigma)
prob = cp.Problem(cp.Minimize(gamma*risk - ret),
               [cp.sum(w) == 1,
                w >= 0])
```

In [ ]:

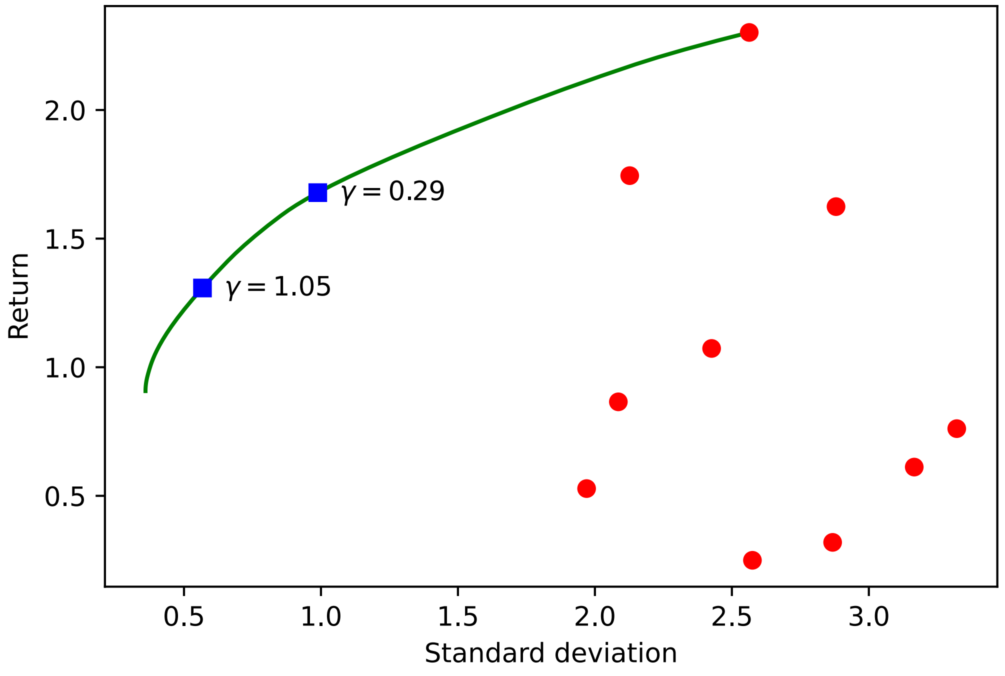
```
# Compute trade-off curve.
from tqdm.auto import tqdm
SAMPLES = 100
risk_data = np.zeros(SAMPLES)
ret_data = np.zeros(SAMPLES)
gamma_vals = np.logspace(-2, 3, num=SAMPLES)
for i in tqdm(range(SAMPLES)):
    gamma.value = gamma_vals[i]
    prob.solve()
    risk_data[i] = cp.sqrt(risk).value
    ret_data[i] = ret.value
```

```
100%|██████████| 100/100 [00:00<00:00, 478.73it/s]
```

In [ ]:

```
# Plot long only trade-off curve.
import matplotlib.pyplot as plt
%matplotlib inline
%config InlineBackend.figure_format = 'svg'

markers_on = [29, 40]
fig = plt.figure()
ax = fig.add_subplot(111)
plt.plot(risk_data, ret_data, 'g-')
for marker in markers_on:
    plt.plot(risk_data[marker], ret_data[marker], 'bs')
    ax.annotate(r"$\gamma = %.2f$" % gamma_vals[marker], xy=(risk_data[marker
for i in range(n):
    plt.plot(cp.sqrt(Sigma[i,i]).value, mu[i], 'ro')
plt.xlabel('Standard deviation')
plt.ylabel('Return')
plt.show()
```



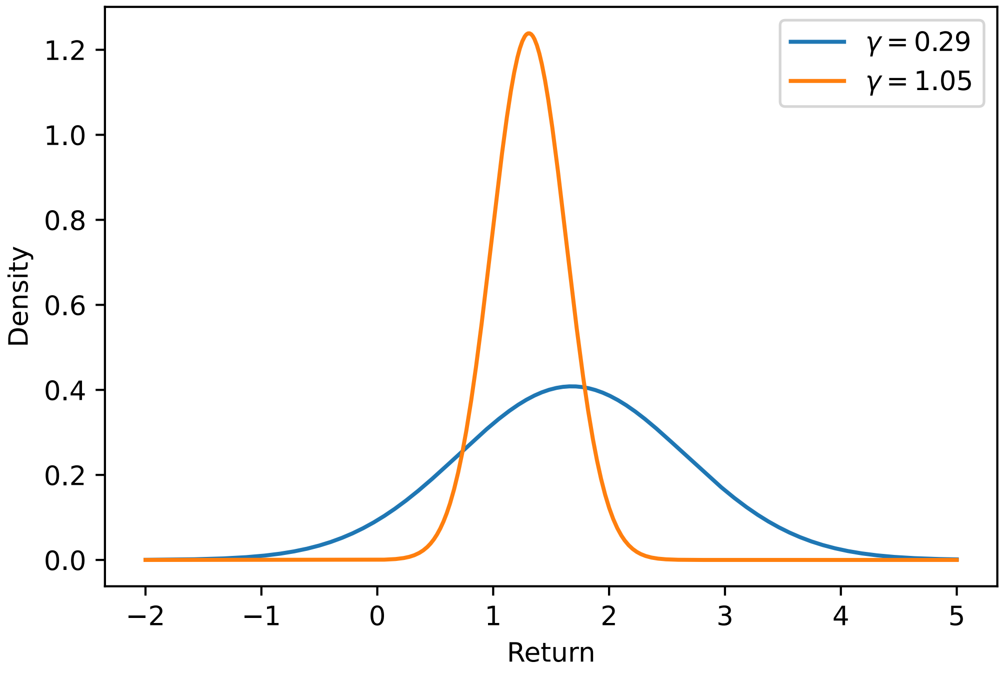
We plot below the return distributions for the two risk aversion values marked on the trade-off curve. Notice that the probability of a loss is near 0 for the low risk value and far above 0 for the high risk value.

In [ ]:

```
# Plot return distributions for two points on the trade-off curve.
import scipy.stats as spstats


plt.figure()
for midx, idx in enumerate(markers_on):
    gamma.value = gamma_vals[idx]
    prob.solve()
    x = np.linspace(-2, 5, 1000)
    plt.plot(x, spstats.norm.pdf(x, ret.value, risk.value), label=r"$\gamma =

plt.xlabel('Return')
plt.ylabel('Density')
plt.legend(loc='upper right')
plt.show()
```



Portfolio constraints

There are many other possible portfolio constraints besides the long only constraint. With no constraint ($\mathcal{W} = \mathbf{R}^n$), the optimization problem has a simple analytical solution. We will look in detail at a *leverage limit*, or the constraint that $\|w\|_1 \leq L^{\max}$.

Another interesting constraint is the *market neutral* constraint $m^T \Sigma w = 0$, where m_i is the capitalization of asset i. $M = m^T r$ is the *market return*, and $m^T \Sigma w = \mathbf{cov}(M, R)$. The market neutral constraint ensures that the portfolio return is uncorrelated with the market return.

Example

In the following code we compute and plot optimal risk-return trade-off curves for leverage limits of 1, 2, and 4. Notice that more leverage increases returns and allows greater risk.

In [ ]:

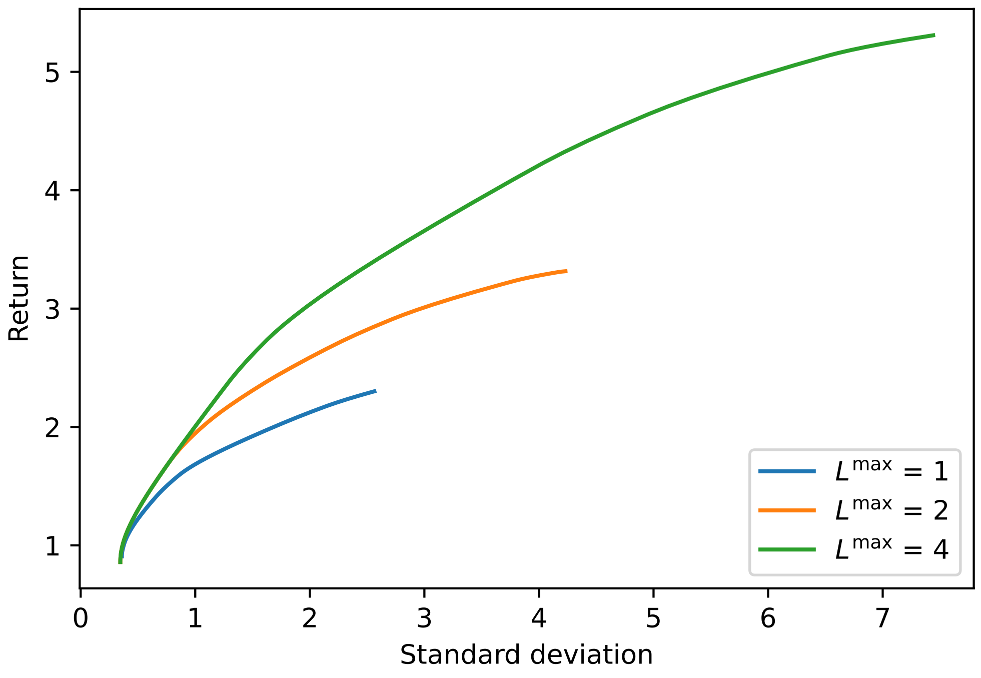
```
# Portfolio optimization with leverage limit.
Lmax = cp.Parameter()
prob = cp.Problem(cp.Maximize(ret - gamma*risk),
               [cp.sum(w) == 1,
                cp.norm(w, 1) <= Lmax])
```

In [ ]:

```
# Compute trade-off curve for each leverage limit.
L_vals = [1, 2, 4]
SAMPLES = 100
risk_data = np.zeros((len(L_vals), SAMPLES))
ret_data = np.zeros((len(L_vals), SAMPLES))
gamma_vals = np.logspace(-2, 3, num=SAMPLES)
w_vals = []
for k, L_val in enumerate(L_vals):
    for i in range(SAMPLES):
        Lmax.value = L_val
        gamma.value = gamma_vals[i]
        prob.solve(solver=cp.CVXOPT)
        risk_data[k, i] = cp.sqrt(risk).value
        ret_data[k, i] = ret.value
```

In [ ]:

```
# Plot trade-off curves for each leverage limit.
for idx, L_val in enumerate(L_vals):
    plt.plot(risk_data[idx,:], ret_data[idx,:], label=r"$L^{\max}$ = %d" % L_
for w_val in w_vals:
    w.value = w_val
    plt.plot(cp.sqrt(risk).value, ret.value, 'bs')
plt.xlabel('Standard deviation')
plt.ylabel('Return')
plt.legend(loc='lower right')
plt.show()
```



We next examine the points on each trade-off curve where $w^T \Sigma w = 2$. We plot the amount of each asset held in each portfolio as bar graphs. (Negative holdings indicate a short position.) Notice that some assets are held in a long position for the low leverage portfolio but in a short position in the higher leverage portfolios.

In [ ]:

```
# Portfolio optimization with a leverage limit and a bound on risk.
prob = cp.Problem(cp.Maximize(ret),
              [cp.sum(w) == 1,
               cp.norm(w, 1) <= Lmax,
               risk <= 2])
```

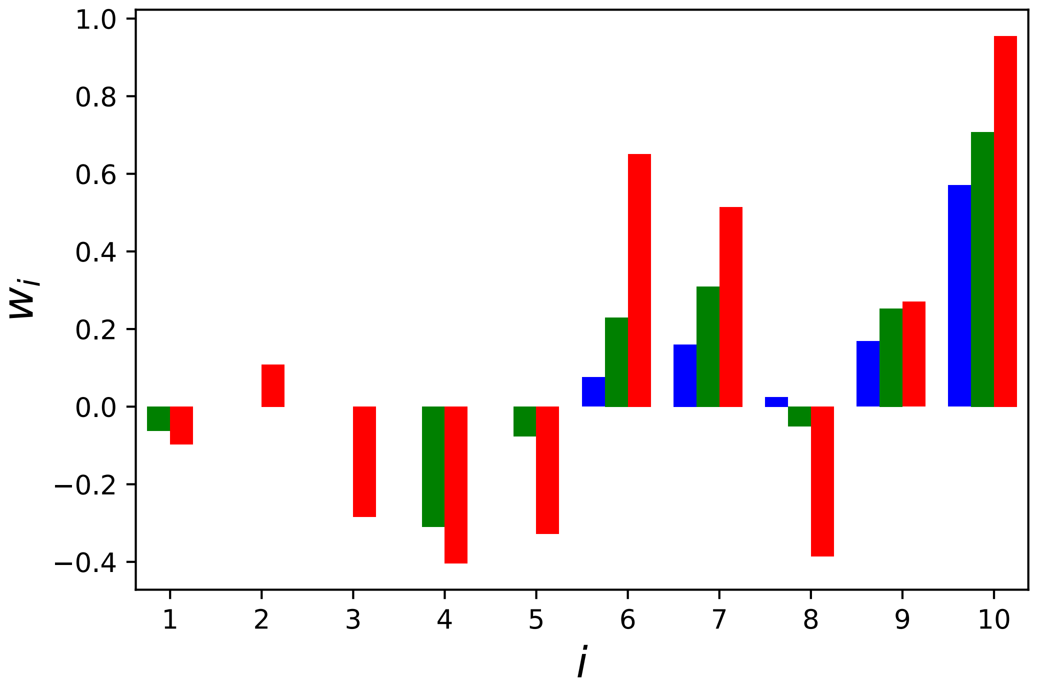
In [ ]:

```
# Compute solution for different leverage limits.
for k, L_val in enumerate(L_vals):
    Lmax.value = L_val
    prob.solve()
    w_vals.append( w.value )
```

In [ ]:

```
# Plot bar graph of holdings for different leverage limits.
colors = ['b', 'g', 'r']
```

```
indices = np.argsort(mu.flatten())
for idx, L_val in enumerate(L_vals):
     plt.bar(np.arange(1,n+1) + 0.25*idx - 0.375, w_vals[idx][indices], color
             label=r"$L^{\max}$ = %d" % L_val, width = 0.25)
plt.ylabel(r"$w_i$", fontsize=16)
plt.xlabel(r"$i$", fontsize=16)
plt.xlim([1-0.375, 10+.375])
plt.xticks(np.arange(1,n+1))
plt.show()
```



Variations

There are many more variations of classical portfolio optimization. We might require that $\mu^T w \geq R^{\min}$ and minimize $w^T \Sigma w$ or $\|\Sigma^{1/2} w\|_2$. We could include the (broker) cost of short positions as the penalty $s^T(w)_-$ for some $s \geq 0$. We could include transaction costs (from a previous portfolio w^{prev}) as the penalty

$$\kappa^T |w - w^{\text{prev}}|^\eta, \quad \kappa \geq 0.$$

Common values of η are $\eta = 1,\ 3/2,\ 2.$

Factor covariance model

A particularly common and useful variation is to model the covariance matrix Σ as a factor model

$$\Sigma = F \tilde{\Sigma} F^T + D,$$

where $F \in \mathbf{R}^{n \times k}$, $k \ll n$ is the *factor loading matrix*. k is the number of factors (or sectors) (typically 10s). F_{ij} is the loading of asset i to factor j. D is a diagonal matrix; $D_{ii} > 0$ is the *idiosyncratic risk*. $\tilde{\Sigma} > 0$ is the *factor covariance matrix*.

$F^T w \in \mathbf{R}^k$ gives the portfolio *factor exposures*. A portfolio is *factor j neutral* if $(F^T w)_j = 0$.

Portfolio optimization with factor covariance model

Using the factor covariance model, we frame the portfolio optimization problem as

$$\begin{array}{ll} \text{maximize} & \mu^T w - \gamma\left(f^T\tilde{\Sigma}f + w^T D w\right) \\ \text{subject to} & \mathbf{1}^T w = 1, \quad f = F^T w \\ & w \in \mathcal{W}, \quad f \in \mathcal{F}, \end{array}$$

where the variables are the allocations $w \in \mathbf{R}^n$ and factor exposures $f \in \mathbf{R}^k$ and $\mathcal{F}$ gives the factor exposure constraints.

Using the factor covariance model in the optimization problem has a computational advantage. The solve time is $O(nk^2)$ versus $O(n^3)$ for the standard problem.

Example

In the following code we generate and solve a portfolio optimization problem with 50 factors and 3000 assets. We set the leverage limit $= 2$ and $\gamma = 0.1$.

We solve the problem both with the covariance given as a single matrix and as a factor model. Using CVXPY with the OSQP solver running in a single thread, the solve time was 173.30 seconds for the single matrix formulation and 0.85 seconds for the factor model formulation. We collected the timings on a MacBook Air with an Intel Core i7 processor.

In [ ]:

```
# Generate data for factor model.
n = 3000
m = 50
np.random.seed(1)
mu = np.abs(np.random.randn(n, 1))
Sigma_tilde = np.random.randn(m, m)
Sigma_tilde = Sigma_tilde.T.dot(Sigma_tilde)
D = np.diag(np.random.uniform(0, 0.9, size=n))
F = np.random.randn(n, m)
```

In [ ]:

```
# Factor model portfolio optimization.
w = cp.Variable(n)
f = F.T*w
gamma = cp.Parameter(nonneg=True)
Lmax = cp.Parameter()
ret = mu.T*w
risk = cp.quad_form(f, Sigma_tilde) + cp.quad_form(w, D)
prob_factor = cp.Problem(cp.Maximize(ret - gamma*risk),
                     [cp.sum(w) == 1,
                      cp.norm(w, 1) <= Lmax])

# Solve the factor model problem.
Lmax.value = 2
gamma.value = 0.1
prob_factor.solve(verbose=True)
```

```
==============================================================================
=
                                     CVXPY
                                     v1.2.0
==============================================================================
=
(CVXPY) Mar 24 01:28:51 PM: Your problem has 3000 variables, 2 constraints, an
d 2 parameters.
/Users/bratishka/anaconda3/lib/python3.9/site-packages/cvxpy/expressions/expre
ssion.py:593: UserWarning:
This use of ``*`` has resulted in matrix multiplication.
Using ``*`` for matrix multiplication has been deprecated since CVXPY 1.1.
    Use ``*`` for matrix-scalar and vector-scalar multiplication.
    Use ``@`` for matrix-matrix and matrix-vector multiplication.
    Use ``multiply`` for elementwise multiplication.
This code path has been hit 1 times so far.

  warnings.warn(msg, UserWarning)
/Users/bratishka/anaconda3/lib/python3.9/site-packages/cvxpy/expressions/expre
ssion.py:593: UserWarning:
This use of ``*`` has resulted in matrix multiplication.
Using ``*`` for matrix multiplication has been deprecated since CVXPY 1.1.
    Use ``*`` for matrix-scalar and vector-scalar multiplication.
    Use ``@`` for matrix-matrix and matrix-vector multiplication.
    Use ``multiply`` for elementwise multiplication.
This code path has been hit 2 times so far.

  warnings.warn(msg, UserWarning)
(CVXPY) Mar 24 01:28:51 PM: It is compliant with the following grammars: DCP,
DQCP
(CVXPY) Mar 24 01:28:51 PM: CVXPY will first compile your problem; then, it wi
ll invoke a numerical solver to obtain a solution.
------------------------------------------------------------------------------
-
                                  Compilation
------------------------------------------------------------------------------
-
(CVXPY) Mar 24 01:28:51 PM: Compiling problem (target solver=OSQP).
(CVXPY) Mar 24 01:28:51 PM: Reduction chain: FlipObjective -> CvxAttr2Constr -
> Qp2SymbolicQp -> QpMatrixStuffing -> OSQP
(CVXPY) Mar 24 01:28:51 PM: Applying reduction FlipObjective
(CVXPY) Mar 24 01:28:51 PM: Applying reduction CvxAttr2Constr
(CVXPY) Mar 24 01:28:51 PM: Applying reduction Qp2SymbolicQp
(CVXPY) Mar 24 01:28:51 PM: Applying reduction QpMatrixStuffing
(CVXPY) Mar 24 01:28:51 PM: Applying reduction OSQP
(CVXPY) Mar 24 01:28:51 PM: Finished problem compilation (took 1.366e-01 secon
ds).
(CVXPY) Mar 24 01:28:51 PM: (Subsequent compilations of this problem, using th
e same arguments, should take less time.)
------------------------------------------------------------------------------
-
                                Numerical solver
------------------------------------------------------------------------------
-
(CVXPY) Mar 24 01:28:51 PM: Invoking solver OSQP  to obtain a solution.
-----------------------------------------------------------------
           OSQP v0.6.2  -  Operator Splitting QP Solver
              (c) Bartolomeo Stellato,  Goran Banjac
        University of Oxford  -  Stanford University 2021
-----------------------------------------------------------------
problem:  variables n = 6050, constraints m = 6052
          nnz(P) + nnz(A) = 172325
settings: linear system solver = qdldl,
          eps_abs = 1.0e-05, eps_rel = 1.0e-05,
          eps_prim_inf = 1.0e-04, eps_dual_inf = 1.0e-04,
          rho = 1.00e-01 (adaptive),
          sigma = 1.00e-06, alpha = 1.60, max_iter = 10000
          check_termination: on (interval 25),
          scaling: on, scaled_termination: off
          warm start: on, polish: on, time_limit: off
```

```
iter   objective    pri res    dua res    rho        time
   1  -2.1359e+03   7.63e+00   3.73e+02   1.00e-01   2.38e-02s
 200  -4.1946e+00   1.59e-03   7.86e-03   3.60e-01   1.82e-01s
 400  -4.6288e+00   3.02e-04   6.01e-04   3.60e-01   3.18e-01s
 600  -4.6444e+00   2.20e-04   7.87e-04   3.60e-01   4.55e-01s
 800  -4.6230e+00   1.09e-04   3.70e-04   3.60e-01   5.91e-01s
1000  -4.6223e+00   8.59e-05   1.04e-04   3.60e-01   7.27e-01s
1200  -4.6205e+00   8.56e-05   9.35e-06   3.60e-01   8.65e-01s
1400  -4.6123e+00   6.44e-05   1.54e-04   3.60e-01   1.00e+00s
1575  -4.6064e+00   2.97e-05   4.06e-05   3.60e-01   1.12e+00s

status:               solved
solution polish:      unsuccessful
number of iterations: 1575
optimal objective:    -4.6064
run time:             1.14e+00s
optimal rho estimate: 3.87e-01

-------------------------------------------------------------------------------
-
                                    Summary
-------------------------------------------------------------------------------
-
(CVXPY) Mar 24 01:28:52 PM: Problem status: optimal
(CVXPY) Mar 24 01:28:52 PM: Optimal value: 4.606e+00
(CVXPY) Mar 24 01:28:52 PM: Compilation took 1.366e-01 seconds
(CVXPY) Mar 24 01:28:52 PM: Solver (including time spent in interface) took 1.
144e+00 seconds
```

Out[ ]:
```
4.606413077728827
```

In [ ]:
```python
# Standard portfolio optimization with data from factor model.
risk = cp.quad_form(w, F.dot(Sigma_tilde).dot(F.T) + D)
prob = cp.Problem(cp.Maximize(ret - gamma*risk),
               [cp.sum(w) == 1,
                cp.norm(w, 1) <= Lmax])

# Uncomment to solve the problem.
# WARNING: this will take many minutes to run.
prob.solve(verbose=True, max_iter=30000)
```

```
===============================================================================
=
                                     CVXPY
                                     v1.2.0
===============================================================================
=
(CVXPY) Mar 24 01:28:54 PM: Your problem has 3000 variables, 2 constraints, an
d 2 parameters.
```

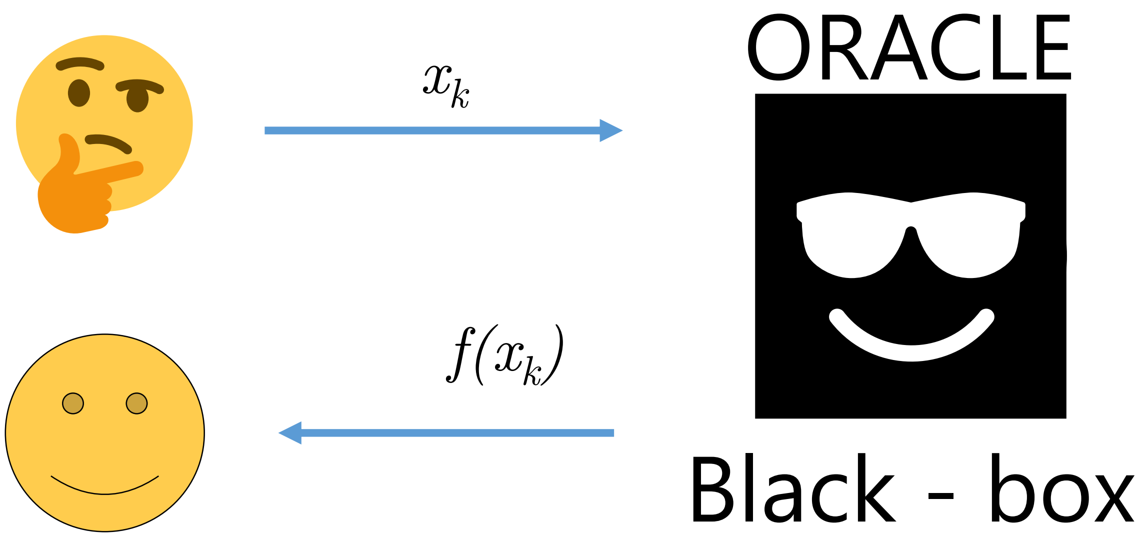
In [ ]:
```python
print('Factor model solve time = {}'.format(prob_factor.solver_stats.solve_ti
print('Single model solve time = {}'.format(prob.solver_stats.solve_time))
```

```
Factor model solve time = 2.1817036670000003
Single model solve time = 447.57964334400003
```

Materials

- Portfolio Optimization Algo Trading colab notebook
- Multi objective portfolio optimization

Zero order methods



Now we have only zero order information from the oracle. Typical speed of convegence of these methods is sublinear. A lot of methods are referred both to zero order methods and global optimization.

Code

- Global optimization illustration - Open in Colab
- Nevergrad library - Open in Colab

Simulated annealing

Problem

We need to optimize the global optimum of a given function on some space using only the values of the function in some points on the space.

$$\min_{x \in X} F(x) = F(x^*)$$

Simulated Annealing is a probabilistic technique for approximating the global optimum of a given function.

Algorithm

The name and inspiration come from annealing in metallurgy, a technique involving heating and controlled cooling of a material to increase the size of its crystals and reduce their defects. Both are attributes of the material that depend on its thermodynamic free energy. Heating and cooling the material affects both the temperature and the thermodynamic free energy. The simulation of annealing can be used to find an approximation of a global minimum for a function with many variables.

Steps of the Algorithm

Step 1 Let $k = 0$ - current iteration, $T = T_k$ - initial temperature.

Step 2 Let $x_k \in X$ - some random point from our space

Step 3 Let decrease the temperature by following rule $T_{k+1} = \alpha T_k$ where $0 < \alpha < 1$ - some constant that often is closer to 1

Step 4 Let $x_{k+1} = g(x_k)$ - the next point which was obtained from previous one by some random rule. It is usually assumed that this rule works so that each subsequent approximation should not differ very much.

Step 5 Calculate $\Delta E = E(x_{k+1}) - E(x_k)$, where $E(x)$ - the function that determines the energy of the system at this point. It is supposed that energy has the minimum in desired value x^*.

Step 6 If $\Delta E < 0$ then the approximation found is better than it was. So accept x_{k+1} as new started point at the next step and go to the step **Step 3**

Step 7 If $\Delta E >= 0$, then we accept x_{k+1} with the probability of $P(\Delta E) = \exp^{-\Delta E/T_k}$. If we don't accept x_{k+1}, then we let $k = k + 1$. Go to the step **Step 3**

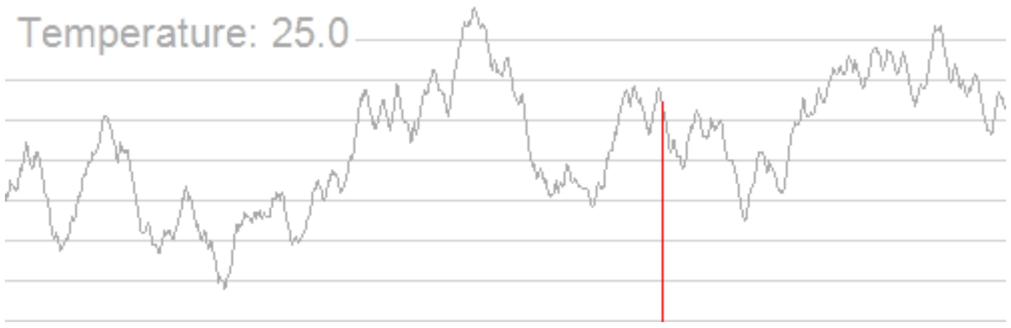
The algorithm can stop working according to various criteria, for example, achieving an optimal state or lowering the temperature below a predetermined level T_{min}.

Convergence

As it mentioned in Simulated annealing: a proof of convergence the algorithm converges almost surely to a global maximum.

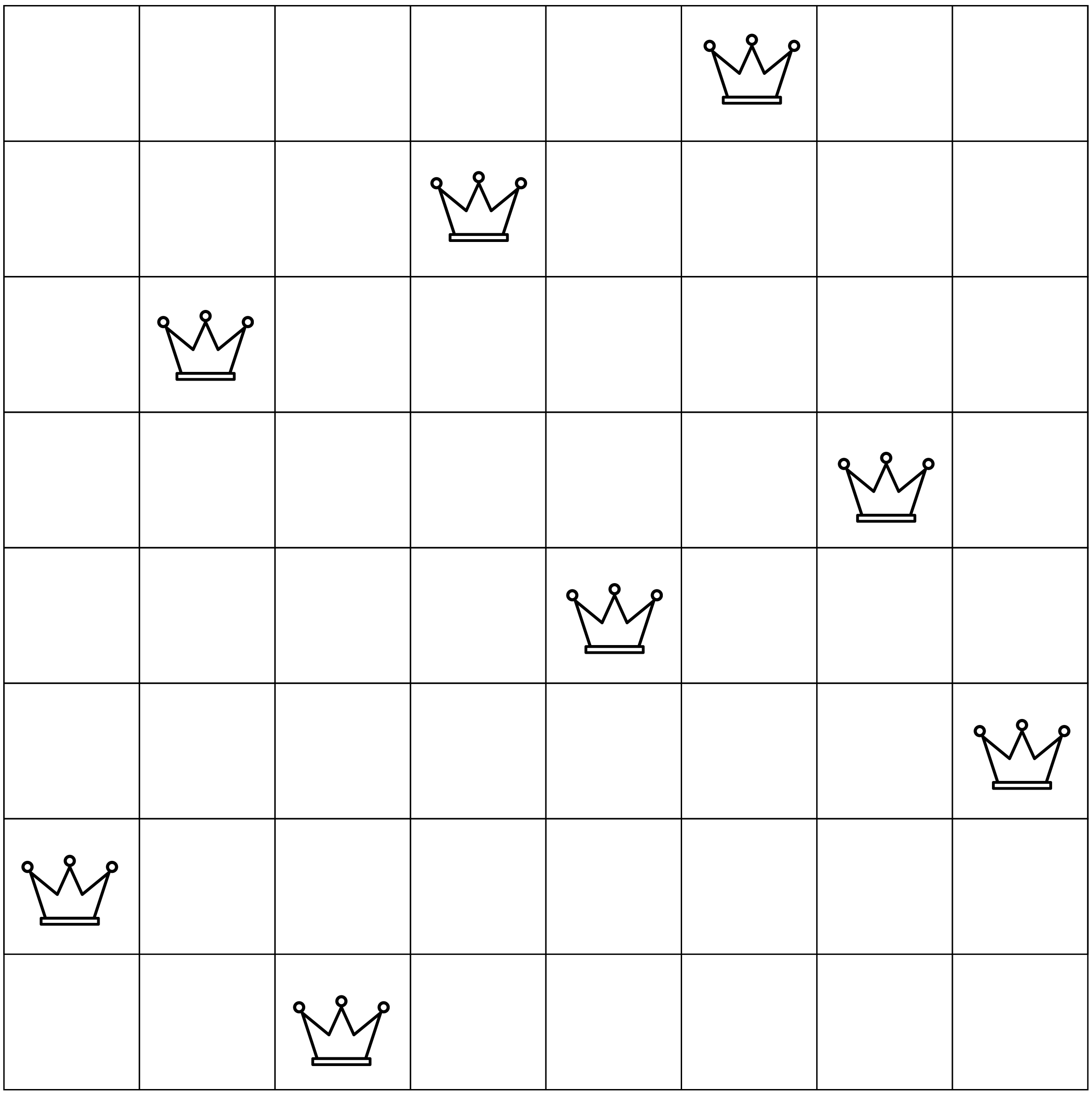
Illustration

A gif from Wikipedia:



Example

In our example we solve the N queens puzzle - the problem of placing N chess queens on an N×N chessboard so that no two queens threaten each other.



The Problem

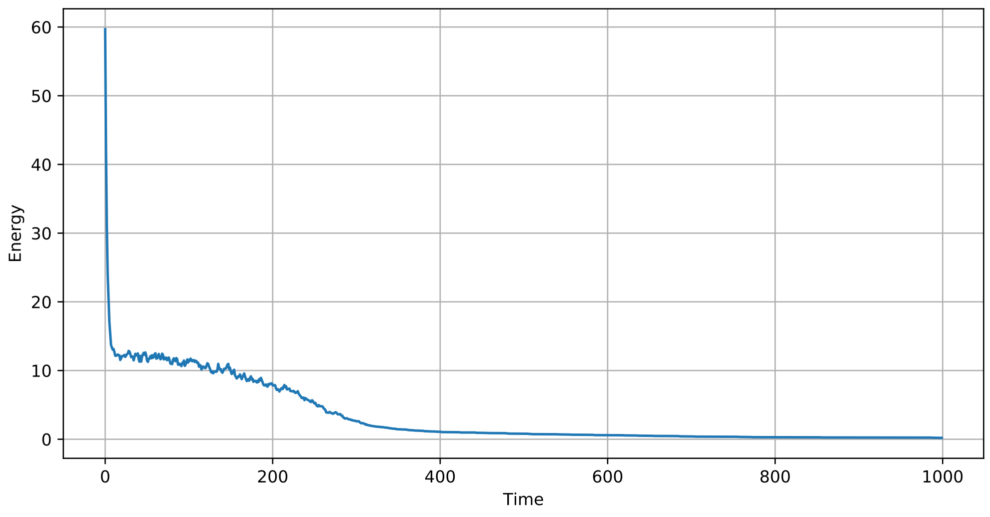
Let $E(x)$ - the number of intersections, where x - the array of placement queens at the field (the number in array means the column, the index of the number means the row).

The problem is to find x^* where $E(x^*) = \min_{x \in X} E(x)$ - the global minimum, that is predefined and equals to 0 (no two queens threaten each other).

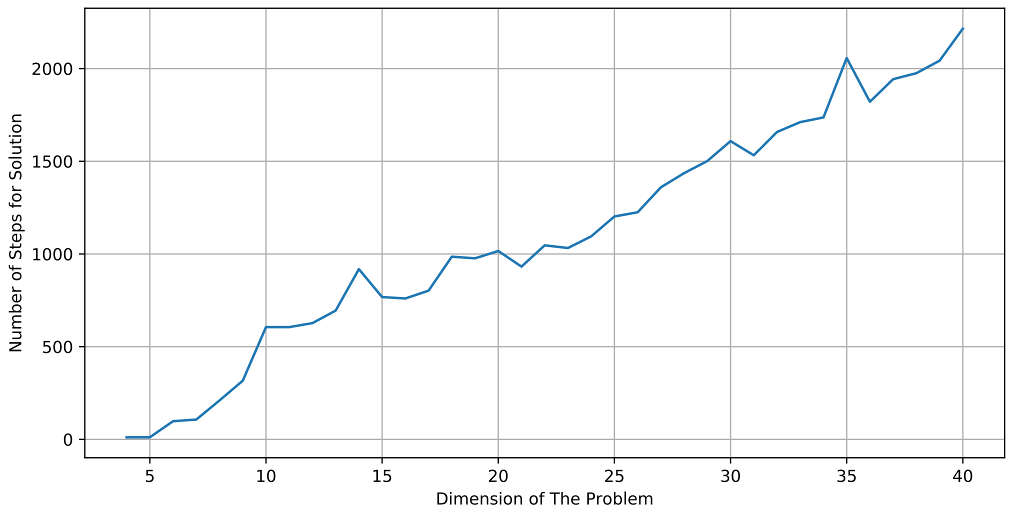
In this code $x_0 = [0, 1, 2, \ldots, N]$ that means all queens are placed at the board's diagonal . So at the beginning $E = N(N-1)$, because every queen intersects others.

Results

Results of applying this algorithm with $\alpha = 0.95$ to the N queens puzzle for $N = 10$ averaged by 100 runs are below:



Results of running the code for N from 4 to 40 and measuring the time it takes to find the solution averaged by 100 runs are below:

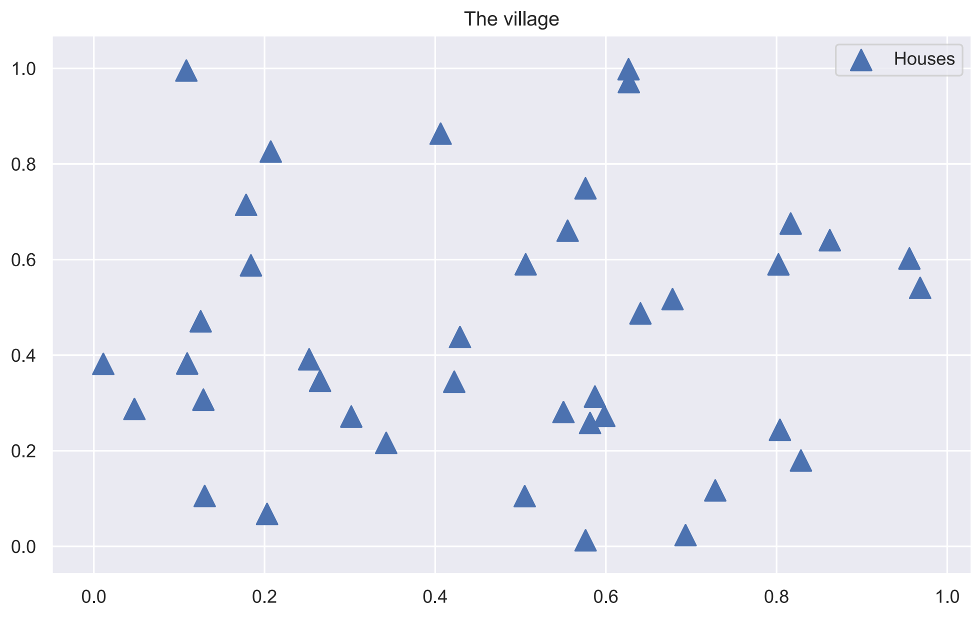


Open in Colab

Genetic algorithm

Problem

Suppose, we have N points in $\mathbb{R}^d$ Euclidian space (for simplicity we'll consider and plot case with $d = 2$). Let's imagine, that these points are nothing else but houses in some 2d village. Salesman should find the shortest way to go through the all houses only once.



That is, very simple formulation, however, implies NP - hard problem with the factorial growth of possible combinations. The goal is to minimize the following cumulative distance:

$$d = \sum_{i=1}^{N-1} \|x_{y(i+1)} - x_{y(i)}\|_2 \to \min_{y},$$

where x_k is the k-th point from N and y stands for the N- dimensional vector of indicies, which describes the order of path. Actually, the problem could be formulated as an LP problem, which is easier to solve.

🧬 Genetic (evolution) algorithm

Our approach is based on the famous global optimization algorithm, known as evolution algorithm.

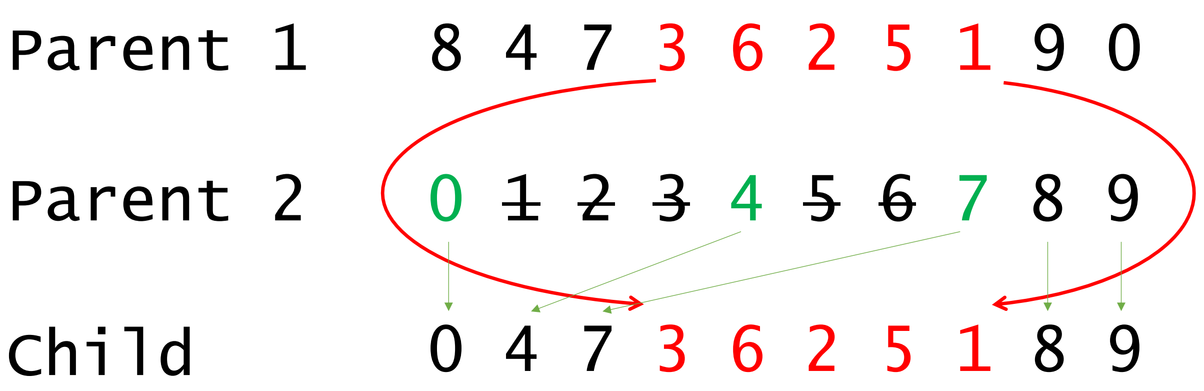
Population and individuals

Firstly we need to generate the set of random solutions as an initialization. We will call a set of solutions $\{y_k\}_{k=1}^n$ as *population*, while each solution is called *individual* (or creature).

Each creature contains integer numbers $1, \ldots, N$, which indicates the order of bypassing all the houses. The creature, that reflects the shortest path length among the others will be used as an output of an algorithm at the current iteration (generation).

Crossing procedure

Each iteration of the algorithm starts with the crossing (breed) procedure. Formally speaking, we should formulate the mapping, that takes two creature vectors as an input and returns its offspring, which inherits parents properties, while remaining consistent. We will use ordered crossover as such procedure.

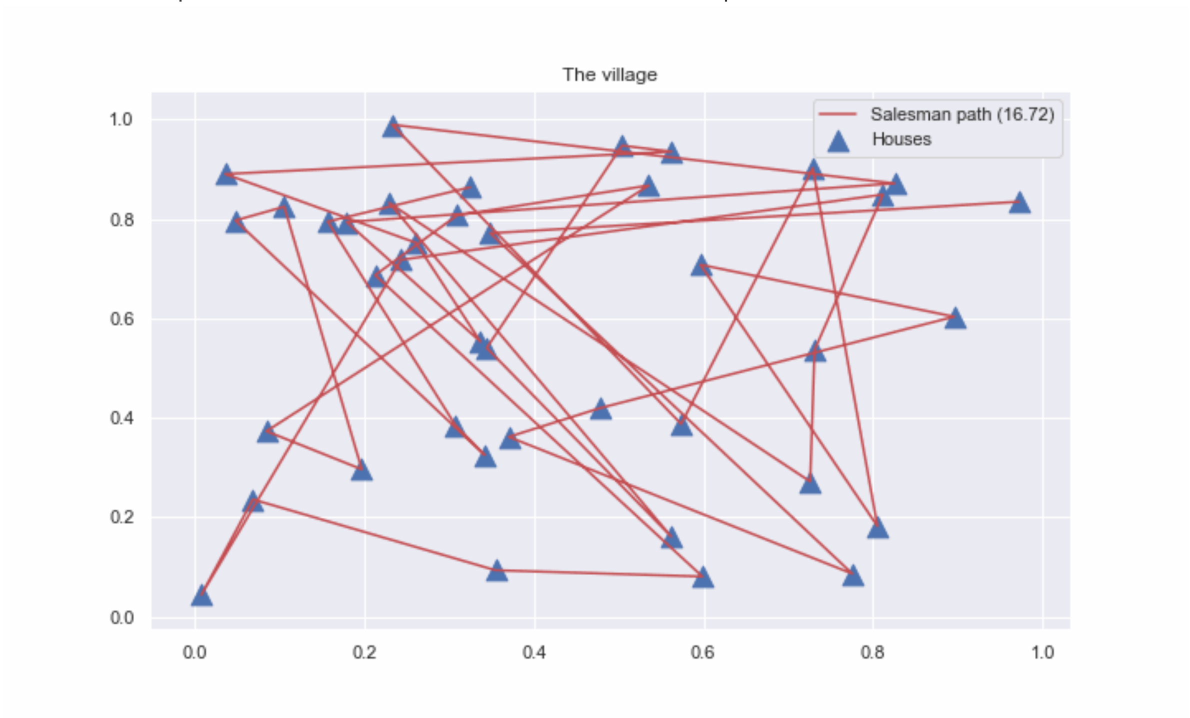


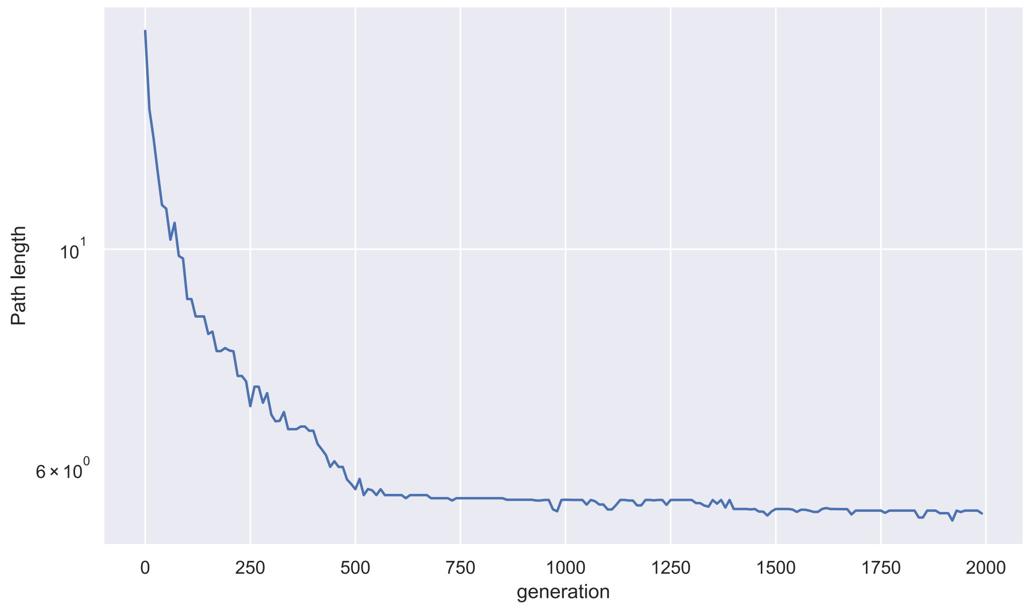
Mutation

In order to give our algorithm some ability to escape local minima we provide it with mutation procedure. We simply swap some houses in an individual vector. To be more accurate, we define mutation rate (say, 0.05). On the one hand, the higher the rate, the less stable the population is, on the other, the smaller the rate, the more often algorithm gets stuck in the local minima. We choose $\text{mutation rate} \cdot n$ individuals and in each case swap random $\text{mutation rate} \cdot N$ digits.

Selection

At the end of the iteration we have increased populatuion (due to crossing results), than we just calculate total path distance to each individual and select top n of them.





In general, for any $c > 0$, where d is the number of dimensions in the Euclidean space, there is a polynomial-time algorithm that finds a tour of length at most $(1 + \frac{1}{c})$ times the optimal for geometric instances of TSP in

$$\mathcal{O}\left(N(\log N)^{(\mathcal{O}(c\sqrt{d}))^{d-1}}\right)$$

Code

Open in Colab

References

- General information about genetic algorithms
- Wiki