Introduction to Algorithms for Data Mining and Machine Learning

Chapter 3: Optimization Algorithms

Xin-She Yang

For details, please read the book:

Xin-She Yang, **Introduction to Algorithms for Data Mining and Machine Learning**, Academic Press/Elsevier, (2019).

Almost Everything is Optimization

Almost everything is optimization ... or needs optimization ...

- Maximize efficiency, accuracy, profit, performance, sustainability, ...
- Minimize costs, wastage, energy consumption, travel distance/time, CO_2 emission, impact on environment, ...

Mathematical Optimization

Objectives: maximize or minimize $\boldsymbol{f}(\boldsymbol{x}) = [f_1(\boldsymbol{x}), f_2(\boldsymbol{x}), ..., f_m(\boldsymbol{x})]$,

$$\boldsymbol{x} = (x_1, x_2, ..., x_D) \in \mathbb{R}^D,$$

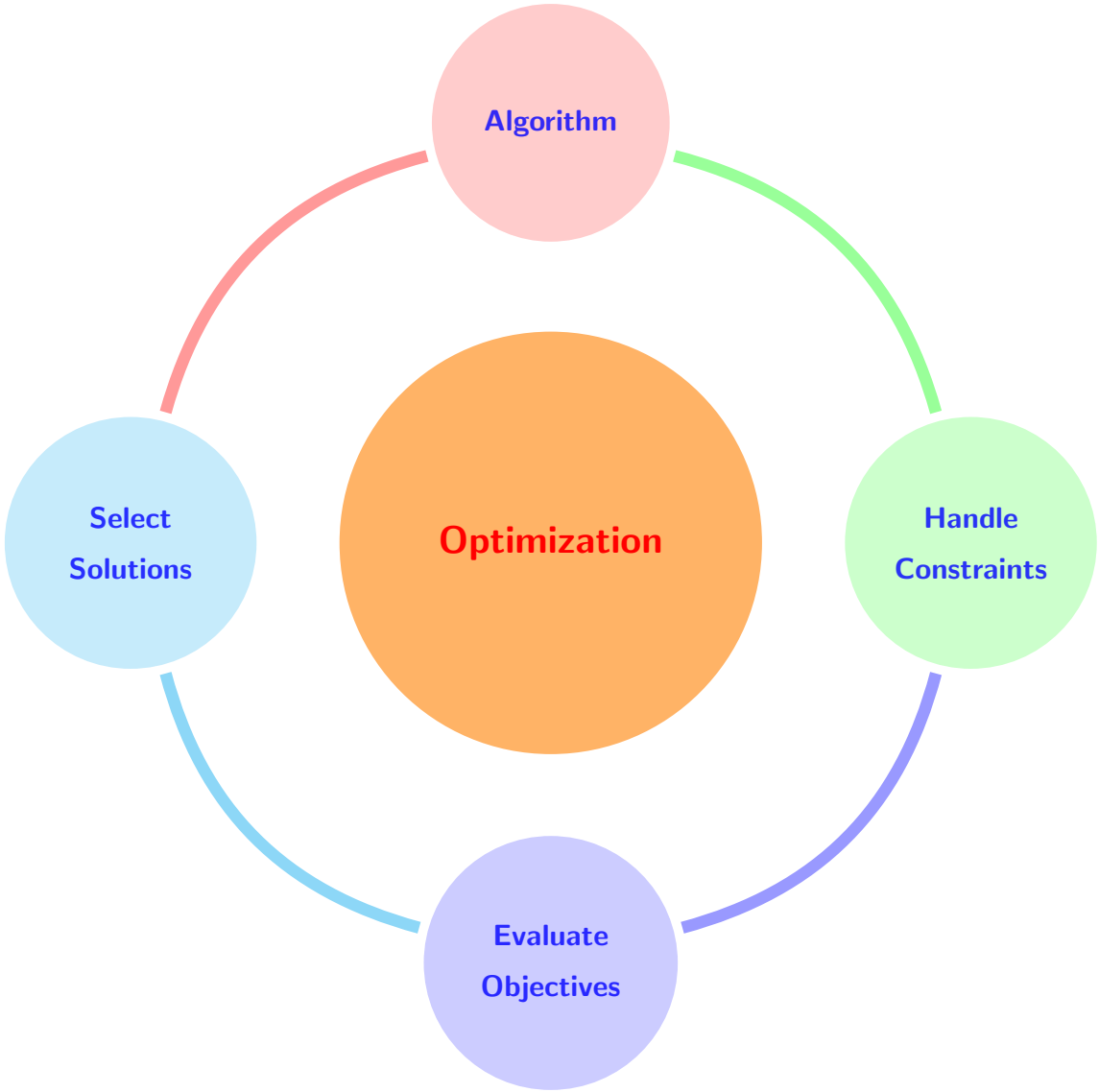
subject to multiple equality and/or inequality design constraints:

$$h_i(\boldsymbol{x}) = 0, \quad (i = 1, 2, ..., M),$$

$$g_j(\boldsymbol{x}) \leq 0, \quad (j = 1, 2, ..., N).$$

In case of $m = 1$, it becomes a single-objective optimization problem.

Key Components for Optimization



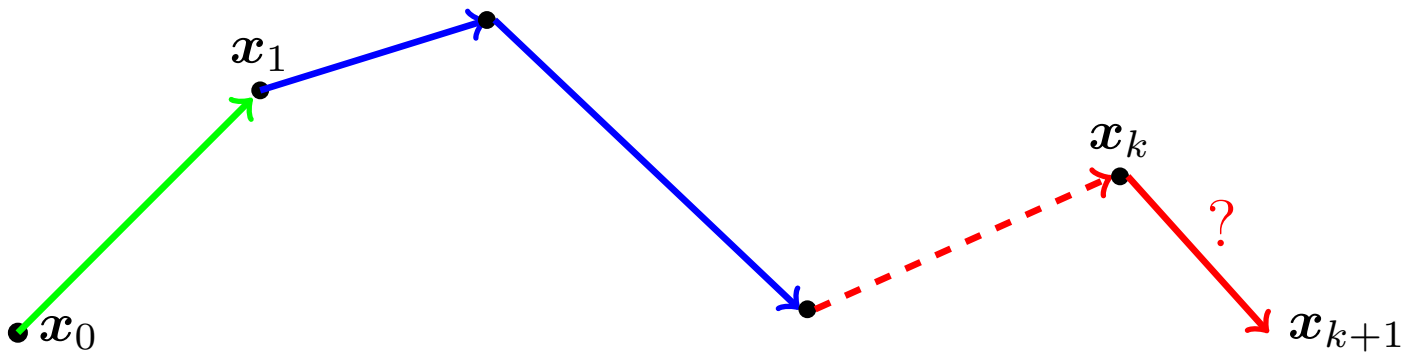
Essence of an Algorithm (Revision)

Algorithm = Iterative Procedure

For a given problem (e.g., optimization), a solution is represented as a vector $\boldsymbol{x}$. To obtain the correct solution, we usually start with an initial guess $\boldsymbol{x}_0$ and try to modify the solution iteratively.

To generate a better solution vector or point $\boldsymbol{x}_{k+1}$ from an existing solution $\boldsymbol{x}_k$ at iteration k, we have

$$\boldsymbol{x}_{k+1} \leftarrow \boldsymbol{x}_k \ \ (\text{modification}).$$



Different algorithms

Different ways for generating new solutions!

Optimization Algorithms

Deterministic

- Newton's method (1669, published in 1711), Newton-Raphson (1690), hill-climbing/steepest descent (Cauchy 1847), least-squares (Gauss 1795),
- Linear programming (Dantzig 1947), conjugate gradient (Lanczos et al. 1952), interior-point method (Karmarkar 1984), etc.

Stochastic

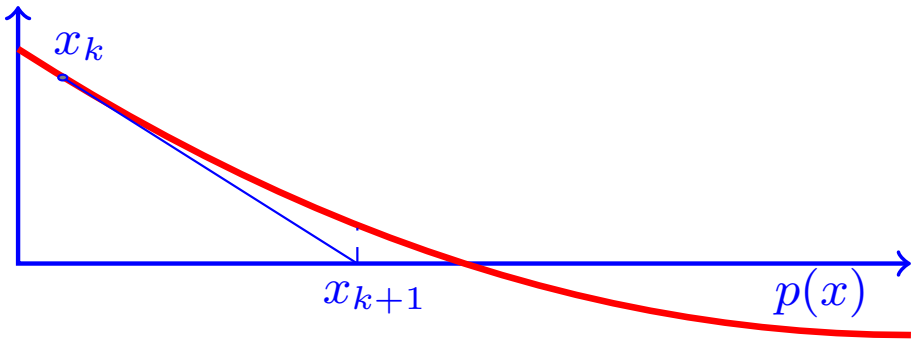
Genetic algorithm (GA), ant colony optimization (ACO), particle swarm optimization (PSO), differential evolution (DE), cuckoo search (CS), firefly algorithm (FA), bat algorithm (BA), flower pollination algorithm (FPA), etc.

Steepest Descent/Hill Climbing

Gradient-Based Methods

Use gradient/derivative information – very efficient for local search.

Newton's Method



Newton's root-finding algorithm

For a polynomial $p(x) = 0$, its roots can be computed by

$$x_{k+1} = x_k - \frac{p(x_k)}{p'(x_k)}, \quad \text{starting from } x_0,$$

where $p'(x)$ is the first derivative of $p(x)$. It usually works well if $p'(x_k) \neq 0$.

Newton's method for optimization

For minimization and maximization of a univariate function $f(x)$, it is equivalent to finding the roots of its gradient $p(x) = f'(x) = 0$. So, we have

$$x_{k+1} = x_k - \frac{p(x_k)}{p'(x_k)} = x_k - \frac{f'(x_k)}{f''(x_k)},$$

where we have also used $f''(x)$, implicitly assuming that these derivatives exist.

Example 1

For a simple function $f(x) = (x-1)^2 = x^2 - 2x + 1$ in the real domain $x \in \mathbb{R}$, we know its global minimum is $f_{\min} = 0$ at $x_* = 1$.

Let us use Newton's formula to find this solution starting from any value $x_0 = a > 0$, we have $f'(x) = 2x - 2$ and $f''(x) = 2$. From

$$x_{k+1} = x_k - \frac{f'(x_k)}{f''(x_k)}),$$

we have

$$x_1 = x_0 - \frac{2x_0 - 2}{2} = a - \frac{2a-2}{2} = 1,$$

which reaches optimality in a single step. This shows that this algorithm is very efficient.

However, this is a special case because the objective $f(x)$ is convex and quadratic. In general, it needs more steps, but still very efficient.

Dependence on x_0

Though Newton's method works well, it has a serious drawback: the final solution may depend on the initial starting point x_0. In some cases, solutions will not converge at all.

Let us demonstrate this dependence using an example ...

Example 2

Function $f(x) = x^2 \exp(-x^2)$ has two maxima at $x_* = \pm 1$ and one minimum at $x_* = 0$.

We know $f'(x) = 2x(1-x^2)\mathrm{e}^{-x^2}$ and $f''(x) = 2\mathrm{e}^{-x^2}(1 - 5x^2 + 2x^4)$. Thus, Newton's iterative formula becomes

$$x_{k+1} = x_k - \frac{f'(x_k)}{f''(x_k)} = x_k - \frac{2x_k(1-x_k^2)\mathrm{e}^{-x_k^2}}{2\mathrm{e}^{-x_k^2}(1-5x_k^2+2x_k^4)} = x_k - \frac{x_k(1-x_k^2)}{1-5x_k^2+2x_k^4},$$

where we have used $\exp(-x_k^2) \neq 0$.

Example 2

Function $f(x) = x^2 \exp(-x^2)$ has two maxima at $x_* = \pm 1$ and one minimum at $x_* = 0$.

We know $f'(x) = 2x(1-x^2)\mathrm{e}^{-x^2}$ and $f''(x) = 2\mathrm{e}^{-x^2}(1 - 5x^2 + 2x^4)$. Thus, Newton's iterative formula becomes

$$x_{k+1} = x_k - \frac{f'(x_k)}{f''(x_k)} = x_k - \frac{2x_k(1-x_k^2)\mathrm{e}^{-x_k^2}}{2\mathrm{e}^{-x_k^2}(1-5x_k^2+2x_k^4)} = x_k - \frac{x_k(1-x_k^2)}{1-5x_k^2+2x_k^4},$$

where we have used $\exp(-x_k^2) \neq 0$.
If we start with $x_0 = 0.8$, we have

$$x_1 = 0.8 - \frac{0.8 \times (1 - 0.8^2)}{1 - 5 \times 0.8^2 + 2 \times 0.8^4} \approx 1.0085747,$$

$$x_2 = 0.999961, \quad x_3 = 0.9999999,$$

which is very close to $x_* = 1.0$. From $x_0 = 0.4$, we can get -1 easily.

Example 2

Function $f(x) = x^2 \exp(-x^2)$ has two maxima at $x_* = \pm 1$ and one minimum at $x_* = 0$.

We know $f'(x) = 2x(1-x^2)\mathrm{e}^{-x^2}$ and $f''(x) = 2\mathrm{e}^{-x^2}(1 - 5x^2 + 2x^4)$. Thus, Newton's iterative formula becomes

$$x_{k+1} = x_k - \frac{f'(x_k)}{f''(x_k)} = x_k - \frac{2x_k(1-x_k^2)\mathrm{e}^{-x_k^2}}{2\mathrm{e}^{-x_k^2}(1-5x_k^2+2x_k^4)} = x_k - \frac{x_k(1-x_k^2)}{1-5x_k^2+2x_k^4},$$

where we have used $\exp(-x_k^2) \neq 0$.
If we start with $x_0 = 0.8$, we have

$$x_1 = 0.8 - \frac{0.8 \times (1 - 0.8^2)}{1 - 5 \times 0.8^2 + 2 \times 0.8^4} \approx 1.0085747,$$

$$x_2 = 0.999961, \quad x_3 = 0.9999999,$$

which is very close to $x_* = 1.0$. From $x_0 = 0.4$, we can get -1 easily.
If we use $x_0 = 0.5$, we have

$$x_1 = 3.5, \quad x_2 \approx 3.66414799, \quad x_3 \approx 3.818812, \tag{1}$$

which gradually moves towards infinity. In this case, the iterative sequence becomes divergent. It never reaches the optimal solution $x_* = 1.0$.

Newton's Method – Higher Dimensions

For a multivariate function $f(\boldsymbol{x})$ where $\boldsymbol{x} = (x_1, x_2, ..., x_n)^T$ in the n-dimensional space, the maxima or minima occur at $f'(\boldsymbol{x}) = 0$. That is, the optimality is reached at the roots of the stationary or critical condition $f'(\boldsymbol{x}) = 0$.

Newton's Method for optimization

$$\boldsymbol{x}_{t+1} = \boldsymbol{x}_t - \frac{f'(\boldsymbol{x}_t)}{\nabla^2 f(\boldsymbol{x}_t)} = \boldsymbol{x}_t - \boldsymbol{H}^{-1}\nabla f,$$

$$\nabla f = \begin{pmatrix} \frac{\partial f}{\partial x_1} \\ \vdots \\ \frac{\partial f}{\partial x_n} \end{pmatrix}, \quad \boldsymbol{H} = \nabla^2 f = \begin{pmatrix} \frac{\partial^2 f}{\partial {x_1}^2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \cdots & \frac{\partial^2 f}{\partial {x_n}^2} \end{pmatrix},$$

where ∇f is the gradient vector and $\boldsymbol{H}$ is the Hessian matrix. Newton's method is also called Newton-Raphson method.

Notations

By convention, t is the iteration counter. Other notations are $\boldsymbol{x}_k$ or $\boldsymbol{x}^{(k)}$. In many cases, without possible confusion with the exponent k, we can also use $\boldsymbol{x}^k$. So, $\boldsymbol{x}_k = \boldsymbol{x}^k = \boldsymbol{x}_t = \boldsymbol{x}^t = \boldsymbol{x}^{(k)} = \boldsymbol{x}^{(t)}$ means the same vector.

Calculations of the Hessian matrix $\boldsymbol{H}$ iteratively can be computationally expensive, especially for higher-dimensional problems. Try to avoid the use of $\boldsymbol{H}$ whenever possible.

Quasi-Newton Method

If $\boldsymbol{H}$ is replaced by $\boldsymbol{I}$, we have

$$\boldsymbol{x}_{t+1} = \boldsymbol{x}_t - \frac{\nabla f(\boldsymbol{x}_t)}{\boldsymbol{H}} = \boldsymbol{x}_t - \alpha \boldsymbol{I} \nabla f(\boldsymbol{x}_t)$$
$$= \boldsymbol{x}_t - \alpha \nabla f(\boldsymbol{x}_t) = \boldsymbol{x}_t + \Delta \boldsymbol{x}_t, \quad \Delta \boldsymbol{x}_t = -\alpha \nabla f(\boldsymbol{x}_t).$$

Here $0 < \alpha < 1$ is a parameter that controls the step length. In principle, α should be related to the eigenvalues of $\boldsymbol{H}$ at each iteration, but we usually set it to a fixed value.

Generation of new moves by gradient.

- The increment/step $\Delta \boldsymbol{x}_t = -\alpha \nabla f(\boldsymbol{x}_t)$ moves along the (negative) gradient direction for minimization, leading to the steepest gradient descent method.
- In machine learning, the parameter α is often called the learning rate. Since the dimensionality of machine learning problems is very high, even the calculations of gradients can be very expensive. Thus, components of gradients may be computed sparely to save time.

  Stochastic gradient descent (SGD) and momentum-based methods such as the Adam optimizer are often used in deep learning.

Line Search

To find the local minimum of the objective function $f(\boldsymbol{x})$, we try to search along a descent direction $\boldsymbol{s}_k$ with an adjustable step size α_k so that

$$\psi(\alpha_k) = f(\boldsymbol{x}_k + \alpha_k \boldsymbol{s}_k),$$

decreases as much as possible, depending on the value of α_k. Loosely speaking, a reasonably right step size should satisfy the Wolfe's conditions:

$$f(\boldsymbol{x}_k + \alpha_k \boldsymbol{s}_k) \le f(\boldsymbol{x}_k) + \gamma_1 \alpha_k \boldsymbol{s}_k^T \nabla f(\boldsymbol{x}_k),$$

$$\boldsymbol{s}_k^T \nabla f(\boldsymbol{x}_k + \alpha_k \boldsymbol{s}_k) \ge \gamma_2 \boldsymbol{s}_k^T \nabla f(\boldsymbol{x}_k),$$

where $0 < \gamma_1 < \gamma_2 < 1$ are algorithm-dependent parameters.
For most functions, we can use $\gamma_1 = 10^{-4}$ to 10^{-2}, and $\gamma_2 = 0.1$ to 0.9.

Algorithm 1: Line Search Method

1 Initial guess $\boldsymbol{x}_0$ at $k = 0$;
2 **while** *($\|\nabla f(\boldsymbol{x}_k)\| >$ accuracy)* **do**
3 Find the search direction $\boldsymbol{s}_k = -\nabla f(\boldsymbol{x}_k)$;
4 Solve for α_k by decreasing $f(\boldsymbol{x}_k + \alpha \boldsymbol{s}_k)$, subject to Wolfe's conditions;
5 Update the result $\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \alpha_k \boldsymbol{s}_k$;
6 $k \leftarrow k + 1$;

Stochastic Gradient

In many optimization problems, especially in deep learning, the objective function or risk function to be minimized can be written in the following form:

$$E(\boldsymbol{w}) = \frac{1}{m}\sum_{i=1}^{m} f_i(\boldsymbol{x}_i, \boldsymbol{w}) = \frac{1}{m}\sum_{i=1}^{m} \left[u_i(\boldsymbol{x}_i, \boldsymbol{w}) - \bar{y}_i\right]^2, \quad f_i(\boldsymbol{x}_i, \boldsymbol{w}) = \left[u_i(\boldsymbol{x}_i, \boldsymbol{w}) - \bar{y}_i\right]^2.$$

Here, $\boldsymbol{w} = (w_1, w_2, ..., w_K)^T$ is a parameter vector [e.g., weights in an artificial neural networks (ANN)]. Also, $\bar{y}_i (i = 1, 2, ..., m)$ are the target or real data, and $u_i(\boldsymbol{x}_i)$ are the predicted values based on the inputs $\boldsymbol{x}_i$ by a model (e.g., trained ANN).

Stochastic Gradient Descent (SGD)

The standard gradient descent uses

$$\boldsymbol{w}^{t+1} = \boldsymbol{w}^t - \frac{\eta}{m}\sum_{i=1}^{m} \nabla f_i(\boldsymbol{w}),$$

where $0 < \eta \leq 1$ is the learning rate. This requires m gradients (expensive if m is large). To save computation, the gradient is approximated by ∇f_i (instead of all m values)

$$\boldsymbol{w}^{t+1} = \boldsymbol{w}^t - \eta_t \nabla f_i, \quad \eta_t = \frac{1}{1+\beta t}, \quad (t = 1, 2, ...)$$

where $\beta > 0$ is a hyper-parameter, which can be set to a fixed value.

Optimizers in Deep Learning

Gradient-based optimizers, including SGD, are widely used in machine learning. However, they can have strong oscillations for objectives with narrow valleys.

Moment-based methods

The main step in gradient-based optimizer is

$$\boldsymbol{x}^{(k+1)} = \boldsymbol{x}^{(k)} - \eta \nabla f(\boldsymbol{x}^{(k)}), \quad (\text{learning rate}) \quad 0 < \eta < 1.$$

Moment-based methods can be considered improved variants to damp potential oscillations. The main additional step is to use

$$\boldsymbol{u}^{(k+1)} = \gamma \boldsymbol{u}^{(k)} + \eta \nabla f(\boldsymbol{x}^{(k)}), \quad 0 < \gamma < 1,$$

then update the increment by

$$\boldsymbol{x}^{(k+1)} = \boldsymbol{x}^{(k)} - \boldsymbol{u}^{(k+1)}.$$

Typically, $\gamma = 0.9$ is used.

AdaDelta and RMSprop Optimizers

AdaDelta Optimizer

The main step of AdaDelta method is

$$\boldsymbol{x}^{(k+1)} = \boldsymbol{x}^{(k)} - \frac{\sqrt{E[(\Delta \boldsymbol{x}_{k-1})^2] + \epsilon}}{\mathrm{RMS}[g_k]} g_k,$$

where $g_k = \nabla f(\boldsymbol{x}^{(k)})$ and $\mathrm{RMS}[g_k] = \sqrt{E[g_k^2] + \epsilon}$ is the root mean squared (RMS) error. The running average formula is given by

$$E[(\Delta x_k)^2] = \gamma E[(\Delta x_{k-1})^2] + (1-\gamma)(\Delta \boldsymbol{x}_k)^2, \quad \Delta \boldsymbol{x}_k = -\frac{\eta}{\mathrm{RMS}[g_k]} g_k.$$

RMSprop

The RMSprop optimizer was introduced by G. Hinton for machine learning, which has some similarity to AdaDelta with $\gamma = 0.9$. The weight update becomes

$$\boldsymbol{x}^{(k+1)} = \boldsymbol{x}^{(k)} - \frac{\eta}{\sqrt{E_k(g_k^2) + \epsilon}} g_k, \quad \eta = 0.001.$$

Adam Optimizer

The Adam optimizer was developed by D. P. Kingma and J. Ba in 2014. It uses both the first moment m_1 and the second moment m_2 at each iteration k.

$$\begin{cases} m_1^{(k)} = \alpha m_1^{(k-1)} + (1-\alpha) g_k, \\ m_2^{(k)} = \beta m_2^{(k-1)} + (1-\beta) g_k^2, \end{cases}$$

where α, β are parameters. The iterative formula becomes

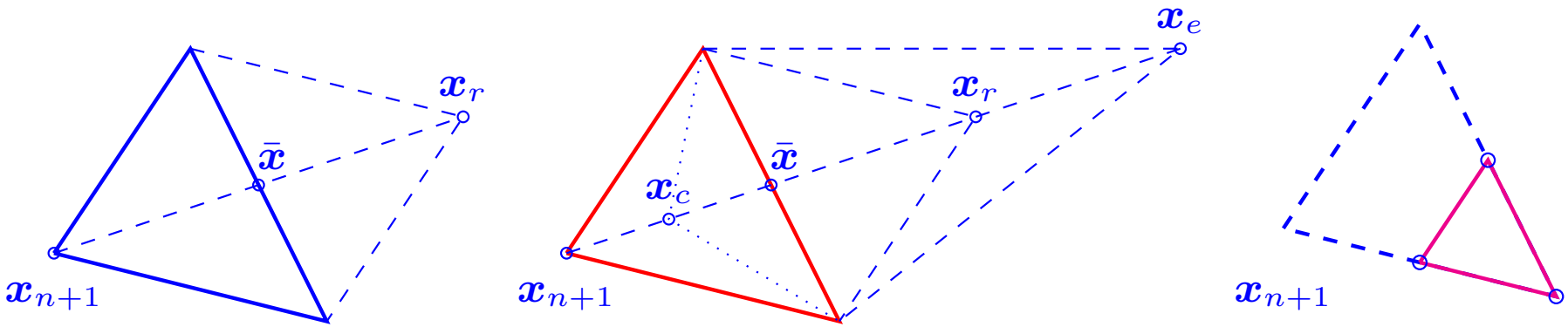
$$\boldsymbol{x}^{(k+1)} = \boldsymbol{x}^{(k)} - \frac{\eta}{\sqrt{\bar{m}_2^{(k)}} + \epsilon} \bar{m}_1^{(k)},$$

$$\bar{m}_1^{(k)} = \frac{m_1^{(k)}}{1-\alpha^k}, \quad \bar{m}_2^{(k)} = \frac{m_2^{(k)}}{1-\beta^k}.$$

Here, α^k and β^k are the values to the power k. For the values of the parameters, $\alpha = 0.9$, $\beta = 0.999$, $\eta = 0.001$, and $\epsilon = 10^{-8}$ can be used. The initial values $m_1^{(0)} = m_2^{(0)} = 0$ can be used.

These optimizers have been implemented in the well-known TensorFlow, etc.

Gradient-free optimizers



Simplex transformation: (a) reflection, (b) expansion and contraction, (c) reduction.

Nelder-Mead Method

The main idea of the Nelder-Mead method is to use the nodal (objective) values of an n-dimensional simplex (with $n+1$ nodes) to determine the moves without using any derivatives. In 2D, the simplex has 3 nodes (a triangle). For a given n-simplex, the objective values are ranked and re-ordered such that

$$f(\boldsymbol{x}_1) \le f(\boldsymbol{x}_2) \le ... \le f(\boldsymbol{x}_n) \le f(\boldsymbol{x}_{n+1}).$$

For minimization, we have a downhill simplex, thus $\boldsymbol{x}_{n+1}$ is the worse point (solution) and $\boldsymbol{x}_1$ is the best (minimum).

There are three types of moves: reflection, expansion/contraction, and reduction.

Algorithm 2: Nelder-Mead (Downhill Simplex) Method.

Initialize a simplex with $n+1$ vertices in n dimension;
while *(stop criterion is not true)* **do**
  Re-order the points such that $f(\boldsymbol{x}_1) \le f(\boldsymbol{x}_2) \le ... \le f(\boldsymbol{x}_{n+1})$ (thus $\boldsymbol{x}_1$ is best);
  Find the centroid $\bar{\boldsymbol{x}}$ using $\bar{\boldsymbol{x}} = \sum_{i=1}^{n} \boldsymbol{x}_i / n$ excluding $\boldsymbol{x}_{n+1}$;
  Generate a trial point via the reflection of the worse vertex;
  $\boldsymbol{x}_r = \bar{\boldsymbol{x}} + \alpha(\bar{\boldsymbol{x}} - \boldsymbol{x}_{n+1})$ where $\alpha > 0$ (typically $\alpha = 1$);
  if $f(\boldsymbol{x}_1) \le f(\boldsymbol{x}_r) < f(\boldsymbol{x}_n)$ **then**
    $\boldsymbol{x}_{n+1} \leftarrow \boldsymbol{x}_r$;
  if $f(\boldsymbol{x}_r) < f(\boldsymbol{x}_1)$ **then**
    Expand in the direction of reflection $\boldsymbol{x}_e = \boldsymbol{x}_r + \beta(\boldsymbol{x}_r - \bar{\boldsymbol{x}})$;
    if $(f(\boldsymbol{x}_e) < f(\boldsymbol{x}_r))$ **then**
      $\boldsymbol{x}_{n+1} \leftarrow \boldsymbol{x}_e$
    else
      $\boldsymbol{x}_{n+1} \leftarrow \boldsymbol{x}_r$
  if $f(\boldsymbol{x}_r) > f(\boldsymbol{x}_n)$ **then**
    Contract by $\boldsymbol{x}_c = \boldsymbol{x}_{n+1} + \gamma(\bar{\boldsymbol{x}} - \boldsymbol{x}_{n+1})$;
    if $f(\boldsymbol{x}_c) < f(\boldsymbol{x}_{n+1})$ **then**
      $\boldsymbol{x}_{n+1} \leftarrow \boldsymbol{x}_c$;
    else
      Reduction $\boldsymbol{x}_i = \boldsymbol{x}_1 + (\boldsymbol{x}_i - \boldsymbol{x}_1), (i = 2, ..., n+1)$;

An animation of how the Nelder-Mead method works can be found here [Wikipedia]

Main Problems with Traditional Algorithms

What's Wrong with Traditional Algorithms?

- Traditional algorithms are mostly local search, thus they cannot guarantee global optimality (except for linear and convex optimization).
- Results often depend on the initial starting points (except linear and convex problems). Methods tend to be problem-specific (e.g., k-opt moves, branch and bound).
- Struggle to cope problems with discontinuity.

Nature-Inspired Optimization Algorithms

Heuristic or metaheuristic algorithms (e.g., ant colony optimization, particle swarm optimization, firefly algorithm, bat algorithm, cuckoo search, differential evolution, flower pollination algorithm, etc.) tend to be a global optimizer so as to

- Increase the probability of finding the global optimality (as a global optimizer)
- Solve a wider class of problems (treating them as a black-box)
- Draw inspiration from nature (e.g., swarm intelligence)

But they can be potentially more computationally expensive.

Stochastic/Metaheuristic Algorithms

- Genetic algorithms (1960s/1970s), evolutionary strategy (Rechenberg & Swefel 1960s), evolutionary programming (Fogel et al. 1960s)
- Simulated annealing (Kirkpatrick et al. 1983), Tabu search (Glover 1980s), ant colony optimization (Dorigo 1992), genetic programming (Koza 1992)
- Particle swarm optimization (Kennedy & Eberhart 1995), differential evolution (Storn & Price 1996/1997), harmony search (Geem et al. 2001), honeybee algorithm (Nakrani & Tovey 2004), artificial bee colony (Karaboga, 2005)
- Firefly algorithm (Yang 2008), cuckoo search (Yang & Deb 2009), bat algorithm (Yang, 2010), flower pollination algorithm (Yang 2012; Yang et al. 2014), ...

For details, please refer to:

Xin-She Yang, Nature-Inspired Optimization Algorithms, Elsevier, (2014).

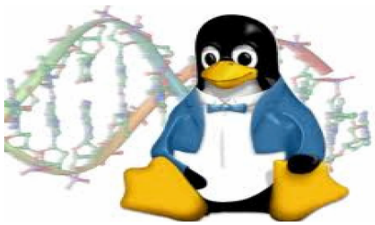
Xin-She Yang, Optimization Techniques and Applications with Examples, John Wiley & Sons, (2018).

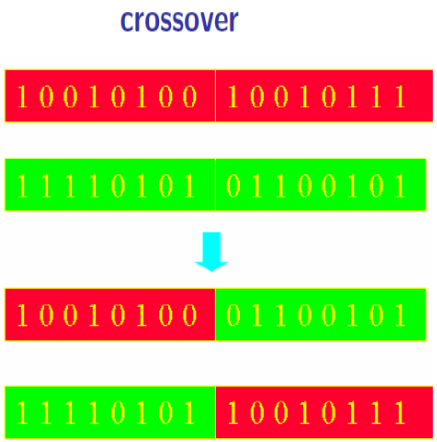
Nature-Inspired Algorithms

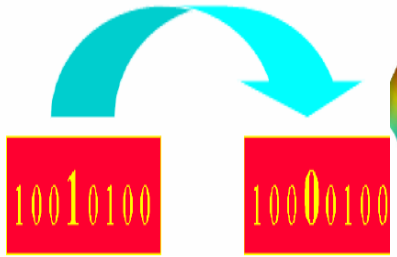
Genetic Algorithm (GA)

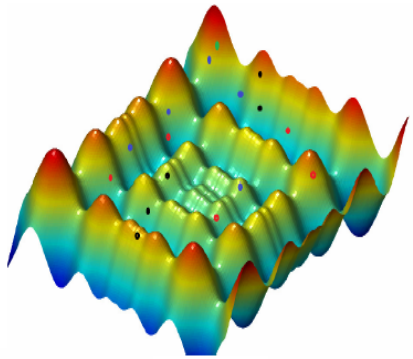
Genetic algorithm, developed by John Holland in the 1960s, is among the most widely used algorithms. It encodes the solutions to a problem as chromosomes and then use genetic operators such as crossover, mutation, and selection (survival of the fittest) to evolve the population.

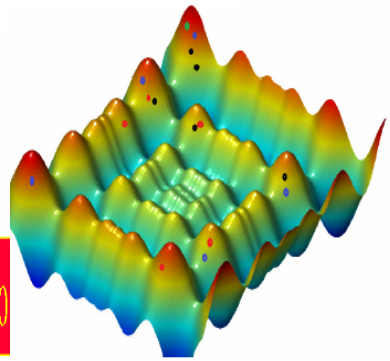
Starting with random sampling, the population will gradually converge.

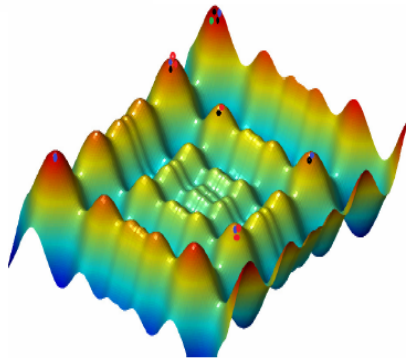












Simulated Annealing

Metal annealing to increase strength $\Longrightarrow$ simulated annealing.

Probabilistic Move:

$$p \propto \exp[-\Delta E/k_B T], \quad \Delta E \propto \Delta f(\boldsymbol{x}).$$

This is equivalent to a random walk $\boldsymbol{x}^{t+1} = \boldsymbol{x}^t + p(\boldsymbol{x}^t, \alpha)$.

k_B = Boltzmann constant (e.g., $k_B = 1$), T = temperature, E = energy.

Cooling schedule

$$T = T_0 \alpha^t \quad \text{(cooling schedule)}, \quad (0 < \alpha < 1).$$

Special case: $T \to 0 \Longrightarrow p \to 0 \quad \Longrightarrow$ hill climbing/Newton's method.

PSO

Particle Swarm Optimization (PSO) (Kennedy and Eberhart, 1995)

Swarming Starlings (YouTube Video)

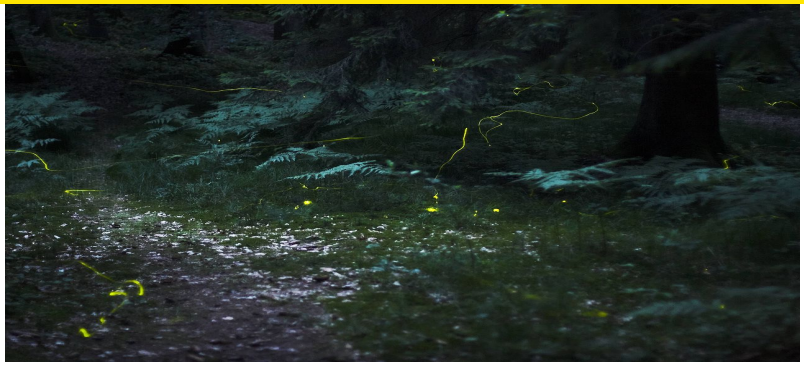
Each particle has a velocity $\boldsymbol{v}_i^t$ and a position vector $\boldsymbol{x}_i^t$ at time t. The whole population has a current best solution $\boldsymbol{g}^*$ and historical best solutions $\boldsymbol{x}_i^*$.

$$\boldsymbol{v}_i^{t+1} = \boldsymbol{v}_i^t + \alpha\epsilon_1(\boldsymbol{g}^* - \boldsymbol{x}_i^t) + \beta\epsilon_2(\boldsymbol{x}_i^* - \boldsymbol{x}_i^t),$$
$$\boldsymbol{x}_i^{t+1} = \boldsymbol{x}_i^t + \boldsymbol{v}_i^{t+1}\Delta t.$$

$\alpha, \beta =$ learning parameters, ϵ_1, ϵ_2=random numbers, and $\Delta t = 1$.

Quite efficient, but it can have premature convergence.

Firefly Algorithm



Firefly Video (YouTube)

Firefly Behaviour and Idealization (Yang, 2008)

- Fireflies are unisex and brightness varies with distance.
- Less bright ones will be attracted to bright ones.
- If no brighter firefly can be seen, a firefly will move randomly.

$$\boldsymbol{x}_i^{t+1} = \boldsymbol{x}_i^t + \beta_0 e^{-\gamma r_{ij}^2} (\boldsymbol{x}_j - \boldsymbol{x}_i) + \alpha\, \epsilon_i^t.$$

- ◇ The objective landscape maps to a light-based landscape, and fireflies swarm into the brightest points/regions.
- ◇ There is no $g*$, therefore, there is no leader. Also, a highly nonlinear iterative system, so subdivision into multiswarms is possible.

Why is FA so efficient?

Advantages of Firefly Algorithm over PSO

- Automatically subdivide the whole population into subgroups, and each subgroup swarms around a local mode/optimum.
- Control modes/ranges by varying γ.
- Control randomization by tuning parameters such as α.
- Suitable for multimodal, nonlinear, global optimization problems.
- FA can find multiple optimal solutions simultaneously (PSO cannot).
- FA has a fractal-like search structure (PSO does not).

Firefly Algorithm (Demo Video at Youtube) [Please click to start]

Cuckoo Search Algorithm (Yang and Deb, 2009)

Cuckoo brood parasitism

- 59 cuckoo species (among 141 cuckoo species) engage the so-called obligate reproduction parasitism strategy.
- Cuckoos lay eggs in the nests of host birds (such as warblers) and let host birds raise their chicks.
- Eggs may be discovered/abandoned with a probability ($p_a \approx 0.25$).
- Co-evolutionary arms race between cuckoo species and host species.

Cuckoo Behaviour (BBC Video)

Cuckoo Search

Local random walk:

$$\boldsymbol{x}_i^{t+1} = \boldsymbol{x}_i^t + s \otimes H(p_a - \epsilon) \otimes (\boldsymbol{x}_j^t - \boldsymbol{x}_k^t).$$

[$\boldsymbol{x}_i, \boldsymbol{x}_j, \boldsymbol{x}_k$ are 3 different solutions, $H(u)$ is a Heaviside function, ϵ is a random number drawn from a uniform distribution, and s is the step size.

Global random walk via Lévy flights:

$$\boldsymbol{x}_i^{t+1} = \boldsymbol{x}_i^t + \alpha L(s, \lambda), \quad L(s, \lambda) \sim \frac{\lambda \Gamma(\lambda) \sin(\pi\lambda/2)}{\pi} \frac{1}{s^{1+\lambda}}, \ (s \gg s_0).$$

Generation of new moves by Lévy flights, random walks and elitism.

Cuckoo Search is a weakly nonlinear system with scale-free search paths.

Cuckoo Search (Demo video at Youtube) [Please click to start]

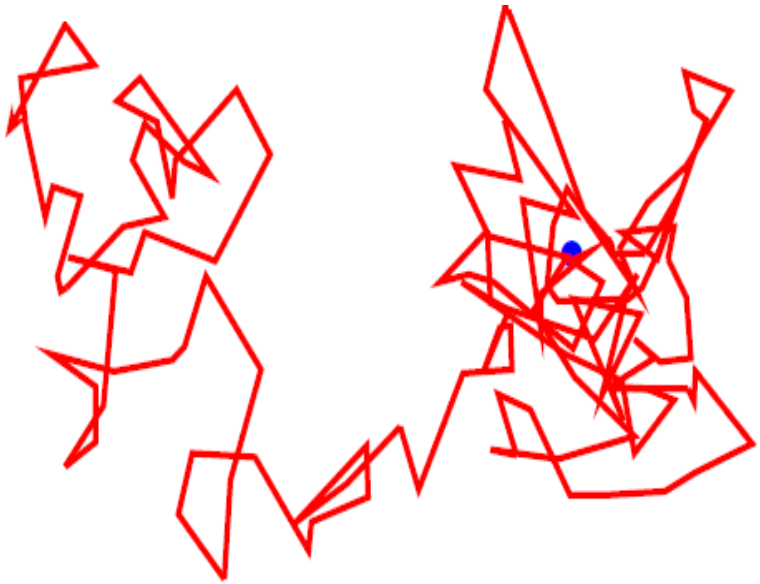
Mathematical Foundation for Cuckoo Search

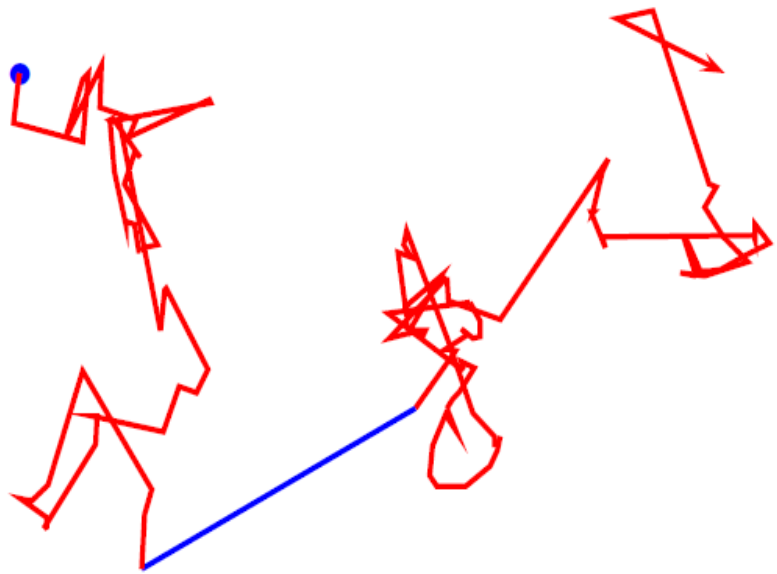
Isotropic andom walks (diffusion)
Gaussian distribution

$$p(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp\Big[-\frac{(x-\mu)^2}{2\sigma^2}\Big],$$

Lévy flights (superdiffusion)
Lévy distribution

$$L(x) = \frac{1}{\pi}\int_0^\infty \cos(tx) e^{-\alpha t^\lambda} dt.$$

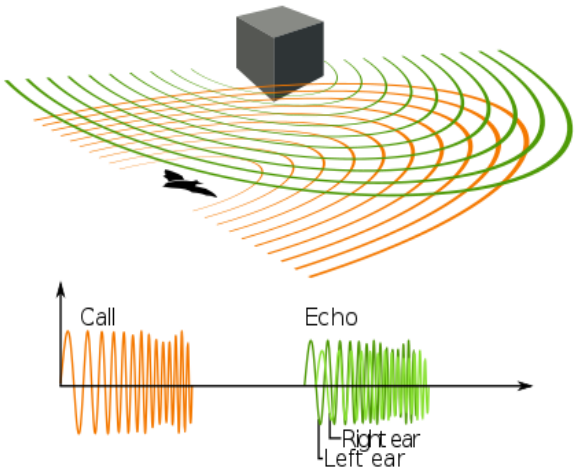




Diffusion distance: $d \sim \sqrt{t}$

$d \sim t^{(3-\lambda)/2}$ (for $1 \le \lambda \le 2$)

Bat Algorithm (Yang, 2010)



BBC Video at Youtube [Click to start]

Microbats use echolocation for hunting

- Ultrasonic short pulses as loud as 110dB with a short period of 5 to 20 ms. Frequencies of 25 kHz to 100 kHz.
- Speed up pulse-emission rate and increase loudness when homing at a prey.

Bat Algorithm

Acoustics of bat echolocation

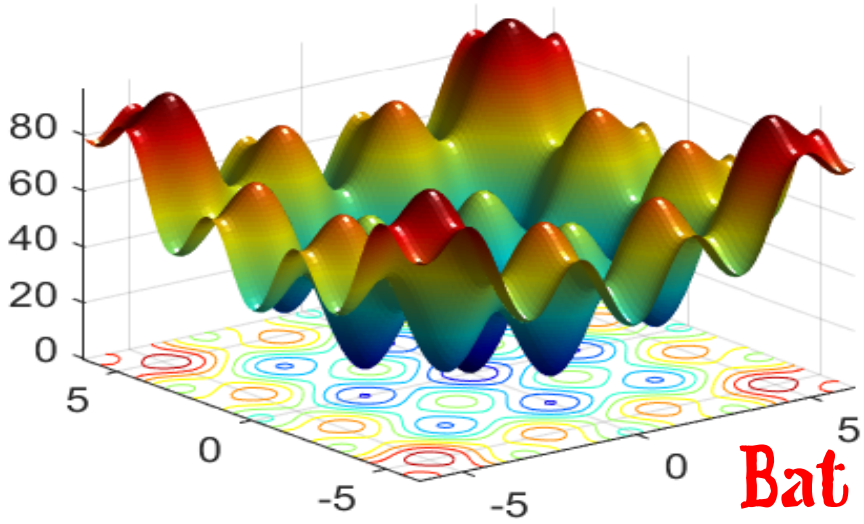
$$\lambda = \frac{v}{f} \sim 2 \text{ mm to } 14 \text{ mm}.$$

$$f_i = f_{\min} + (f_{\max} - f_{\min})\beta, \quad \beta \in [0, 1],$$

$$\boldsymbol{v}_i^{t+1} = v_i^t + (\boldsymbol{x}_i^t - \boldsymbol{x}_*) f_i, \quad \boldsymbol{x}_i^{t+1} = \boldsymbol{x}_i^t + \boldsymbol{v}_i^t \Delta t, \quad (\Delta t = 1).$$

Variations of Loudness and Pulse Rate

$$A_i^{t+1} \leftarrow \alpha A_i^t, \quad \alpha \in (0, 1], \quad r_i^{t+1} = r_i^0 [1 - \exp(-\gamma t)].$$



Bat Algorithm (Demo at Youtube)

Flower Pollination Algorithm (FPA) [Yang, 2012]


Pollination Characteristics

Flowering plants have been evolving for at least 125 million years, and about 90% of flower pollination needs pollinators. There are about 200 000 varieties of pollinators such as insects and birds. **BBC Video (Flower Pollination) at Youtube** [click to start]

- Biotic pollination and cross pollination (about 90% of flowering plants) via pollinators (e.g., insects and animals). [Feature 1]
- Abiotic and self-pollination (local, winds) (about 10%). [Feature 2]
- Flower constancy (e.g., hummingbirds)/flower-pollinator co-evolution. [Feature 3]
- Pollinators can fly for a long distance (thus possible Lévy flights). [Feature 4]

Yang, X.S., Flower pollination algorithm for global optimization. In: Unconventional Computation and Natural Computation. Lecture Notes in Computer Science vol. 7445, 240–249, Springer, Berlin (2012).

Flower Pollination Algorithm

Global pollination

$$\boldsymbol{x}_i^{t+1} = \boldsymbol{x}_i^t + \gamma L(\lambda)\,(\boldsymbol{g}_* - \boldsymbol{x}_i^t),$$

where $\boldsymbol{x}$ = a solution vector. $\boldsymbol{g}_*$ = best solution (so far). Step sizes are drawn randomly from the Lévy distribution:

$$L(\lambda) \sim \frac{\lambda\Gamma(\lambda)\sin(\pi\lambda/2)}{\pi}\frac{1}{s^{1+\lambda}}.$$

Local pollination

$$\boldsymbol{x}_i^{t+1} = \boldsymbol{x}_i + \epsilon\,(\boldsymbol{x}_j^t - \boldsymbol{x}_k^t),$$

where ϵ = uniformly distributed random number.

Switching Probability p

Control which branch/equation for search.

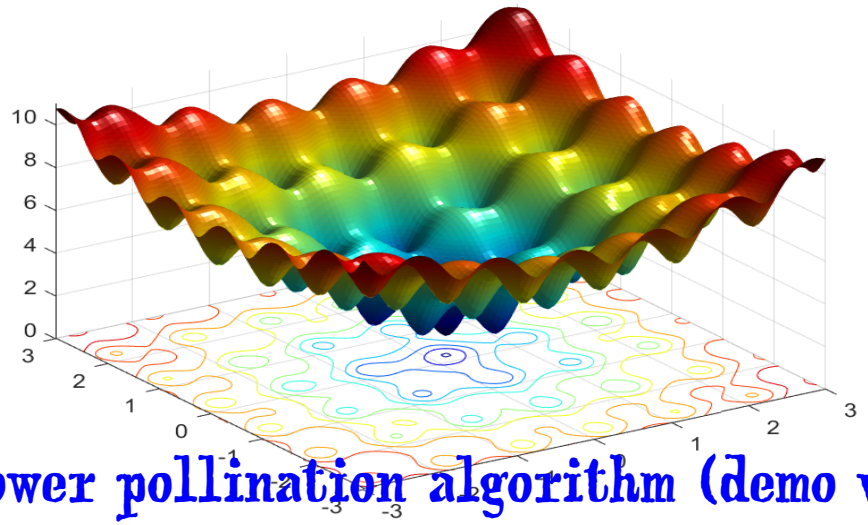
Typical Parameter Values

- Population size: $n = 20$ to 40 (up to 100 if necessary).
- Frequency: $f_{\min} = 0$, $f_{\max} = O(1)$ (typically $f_{\max} = 1$ or 2).
- Loudness: $A_0 = 1$, $\alpha = 0.9$ to 0.99 (typically $\alpha = 0.97$).
- Pulse emission rate: $r_0 = 1$, $\gamma = 0$ to 0.5 (typically $\gamma = 0.1$). Scaling: $\sigma = 0.5$.
- Number of iterations $t_{\max} = 100$ to 1000.

Demo: Eggcrate Function

$$f(x, y) = x^2 + y^2 + 25(\sin^2 x + \sin^2 y), \quad (x, y) \in [-2\pi, 2\pi]^2.$$

Optimal solution $f_{\min} = 0$ at $(0, 0)$.



Flower pollination algorithm (demo video at Youtube) [Click to start]

References

References

- Xin-She Yang, **Introduction to Algorithms for Data Mining and Machine Learning**, Academic Press/Elsevier, (2019).
- Xin-She Yang, Nature-Inspired Optimization Algorithms, Elsevier Insights, (2014).
- Xin-She Yang, **Optimization Techniques and Applications with Examples**, John Wiley & Sons, (2018).

Notes on Software

- The Matlab codes for nature-inspired algorithms (Bat algorithm, Cuckoo Search, Firefly Algorithm, and Flower Pollination Algorithms) can be downloaded from https://uk.mathworks.com/matlabcentral/profile/authors/3659939-xs-yang
- Also, Matlab has the optimization toolbox, including `fmincon`, `quadprog`, and multi-objective genetic algorithm `gamultiobj`.
- R has a relatively general purpose optimization solver `optimr` with `optim()` using Nelder-Mead method, simulated annealing and others. It also has a quadratic programming `solve.QP()` and least-squares solver `solve.qr()` as well as the firefly algorithm
- Python does have good optimization capabilities via `scipy.optimize()`, which includes the conjugate gradient, Newton's method, trust-region method, and least-square minimization.

Any questions? **Thank you :)**