PART II. OPTIMIZATION: NUMERICAL APPROACHES
(LECTURE 2)

Shpilev Petr Valerievich
Faculty of Mathematics and Mechanics, SPbU

September, 2025

Санкт-Петербургский государственный университет

Comments

In this lecture, we continue exploring numerical methods for optimization, beginning with the DIRECT algorithm in higher dimensions. We analyze its splitting strategy, interval structure, and reparameterization, illustrating the algorithm's behavior through multivariate examples and step-by-step implementation details. Building on deterministic global search, we then transition to stochastic approaches. We introduce noisy descent, its motivation, update rules, and convergence properties, followed by stochastic gradient descent and its search dynamics. The lecture also covers mesh adaptive direct search (MADS), highlighting step-size control, direction sampling, and practical realizations in two dimensions. Finally, we study simulated annealing, focusing on its probabilistic search principle, annealing schedules, and implementation, concluding with adaptive simulated annealing methods that refine the basic scheme to improve efficiency in complex optimization landscapes.

DIRECT: Behavior in High Dimensions

Splitting Strategy in Multivariate DIRECT:

- The search space is normalized to a unit hypercube so that $x \in [0,1]^d$
- Only the longest edges of each hyper-rectangle are split.
- Splitting produces 3^d subrectangles if all edges are equal.

Rectangle Selection:

- Each subrectangle is represented by a pair $(\delta_i, f(c_i))$, where:
  - c_i: center point,
  - δ_i: center-to-vertex distance.
- A rectangle is **potentially optimal** if the corresponding pair lies on the lower-right boundary of the convex hull of all points $(\delta_i, f(c_i))$.

Limitation:

- To shrink all dimensions evenly, DIRECT must pass through many intermediate rectangles.
- Leads to slow convergence in high dimensions.

Comments

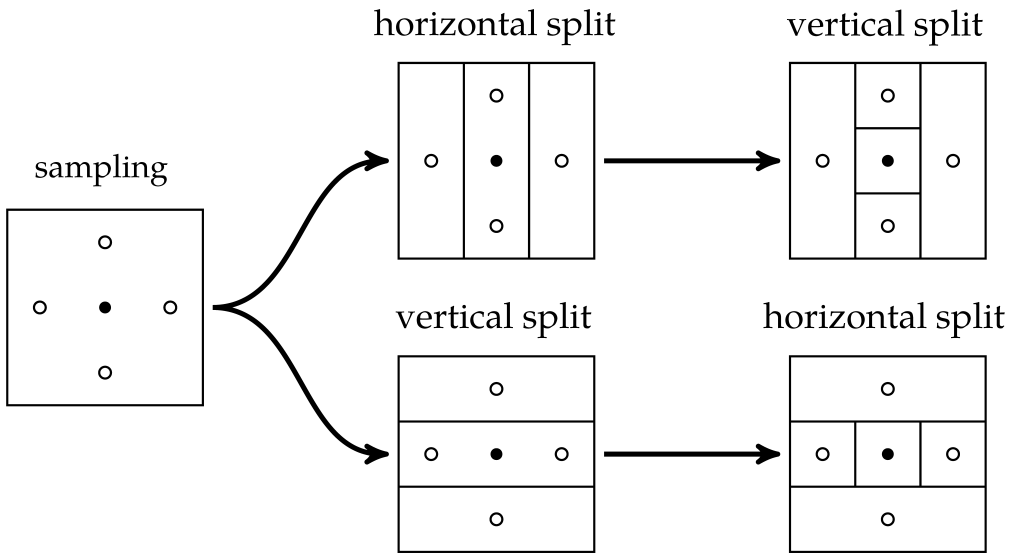
Before starting the search, DIRECT normalizes the entire search domain to the unit hypercube: each coordinate x_i is rescaled to lie within the interval from 0 to 1. This ensures that all variables share the same scale and that space is divided uniformly across dimensions.

Initially, the algorithm evaluates the objective function at the center of the hypercube. At each iteration, it selects potentially optimal hyperrectangles — those with the best lower bound for some admissible value of the Lipschitz constant. Only these hyperrectangles are subject to further division. On the first iteration, the entire hypercube is considered the sole potentially optimal region.

The division process is not applied along all coordinates at once, but only along the directions with the largest edge length. If multiple such directions exist, one of them is selected. The chosen coordinate direction is then divided into three equal parts, creating new hyperrectangles of equal width along that axis. The center of each new hyperrectangle becomes the point where the objective function is evaluated.

For each hyperrectangle, the function value at the center and the distance to the farthest vertex are used to compute a lower bound. Then the selection of potentially optimal hyperrectangles is repeated, and the division process continues.

This approach efficiently combines local and global search, but scales poorly: to shrink the region along all coordinates, it must be split repeatedly along different axes in sequence. As a result, the method works well in problems with 1–3 variables, acceptably in 4–6, but quickly loses effectiveness for higher-dimensional problems.



Comments

Let us illustrate how region subdivision works in the two-dimensional case. The entire domain is initially normalized into a unit square, and the function is first evaluated at the center. On the first iteration, since both coordinate directions are of equal length, they are considered eligible for subdivision. However, DIRECT does not select a direction at random: for each eligible axis, two new points are computed by offsetting the center left and right (or up and down), and the function is evaluated at those points. The axis along which the function exhibits the smallest value is chosen for subdivision. In this example, that might be the horizontal axis, if function values decrease more in that direction or the vertical axis, otherwise. As a result, the square is split into three equal rectangles along the horizontal axis (or the vertical axis), and the function is sampled at the centers of those new rectangles.

On the second iteration, the central rectangle may be selected as potentially optimal if its center value is best. Now, since the horizontal (or vertical) direction was already divided, the next eligible axis is vertical (or horizontal). Again, offset evaluations are performed and the axis is chosen where the function drops more sharply. This results in adaptive, alternating subdivision across coordinate directions, focused where the function appears to decrease most promisingly.

Algorithm 8: DIRECT

```
1  function direct(f, a, b, ϵ, k_max)
2  # Input: objective function f, vector of lower bounds a,
3  # vector of upper bounds b, tolerance parameter ϵ, number of iterations k_max.
4      g = reparameterize_to_unit_hypercube(f, a, b)    # normalize domain
5      intervals = Intervals()                          # interval storage
6      n = length(a)                                    # problem dimension
7      c = fill(0.5, n)                                 # center of domain
8      interval = Interval(c, g(c), fill(0, n))         # initial interval
9      add_interval!(intervals, interval)               # add to structure
10     c_best, y_best = copy(interval.c), interval.y    # best known point so far
11
12     for k in 1:k_max                                 # main loop
13         S = get_opt_intervals(intervals, ϵ, y_best) # select optimal intervals
14         to_add = Intervals.empty_list()              # prepare new intervals
15         for interval in S
16             append!(to_add, divide(g, interval))       # add new intervals
17             dequeue!(intervals[vertex_dist(interval)]) # remove old interval
18         end
```


Comments

This is the first part of the implementation of the main DIRECT algorithm.The code is written in Julia — a high-level programming language designed specifically for numerical and scientific computing.

It combines the expressiveness of languages like Python with performance close to C.Some key features of Julia that are relevant here: vectors are indexed starting from one; variables are passed by reference; and functional programming is used extensively.

The function direct takes the following arguments:

f — the objective function to be minimized;

a and b — vectors of lower and upper bounds of the feasible region;

ε — the tolerance parameter;

k_{max} — the maximum number of iterations.

The goal of the algorithm is to find the point where the function attains its global minimum, or comes as close as possible to it, within a limited number of iterations.

The algorithm works iteratively. At each step, it selects potentially optimal rectangles, divides them along suitable coordinates, computes function values at the new centers, and updates the best value found so far.

The search is not performed in the original domain, but in a normalized one — the entire domain is transformed to the unit hypercube before the search begins. Now let's briefly comment on the code lines:

Line 4: Here the function `reparameterize_to_unit_hypercube` is called. It returns a new function g, which is equivalent to the original function f, but now defined on the unit hypercube.

Line 5: The intervals data structure is initialized. It is used to store all current rectangles. Internally, it is implemented as a dictionary: the key is the distance from the center of the rectangle to a vertex, and the values are priority queues ordered by the function value at the center.

Line 6: The problem's dimension is determined.

Line 7: The center of the entire search space is initialized — a vector c consisting entirely of 0.5. This point is the center of the unit hypercube.

Algorithm 8: DIRECT

```
1  function direct(f, a, b, ϵ, k_max)
2  # Input: objective function f, vector of lower bounds a,
3  # vector of upper bounds b, tolerance parameter ϵ, number of iterations k_max.
4      g = reparameterize_to_unit_hypercube(f, a, b)    # normalize domain
5      intervals = Intervals()                          # interval storage
6      n = length(a)                                    # problem dimension
7      c = fill(0.5, n)                                 # center of domain
8      interval = Interval(c, g(c), fill(0, n))         # initial interval
9      add_interval!(intervals, interval)               # add to structure
10     c_best, y_best = copy(interval.c), interval.y    # best known point so far
11
12     for k in 1:k_max                                 # main loop
13         S = get_opt_intervals(intervals, ϵ, y_best) # select optimal intervals
14         to_add = Intervals.empty_list()              # prepare new intervals
15         for interval in S
16             append!(to_add, divide(g, interval))       # add new intervals
17             dequeue!(intervals[vertex_dist(interval)]) # remove old interval
18         end
```


Comments

Line 8: The initial interval covering the whole hypercube is created. It has three components:

The center coordinates c;

The function value at the center, computed as g(c);

A depth vector — this is a vector of length n, where each value indicates how many times the region has been divided along the corresponding coordinate. Since no divisions have occurred yet, all values are initially zero.

Line 9: This interval is added to the intervals data structure using the `add_interval` function which is defined bellow.

Line 10: The initial best approximation to the minimum is stored: both the coordinates and the function value at the center. The copy function is used to explicitly create a copy of the center vector. This is important because in Julia, variables are passed by reference — and without copying, the structure could be corrupted during further modifications.

Line 12: The outer loop begins. It will run k_{max} times — that is, as many iterations as are allowed for the algorithm.

Line 13: From the current set of rectangles, the potentially optimal ones are selected.This is done by calling the `get_opt_intervals` function, which receives the interval structure, the tolerance ε, and the current best function value. This procedure implements selection by lower bound estimates — following the idea of the DIRECT algorithm.

Line 14: A temporary list `to_add` is created. It will be used to store new rectangles obtained by subdividing the selected ones.

Lines 15: We iterate over each interval in S, the list of potentially optimal rectangles:

Lines 16: The divide function is called on the interval and produces a list of smaller subintervals; The result is appended to the temporary list `to_add`;

Lines 17: The `vertex_dist` function computes the L_2 distance from the center to a vertex of the original rectangle; The rectangle is removed from the interval structure using dequeue!, since it is now replaced by its subintervals.

```
        for interval in to_add  # iterate through new intervals
            add_interval!(intervals, interval)  # insert into data structure
            if interval.y < y_best              # update best solution if needed
                c_best, y_best = copy(interval.c), interval.y
            end
        end
    end
    return rev_unit_hypercube_parameterization(c_best, a, b)
end
```

```
function reparameterize_to_unit_hypercube(f, a, b)
    Δ = b - a
    return x -> f(x.* Δ + a)  # scale and shift to map from [0,1]^n to [a,b]
end

function rev_unit_hypercube_parameterization(x, a, b)
    return x.* (b - a) + a    # map from unit hypercube back to original space
end
```


Comments

This slide completes the direct function and shows two helper functions responsible for coordinate transformation between the original domain and the normalized unit hypercube.

We iterate through all intervals that were created by splitting potentially optimal rectangles. Each new interval is added to the main interval structure. If the function value at its center is smaller than the current best value, we update the record of the best point found so far. Note that copy(interval.c) is used to avoid referencing a mutable object — a common issue in Julia.

After completing all iterations, we return the best point, transformed back from normalized space to the original variable domain. The transformation is done via the function `rev_unit_hypercube_parameterization`, shown below.

The function `reparameterize_to_unit_hypercube` takes the original objective function f, and bounds a and b, and returns a new function g. This new function operates on the unit hypercube, mapping points back into the original domain using the corresponding formula.

The function `rev_unit_hypercube_parameterization` does exactly the reverse: it converts a point from the unit hypercube back into the original domain. This is used to present the final answer in the same coordinate system as the original input.

Interval Structure in DIRECT

```julia
using DataStructures  # import priority queues and dictionaries
struct Interval
    c       # center of the rectangle
    y       # function value at center
    depths  # division depth for each coordinate
end
min_depth(interval) = minimum(interval.depths)  # smallest depth value
# Euclidean distance from center to vertex
vertex_dist(interval) = norm(0.5 * 3.0.^ (-interval.depths), 2)

# main data structure: maps distances to priority queues of intervals
const Intervals = OrderedDict{Float64, PriorityQueue{Interval, Float64}}()
function add_interval!(intervals, interval)
    d = vertex_dist(interval)
    if !haskey(intervals, d)
        intervals[d] = PriorityQueue{Interval, Float64}()
    end
    return enqueue!(intervals[d], interval, interval.y)
end
```


Comments

This slide defines the Interval structure and the data structure used to organize intervals in the DIRECT algorithm.

The Interval struct stores the center point c, the function value at the center y, and a depth vector depths, which tracks how many times the interval has been divided along each coordinate.

The function `min_depth` returns the smallest value in the depth vector, indicating the direction of least refinement. The function `vertex_dist` computes the Euclidean distance from the center to a vertex of the rectangle — this serves as a measure of interval size.

All intervals are stored in an OrderedDict called Intervals, where each key is a distance (from `vertex_dist`), and the value is a priority queue that stores intervals with that size. The priority queue ensures that intervals with lower function values are accessed first, which is crucial for efficiently selecting potentially optimal regions.

The function `add_interval!` inserts a new interval into the appropriate priority queue, creating the queue if needed. This structure groups intervals by size and sorts them by objective value within each group — a key design for fast, structured global search.

get_opt_intervals Function

```julia
function get_opt_intervals(intervals, ϵ, y_best)
    stack = Interval[]  # Stack for storing potentially optimal intervals
    for (x, pq) in intervals  # Iterate over interval groups by distance
        if !isempty(pq)
            interval = DataStructures.peek(pq)[1]  # Interval with best value
            y = interval.y

            # Check dominance against last two in stack
            while length(stack) > 1
                interval1 = stack[end]
                interval2 = stack[end-1]
                x1, y1 = vertex_dist(interval1), interval1.y
                x2, y2 = vertex_dist(interval2), interval2.y
                ℓ = (y2 - y) / (x2 - x)  # Slope of lower bound line
                if y1 <= ℓ*(x1 - x) + y + ϵ
                    break  # Interval is dominated - stop checking
                end
                pop!(stack)  # Remove dominated interval
            end
```


Comments

The function `get_opt_intervals` identifies the set of potentially optimal intervals among all current intervals stored in the intervals data structure. In the DIRECT algorithm, these are the regions most promising for further exploration.

The core idea is to treat each interval as a point in 2D:

x-axis: the distance from the center to the vertex (computed by `vertex_dist`)

y-axis: the function value at the center

Among all such points, only those on the lower convex hull can be optimal under some value of the unknown Lipschitz constant. These are the intervals the algorithm continues to divide.

The function loops through all interval groups (clustered by equal distance) and selects the best interval from each group (i.e., the one with the lowest function value). It then performs a geometric test to see if each interval is dominated by others — meaning it can't be optimal for any slope. This test compares slopes between points and removes dominated ones from the stack.

```
            if !isempty(stack) && interval.y > stack[end].y + ϵ
                continue  # New interval is worse - skip it
            end
            push!(stack, interval)  # Add new potentially optimal interval
        end
    end
    return stack
end
```

```
function divide(f, interval)
    #center, min depth, dimension
    c, d, n = interval.c, min_depth(interval), length(interval.c)
    # returns vector of index for elements with minimal depth
    dirs = findall(interval.depths.== d)
    # new centers along each direction
    cs = [(c + 3.0^(-d-1) * basis(i, n),
           c - 3.0^(-d-1) * basis(i, n))
           for i in dirs]
```


Comments

At the end, the function returns a filtered list (stack) containing only potentially optimal intervals that will be divided in the next iteration.

The function devide performs the core geometric operation in the DIRECT algorithm — splitting a rectangle into smaller rectangles. The input is an Interval object, which includes the center c, the function value at the center, and a vector of depths (indicating how many times the region has been split along each axis).

The function begins by identifying all directions (axes) that have been split the least — i.e., those with minimal depth. These are the candidate directions for the next division. For each of these directions, two new centers are generated by shifting the original center slightly in both directions along the corresponding axis. The magnitude of the shift is proportional to $3^{-(d+1)}$, where d is the current minimal depth. This ensures that each subdivision becomes progressively finer.

```
    # returns list of pairs of function values at new centers
    vs = [(f(C[1]), f(C[2])) for C in cs]
    # returns list of smallest values from each pair in vs
    minvals = [min(V[1], V[2]) for V in vs]
    intervals = Interval[]
    # copy(...) → returns a shallow copy of the depths vector
    depths = copy(interval.depths)
    # sortperm(...) → returns indices that would sort minvals
    for j in sortperm(minvals)
        depths[dirs[j]] += 1
        C, V = cs[j], vs[j]
        # push! → appends new Interval objects to the list
        push!(intervals, Interval(C[1], V[1], copy(depths)))
        push!(intervals, Interval(C[2], V[2], copy(depths)))
    end
    # original interval is also added with updated depths
    push!(intervals, Interval(c, interval.y, copy(depths)))
    return intervals
end
```


Comments

Function values at the new centers are then computed and used to determine along which direction the function appears to be most promising (i.e., has the lowest value). The directions are sorted accordingly.

For each such direction, two new intervals are created (one for each shifted center), with the depth along the current direction incremented by one. The original center is also added at the end with the updated depth vector. In total, this results in $2k + 1$ new rectangles, where k is the number of directions with minimal depth.

The function returns the full list of these new Interval objects.

DIRECT: First Iteration

Example $\min_{x} f(x) = \|x\| + \sin(4 \cdot \arctan(x_2, x_1)),\ x \in [-1, 3] \times [-2, 1]$.

Step 1. Normalize domain to unit hypercube: $f'(x') = f(4x'_1 - 1,\ 3x'_2 - 2)$

Step 2. Evaluate function in center $x' = (0.5, 0.5)$: $f'(0.5,\ 0.5) = 0.158$

Step 3. Initial interval: center $= (0.5, 0.5)$, sides $= (1, 1)$, depth $= (0,\ 0)$.

Step 4. Minimal depth $= 0 \Rightarrow$ evaluate in both directions ($\delta = \frac{1}{3}$, step $= \frac{1}{6}$):

$$\text{along } x_1: \quad f'(1/6,\ 1/2) = 0.500; \quad f'(5/6,\ 1/2) = 1.231$$

$$\text{along } x_2: \quad f'(1/2,\ 1/6) = 2.029; \quad f'(1/2,\ 5/6) = 1.861$$

Step 5. Select both directions for division: Best function value among sampled points is 0.5 (along x_1), next best is 1.861 (along x_2) $\Rightarrow$ divide along x_1, then x_2

Step 6. After two divisions, five subintervals are created:

interval	center	side lengths	vertex distance	center value
1	$[1/2, 1/2]$	$[1/3, 1/3]$	0.236	0.158
2	$[1/6, 1/2]$	$[1/3, 1]$	0.527	0.500
3	$[5/6, 1/2]$	$[1/3, 1]$	0.527	1.231
4	$[1/2, 5/6]$	$[1/3, 1/3]$	0.236	1.861
5	$[1/2, 1/6]$	$[1/3, 1/3]$	0.236	2.029

Comments

In this example, we illustrate the first iteration of the DIRECT algorithm applied to the so-called flower function — a non-convex test function often used in optimization benchmarks.

We start by normalizing the search space to a unit square, which allows us to apply the algorithm uniformly regardless of the original domain.

The algorithm evaluates the objective function at the center of the hypercube and stores the result. Since all directions initially have the same minimal depth, evaluations are made in both coordinate directions.

By comparing the function values at shifted points, we select directions where improvement is most promising. The interval is then divided accordingly — first in the better direction, then in the next one.

As a result, five new intervals are formed. These represent the new candidates for exploration in the next iteration, maintaining DIRECT's balance between global search and local refinement.

Multivariate DIRECT: Example

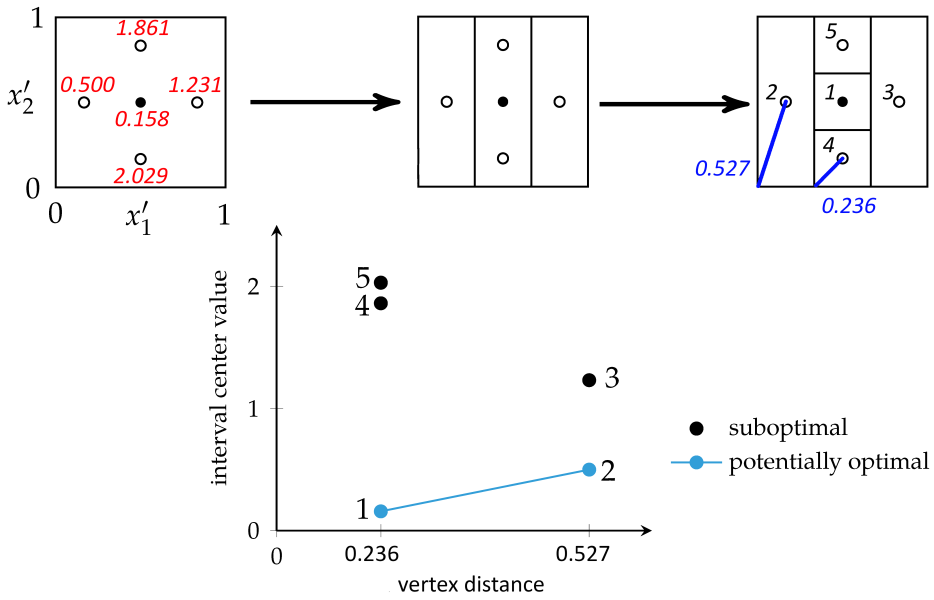


Figure: We next split on the two intervals centered at point 1 and point 2.

Comments

This slide visualizes the first iteration of the DIRECT algorithm.

In the top row, we see the evolution of the domain:

The function is first evaluated at the center and at points along coordinate directions.

Based on those evaluations, the region is divided first along the x_1-axis, then along the x_2-axis for the most promising interval.

As a result, five smaller subregions are created.

In the bottom plot, each subinterval is represented as a point.The horizontal axis shows the distance from the center to the corners — a measure of the interval's size — while the vertical axis shows the function value at the center.

DIRECT selects potentially optimal intervals by balancing function value and size.In this case, the points corresponding to intervals 1 and 2 lie on the lower-right envelope of the bottom scatter plot. These points are not dominated by any other: there is no interval that is both narrower and has a better (lower) function value. Therefore, DIRECT selects these intervals as potentially optimal.

Stochastic Methods: Introduction

Key Idea:

- Introduce controlled randomness into the optimization process.
- Explore the design space more broadly and escape local optima.
- Especially useful for non-convex, noisy, or black-box problems.

Motivation:

- Deterministic methods can get stuck in saddle points or poor local minima.
- Randomization increases the chance of finding the global minimum.
- Pseudo-random generators ensure repeatability of experiments.

Note: Too much randomness can lead to inefficiency — methods must carefully balance exploration and exploitation.

Comments

The defining feature of stochastic methods is the use of randomness as a strategic tool to improve optimization performance. Rather than following a fixed rule for selecting each new iterate, these methods introduce controlled variability — either in the decision variables, the objective evaluation, or both. The goal is to enhance the exploration of the search space and avoid getting stuck in suboptimal regions.

This use of randomness is not arbitrary. On the contrary, the stochastic components are carefully designed to complement the structure of the problem. For example, by occasionally allowing moves that worsen the objective, these methods can escape local minima that would trap a purely descent-based method. Another key aspect is that randomness often helps in dealing with noisy functions or models for which derivative information is unavailable or unreliable.

The challenge, of course, is balance: too much randomness and the method becomes unstable or slow; too little, and it behaves like a standard deterministic method, with the same pitfalls. Many stochastic algorithms manage this trade-off by controlling the intensity of noise or randomness over time — a theme that will appear throughout this part.

Another important motivation for stochastic methods is their ability to mitigate the curse of dimensionality. In high-dimensional spaces, deterministic methods can become inefficient due to the exponential growth of the search space. Stochastic sampling can avoid this by probing promising directions without exhaustively covering the entire space, making it possible to find good solutions even in hundreds or thousands of dimensions.

Noisy Descent: Motivation and Update Rule

Main Idea: Adding noise to descent steps helps the algorithm escape saddle points and explore flat regions more effectively.

- **Saddle points** often have gradients close to zero
- **Noisy updates** prevent the method from stagnating
- **Useful in machine learning** where full gradient computation is expensive

Classic Gradient Descent:

$$x^{(k+1)} = x^{(k)} + \alpha g^{(k)}$$

- α - step size; $g^{(k)}$ - descent direction (often $-\nabla f(x^{(k)})$)

Stochastic Variant with Noise:

$$x^{(k+1)} = x^{(k)} + \alpha g^{(k)} + \varepsilon^{(k)}$$

- $\varepsilon^{(k)}$: zero-mean Gaussian noise with standard deviation $\sigma(k)$
- Typical choice: $\sigma(k) = \frac{1}{k}$ - decreasing noise over time

Key Benefit: Combining gradient information and stochasticity helps avoid stagnation near saddle points and improves global search behavior.

Comments

Gradient descent is one of the most fundamental optimization techniques. At each step, it moves in the direction opposite to the gradient, aiming to reduce the objective function. However, in complex, high-dimensional landscapes, this basic approach encounters serious limitations. One of the most common issues is the presence of saddle points — locations where the gradient is zero but the point is not a minimum. Since the step direction is entirely determined by the gradient, the algorithm can become stuck or slow to escape from these flat regions.

Introducing noise into the update helps overcome this issue. The core idea is to combine the gradient-based step with a small random perturbation. The added randomness enables the algorithm to explore directions that would be ignored by a purely gradient-based approach. Crucially, this noise must be carefully controlled. If it remains constant, convergence may be compromised; if it decreases over time, the method can explore at first and later settle into a solution.

This strategy is particularly useful in large-scale optimization problems such as training deep neural networks. There, computing exact gradients over the entire dataset is often infeasible. Instead, gradients are approximated using small random subsets, or mini-batches, and this naturally introduces noise into the updates. This noise not only improves computational efficiency but also helps the model generalize better by escaping poor local minima or saddle plateaus — a desirable side effect known from practice.

Algorithm 9: Noisy Descent.

```
mutable struct NoisyDescent <: DescentMethod
    submethod     # inner descent method (e.g. gradient descent)
    σ             # noise schedule function
    k             # iteration counter
end

function init!(M::NoisyDescent, f, ∇f, x)
    init!(M.submethod, f, ∇f, x)
    M.k = 1
    return M
end
function step!(M::NoisyDescent, f, ∇f, x)
    x = step!(M.submethod, f, ∇f, x)
    σ = M.σ(M.k)
    x += σ .* randn(length(x))
    M.k += 1
    return x
end
```


Comments

This implementation shows how the noisy descent algorithm can be structured in practice. The key idea is to augment an existing descent method with an additional stochastic step. Rather than writing a completely new optimization method from scratch, we create a wrapper — a higher-level object that calls the original descent method and then adds noise to the result.

The implementation defines a new Julia structure NoisyDescent, which contains three fields: the inner method (e.g., gradient descent), a noise schedule function, and an iteration counter. The noise schedule function $\sigma(k)$ determines how much noise to apply at each iteration — typically decreasing over time.

The initialization function `init!` delegates to the inner method's own initialization routine and resets the iteration counter. The main logic happens in `step!`, where we take one step using the base descent method, then generate random Gaussian noise of the appropriate magnitude and add it to the current point. Finally, the iteration counter is incremented.

This modular approach makes it easy to combine stochastic behavior with any standard descent method. The structure is clean and general: we can use the same wrapper with steepest descent, Newton's method, or even quasi-Newton schemes. By decoupling the descent logic from the noise injection, we gain both flexibility and clarity in implementation.

Julia Syntax: Key Notions

mutable struct defines a modifiable data container.

- Like a class in Python or a 'struct' with fields in C++.
- Fields can be updated after creation.

Function calls inside functions:

- Example: `step!(M.submethod, f, ∇ f, x)` means:
- Call the method `step!` defined for the inner method stored in `M.submethod`.
- This allows algorithm composition — e.g., combine gradient descent with noise.

Meaning of the `!` in function names:

- By convention in Julia, `f!(x)` modifies its argument(s) in-place.
- `init!`, `step!` change internal state of `M`.

Comments

To understand the implementation of Noisy Descent, it's useful to clarify a few syntactic conventions from the Julia programming language. First, the keyword mutable struct defines a structure whose fields can be changed after creation. This is similar to a class in Python with modifiable attributes, or a record with writable fields in languages like C++.

Second, Julia supports nested function calls. For instance, the expression `step!(M.submethod, f, ∇f, x)` means we are calling the `step!` function for the descent method stored in M.submethod. This is a powerful design pattern that enables composition — we can reuse existing algorithms by wrapping them with additional behavior, such as adding noise.

Finally, the exclamation mark at the end of a function name — for example, in `step!` or `init!` — is a naming convention that indicates the function modifies its arguments. This is not enforced by the language but is widely respected. It helps readers of the code quickly distinguish between functions that mutate input and those that don't.

Convergence of Stochastic Descent

Step Size Schedule for Convergence:

$$\sum_{k=1}^{\infty} \alpha_k = \infty, \qquad \sum_{k=1}^{\infty} \alpha_k^2 < \infty$$

These classical Robbins-Monro conditions ensure convergence in expectation.

Interpretation and Typical Choices:

- The first condition ensures **persistent exploration** — steps never vanish too quickly.
- The second condition ensures **control of stochastic noise**.
- A common rule is $\alpha_k = \frac{1}{k^a}$ with $a \in (0.5, 1)$.
- $\alpha_k = \frac{1}{k}$ is the borderline case satisfying both conditions.

Stochastic Approximation Setting

Let $g^{(k)}$ be a noisy estimate of $\nabla f(x^{(k)})$, such that:

$$\mathbb{E}[g^{(k)} \mid x^{(k)}] = \nabla f(x^{(k)}), \qquad \mathbb{E}[\|g^{(k)}\|^2] < \infty$$

Then, under suitable conditions on f, the iterates converge in expectation.

Note: Step sizes must decay slowly enough to allow movement, but fast enough to suppress noise — this is a delicate balance.

Comments

The step size schedule is a central element in the convergence of stochastic descent methods. The two conditions shown here — first proposed by Robbins and Monro — are classical. They ensure that, even in the presence of noise, the iterates will converge to a solution in expectation.

The intuition is straightforward. If step sizes decay too quickly, the algorithm loses the ability to explore and gets stuck. If they decay too slowly, the accumulated noise prevents convergence. That's why the first series must diverge, and the second must converge. The common rule alpha sub k equals one divided by k raised to the power of a with a is in the open interval from zero point five to one ($0.5 < a < 1$) satisfies both requirements.

We also place this in the context of stochastic approximation. In practice, we don't use exact gradients, but noisy estimates. As long as these estimates are unbiased and have bounded variance, convergence can still be guaranteed — provided that step sizes meet the above conditions.

The practical consequence is that careful scheduling of step sizes is essential. Too aggressive or too conservative a decay can lead to failure. Balancing exploration and stability is one of the fundamental themes in stochastic optimization.

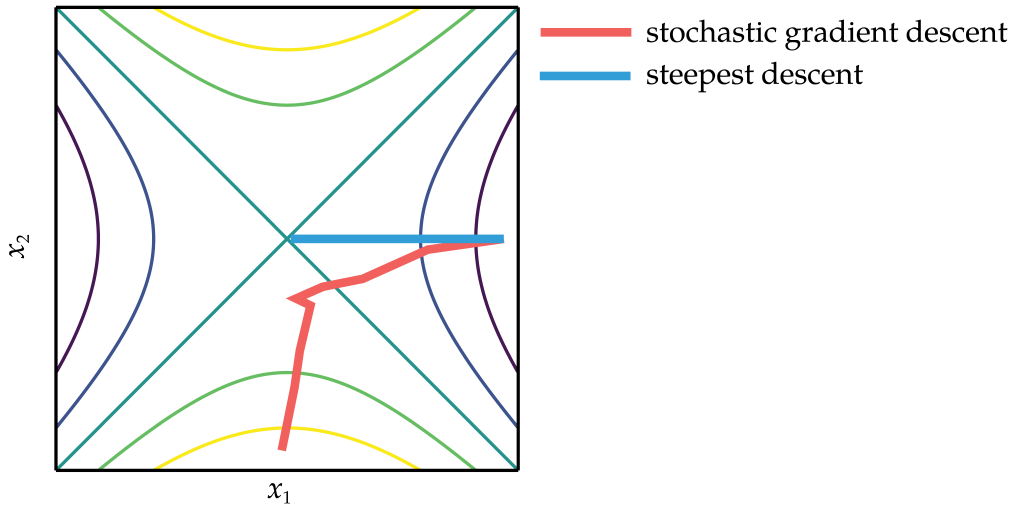


Figure: Adding stochasticity to a descent method helps with traversing saddle points such as $f(x) = x_1^2 - x_2^2$ shown here.

Comments

This figure provides a visual comparison between standard gradient descent and noisy gradient descent. The plot shows how the two methods behave in a two-dimensional non-convex landscape. Specifically, the underlying function has a saddle point, and both algorithms are initialized from the same starting point.

In the figure, we see the trajectory of the standard gradient descent (blue line). Since the gradient near the saddle point becomes very small, the algorithm stalls and converges to that region. The path becomes flat, and no further progress is made. This illustrates the known issue that gradient descent may get stuck in saddle points, especially in high-dimensional settings where such points are ubiquitous.

Also, we can observe the trajectory of the noisy descent (red line). Each update is perturbed with a small amount of random noise. As a result, the trajectory deviates from the saddle point and continues exploring the landscape. The noisy perturbations help the algorithm escape the flat region and eventually reach a more promising area.

This example illustrates a key advantage of stochastic optimization: adding carefully controlled randomness allows the method to escape from plateaus and saddle regions that would trap deterministic methods. In practice, this kind of behavior often makes the difference between failure and successful optimization, especially in deep learning and other non-convex settings.

Mesh Adaptive Direct Search (MADS): Overview

What is MADS?

- A derivative-free optimization method.
- Generalization of classic direct search techniques (e.g., pattern search).
- Evaluates objective at discrete mesh points that evolve over time.

Key Ideas:

- Maintain a mesh — a discretized subset of the search space.
- At each iteration, evaluate objective at nearby mesh points.
- Mesh size is adapted dynamically based on progress.
- Use flexible search and polling directions for robustness.

Note: MADS is well suited for expensive black-box functions and can handle nonsmooth and nonconvex problems.

Comments

Mesh Adaptive Direct Search, or MADS, is a class of derivative-free optimization algorithms designed for problems where gradients are unavailable, unreliable, or too expensive to compute. It builds on the ideas of classical direct search methods which we discussed earlier but introduces two key enhancements: the use of an adaptive mesh, and more flexible sets of search directions.

At each iteration, MADS defines a finite set of points, called the mesh, around the current iterate. These points form a discretized grid, and the algorithm evaluates the objective function only at mesh points. The idea is to control the resolution of the search through a mesh size parameter — if improvement is observed, the mesh is refined; if not, it is coarsened.

Another important concept is the polling step, where the algorithm probes directions around the current point. The set of polling directions can vary from iteration to iteration and need not be limited to coordinate axes. This makes MADS more robust and efficient, especially in higher dimensions or irregular landscapes.

MADS is particularly well suited for black-box problems, where the objective function is expensive, noisy, or nondifferentiable. Examples include engineering design, simulations, and optimization under uncertainty. The method has theoretical convergence guarantees under mild assumptions, making it both practical and rigorous.

MADS: Step Size and Direction Sampling

Step Size $\alpha(\mathrm{k})$:

- Controls mesh resolution: smaller $\alpha \rightarrow$ finer search.
- Initialized as $\alpha(1) = 1$, always a power of 4.
- Updated as:

$$\alpha(\mathrm{k}+1) = \begin{cases} \alpha(\mathrm{k})/4 & \text{if no improvement} \\ \min(1, 4\alpha(\mathrm{k})) & \text{if successful} \end{cases}$$

Generating a Positive Spanning Set:

- Build an $\mathrm{n} \times \mathrm{n}$ lower-triangular matrix L:
  - Diagonal entries: randomly $\pm 1/\sqrt{\alpha(\mathrm{k})}$
  - Lower-triangle: random integers in

$$\left\{ -\frac{1}{\sqrt{\alpha(\mathrm{k})}} + 1, \ldots, \frac{1}{\sqrt{\alpha(\mathrm{k})}} - 1 \right\}$$

- Permute rows and columns of L to obtain $\mathrm{D} \in \mathbb{R}^{\mathrm{n} \times \mathrm{n}}$
- Add direction $\mathrm{d}^{(\mathrm{n}+1)} = -\sum_{\mathrm{i}=1}^{\mathrm{n}} \mathrm{d}^{(\mathrm{i})}$

Note: The resulting $\mathrm{n}+1$ directions form a positive spanning set — any vector can be expressed as a nonnegative combination of them.

Comments

The step size $\alpha(\mathrm{k})$ is the main parameter controlling the resolution of the mesh. It determines how finely the algorithm samples the space around the current point. Smaller values of alpha correspond to a finer mesh, enabling local search, while larger values promote exploration. The step size is always a power of four. It is initialized as one, and at each iteration it is either divided by four if no progress was made, or multiplied by four if the iteration was successful — but never increased beyond one.

To generate directions, MADS builds a lower-triangular matrix L of size n by n. The diagonal entries are randomly chosen from plus or minus one divided by the square root of $\alpha(\mathrm{k})$. The entries below the diagonal are random integers from the set of integers strictly between $1 - \frac{1}{\sqrt{\alpha(\mathrm{k})}}$ and $\frac{1}{\sqrt{\alpha(\mathrm{k})}} - 1$. These entries ensure that the matrix has bounded entries depending on the current step size.

After that, the rows and columns of L are randomly permuted to produce a direction matrix D. This matrix contains n linearly independent directions. A final direction is added: minus the sum of the previous n directions. The resulting $\mathrm{n}+1$ directions are guaranteed to form a positive spanning set — meaning that any vector in the space can be written as a nonnegative combination of them. This guarantees full coverage of the space.

MADS: Example in 2D

Setup:

- Dimension $n = 2$, current iterate $k = 2$ (on the first step L is the identity matrix).
- Step size $\alpha(k) = 1/4 \Rightarrow \sqrt{\alpha(k)} = 1/2$
- Diagonal entries $= \pm 2$ (since $1/\sqrt{\alpha(k)} = 2$)

Construct L and D (random permutation of rows and columns of L):

$$L = \begin{bmatrix} -2 & 0 \\ 1 & 2 \end{bmatrix}, \; D = \begin{bmatrix} 1 & 2 \\ -2 & 0 \end{bmatrix} \Rightarrow d^{(3)} = -d^{(1)} - d^{(2)} = \begin{bmatrix} -3 \\ 2 \end{bmatrix}$$

- Evaluate f at $x^{(k)} + \sqrt{\alpha(k)} \cdot d^{(i)}$ for $i = 1, 2, 3$
- If improvement found at any direction, set $x^{(k+1)} = x^{(k)} + \sqrt{\alpha(k)} \cdot d^{(i)}$. Otherwise $x^{(k+1)} = x^{(k)}$.
- Update step size:

$$\alpha(k+1) = \begin{cases} \alpha(k)/4 = 1/16 & \text{if no improvement} \\ \min(1, 4\alpha(k)) = 1 & \text{otherwise} \end{cases}$$

Note: In 2D, MADS explores 3 directions per step forming a positive spanning set in $\mathbb{R}^2$.

Comments

This example illustrates how the Mesh Adaptive Direct Search algorithm works in two dimensions. We assume that we have already performed the first iteration, for which the matrices L and D coincide with the identity, so k is equal to 2 and the step size $\alpha(k)$ is one fourth. Then, the square root of alpha is one half, and the inverse of that is two. According to the algorithm, we place randomly values plus or minus two on the diagonal of matrix L.

In this specific case, we build a two-by-two lower triangular matrix L with diagonal entries minus two and plus two, and a lower triangular entry equal to one. We then randomly permute rows and columns to produce the matrix D, which defines our search directions.

To form a positive spanning set, we add a third direction: minus the sum of the first two columns of D. This gives us a total of three directions to explore from the current point. Each direction is scaled by the $\sqrt{\alpha(k)}$, which in this case is one half.

The algorithm evaluates the objective function at each of these three candidate points. If none improve the objective, the step size alpha is divided by four. If any direction leads to improvement, the algorithm moves to that point and multiplies alpha by four, but never lets it exceed one.

This process adapts both the geometry and resolution of the search, and ensures that MADS retains coverage of the entire space while still focusing locally.

```
function rand_positive_spanning_set(α, n)
    δ = round(Int, 1 / sqrt(α))                  # Discretization level
    L = Matrix(Diagonal(δ .- rand([1, -1], n)))
    for i in 1:n-1
        for j in 1:i-1
            L[i, j] = rand(-δ+1 : δ-1)  # Random integers in lower triangle
        end
    end
    D = L[randperm(n), :]       # Random row  permutation
    D = D[:, randperm(n)]       # Random column permutation
    D = hcat(D, -sum(D, dims=2)) # Combining the matrix and column
    return [D[:, i] for i in 1:n+1]
end
```

Algorithm 10: MADS

```
function mesh_adaptive_direct_search(f, x, ϵ)
    α = 1                       # Initial step size
    y = f(x)                    # Current objective
```


Comments

These two functions implement the core logic of the Mesh Adaptive Direct Search method. The first function, called `rand_positive_spanning_set`, is responsible for generating a set of directions that positively span the space. This means that any vector in the space can be expressed as a nonnegative combination of these directions. The method ensures this by constructing a lower triangular matrix with randomly signed diagonal entries and randomly filled lower triangle. After that, it permutes the rows and columns to introduce more variability. Finally, it adds one more direction: the negative sum of all previously generated directions. This guarantees that the resulting set spans the space in a positively complete way.

The second function, `mesh_adaptive_direct_search`, is the main optimization loop. It starts from a given initial point and uses a step size alpha to probe trial points.

MADS: realizion (final)

```
    while α > ϵ
        improved = false
        for (i, d) in enumerate(rand_positive_spanning_set(α, length(x)))
            x′ = x + α * d                                  # Trial point
            y′ = f(x′)                                  # Evaluate objective
            if y′ < y
                x, y, improved  = x′, y′,true      # Accept improvement
                x′ = x + 3α * d       # Opportunistic second step (3×α)
                y′ = f(x′)
                if y′ < y
                    x, y = x′, y′           # Accept second improvement
                end
                break                        # Skip remaining directions
            end
        end
        α = improved ? min(4α, 1) : α / 4       # Update mesh step size
    end
    return x                                      # Final design point
end
```


Comments

On each iteration, a new set of directions is generated using the first function. Then, the algorithm evaluates the objective function at trial points obtained by stepping in those directions.

If a trial point improves the objective, it is immediately accepted. The method also includes an opportunistic step: it tries moving even farther in the same direction to potentially accelerate progress. If that second point is also better, it is accepted too. After each iteration, the step size is updated — increased if improvement occurred, or reduced otherwise. This dynamic adaptation allows the method to focus locally when necessary, but also escape local traps when needed.

Together, these two functions define a robust derivative-free optimization strategy that is well suited to black-box, noisy, or nonsmooth problems.

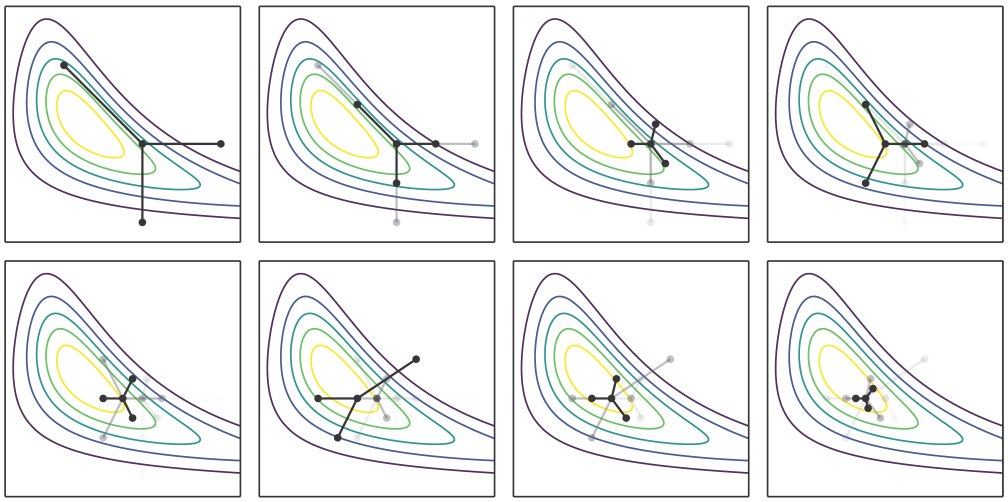


Figure: MADS: Search Process

Comments

This figure illustrates the behavior of mesh adaptive direct search as it explores the search space. The trajectory proceeds from left to right and top to bottom, showing the sequence in which points are evaluated and accepted. At each iteration, the algorithm samples a set of positive spanning directions and evaluates points by stepping in those directions using a mesh-defined step size.

When a trial point improves the objective function, it is accepted, and the mesh is refined by increasing the step size. If no improvement occurs, the step size is reduced. This adaptivity balances local refinement and global exploration.

While the algorithm also includes an opportunistic second step in the same direction after a successful move, such steps are not explicitly illustrated in the figure. The plot focuses on showing accepted points and how the method adapts its resolution as it moves across the domain.

Overall, the figure demonstrates how MADS moves through the space in a structured, direction-driven fashion, dynamically adjusting its granularity in response to the optimization landscape. It highlights the algorithm's ability to balance exploration and refinement even without derivative information.

Simulated Annealing: Core Idea

Inspired by metallurgy:

- Temperature controls the amount of randomness.
- High temperature → free exploration.
- Gradual cooling → convergence to a minimum.

Main Idea:

- At each step, propose a candidate $x' \sim T(x)$ ($T(x)$ is a transition distribution).
- Accept with Metropolis criterion:

$$\mathbb{P}(\text{accept}) = \begin{cases} 1, & \Delta y \leq 0 \\ e^{-\Delta y / t}, & \Delta y > 0 \end{cases}$$

- $\Delta y = f(x') - f(x)$, t is the temperature.

Key Benefit: The method escapes local minima when temperature is high, and converges as the temperature cools.

Comments

Simulated annealing is a stochastic optimization algorithm inspired by physical annealing in metallurgy. In that process, a material is first heated and then cooled slowly. When hot, the atoms are highly mobile and can rearrange themselves into more stable configurations. As the temperature decreases, the material settles into a low-energy, ordered structure. Simulated annealing mimics this by using a "temperature" parameter to control the level of randomness during optimization.

At each iteration, a new candidate point is proposed using a transition distribution T. If the new point reduces the objective function, it is always accepted. If it increases the objective, it may still be accepted with a certain probability. This probability decreases as the increase in the objective grows, and also as the temperature goes down. The acceptance rule is called the Metropolis criterion.

This rule is what allows the method to escape local minima. When the temperature is high, even worse points may be accepted, which allows broader exploration of the search space. As the temperature decreases, the algorithm becomes more conservative, eventually converging to a local or global minimum.

Simulated Annealing: Annealing Schedules

The temperature schedule $t(k)$ controls the acceptance probability.

- Slow cooling ensures convergence to the global optimum.
- Rapid cooling may miss global minima.

Standard Annealing Schedules:

- Logarithmic:
$$t(k) = \frac{t(1)\ln 2}{\ln(k+1)}$$
- Exponential:
$$t(k+1) = \gamma^k \cdot t(1), \quad \gamma \in (0,1)$$
- Fast:
$$t(k) = \frac{t(1)}{k}$$

Note: Only logarithmic cooling guarantees convergence, but it is too slow for most practical problems.

Comments

The parameter called temperature in simulated annealing is a numerical value that controls how strongly the algorithm prefers lower objective values over higher ones. It defines the level of randomness in decision-making during optimization.

At every iteration, a candidate solution is proposed. If the candidate has a lower objective value, it is accepted. If it has a higher value, it can still be accepted, but only with a probability that depends on the temperature: the higher the temperature, the more likely it is to accept worse points.

As optimization proceeds, the temperature is decreased according to a predefined rule, called the annealing schedule. This is what allows the algorithm to explore the space widely at first and then focus later.

Three commonly used cooling schedules are logarithmic, exponential, and fast. Logarithmic cooling uses the rule $t(k) = \frac{t(1)\ln 2}{\ln(k+1)}$. This is the only schedule that theoretically guarantees convergence to the global minimum, but it is very slow in practice and rarely used.

The exponential schedule reduces temperature by multiplying it by a fixed constant gamma between zero and one. This is the most commonly used rule in real applications.

The fast schedule reduces temperature as $t(k) = \frac{t(1)}{k}$. It decreases quickly but not as aggressively as the exponential rule, which makes it a useful compromise.

Choosing the right schedule is a trade-off between theoretical guarantees and practical performance. In most real-world applications, exponential or fast schedules are used.

Algorithm 11: Basic simulated annealing.

```
function simulated_annealing(f, x, T, t, k_max)
    y = f(x)
    x_best, y_best = x, y
    for k in 1:k_max
        x′ = x + rand(T)              # propose new point
        y′ = f(x′)                    # evaluate objective
        Δy = y′ - y
        if Δy ≤ 0 || rand() < exp(-Δy / t(k))
            x, y = x′, y′             # accept new point
        end
        if y′ < y_best
            x_best, y_best = x′, y′   # track best seen
        end
    end
    return x_best
end
```


Comments

This is the basic implementation of simulated annealing. The function takes five inputs: the objective function f, the initial point x, the transition distribution T, the annealing schedule t, and the maximum number of iterations k max.

At each iteration, the algorithm samples a new point x prime by adding a random perturbation drawn from the distribution T. The candidate point is then evaluated using the objective function.

The difference in objective values, Δy, is calculated. If the new point is better, meaning $\Delta y \leq 0$, the new point is accepted. If the new point is worse, it is accepted only with a probability equal to the exponential of minus Δy divided by the current temperature.

This acceptance rule allows the algorithm to explore the space and occasionally accept worse points, especially at high temperatures. As the temperature decreases, the method becomes more selective.

The algorithm also tracks the best point found so far. This is important because the current point may fluctuate, especially at high temperatures.

The function returns the best point found during the optimization.

Adaptive Simulated Annealing (Corana et al., 1987)

Main Idea: Adapt the step size v_i for each coordinate i during the search to maintain an acceptance rate near 50%.

Sampling Rule: For each coordinate i, propose:

$$x' = x + r v_i e_i \qquad \text{where } r \sim \mathcal{U}[-1, 1]$$

- e_i — unit vector in direction i.
- v_i — current step size.

Step Size Update (every n_s cycles):

$$v_i \leftarrow \begin{cases} v_i \cdot \left(1 + c_i \dfrac{a_i/n_s - 0.6}{0.4}\right), & a_i > 0.6 n_s \\ v_i / \left(1 + c_i \dfrac{0.4 - a_i/n_s}{0.4}\right), & a_i < 0.4 n_s \\ v_i, & \text{otherwise} \end{cases}$$

- a_i — number of accepted moves along coordinate i.
- c_i — update coefficient (typically $c_i = 2$).
- Balances efficiency: too many rejections → reduce step; too many acceptances → increase step.

Comments

Simulated annealing is usually implemented using full-dimensional updates, where each candidate point is generated by adding a random vector to the current point. However, in high-dimensional spaces, this approach can be inefficient. It is difficult to tune the size and direction of updates globally.

To address this, Corana and collaborators proposed a coordinate-wise version of simulated annealing. Instead of updating all coordinates at once, this method updates one coordinate at a time. At every iteration, a candidate point is generated by taking a step along this coordinate, where the step length is a random number between -1 and 1, multiplied by v_i - the current step size for that coordinate.

Accepted moves are tracked individually for each direction. After a fixed number of cycles, the step sizes are updated. If the acceptance rate in a coordinate is above sixty percent, the step size is increased. If the acceptance rate is below forty percent, it is decreased. Otherwise, it is left unchanged.

The goal is to keep the acceptance rate around 50 percent. Accepting too many moves means the steps are too small and inefficient. Rejecting too many means the steps are too large and likely overshoot. The update rule adjusts the steps accordingly using a scaling factor, typically set to two.

This coordinate-wise and adaptive control improves efficiency in problems where different variables behave differently. It avoids the need for manual step-size tuning and adapts automatically during optimization.

```julia
function corana_update!(v, a, c, ns)
     for i in 1:length(v)
          ai, ci = a[i], c[i]  # accepted count & scaling factor
          if ai > 0.6 * ns     # too many acceptances → increase step
               v[i] *= (1 + ci * (ai/ns - 0.6) / 0.4)
          elseif ai < 0.4 * ns # too few → reduce step
               v[i] /= (1 + ci * (0.4 - ai/ns) / 0.4)
          end
     end
     return v
end
```

Algorithm 12: Coordinate-wise adaptive SA.

```julia
function adaptive_simulated_annealing(f, x, v, t, ϵ;
    ns=20, nϵ=4, nt=max(100, 5 * length(x)),
    γ=0.85,  c=fill(2, length(x)) )
    y = f(x)                        # initial value
    x_best, y_best = x, y          # best design seen
```


Comments

We begin with the implementation of Corana's update rule. The function `corana_update!` adjusts step sizes for each coordinate individually based on the number of accepted proposals. If the acceptance rate in a coordinate exceeds 60%, the step is increased. If it's below 40%, the step is reduced. The update is multiplicative and aims to keep acceptance around 50% for optimal balance between exploration and convergence.

The main algorithm `adaptive_simulated_annealing` initializes the current and best design points, as well as all necessary parameters: the step vector v, the current temperature t, and the counters. It also creates the vector c, which controls how aggressively the steps are updated. The default for c is 2, as recommended in the original paper.

```
    y_arr = [] # state history
    n, U = length(x), Uniform(-1.0, 1.0)  # dim, distribution
    a, cycles, resets = zeros(n), 0, 0 #accepted counts, cycle/reset counters
    while true
        for i in 1:n
            x′ = x + basis(i, n) * rand(U) * v[i]  # propose in coordinate i
            y′ = f(x′)
            Δy = y′ - y
            if Δy < 0 || rand() < exp(-Δy / t)     # Metropolis acceptance
                x, y = x′, y′
                a[i] += 1
            if y′ < y_best; x_best, y_best = x′, y′; end
        end
        cycles += 1
        if cycles < ns
          continue                                # wait until enough samples
        end
        cycles = 0
```


Comments

The core loop runs indefinitely until the convergence condition is met. In each cycle, the algorithm performs coordinate-wise updates: one coordinate at a time is perturbed with a step scaled by the current step size and a uniform random factor in $[-1, 1]$. The resulting point is evaluated and either accepted if it improves the objective, or — with some probability — even if it worsens it, according to the Metropolis criterion.

Accepted moves increment the corresponding entry in the acceptance vector a. The best solution seen so far is updated whenever an improvement is found. Once the number of coordinate updates reaches ns, we proceed to update the step sizes.

```
        corana_update!(v, a, c, ns)                # adjust step sizes
        fill!(a, 0)                                # reset accepted counters
        resets += 1
        if resets < nt
            continue                               # wait until enough resets
        end

        t *= γ                                     # reduce temperature
        resets = 0
        push!(y_arr, y)                            # save history
        if !(length(y_arr) > nϵ && y_arr[end] - y_best ≤ ϵ &&
            all(abs(y_arr[end] - y_arr[end-u]) ≤ ϵ for u in 1:nϵ))
            x, y = x_best, y_best            # restart from best if not stable
        else
            break                                  # convergence detected
        end
    end
    return x_best                                  # return best design
end
```


Comments

After updating the step sizes, the algorithm checks whether a full temperature cycle has completed. If so, the temperature is reduced by the cooling rate γ. It also appends the current value to the history array.

Finally, convergence is detected if the objective value has not changed significantly for the last $n\epsilon$ resets and is close to the current best value. If the algorithm stagnates, it restarts from the best point found. Otherwise, it exits and returns that point as the final design.

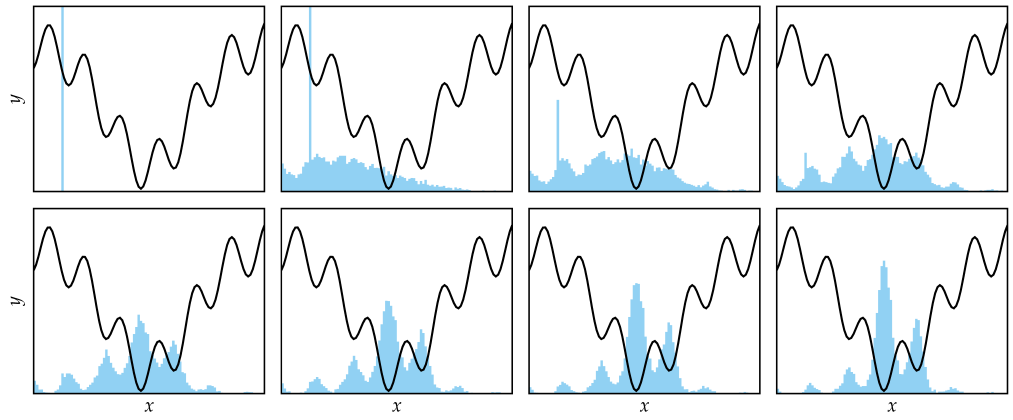


Figure: The adaptive simulated annealing.

Comments

This figure illustrates how simulated annealing behaves over time under exponential temperature decay. Each histogram shows the distribution of the algorithm's position at different stages of the optimization process.

At high temperatures, the search is nearly uniform — the algorithm explores the space broadly, accepting both better and worse proposals. As the temperature decreases, the acceptance probability for uphill steps diminishes, and the algorithm starts to concentrate in regions of lower function values.

In the final stages, the distribution becomes narrower, with most of the samples clustered near minima. However, we still observe three distinct peaks rather than a single one.

This occurs because the temperature has not dropped low enough for the algorithm to fully commit to a single minimum. The system still has enough randomness to occasionally visit nearby local minima, which preserves residual probability mass in multiple regions. This means the algorithm has not yet fully converged — it is still transitioning from global exploration to local refinement.

With further cooling and more iterations, the distribution would eventually collapse to a sharp single peak centered at the global minimum. So, this figure shows a transient stage, not the final convergence.