PART II. OPTIMIZATION: NUMERICAL APPROACHES
(LECTURE 1)

Shpilev Petr Valerievich

Faculty of Mathematics and Mechanics, SPbU

September, 2025

Санкт-Петербургский государственный университет

Comments

We begin the next part of our course devoted to numerical approaches to optimization, In this lecture we focus on direct methods for unconstrained optimization problems. We start by introducing cyclic coordinate search and its variant, cyclic coordinate descent, with and without an acceleration step, to optimize functions over a set of coordinates iteratively. Next, we explore Powell's method, its underlying motivation, and the search process that combines direction sets to find optimal solutions. We then cover the Hooke-Jeeves method, which employs pattern searches for exploration, and introduce the generalized pattern search (GPS) algorithm with its positive spanning set. The lecture also delves into the Nelder-Mead method, focusing on its geometric operations and search techniques for multidimensional optimization. Finally, we introduce the DIRECT (Divided Rectangles) method for global optimization, explaining how it approximates the global lower bound and applies the Lipschitz condition to select promising intervals for further exploration, ensuring efficient global search strategies.

Direct Methods: Introduction

What are Direct Methods?

- Also known as zero-order, black box, pattern search, or derivative-free methods.
- Rely solely on the objective function f(x).

Key Characteristic:

- Do not use derivative information ($\nabla f(x)$, $\nabla^2 f(x)$) to guide the search or identify convergence.
- Use other criteria for choosing search directions and judging convergence.

Why use them?

- Useful when derivatives are unavailable, difficult, or expensive to compute.
- Common in engineering, simulations, or when f(x) is a "black box" output.

Comments

Direct methods are a fundamental class of optimization techniques that do not rely on derivative information. Instead, they operate by evaluating the objective function — which we typically denote by f — at carefully chosen points, making decisions based solely on these function values. That is why they are often called zero-order methods, meaning they use no first or second-order derivative information.

Another term you will often encounter is "black box methods." This reflects the fact that these algorithms treat the objective function as a black box — they can query it, meaning they can compute f(x) at different points, but they have no access to its internal structure, formula, or derivatives.

Direct methods are especially useful for problems where derivatives are unavailable, unreliable, or expensive to compute. This can happen in simulation-based optimization, engineering design, or machine learning hyperparameter tuning, where evaluating f may involve running an experiment or a complex computational model.

The way these methods choose where to evaluate f next often follows a systematic rule or heuristic. A heuristic, simply put, is a practical rule or strategy designed to make the search more efficient, even though it does not guarantee an optimal decision at each step.

For example, some direct methods search along coordinate directions, others use geometric patterns or randomly generated search directions. Regardless of the specifics, they all share the key feature of using only function evaluations to guide the search process.

However, one limitation to keep in mind is that direct methods tend to converge more slowly than gradient-based methods on smooth, well-behaved functions. When derivative information is available and reliable, gradient-based methods can exploit it to accelerate convergence.

Despite this, direct methods remain widely used due to their simplicity, broad applicability, and robustness, especially when the problem structure is unknown or the objective is noisy or non-differentiable.

In today's lecture, we will look at a few popular and widely used in practice direct methods.

Cyclic Coordinate Search

Cyclic coordinate search, also known as coordinate descent or taxicab search, is a simple direct method that optimizes one coordinate at a time.

Algorithm Overview:

- Start from an initial point $x^{(1)}$.
- Perform a line search along the first coordinate direction $e^{(1)}$:
$$x^{(2)} = \arg\min_{x_1} f(x_1, x_2^{(1)}, \ldots, x_n^{(1)})$$
- Then optimize the next coordinate (along $e^{(2)}$):
$$x^{(3)} = \arg\min_{x_2} f(x_1^{(2)}, x_2, x_3^{(2)}, ..., x_n^{(2)})$$
- Proceed sequentially along each coordinate direction $e^{(i)}$ (the i-th basis vector).
- One full pass over all coordinates is called a cycle.

Note: Like steepest descent, cyclic coordinate search guarantees non-increasing objective value at each step, but may get stuck if optimal directions lie between coordinate axes.

Comments

Cyclic coordinate search is one of the simplest direct optimization methods. It is also referred to as coordinate descent or, sometimes, taxicab search. The key idea is that instead of searching in arbitrary directions or requiring gradient information, this method optimizes the objective function by iteratively adjusting one coordinate at a time, while keeping the others fixed.

The process begins with an initial guess, denoted as $x^{(1)}$. The algorithm then performs a line search along the first coordinate direction. That means, it varies only the first component, x_1, while keeping the remaining components fixed, and finds the value that minimizes the objective function f along that axis.

Once the first coordinate is optimized, the algorithm proceeds to the second coordinate, repeating the same procedure. This sequential process continues until all n coordinates have been optimized once — this is called a full cycle. After each cycle, the method checks for convergence, often based on how much the solution has changed or how much the objective function has improved. No significant improvement after a full cycle over all coordinates indicates that the method has converged.

To perform these searches, the algorithm uses what we call basis vectors. For an n-dimensional space, the i-th basis vector, $e^{(i)}$, is a vector with all zeros except for a one in the i-th position. These vectors define the directions along which the line searches occur.

A key advantage of this method is that, like steepest descent, it guarantees that the objective function value does not increase at each step. However, cyclic coordinate search can get stuck in cases where the optimal direction is not aligned with any coordinate axis. This is a limitation inherent to its simplicity.

Despite this, the method is widely used due to its ease of implementation and low computational cost, especially for high-dimensional problems.

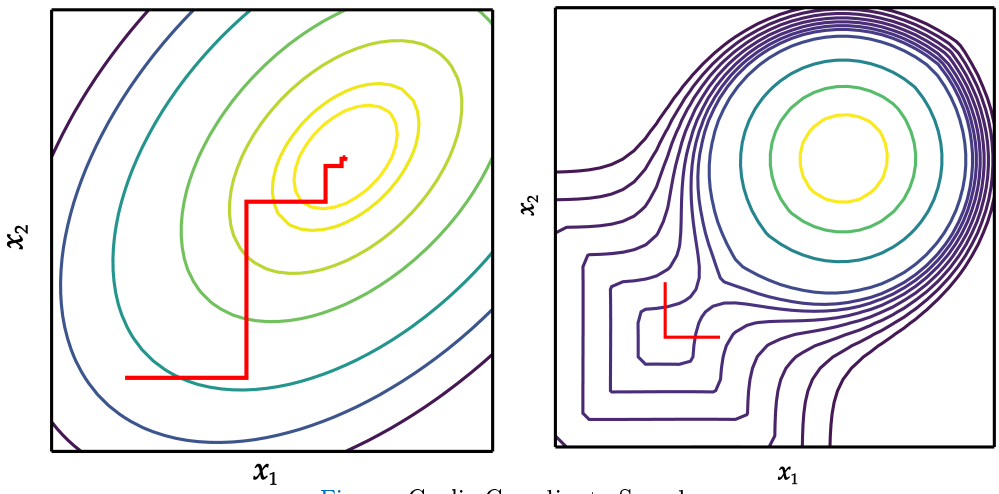


Figure: Cyclic Coordinate Search

Comments

These figures illustrate both the core idea and the main limitation of cyclic coordinate descent.

In left figure, you can see how the method alternates between coordinate directions during optimization. The trajectory consists of straight-line segments that align strictly with the coordinate axes. First, the algorithm moves along the horizontal axis, then along the vertical axis, and so on. This step-by-step movement reflects the simple nature of the method — it optimizes one variable at a time while keeping the others fixed.

However, the right figure demonstrates a fundamental drawback of this approach. The current point is located at a position where moving in either coordinate direction — horizontally or vertically — only increases the objective function f. In other words, a line search along those directions cannot improve the solution. Yet, there exists a diagonal direction where f decreases.

Unfortunately, cyclic coordinate search does not consider diagonal or arbitrary directions, so it gets stuck in such situations.

This example highlights that while the method is simple and reliable along coordinate directions, it may fail to find even a local minimum if the optimal descent direction does not align with the axes.

Cyclic Coordinate Descent: Basic Algorithm

Key Idea: The cyclic coordinate descent iteratively updates each coordinate using one-dimensional line searches until convergence.

Algorithm Cyclic Coordinate Descent

```
function cyclic_coordinate_descent(f, x, ϵ)
    Δ = ∞, n = length(x)
    while abs(Δ) > ϵ do
        x′ = copy(x)
        for i in 1 : n do
            d = basis(i, n)                ▷ returns the basis vector e^(i)
            x = line_search(f, x, d)
        end for
        Δ = ‖x − x′‖
    end while
    return x
end function
```

Note: This method may get stuck far from a local minimum if optimal descent directions do not align with coordinate axes.

Comments

This slide summarizes the practical implementation of the basic cyclic coordinate descent method, a classic approach for derivative-free optimization.

The algorithm receives two inputs: the objective function f and an initial point x. It also requires a convergence tolerance ϵ, which determines when the algorithm should stop. The core idea is straightforward — at each iteration, the method performs a line search along each coordinate direction sequentially.

The loop starts by saving a copy of the current point x′. Then, for each coordinate i, a basis vector d is constructed — this is a vector with 1 at the i-th position and zeros elsewhere. A one-dimensional line search is then performed along d, optimizing f with respect to the i-th variable.

After all coordinates have been updated, the algorithm computes the total change in x. If this change is smaller than the tolerance ϵ, the method terminates. Otherwise, another full cycle of coordinate updates begins.

It is important to note that while this method is simple and easy to implement, it may fail to find even a local minimum if the optimal search direction is not aligned with the coordinate axes, as illustrated earlier.

Cyclic Coordinate Descent with Acceleration Step

Key Idea: The acceleration step addresses limitations of axis-aligned searches by introducing an additional line search after each full coordinate cycle.

Algorithm Cyclic Coordinate Descent with Acceleration

```
function cyclic_coordinate_descent_with_acceleration_step(f, x, ϵ)
    Δ = ∞, n = length(x)
    while abs(Δ) > ϵ do
        x′ = copy(x)
        for i in 1 : n do
            d = basis(i, n)
            x = line_search(f, x, d)
        end for
        x = line_search(f, x, x − x′)                ▷ acceleration step
        Δ = ‖x − x′‖
    end while
    return x
end function
```

Note: The acceleration step allows exploration of diagonal directions, improving convergence in curved or narrow valleys.

Comments

This slide presents an improved version of the basic cyclic coordinate descent algorithm, which includes an acceleration step to overcome the limitations of purely axis-aligned searches.

As we saw earlier, standard coordinate descent optimizes along each coordinate direction sequentially. However, this approach can fail to make progress if the optimal descent direction lies diagonally or along some combination of coordinates. The acceleration step provides a simple yet effective remedy for this issue.

After completing a full cycle of updates along all coordinates, the algorithm performs an additional line search. This search follows the vector connecting the starting point of the cycle, x', to the point after the last coordinate update, x. Essentially, this direction approximates how the algorithm moved through the parameter space during the cycle and often captures useful diagonal directions that standard coordinate steps miss.

The added line search requires minimal computational effort, yet it significantly improves the algorithm's ability to traverse curved valleys or narrow ridges in the objective function landscape. As a result, convergence is typically faster and more reliable, especially in problems where the coordinate axes are not aligned with the function's inherent structure.

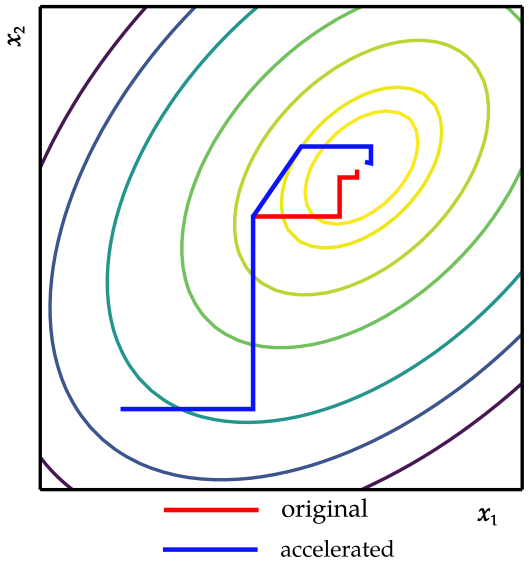


Figure: Cyclic Coordinate Search with Acceleration Step

Comments

This figure visually demonstrates the practical benefit of adding an acceleration step to the cyclic coordinate descent method.

The plot compares two search trajectories for the same optimization problem: the original cyclic coordinate descent and its accelerated version. Both methods start from the same initial point, and six steps of each method are shown.

In the case of the basic method, the trajectory follows a characteristic staircase pattern. Each step moves strictly along one coordinate direction, either horizontally or vertically. While this pattern is simple and systematic, it can be inefficient when the optimal descent path lies diagonally or along a curved valley, as is common in many real-world problems.

The accelerated version includes an additional line search along the direction connecting the start and end points of each full coordinate cycle. Thanks to this step, the trajectory cuts across the valley more effectively, moving in directions that combine coordinate axes. As a result, the method progresses faster toward the minimum.

This example clearly illustrates how even a minor algorithmic modification can significantly improve performance, particularly when dealing with curved or narrow optimization landscapes.

Powell's Method: Concept and Motivation

Powell's method is a powerful enhancement of coordinate descent that searches along directions that adapt over time, rather than being limited to coordinate axes.

Key Advantages:

- Allows searches along arbitrary directions, not just axis-aligned.
- Efficient for long, narrow valleys where coordinate descent struggles.
- Adapts search directions based on accumulated progress.
- Particularly effective for poorly scaled or ill-conditioned problems.

Note: Powell's method can "learn" conjugate directions, leading to faster convergence for quadratic or near-quadratic objective functions.

Illustration:

- Initially follows coordinate descent.
- Progressively updates directions based on the path taken.

Comments

Powell's method is a refinement of coordinate descent that overcomes one of its key limitations — the inability to efficiently handle narrow, curved valleys in the optimization landscape. In standard coordinate descent, we only search along axis-aligned directions, which can lead to slow progress in such valleys. Powell's method addresses this by adaptively updating the search directions based on the history of the optimization process.

The method begins similarly to coordinate descent, using the coordinate axes as the initial set of directions. However, after completing a full cycle of searches along these directions, it replaces one of them — typically the oldest — with a direction representing the net progress made over the cycle. This new direction often captures the curved structure of the valley, allowing the method to make faster progress in subsequent iterations.

An important property of Powell's method is its ability to "learn" conjugate directions. For quadratic functions, these directions guarantee that the function can be minimized efficiently, often in a small number of iterations.

However, it's important to note that while Powell's method improves upon coordinate descent, it still relies on line searches in each chosen direction. Its success depends on the quality of those searches and the structure of the objective function.

In practice, Powell's method is especially effective for problems where the objective function exhibits elongated, curved valleys — a common feature in many engineering and machine learning optimization tasks.

Search Directions and Updates:

- Maintain n search directions $u^{(1)}, \ldots, u^{(n)}$.
- Initially set $u^{(i)} = e^{(i)}$ — standard basis vectors.
- For each direction $u^{(i)}$, perform a line search:

$$x^{(i+1)} \leftarrow \text{line\_search}(f, x^{(i)}, u^{(i)})$$

- After all n searches:
  - Shift directions: $u^{(i)} \leftarrow u^{(i+1)}$ for $i = 1, \ldots, n-1$.
  - Replace last direction with overall progress:

$$u^{(n)} \leftarrow x^{(n+1)} - x^{(1)}$$

- Perform one more line search along $u^{(n)}$.

Note: Replacing old directions with accumulated progress captures curved search paths and improves efficiency.

Comments

Powell's method proceeds by maintaining and systematically updating a set of search directions. Initially, these directions are simply the standard basis vectors aligned with the coordinate axes. This makes the method start similarly to coordinate descent.

During each iteration, a line search is performed along each of these directions in sequence. After completing all n searches, we shift the list of directions: the first direction is discarded, each remaining direction is shifted down by one position, and the last direction is replaced with the vector representing the net progress made during the entire iteration.

This final direction — the difference between the final and initial points of the cycle — effectively captures the overall trend of the optimization trajectory. By adding this direction, Powell's method "learns" from the optimization history, enabling it to adapt to curved or non-axis-aligned valleys in the objective function.

Finally, an additional line search is performed along this new direction to complete the iteration.

This simple yet elegant update mechanism allows Powell's method to accelerate convergence, particularly in functions with elongated, curved valleys, which often cause difficulties for purely axis-aligned methods.

It is important to note that this process introduces flexibility to the search but also potential instability if the directions become linearly dependent — a topic addressed in later slides.

Powell's Method: Implementation

Algorithm Powell's Method

```
1: function powell(f, x, ϵ)
2:     n = length(x)
3:     U = [basis(i, n) for i in 1 : n]
4:     Δ = ∞
5:     while Δ > ϵ do
6:         x′ = x
7:         for i in 1 : n do
8:             x = line_search(f, x, U[i])
9:         end for
10:        for i in 1 : n − 1 do
11:            U[i] = U[i + 1]
12:        end for
13:        U[n] = x − x′
14:        x = line_search(f, x, U[n])
15:        Δ = ‖x − x′‖
16:    end while
17:    return x
18: end function
```


Comments

This slide outlines the full pseudocode of Powell's method. The method begins with an initial guess x and a set of directions, which are initialized to the standard basis vectors. The algorithm continues iterating until the step size over one full iteration — denoted by Δ, the norm of the change in x — becomes smaller than a chosen tolerance ϵ.

Within each iteration, the method first saves the current point x'. It then performs a sequence of n line searches, each along one of the directions in the list U. These searches sequentially update the solution vector.

Once all line searches are complete, the directions are updated: all existing directions are shifted forward by one position, and the last direction is replaced by the vector difference $x - x'$, which captures the net progress over the cycle. A final line search is performed along this new direction, and then the step size is recomputed.

This cycle is repeated until convergence. The algorithm is efficient in terms of iteration count, but note that it requires $n + 1$ line searches per iteration, which can be expensive if each function evaluation is costly.

The procedure of dropping the oldest search direction in favor of the overall direction of progress can lead the search directions to become linearly dependent. Without search vectors that are linearly independent, the search directions can no longer cover the full design space, and the method may not be able to find the minimum. This weakness can be mitigated by periodically resetting the search directions to the basis vectors. A typical approach is to reset every n or $n + 1$ iterations.

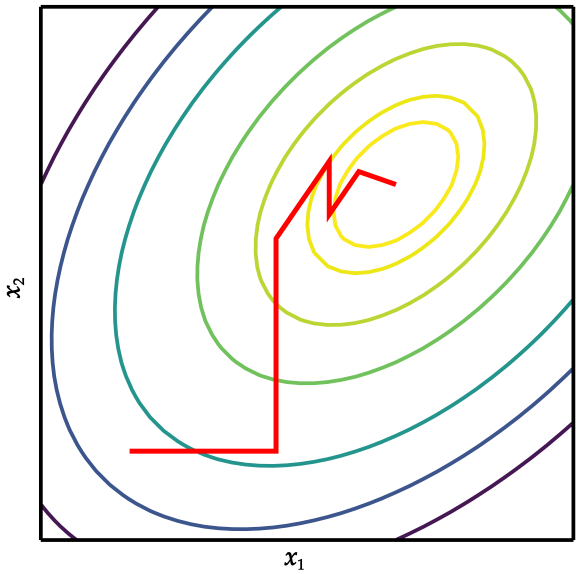


Figure: Powell's Method's Search Process

Comments

This figure illustrates the trajectory of Powell's method during optimization. Initially, the search follows the same pattern as cyclic coordinate descent, moving along coordinate directions. These first steps are axis-aligned, reflecting Powell's use of standard basis vectors as the initial set of directions. However, the key feature of Powell's method emerges as the process continues: it learns new directions that are not limited to the coordinate axes. After each full cycle over the current directions, the method computes an overall direction of progress — the vector from the starting point of the cycle to the final point. This new direction is then added to the list, replacing the oldest one.

As shown in the figure, subsequent steps are no longer aligned with the coordinate axes, but follow directions that better match the curvature of the objective function. The figure highlights how Powell's method improves over standard coordinate descent by using conjugate directions, which are better aligned with narrow valleys. This adaptive behavior allows the method to converge more quickly and efficiently in challenging optimization scenarios where axis-aligned steps would require many iterations.

Hooke-Jeeves Method: Overview

The Hooke-Jeeves method explores the search space using coordinate-wise steps from an anchor point. It relies on function evaluations at small positive and negative steps in each coordinate direction and accepts any improvement found.

Core Procedure:

- From anchor point x, evaluate $f(x \pm \alpha e^{(i)})$ for a given step size α in every coordinate direction.
- Accept any point that yields lower function value.
- If no improvement: reduce step size $\alpha \leftarrow \alpha \cdot \gamma$.
- Repeat until α falls below given tolerance ϵ.

Properties:

- Requires 2n function evaluations per iteration in n dimensions.
- No need for gradients; purely based on evaluating f.
- Can get stuck in local minima.
- Proven to converge for certain function classes.

Comments

The Hooke-Jeeves method is a classic example of a derivative-free, coordinate-based search technique. At its core, the algorithm evaluates small steps in both positive and negative directions along each axis. If any of these steps result in a lower objective value, the algorithm updates the anchor point to this new location. This behavior makes it a greedy method, in the sense that it accepts any improvement found, without evaluating the global picture or future impact.

A key component of the method is the step size α. Initially, it can be relatively large to explore broadly. But once no coordinate direction offers further improvement, this is interpreted as a sign of stagnation — and the step size is decreased by a constant factor γ, often set to 0.5. This decay allows the method to gradually refine the solution with increasingly finer resolution.

An important trade-off in the Hooke-Jeeves method is computational cost: each iteration requires 2n function evaluations. For high-dimensional problems, this becomes expensive quickly. Also, like many local methods, Hooke-Jeeves is prone to getting stuck in local minima if the initial point is not well chosen or the landscape is complex.

However, the method is easy to implement, doesn't require gradient information, and is useful when the objective is noisy, non-differentiable, or hard to model analytically. Under certain regularity conditions, it has been shown to converge to a local minimum.

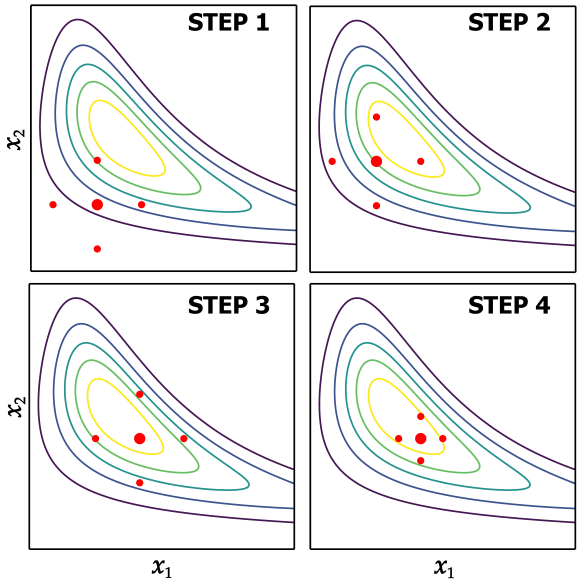


Figure: Hooke-Jeeves Method's Search Process

Comments

This figure illustrates the first four iterations of the Hooke-Jeeves method applied to a two-dimensional optimization problem. Each panel corresponds to a separate iteration. The red dots show the anchor point and the trial points tested in each coordinate direction.

Step 1: The method starts at an initial point and evaluates the objective function at nearby points displaced by a fixed step size along each coordinate axis. Upon finding an improvement, it updates the anchor.

Step 2: The process is repeated from the new anchor. If a better point is found, it becomes the new center for the next step.

Step 3: As the search continues, trial points begin to cluster near regions where the function values are lower.

Step 4: The method continues refining the position, now exploring in a more localized region around the minimum.

This visual sequence highlights the method's systematic exploratory behavior and shows how it gradually narrows down the region of interest. Since it only evaluates the objective function and not its derivatives, the method remains applicable even to non-smooth or noisy functions.

Algorithm Hooke-Jeeves Method

```
function Hooke_Jeeves(f, x, α, ϵ, γ = 0.5)
    y = f(x), n = length(x)
    while α > ϵ do
        improved = false
        x_best = x, y_best = y
        for i in 1 : n do
            for sgn in (−1, 1) do                    ▷ Loop for 2 values: sgn = ±1
                x′ = x + sgn · α · basis(i, n)
                y′ = f(x′)
                if y′ < y_best then
                    x_best = x′, y_best = y′, improved = true
                end if
            end for
        end for
        x = x_best, y = y_best
        if not improved then
            α = α · γ
        end if
    end while
    return x
end function
```


Comments

Let us go step by step through the Hooke–Jeeves method. The main idea is very simple: we do not compute derivatives, we only test how the function value changes when we move in different directions.

We start from an initial guess x and a step size α. You can think of α as the distance of our trial moves. At the beginning, α is relatively large, so we can explore the space broadly. As the algorithm progresses, α will be reduced and we will search more locally.

At each iteration, we perform what is called an exploratory search. For each coordinate direction $i = 1, 2, \ldots, n$ we try two moves: $x' = x + \alpha e_i$ and $x' = x - \alpha e_i$, where e_i is the i-th basis vector. Each trial point is evaluated: $y' = f(x')$. If any of these candidates has a smaller objective value than our current best y, then we accept it as the new best position.

If at least one successful move is found, we update our current point to this better location. Otherwise, if none of the trial moves improves the situation, the algorithm reduces the step size: $\alpha \leftarrow \gamma\alpha$ with $\gamma < 1$ (typically $\gamma = 0.5$). This way, the search becomes more fine-grained.

The process continues until the step size becomes smaller than the tolerance ϵ. At that moment we consider the search finished and return the current best solution.

So the logic is: explore systematically along coordinate axes, accept improvements whenever possible, and if no progress is made, shrink the step size. Over time, the method zooms in on a good solution without ever using gradients.

Generalized Pattern Search (GPS)

GPS generalizes coordinate-based methods by allowing arbitrary search directions. A pattern is defined as:

$$\mathcal{P} = \{x + \alpha d \mid d \in \mathcal{D}\}$$

Here, x is the anchor point, α is the step size, and $\mathcal{D}$ is a direction set.

Comparison with Hooke-Jeeves:

- Hooke-Jeeves: 2n coordinate directions.
- GPS: $\geq n + 1$ arbitrary directions.
- Allows movement outside axis-aligned constraints.

Positive Spanning Set: A set $\mathcal{D}$ is positively spanning if any vector in $\mathbb{R}^n$ can be written as a nonnegative linear combination of its elements.

Key Idea: If the set of directions $\mathcal{D}$ forms a positive spanning set, then from any non-stationary point there exists at least one direction in $\mathcal{D}$ along which the objective function decreases.

Motivation: Generalized directions better align with curved landscapes or ridged contours, improving convergence behavior.

Comments

The Generalized Pattern Search, or GPS, represents a powerful extension of coordinate-based methods like Hooke-Jeeves. Unlike methods that restrict movement to axis-aligned directions, GPS allows searches in arbitrary directions. This flexibility enables the algorithm to adapt more efficiently to objective functions with narrow ridges, valleys, or irregular curvatures.

The key idea behind GPS is the concept of a pattern $\mathcal{P}$ — a set of candidate points constructed from an anchor point x by adding a step α in each direction d from a chosen set $\mathcal{D}$. The pattern is given by the set of all x plus alpha times d, such that d belongs to $\mathcal{D}$. The success of the method depends heavily on the properties of $\mathcal{D}$.

A positive spanning set is a special set of directions such that any vector in R^n can be expressed as a nonnegative linear combination of these directions.

The key idea of generalized pattern search is that if the direction set $\mathcal{D}$ is a positive spanning set, then from any point where the gradient is nonzero, there exists at least one direction in $\mathcal{D}$ that leads to a decrease in the objective function. This geometric property ensures that the method can always make progress unless it is already at a stationary point.

In contrast to Hooke-Jeeves, which uses 2n directions, GPS can work with as few as $n + 1$. This not only reduces the number of function evaluations per iteration but also allows movement along directions that may be more aligned with the structure of the objective function, especially when it is non-separable or ill-conditioned.

Finally, GPS offers practical advantages in higher dimensions, where axis-aligned moves become inefficient. By choosing directions wisely — and ensuring they form a positive spanning set — GPS achieves more robust convergence in a wider class of problems.

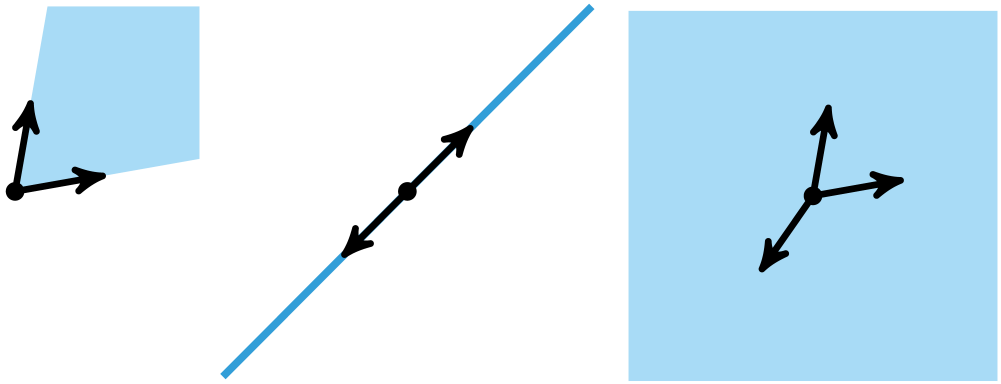


Figure: In the left image, the two directions shown span only a narrow cone. The middle image depicts a degenerate case where the set spans only a one-dimensional subspace. In the right image does the set of directions positively span the full $\mathbb{R}^2$ space.

Comments

This slide provides visual intuition behind the concept of a positive spanning set by comparing three different cases in two dimensions.

In the left image, the two directions shown span only a narrow cone. Any vector within this cone can be expressed as a nonnegative linear combination of the two given directions, but vectors outside the cone—especially in the opposite quadrant—cannot. Therefore, this set does not positively span the full R^2 space. Such a set may miss descent directions, limiting its usefulness in optimization.

The middle image depicts a degenerate case where the set spans only a one-dimensional subspace. Here, both directions lie along the same line, so even their linear combinations remain confined to that line. Consequently, this set also fails to positively span the entire space, providing no guarantee of progress unless the optimum lies directly on that line.

Only in the right image does the set of directions positively span the full R^2 space. This means that any vector in two dimensions can be represented as a nonnegative linear combination of these directions. Therefore, at least one direction in the set will always lead downhill when the gradient is nonzero, which is critical for the convergence of pattern search algorithms.

This geometric property ensures the algorithm is not "blind" to important directions of improvement. Without a positive spanning set, a search method may stall even far from a local minimum.

Generalized Pattern Search Algorithm

Algorithm Generalized Pattern Search (GPS)

```
function generalized_pattern_search(f, x, α, D, ϵ, γ = 0.5)
    y = f(x), n = length(x)
    while α > ϵ do
        improved = false
        for (i, d) in enumerate(D) do                ▷ Loop over indexed directions
            x′ = x + α · d
            y′ = f(x′)
            if y′ < y then
                x = x′, y = y′, improved = true
                D = pushfirst!(deleteat!(D, i), d)   ▷ Promote successful direction
                break
            end if
        end for
        if not improved then
            α = α · γ                                ▷ Reduce step size
        end if
    end while
    return x
end function
```


Comments

This slide presents the full implementation of the Generalized Pattern Search (GPS) method in pseudocode. The algorithm starts from an initial point x with step size α, and a direction set D. This set must form a positive spanning set, ensuring that descent is always possible when not at a stationary point.

In each iteration, the algorithm checks candidate points of the form "x + $\alpha \cdot$ d", looping through all directions lowercase d in D. If a better value is found, the corresponding direction is promoted to the front of the list. This dynamic reordering (known as opportunistic direction promotion) improves convergence by exploiting promising directions early.

Note that as soon as an improvement is found, the search terminates the current iteration early (via break). This makes GPS more efficient than fully evaluating all directions in each iteration.

If no improvement occurs, the step size is reduced by multiplying it with a decay factor γ. The loop continues until the step size becomes smaller than a given tolerance ε.

The implementation includes a key enhancement over simpler methods: the dynamic adjustment of the direction set, which helps adapt the search to the local geometry of the function.

Also note the use of enumerate(D), which is used to obtain both the index and the direction vector which allows tracking the index i of each direction. This is necessary because we reorder the list when a direction leads to improvement. Without indexing, that operation would not be possible.

This algorithm strikes a balance between simplicity, flexibility, and convergence guarantees, and works well when derivatives are unavailable or unreliable.

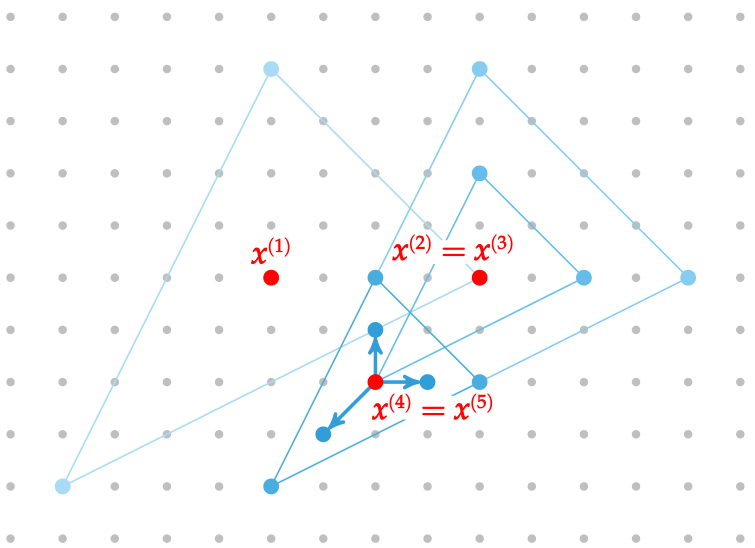


Figure: Generalized Pattern Search Process

Comments

This slide visualizes the behavior of the Generalized Pattern Search (GPS) method through a series of steps. The image shows how the algorithm explores the search space starting from an initial anchor point and evaluating potential new points defined by a positive spanning set of directions.

At each iteration, the GPS method constructs a pattern by adding scaled versions of its current direction vectors to the anchor point. The algorithm then evaluates the objective function at each of these trial points. If any of them results in an improvement, the corresponding point becomes the new anchor, and the direction that led to the improvement is promoted to the front of the list to be tried earlier in the next iteration. This dynamic ordering accelerates convergence by prioritizing successful directions.

Importantly, the pattern of directions is not restricted to coordinate axes. This flexibility allows the algorithm to move in oblique directions that better align with the landscape of the objective function, especially when the function is non-separable or the search space is ill-conditioned.

The visualization illustrates how the algorithm progresses along a virtual mesh formed by combinations of the direction vectors. While this mesh is never constructed explicitly, the evaluated points lie on it. Over time, the steps become finer as the step size is reduced in the absence of improvement.

Overall, this figure emphasizes the core strengths of GPS: flexibility in direction, rapid adaptation through opportunistic updates, and convergence in complex landscapes where more rigid methods struggle.

Nelder-Mead Method: Geometry and Operations

The Nelder-Mead method maintains a **simplex** — a geometric figure with $n+1$ vertices in $\mathbb{R}^n$ (e.g., a triangle in 2D, a tetrahedron in 3D). The simplex adapts its shape and location to find a local minimum of the function.

- Let $x^{(1)}, \ldots, x^{(n+1)}$ be the simplex vertices.
- x_h: vertex with the **highest** function value.
- x_l: vertex with the **lowest** function value.
- x_s: **second-highest** function value.
- $\overline{x}$: centroid of all points except x_h.

Simplex Operations:

- **Reflection:** $x_r = \overline{x} + \alpha(\overline{x} - x_h)$, usually with $\alpha = 1$.
- **Expansion:** $x_e = \overline{x} + \beta(x_r - \overline{x})$, $\beta > \alpha$, typically $\beta = 2$.
- **Contraction:** $x_c = \overline{x} + \gamma(x_h - \overline{x})$, $\gamma \in (0, 1)$, typically 0.5.
- **Shrinkage:** All vertices move (typically halfway) toward x_l.

Note: Convergence is based on the **standard deviation** of objective values, not distance in parameter space.

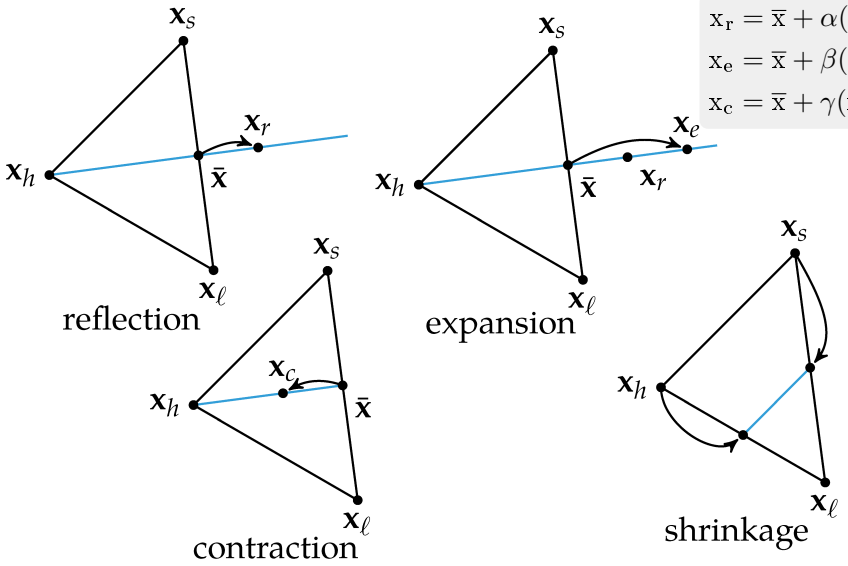
Comments

This slide introduces the Nelder–Mead simplex method — one of the most well-known derivative-free optimization algorithms. It relies on a geometric structure called a simplex: the simplest possible polytope, with n+1 vertices in n-dimensional space. For example, in two dimensions, a simplex is a triangle; in three, it is a tetrahedron.

The algorithm uses the simplex to explore the objective function. On each iteration, function values are evaluated at the vertices, and one of four operations is performed: reflection, expansion, contraction, or shrinkage. These allow the simplex to adaptively move and reshape toward a minimum.

The key feature of Nelder–Mead is that it does not use derivatives, relying instead on comparing function values. This makes it particularly suitable for optimizing non-smooth or derivative-free functions.

Its convergence criterion is based not on the distance between points but on the standard deviation of the function values at the simplex vertices. When these values are close together, the simplex lies in a relatively flat region, and the algorithm terminates.



$$x_r = \bar{x} + \alpha(\bar{x} - x_h)$$
$$x_e = \bar{x} + \beta(x_r - \bar{x})$$
$$x_c = \bar{x} + \gamma(x_h - \bar{x})$$

Operations of N-M Method: 2D-example

Comments

The algorithm starts by initializing a simplex consisting of $n+1$ points. This is the minimal number of vertices needed to define a polytope in n-dimensional space.

At each iteration, three key vertices are identified: x_h is the point with the highest objective value, x_s is the second highest, and x_l is the point with the lowest value.

The centroid $\bar{x}$ is computed as the mean of all vertices except x_h. It serves as the reference for the next transformations.

The first operation is reflection: the reflected point x_r is computed as $\bar{x} + \alpha(\bar{x} - x_h)$. This typically moves the simplex toward better regions.

If x_r is better than x_l, expansion is attempted. The expansion point x_e is calculated as $\bar{x} + \beta(x_r - \bar{x})$. This extends the move further in the same direction.

If the reflected point is not better, contraction is tested. The contraction point x_c is computed as $\bar{x} + \gamma(x_h - \bar{x})$. This shrinks the simplex toward the centroid.

If contraction fails, shrinkage is applied: all points are moved closer to x_l, usually by halving the distance.

Convergence is determined by checking the standard deviation of the function values over the simplex. If it drops below the given tolerance epsilon, the algorithm stops.

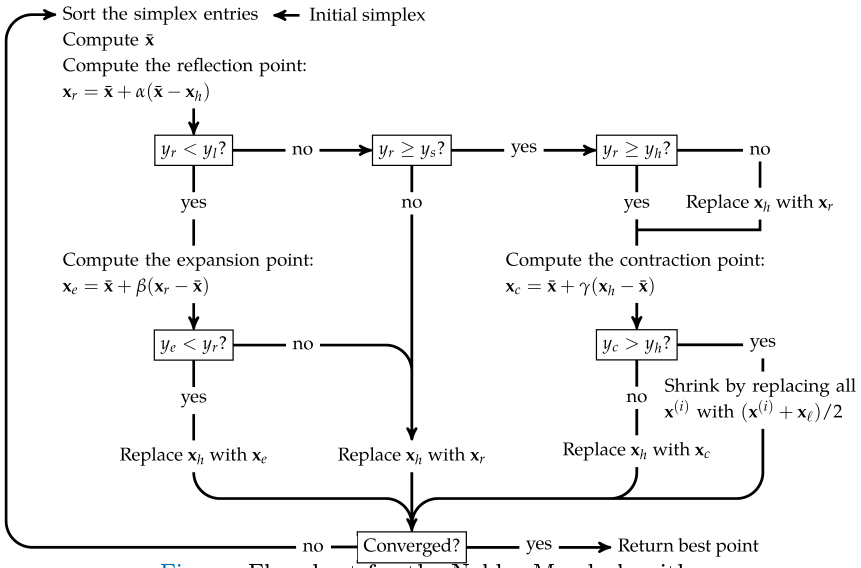


Figure: Flowchart for the Nelder-Mead algorithm

Comments

This diagram represents the flow of the Nelder-Mead algorithm. The algorithm, as we discussed, operates on a simplex, which in two dimensions is a triangle, and in higher dimensions is a polytope with $n + 1$ vertices.

The process starts by evaluating the objective function at all simplex points. Then, these points are sorted according to their function values: the lowest point (best solution so far), the highest point (worst solution), and the second-highest point are identified. The centroid of all points except the worst one is computed; this centroid acts as a reference for exploring new candidate solutions.

The next step is the reflection. The worst point is reflected across the centroid to test whether moving in the opposite direction improves the function value. If this reflection produces the best value so far, the algorithm attempts an expansion, pushing further in that promising direction. If the reflection is worse than the second-worst point, the method instead tries a contraction, moving the worst point closer to the centroid.

If contraction fails to improve the solution, the algorithm performs a shrink step: all points except the best one are moved closer to the best point. This reduces the size of the simplex and forces the search into a smaller region.

The process repeats iteratively: reflection, expansion, contraction, or shrink, depending on the comparisons of function values. The stopping condition is typically when the function values at the simplex vertices are close to each other (measured by standard deviation), indicating convergence. At that stage, the algorithm outputs the best vertex as the approximate solution.

Overall, the diagram shows how Nelder-Mead balances exploration and contraction of the simplex, guiding it step by step toward a local minimum without requiring derivatives of the objective function.

Algorithm 7 Nelder-Mead Method

```
function nelder_mead(f, S, ϵ; α=1.0, β=2.0, γ=0.5)
    Δ, y_arr = Inf, f.(S)
    while Δ > ϵ
        p = sortperm(y_arr) # sort lowest to highest
        S, y_arr = S[p], y_arr[p]
        xl, yl = S[1], y_arr[1] # lowest
        xh, yh = S[end], y_arr[end] # highest
        xs, ys = S[end-1], y_arr[end-1] # second-highest
        xm = mean(S[1:end-1]) # centroid
        xr = xm + α*(xm - xh) # reflection point
        yr = f(xr)
        if yr < yl
            xe = xm + β*(xr - xm) # expansion point
            ye = f(xe)
            S[end], y_arr[end] = ye < yr ? (xe, ye) : (xr, yr)
        elseif yr ≥ ys
        if yr < yh
```


Comments

Here is a software implementation of the Nelder-Meade method, which we discussed with you.

In this implementation, the function `nelder_mead` accepts three required arguments: `f` is the objective function we want to minimize, `S` is the initial set of points (which defines the starting simplex), and ϵ is a tolerance value for the stopping condition.

The first step initializes Δ (a measure of the "spread" of the simplex) and `y_arr` (the array of function values at the current simplex points). The `while` loop continues until Δ becomes smaller than the tolerance ϵ, which means the algorithm is sufficiently close to the minimum.

In each iteration, the `sortperm` function is used to order the points in the simplex (`S`) by their corresponding function values (`y_arr`), from the lowest to the highest. The first point in the sorted list, `xl` (with function value `yl`), represents the lowest value, and the last point, `xh` (with function value `yh`), represents the highest value. The second-highest point `xs` and `ys` are also identified for later comparisons.

The next step involves computing the centroid `xm` of the simplex (excluding the highest point), which will be used to determine the reflection point `xr`. The reflection point is calculated by moving away from the highest point in the direction of the centroid. If the function value at `xr` is better than the current lowest point `yl`, we will consider further steps to explore the space for a better solution.

The use of the parameters α, β, and γ governs the scaling of the reflection, expansion, and contraction steps, respectively.

```
                xh, yh, S[end], y_arr[end] = xr, yr, xr, yr
            end
            xc = xm + γ*(xh - xm) # contraction point
            yc = f(xc)
            if yc > yh
                for i in 2 : length(y_arr)
                    S[i] = (S[i] + xl)/2
                    y_arr[i] = f(S[i])
                end
            else
                S[end], y_arr[end] = xc, yc
            end
        else
            S[end], y_arr[end] = xr, yr
        end
        Δ = std(y_arr, corrected=false)
    end
    return S[argmin(y_arr)]
end
```


Comments

In this part of the algorithm, we handle the different scenarios based on the function value at the reflection point `xr` (denoted by `yr`).

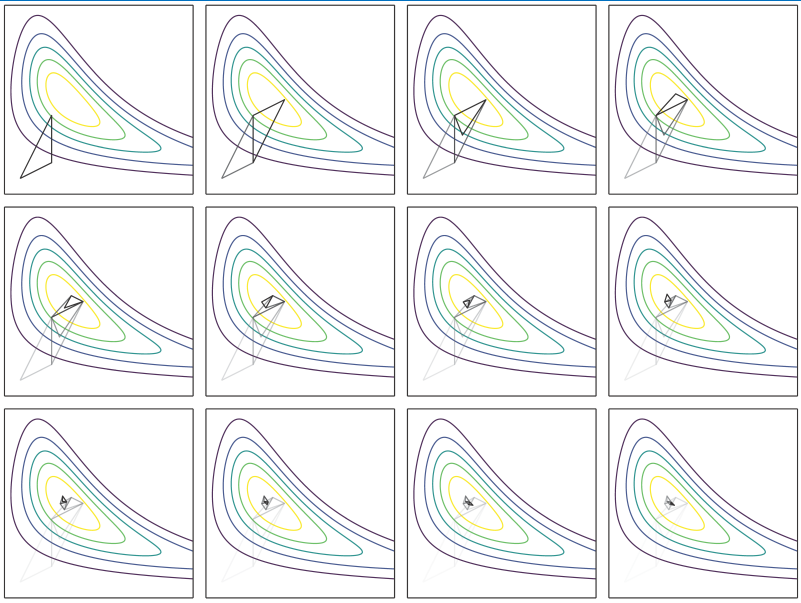
1. Expansion: If the function value at `xr` is better than `yl`, this suggests that the algorithm has found a potentially better region, and thus an expansion step is considered. The expansion point `xe` is calculated by moving beyond `xr` in the direction of the centroid. If `xe` leads to an even better function value, it becomes the new highest point in the simplex. If not, `xr` is retained.

2. Contraction: If the function value at `xr` is not better than `ys`, a contraction step is performed. The contraction point `xc` is calculated by moving towards the centroid but by a smaller amount, controlled by the parameter γ. If the function value at `xc` is worse than the current highest point `yh`, this means the simplex needs to be contracted, and all points (except the lowest) are moved towards the lowest point.

3. Updating the Simplex: Based on the results of the expansion or contraction steps, the simplex is updated accordingly. The best points are kept, and the worst points are discarded or replaced by new ones.

Finally, the stopping condition is checked by calculating the standard deviation Δ of the function values `y_arr`. When Δ is smaller than the tolerance ϵ, the algorithm terminates. The function then returns the point corresponding to the minimum value of the function.

In essence, the algorithm iteratively refines the simplex, reflecting, expanding, and contracting the simplex points based on their function values, gradually converging to a minimum.



Comments

This figure illustrates the dynamics of the Nelder-Mead algorithm applied to a two-dimensional objective function. Each subplot shows a stage in the evolution of the simplex, which is the triangular shape composed of three points in the plane.

The background contours represent level sets of the objective function. The innermost contour indicates the region with the lowest function values.

As the algorithm proceeds, the simplex adapts its shape and location to explore the function landscape. You can observe the key steps of the algorithm visually:

Reflection: In early frames, one vertex is reflected across the centroid of the other two.

Expansion: When the reflected point shows significant improvement, the algorithm explores further in that direction.

Contraction and shrinkage: If no improvement is made, the simplex contracts or shrinks toward the best point.

The sequence of plots shows how the simplex becomes smaller and gradually settles near the function's minimum. This dynamic behavior is crucial in navigating curved or non-convex landscapes without gradients.

By the final frames, the simplex has nearly collapsed to a single point — the estimated local minimum.

DIRECT: Divided Rectangles Method

DIRECT (DIvided RECTangles): A global optimization method based on Lipschitz continuity, but without requiring a known Lipschitz constant. Works well in multidimensional settings.

Lipschitz Continuity Reminder:

- f is Lipschitz continuous if its rate of change is bounded:

$$|f(x) - f(y)| \leq \ell \|x - y\|_2 \quad \forall x, y$$

- ℓ defines how sharply f can vary — no derivative needed.
- Lower bound near sample $x^{(i)}$: $f(x) \geq f(x^{(i)}) - \ell \|x - x^{(i)}\|_2$

DIRECT's Strategy:

- Domain is split into hyper-rectangles centered at sampled points.
- Each region has a local Lipschitz-type bound, without global ℓ.
- This avoids evaluating complex intersections of lower bounds.

Key Point: Even for smooth functions, DIRECT works without gradients — it builds provable bounds based on sampled values only.

Comments

Lipschitz continuity gives a way to bound the behavior of a function without using derivatives. If a function is Lipschitz continuous, then its values cannot change too rapidly — the Lipschitz constant defines this maximum rate.

This is useful even for smooth functions. Derivatives might be available, but in global optimization we care more about global guarantees. Gradients point in local directions, while Lipschitz bounds define hard limits over entire regions. That's why DIRECT — even though derivative-free — is still very relevant.

The key is that around each sample point, we can build a guaranteed lower estimate for the function. This estimate decreases linearly with distance, forming a cone-shaped lower bound.

In high dimensions, combining such cones is geometrically difficult. DIRECT simplifies this: it breaks the domain into rectangles, and assigns one cone to each rectangle. It no longer computes the maximum over cones — it just compares the bounds inside individual regions. This makes the search tractable.

DIRECT doesn't even need to know the true Lipschitz constant. It proceeds as if there exists some valid constant and uses geometry to identify promising regions.

DIRECT: Approximating the Global Lower Bound

Global Lipschitz Lower Bound:
$$f(x) \geq \max_{i=1,\ldots,m} \left(f(x^{(i)}) - \ell \|x - x^{(i)}\|_2 \right)$$

Interpretation:

- Each function evaluation defines a local cone-shaped bound.
- The maximum of these cones bounds $f(x)$ from below everywhere.

Problem: This bound is continuous but geometrically complex in high dimensions.

DIRECT's Idea: Instead of working with the global envelope, intersect each cone with a hyper-rectangle R_i centered at $x^{(i)}$ and evaluate the minimum of that cone only within its rectangle:
$$\min_{x \in R_i} \left(f(x^{(i)}) - \ell \|x - x^{(i)}\|_2 \right)$$

Effect:

- The complex lower surface is approximated by independent bounds in separate boxes.
- This approximation is computationally cheap and valid from below.
- It enables comparing boxes and refining those with the lowest estimated minima.

Comments

The true Lipschitz lower bound for a function is given by the maximum over all cone-shaped estimates based on evaluated points. This global bound is continuous but difficult to compute and manipulate, especially in high dimensions, as it requires evaluating the upper envelope of all cones at each point.

DIRECT avoids this complexity by using a local approximation. Each function evaluation defines a cone, and instead of globally maximizing over all cones, DIRECT restricts each one to its associated hyper-rectangle. It then computes the minimum of that cone only within the box. This gives a conservative lower bound for the function in that region.

These local bounds are used to compare boxes. The boxes with the lowest such bounds are seen as most promising and are refined further. The method thus balances global exploration and local descent, all without having to explicitly resolve the global lower surface.

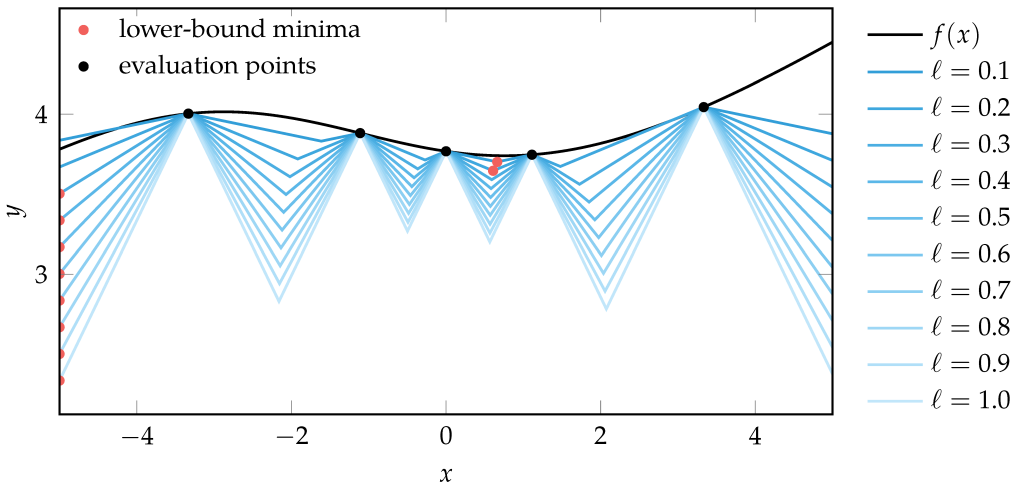


Figure: The Lipschitz lower bound for different Lipschitz constants ℓ. Not only does the estimated minimum change locally as the Lipschitz constant is varied, the region in which the minimum lies can vary as well

Comments

This figure illustrates how the classical Lipschitz lower bound depends on the choice of the Lipschitz constant ℓ. For each value of ℓ, we construct the global lower bound as the maximum of all cone-shaped functions centered at previously evaluated points.

When ℓ is small, these cones are shallow. As a result, the lower bound tends to be higher near the best evaluated points — so the estimated minimum lies close to the existing samples.

As ℓ increases, the cones become steeper. The lower envelope now decreases more aggressively with distance, and the estimated minimum moves farther away from known points. This shows how sensitive the lower bound is to the value of ℓ, and how it can shift the search behavior between exploitation and exploration.

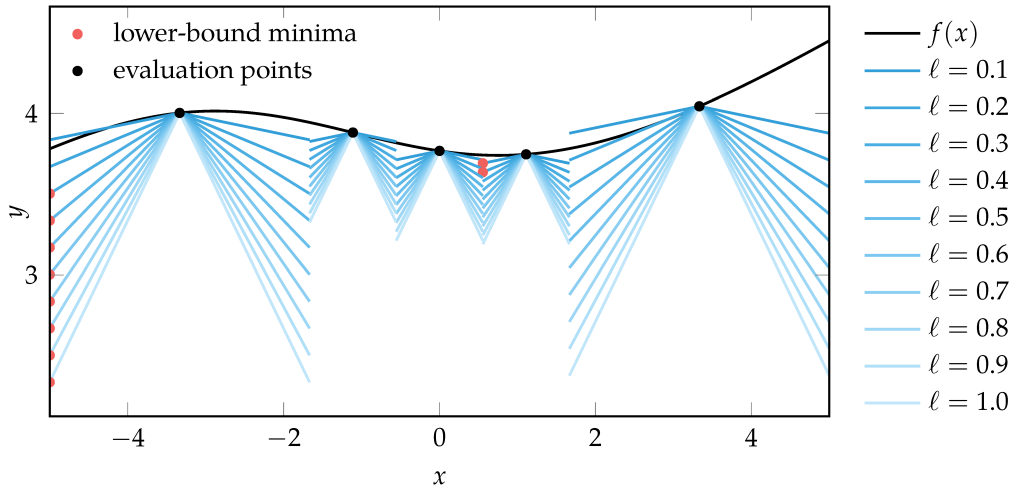


Figure: The DIRECT lower bound for different Lipschitz constants ℓ. The lower bound is not continuous. The minimum does not change locally but can change regionally as the Lipschitz constant changes

Comments

This figure illustrates how the behavior of the DIRECT method depends on the chosen value of the Lipschitz constant ℓ. Each rectangle is centered at a previously evaluated point and serves as a local region for estimating the function's lower bound.

For each rectangle, DIRECT constructs a simple cone-shaped bound: it decreases linearly with distance from the center, at a rate determined by ℓ. The lower bound inside a rectangle is then defined as the minimum of this cone over the rectangle.

When ℓ is small, the cone is shallow, and the minimum tends to lie near the center — favoring rectangles with low function values. When ℓ is large, the cone drops more steeply, and large rectangles can yield very low estimates even if their center values are not optimal.

Thus, different values of ℓ change which rectangles appear most promising. DIRECT implicitly explores a range of such values in parallel and selects rectangles that could yield a global minimum, balancing local exploitation and global exploration.

Univariate DIRECT: Interval Division and Lower Bound

Basic Procedure:

- Start with interval $[a, b]$, center $c = \frac{a+b}{2}$.
- Evaluate $f(c)$ and construct local lower bound using unknown ℓ.
- Estimate:
$$f(x) \geq f(c) - \ell|x - c|$$
- Minimum lower bound on $[a, b]$ is:
$$f(c) - \ell(b - a)/2$$

Comparison Between Intervals:

- If two intervals have equal width, the one with lower $f(c)$ has lower bound.
- Even without knowing ℓ, we can prefer shorter intervals with low $f(c)$.
- DIRECT uses this idea to guide which intervals to subdivide.

Key Idea: DIRECT balances global exploration and local refinement by recursively selecting the most promising intervals based on their center values and widths.

Comments

This slide explains how the univariate version of the DIRECT algorithm works. The method begins with a full interval and samples the function at the center point c. The key assumption is that the objective function is Lipschitz continuous with some unknown constant ℓ, which allows us to bound the function from below.

At every center point c, we construct a conical lower bound of the form $f(x) \geq f(c) - \ell|x - c|$. The minimum of this bound on is reached at the endpoints and equals $f(c) - \ell(b - a)/2$. Although ℓ is not known, we can still compare two intervals of the same length: the one with a smaller function value at the center will have a tighter lower bound.

More importantly, we can also compare intervals of different lengths. An interval with a slightly higher function value but much smaller width might still be preferred. This comparison enables the method to balance between local exploitation (narrow intervals with low values) and global exploration (wide intervals that might hide new minima).

Based on this logic, DIRECT selects intervals that are potentially optimal — those that could minimize the unknown lower bound for some feasible value of ℓ. It then subdivides these intervals into thirds and repeats the process. This simple rule allows the method to adaptively explore the domain without requiring gradient information or knowledge of ℓ, making it particularly suitable for black-box or noisy optimization problems.

Selecting Potentially Optimal Intervals

Lower Bound in Interval $[a_i, b_i]$: Let $c_i = (a_i + b_i)/2$, and $\delta_i = (b_i - a_i)/2$. Then:

$$f(x) \geq f(c_i) - \ell\delta_i, \quad \text{for all } x \in [a_i, b_i]$$

- For a fixed $\ell > 0$, the lower bound of interval i is $f(c_i) - \ell\delta_i$.
- Intervals that minimize this bound for given ℓ are **potentially optimal**.
- Solving this for multiple values of ℓ yields a set of such intervals.

Geometric Selection Rule: Potentially optimal intervals form the lower-right convex hull of points $(\delta_i, f(c_i))$.

- This captures all intervals that are optimal under at least one Lipschitz constant.
- These intervals are refined in the next iteration by dividing them into thirds.

Comments

The goal is to select intervals where the function may attain its minimum. Since we only evaluate the function at the center of each interval, we construct a Lipschitz-based lower bound: $f(c_i) - \ell\delta_i$, where δ_i is half the interval's width. A wider interval gives a looser bound — even if $f(c_i)$ is not small, the bound could still be low.

This is crucial: the lower bound depends not only on the center value f(ci) but also on how wide the interval is. So, a wide interval with a moderate function value might still be promising compared to a narrow one with a slightly better value.

DIRECT selects all intervals for which there exists some given $\ell > 0$ such that this bound is the lowest among all intervals. These are called potentially optimal intervals. To find them, we plot all intervals as points $(\delta_i, f(c_i))$ and construct the lower-right convex hull. Each vertex on this hull corresponds to an interval that minimizes the bound for some slope ℓ.

The key insight is that DIRECT doesn't just focus on the interval with the lowest $f(c_i)$. It considers how much "room" each interval has to potentially hide a minimum. By including both small and large intervals, it balances global exploration with local refinement — without requiring any knowledge of ℓ.

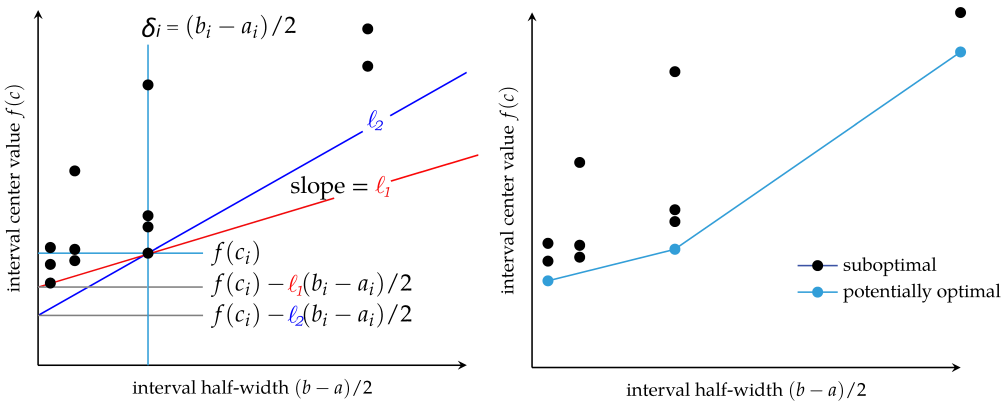
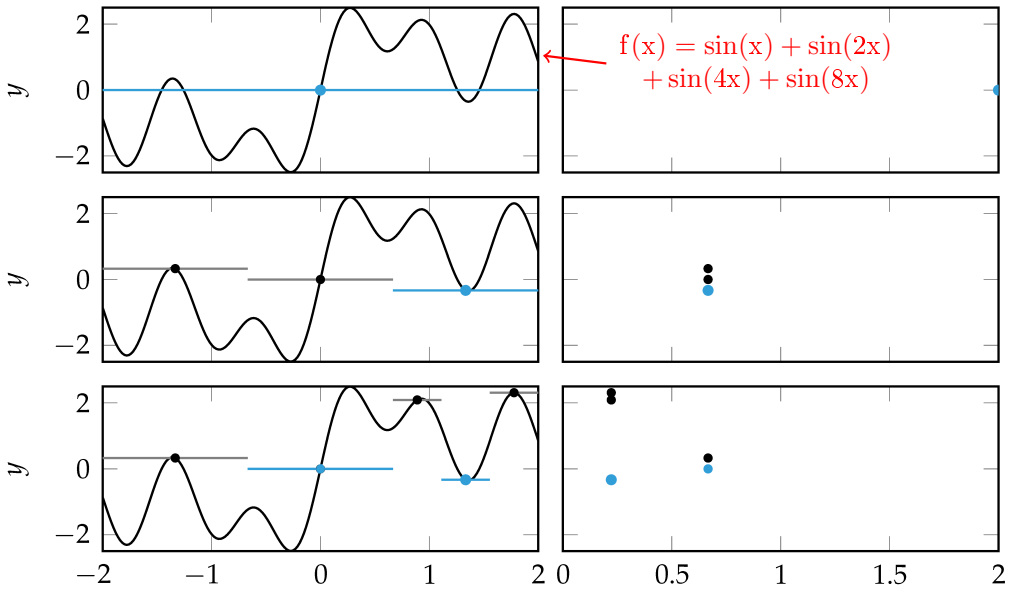


Figure:
The left plot illustrates how lower bounds are determined for a given Lipschitz constant. The right plot shows the set of potentially optimal intervals.

Comments

The left plot illustrates how lower bounds are determined for a given Lipschitz constant. The horizontal axis shows the interval half-width, that is, (b − a)/2, while the vertical axis represents the function value at the interval center, denoted f(c). Each dot on the plot corresponds to one interval. Two lines with slopes l_1 and l_2 are drawn through the same point. These lines define lower bounds: any other point on the plot must lie on or above these lines indicating that they do not provide better lower estimates for these values of l. If a point lies below such a line, the corresponding interval cannot be considered potentially optimal for that value of l. The steeper the slope, the more the bound emphasizes the interval width rather than its center value.

The right plot shows the set of potentially optimal intervals. These intervals are defined as those for which no other interval has both a smaller width and a lower function value. All such intervals form the lower-right convex hull of the points on the plot. These are the intervals selected by the DIRECT method for further subdivision.

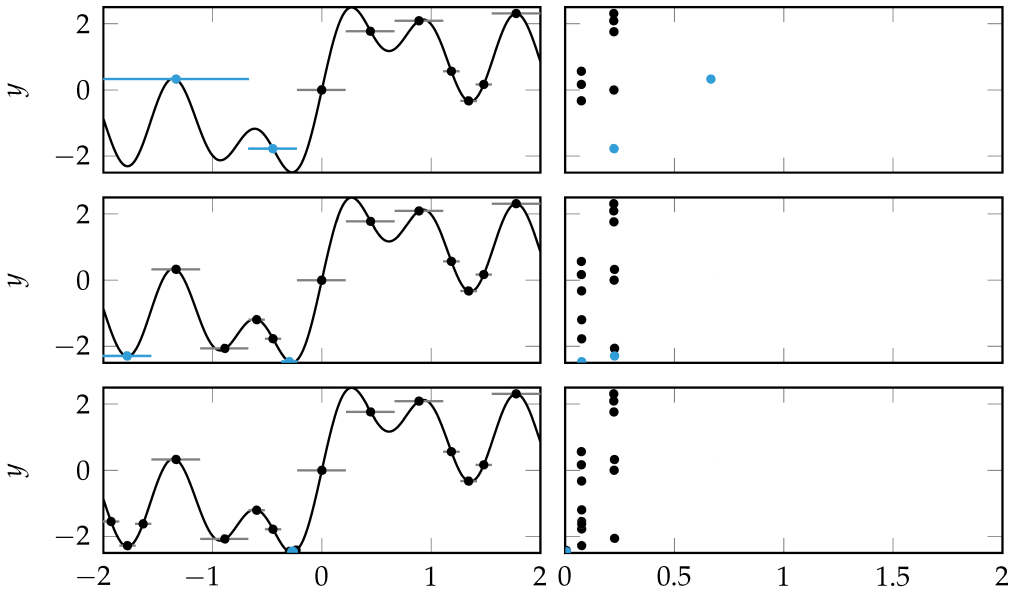


Comments

This example illustrates how the DIRECT algorithm progressively identifies and subdivides promising intervals to locate the global minimum. On the left of each panel, we see the current approximation of the function, with sampled points and subdivisions. On the right, each dot represents an interval, plotted by its half-width and function value at its center. Black dots are all current intervals, and blue ones indicate those deemed potentially optimal at that iteration.

Initially, the function is evaluated only at the center of the entire domain. The interval is then divided into three parts, with new evaluations at the outer thirds. In each iteration, DIRECT identifies intervals that could, for some Lipschitz constant, provide the lowest bound on the function. These are the potentially optimal intervals. They may not contain the lowest function values but offer the best trade-off between interval size and central value.

Starting from the fourth iteration, the algorithm shifts attention to the left-hand side, where the true minimum lies. With each step, it narrows its search and focuses subdivisions around that region. By iteration six, DIRECT has successfully homed in on the global minimum with high accuracy.



Comments

Starting from the fourth iteration, the algorithm shifts attention to the left-hand side, where the true minimum lies. With each step, it narrows its search and focuses subdivisions around that region. By iteration six, DIRECT has successfully homed in on the global minimum with high accuracy.