OPTIMIZATION TECHNIQUES

Deterministic/stochastic global optimization

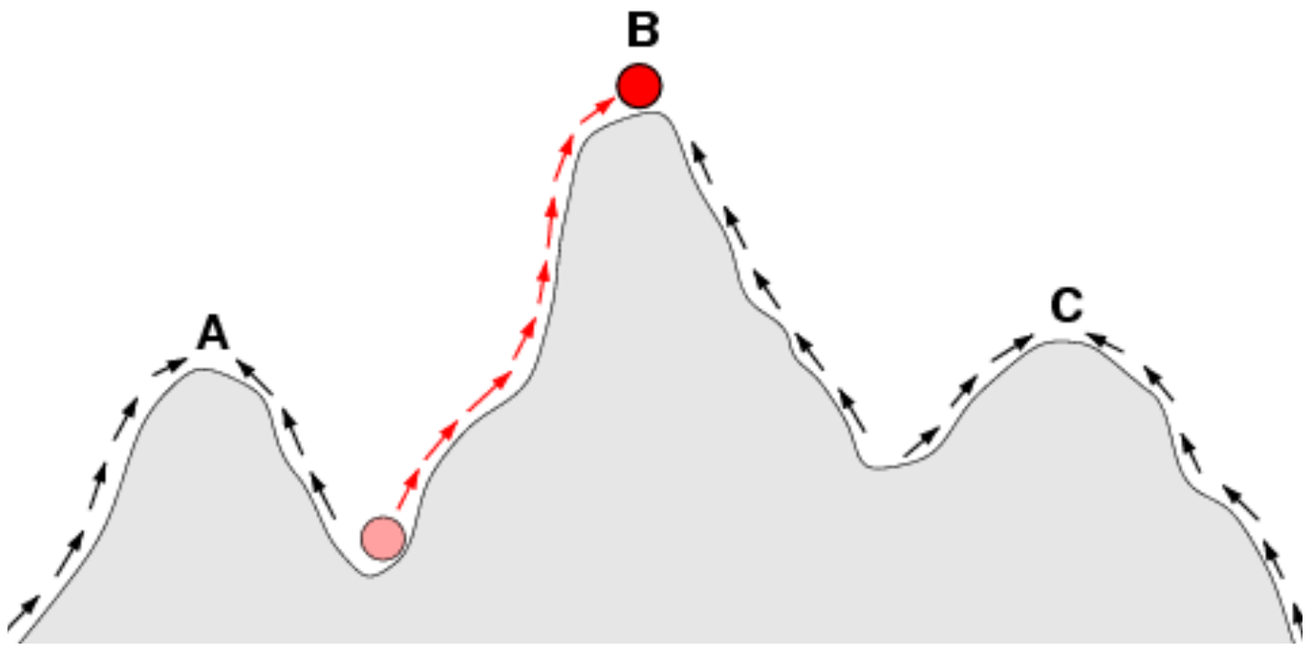
Prof. Giovanni Iacca

giovanni.iacca@unitn.it

UNIVERSITY OF TRENTO - Italy

Information Engineering
and Computer Science Department

Recall from first lecture



OPTIMIZATION

WHAT IS OPTIMIZATION?

Solving an optimization problem = finding the minimum/maximum of one/more objective functions

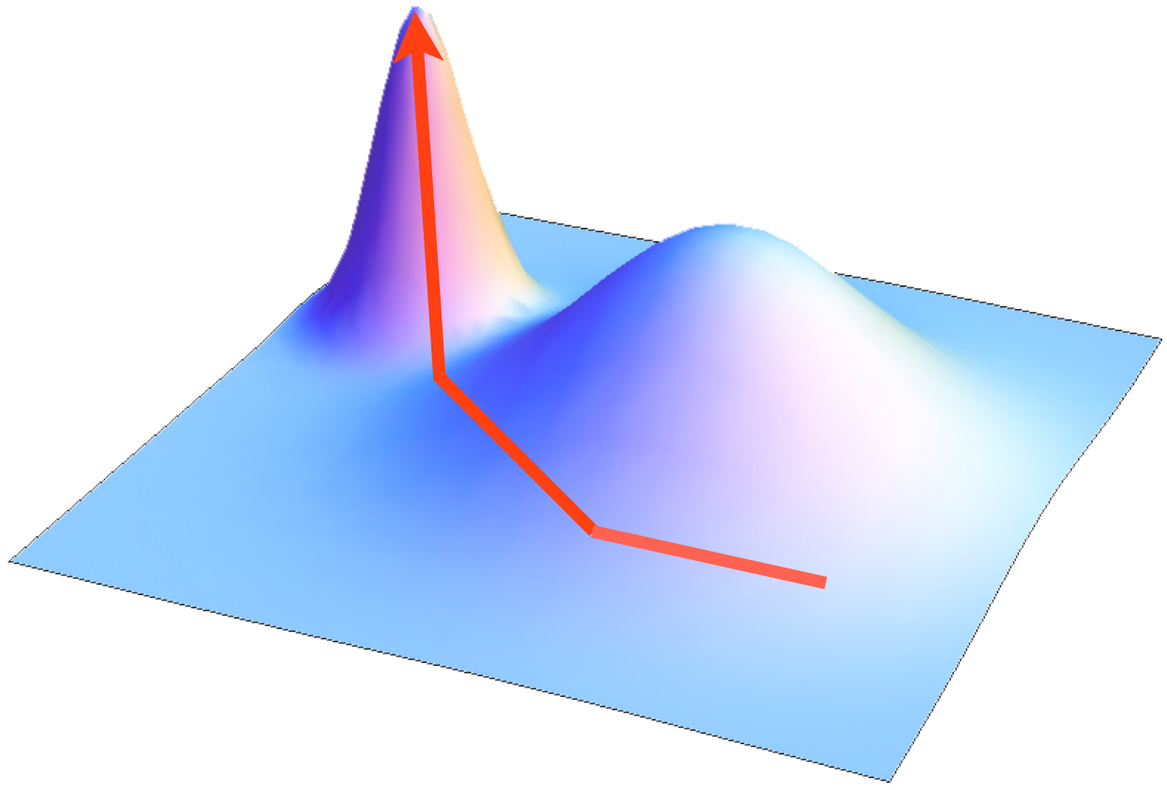
$$\text{find } x^* \ni f(x^*) = \min_{x \in D} \{f(x)\}$$

$$\text{s.t:} \begin{cases} g_j(x) \leqslant 0 & j = 1, 2, \ldots, j_{max} \\ h_k(x) = 0 & k = 1, 2, \ldots, k_{max} \end{cases}$$

$$f(x) = \begin{cases} f_1(x) & : D \rightarrow F_1 \\ f_2(x) & : D \rightarrow F_2 \\ & \ldots \\ f_m(x) & : D \rightarrow F_m \end{cases}$$

- Decision (design) variables:
  $x = [x(1), x(2), \ldots x(n)]$
- Objective function(s): $f(x)$
- Decision (design, search, solution) space: D
- Constraints: $g(x)$ and $h(x)$
- (Global/local) optimum: x^*

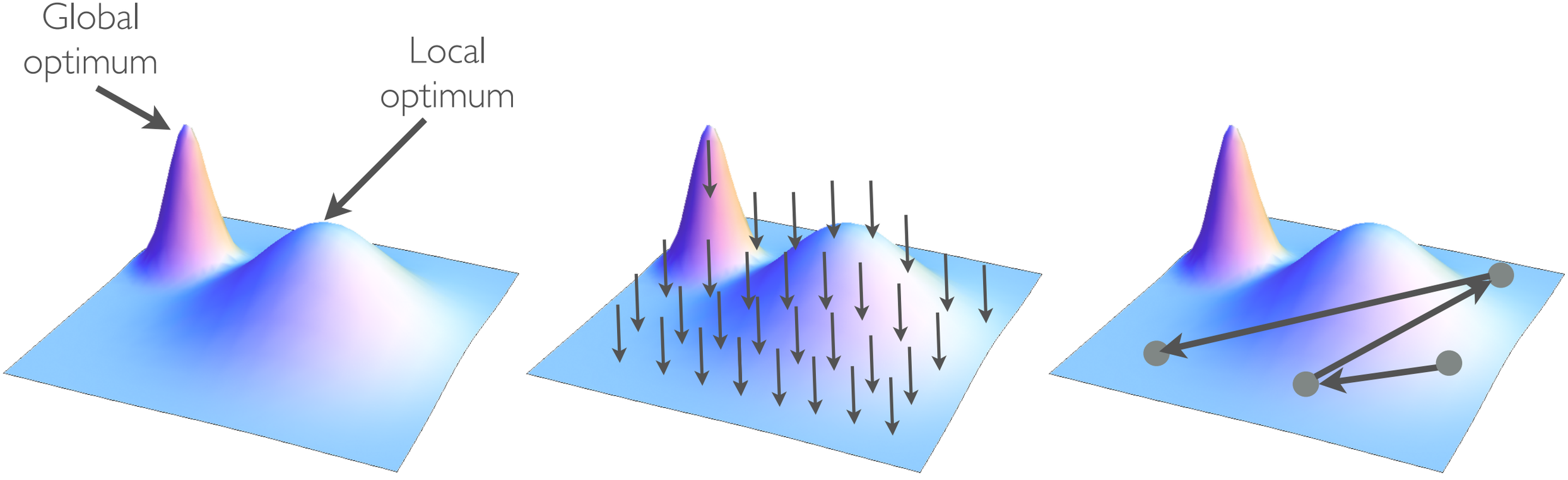
IMPORTANT NOTE
minimize $f(x)$ = maximize $-f(x)$



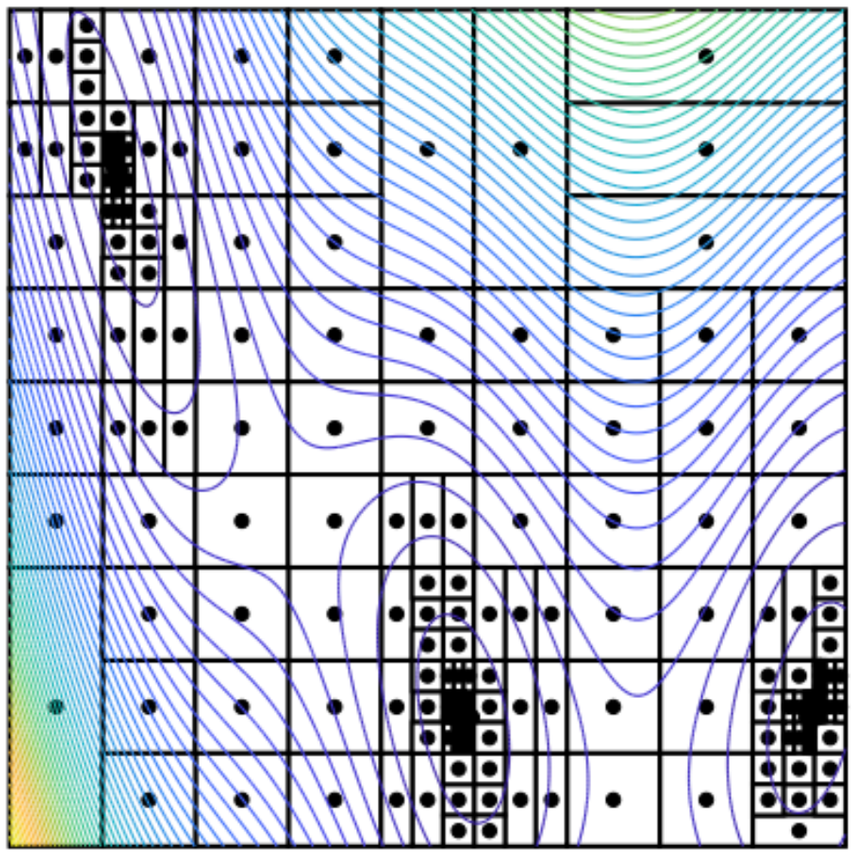
BASIC OPTIMIZATION ALGORITHMS

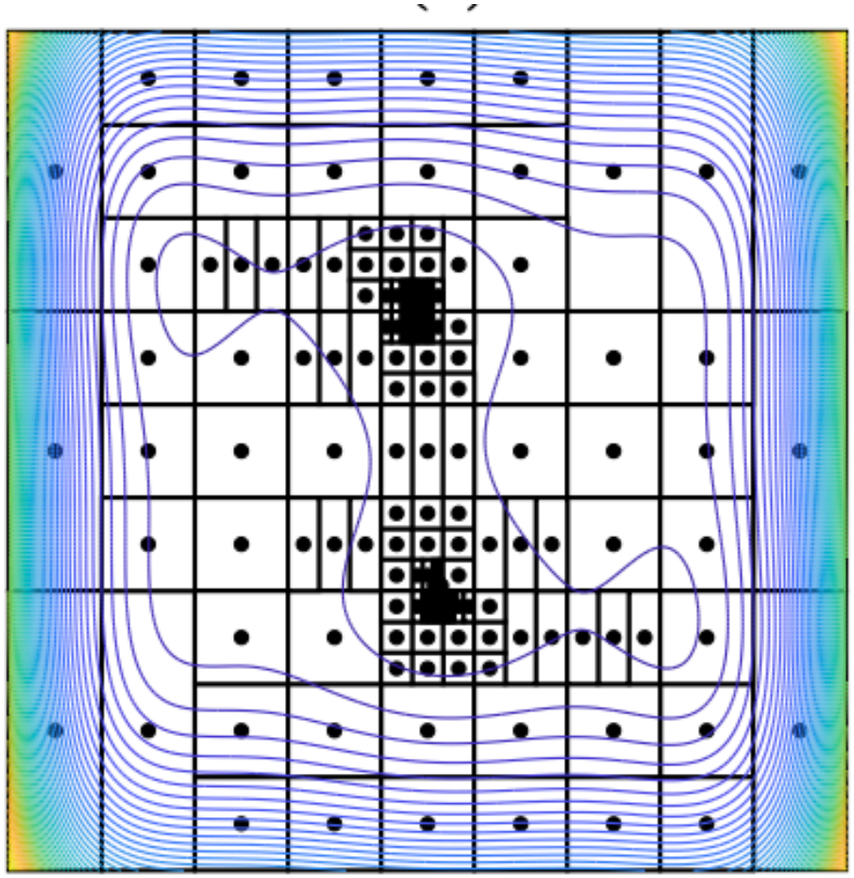
GLOBAL OPTIMIZATION: FIND THE GLOBAL OPTIMUM

- Unimodal vs multimodal functions: one vs many optima
- Approaches:
  - **Deterministic**: brute force (discretize the search space and evaluate **all points**)
  - **Stochastic**: random search (start from an initial point, and perturb it "walking" randomly in the search space, otherwise just sample a new point at each step)
  - More advanced methods: DIRECT, basin hopping, etc.



DIviding REctangles (DIRECT) algorithm





DIVIDING RECTANGLES (DIRECT)

GENERAL INFORMATION

- Introduced by Jones et al. in 1993:
  Jones, D. R., Perttunen, C. D., Stuckman, B. E.: Lipschitzian optimization without the Lipschitz constant. J. Optim. Theory Appl. 79(1), 157–181 (1993). https://doi.org/10.1007/BF00941892

- Original version has proven effective for deterministic, low-dimensional (< 10D) problems.

- Widely applicable, requiring only the objective function to be Lipschitz continuous.
  Several complex applications including genetics, surgery, air transport, power generation, astronomy, hydrogen fuel cell management, photovoltaic systems, hard disk design, aircraft routing reported in the literature.

- Available in Python, MATLAB, etc.

- **PROS** 👍
  - Does not require calculation of a gradient
  - Relatively simple (only one tuning parameter, i.e., the desired accuracy)

- **CONS** 👎
  - Less effective in large dimensionalities (but, more recent variants handle them, e.g. glcCluster)
  - Fast at finding the global optimum's basin, BUT slow at fine-tuning the solution to high accuracy

DIVIDING RECTANGLES (DIRECT)

WHAT IS A LIPSCHITZIAN CONTINUOUS FUNCTION?

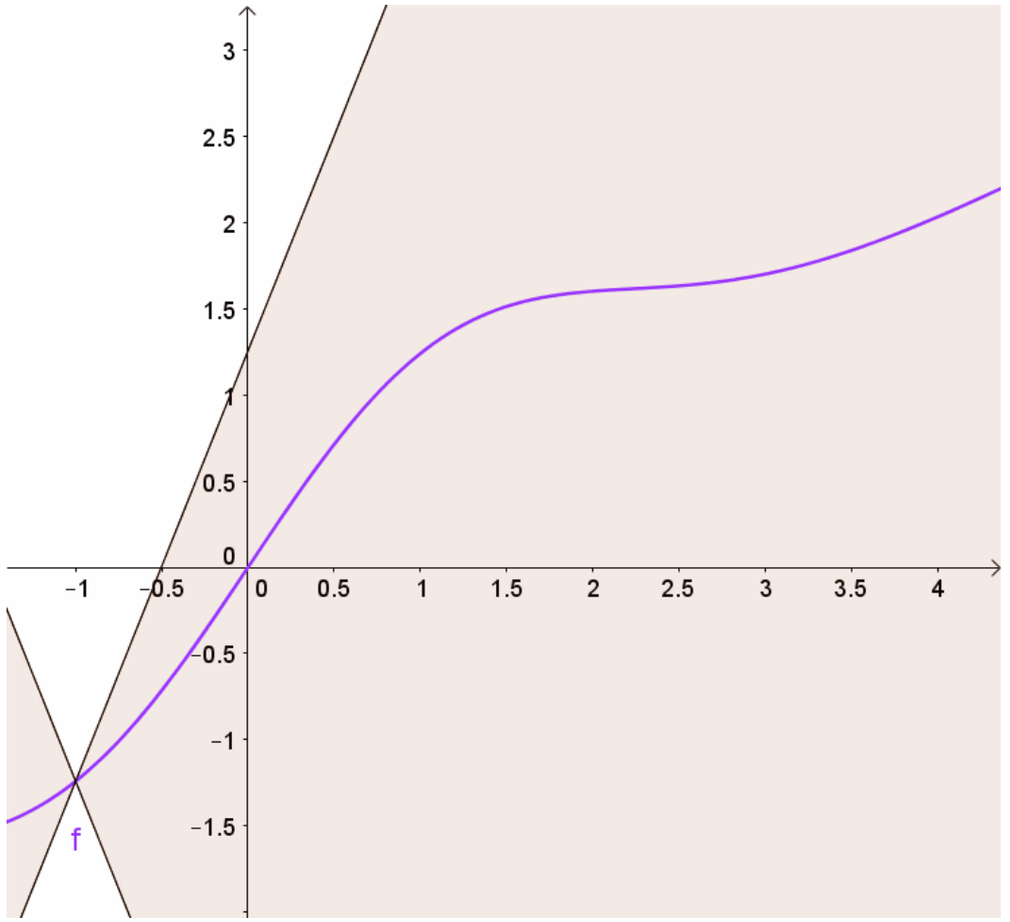
- A continuous function $f(x)$ is said to be L-Lipschitz continuous iff:

  $$|f(x) - f(x_0)| \leq L||x - x_0||, \forall x, x_0 \in \mathbb{R}^d$$

  which can be reformulated as:

  $$f(x_0) - L\big||x - x_0|\big| \leq f(x) \leq f(x_0) + L||x - x_0||$$

  where $L \geq 0$ is a real value called *Lipschitz constant* or *modulus of uniform continuity*.



- Lipschitz continuity DOES NOT imply differentiability!

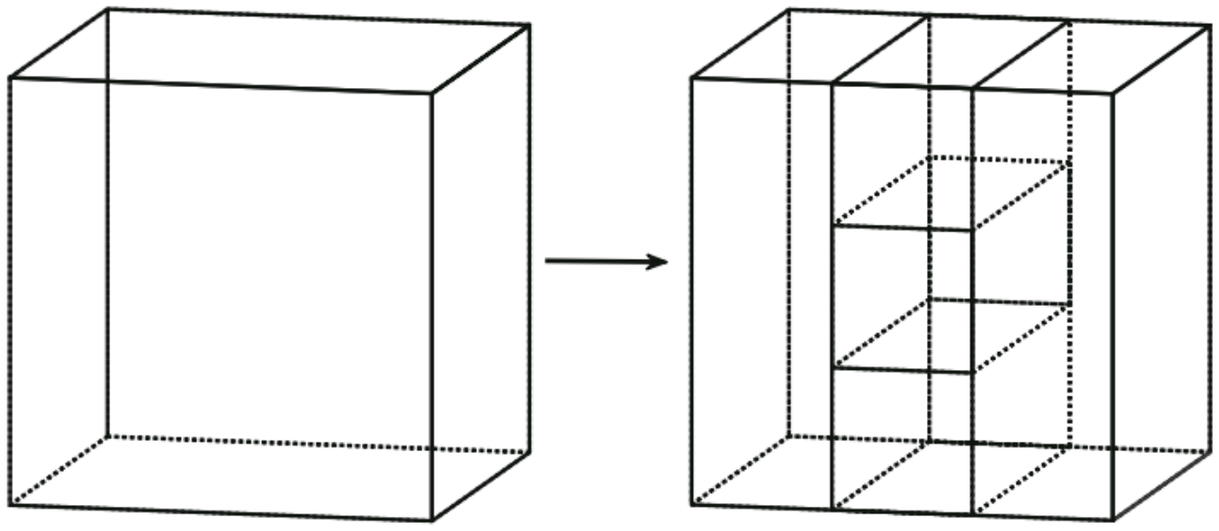
- BUT, it holds that:
  Continuously differentiable ⊂ Lipschitz continuous functions

- Essentially, a "strong" form of uniform continuity that limits "how fast" $f(x)$ can change

DIVIDING RECTANGLES (DIRECT)

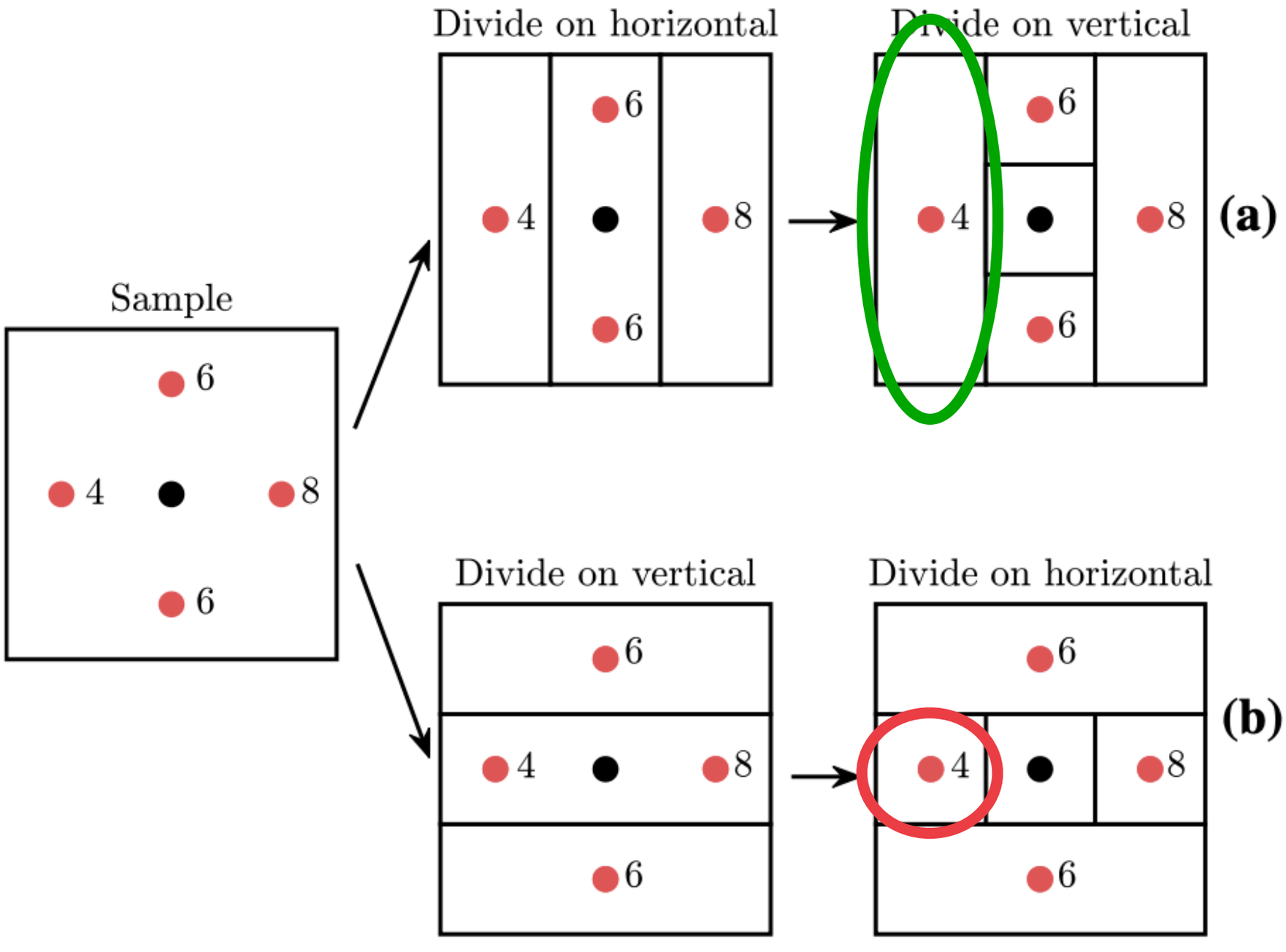
MAIN GIST OF THE ALGORITHM

- Basically a partitioning algorithm
- Combines global and (somehow more limited) local search
- Given enough function evaluations and a highly sensitive stopping criterion, DIRECT will always find the global optimum of a function within a given domain. Of course, this exhaustive search can be computationally expensive and thus impractical in most cases.
- Still, DIRECT can easily find the sub-region within the search space that contains the global optimum, while simultaneously reducing the chance of searching in regions that contain local optima.



DIVIDING RECTANGLES (DIRECT)

PARTITIONING MECHANISM



In principle, both splits (a and b) are possible. However, DIRECT splits the rectangles in order to have larger rectangles containing better fitness value (to increase attractiveness of those regions), so in this case (a), assuming minimization.

NOTE: it would also be possible to split along random directions, but we'd lose determinism.

DIVIDING RECTANGLES (DIRECT)

PARTITIONING MECHANISM (CONT'D)

Algorithm 1 Procedure for dividing rectangles

1: Identify the set M of dimensions with the maximum side length. Let δ equal one-third of this maximum side length.
2: Sample the function at the points $\mathbf{c} \pm \delta \mathbf{e}_k$ for all $k \in M$, where $\mathbf{c}$ is the center of the rectangle and $\mathbf{e}_k$ is the kth unit vector.
3: Divide the rectangle containing $\mathbf{c}$ is into thirds along the dimensions in M, starting with the dimension with the lowest value of $w_k = \min\{f(\mathbf{c} + \delta \mathbf{e}_k), f(\mathbf{c} - \delta \mathbf{e}_k)\}$, and continuing to the dimension with the highest w_k.

- Once the initial hypercube has been divided, some of the subregions will be rectangular. While dividing such rectangles, DIRECT only considers the long dimensions.

- By dividing only along the long dimensions, we ensure that the rectangles shrink in every dimension.

DIVIDING RECTANGLES (DIRECT)

WHY LIPSCHITZIAN CONTINUITY MATTERS

- Let c_i be the center of rectangle i in the partition, and let d_i be the Euclidean distance between c_i and the vertices of rectangle i. Given that the function has a value $f(c_i)$ at the center of rectangle i and that the maximum distance between c_i and any point in rectangle i is d_i, it follows that a valid lower bound for f over rectangle i is:

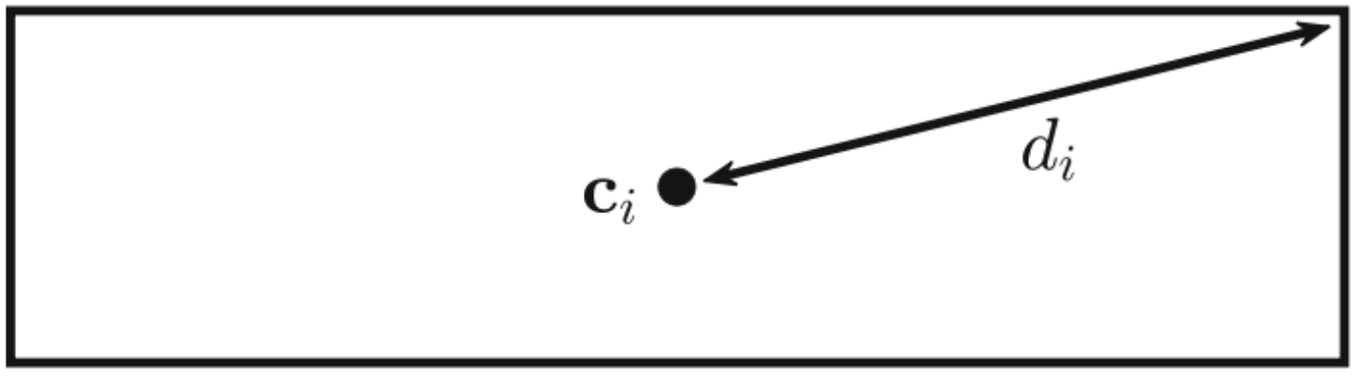
$$f(c_i) - Kd_i$$

Lower when $f(c_i)$ is low → local search

high (low) K emphasizes global (local) search

Lower when d_i is large → exploration

- If K is a valid Lipschitz constant (i.e., the upper bound on the rate of change of the objective function), then one can prove that, for any small positive number $\varepsilon > 0$, the algorithm will find a solution within ε of the optimum in a finite number of iterations.



DIVIDING RECTANGLES (DIRECT)

WHY LIPSCHITZIAN CONTINUITY MATTERS

- Since the Lipschitz constant must be an upper bound on the rate of change of the objective function, it is generally quite high. Standard Lipschitzian optimization (a predecessor of DIRECT based on branch-and-bound, where bounds are computed based on knowledge of a Lipschitz constant for the objective function) would place a high emphasis on global search and, as a result, it may exhibit slow convergence.

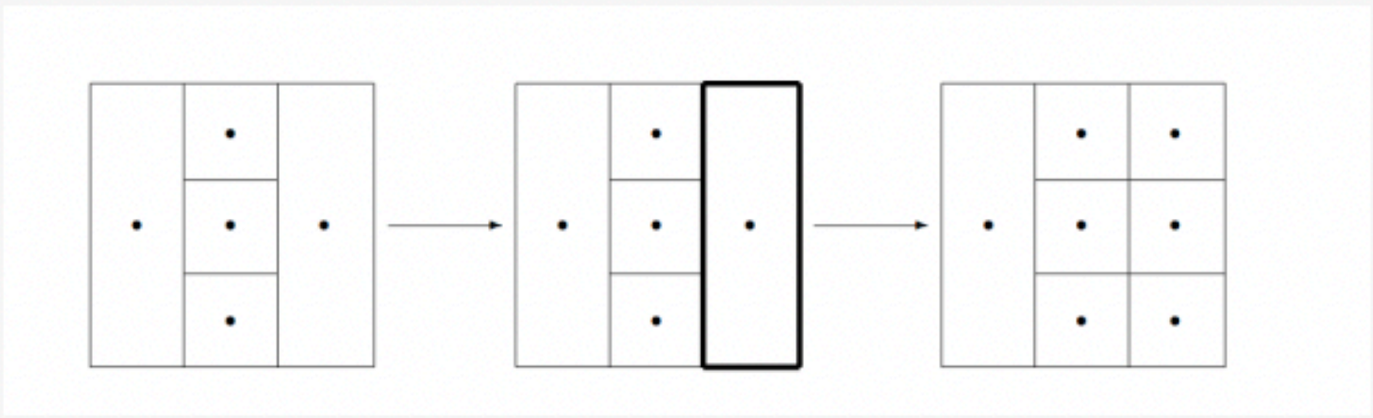
- Once the basin of the optimum is found, the search would proceed more quickly if *K* could be reduced, thereby putting more emphasis on local search.

- DIRECT avoids the drawback of using a single, large Lipschitz constant in a different way. In particular, in each iteration it selects for further search *any* rectangle *i* that *could* have the lowest Lipschitzian lower bound *for some Lipschitz constant* $K > 0$, subject to the constraint that the resulting lower bound is "non-trivially better" than the current best solution f_{min}. We call the rectangles that meet this criterion "potentially optimal" rectangles (*see next slides*).

DIVIDING RECTANGLES (DIRECT)

MAIN GIST OF THE ALGORITHM

- Firstly, normalize the search space into a hypercube in $[0,1]^D$, being D the problem dimension.
- Evaluate the objective function at the center of this hypercube, c.
- Initial sampling: divide the hypercube into smaller hyper-rectangles by evaluating the objective function at one-third of the distance, δ, from the center in all coordinate directions $c \pm \delta e_i$, where e_i are the basis vectors, for $i = 1, 2, \ldots D$. As shown in the previous slide, the division is performed so that the region with the best function value is given the largest space.

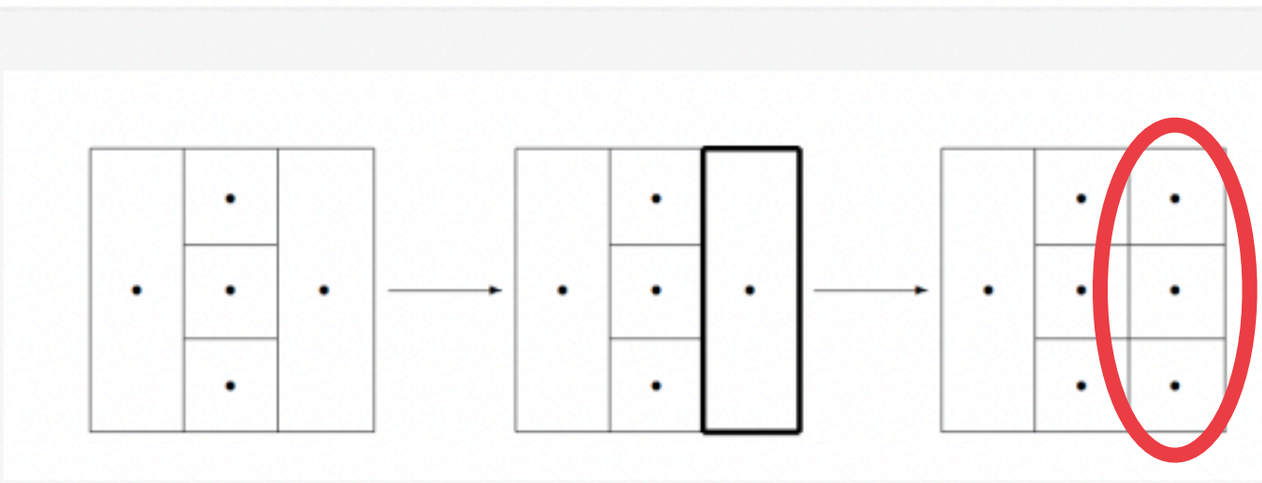


After the initial sampling phase, the rectangle on the far right is potentially optimal, and is therefore divided into thirds along its longest side.

DIVIDING RECTANGLES (DIRECT)

MAIN GIST OF THE ALGORITHM

- Then, identify the set *S* of **"potentially optimal" boxes** *(see next slide)*. For each of these boxes, unless the box is a (hyper)cube, divide it into thirds along its longest side. The objective function is evaluated at points $c \pm \delta e_i$ for all the longest dimensions *i*.



After the initial sampling phase, the rectangle on the far right is potentially optimal, and is therefore divided into thirds along its longest side.

DIVIDING RECTANGLES (DIRECT)

HOW TO IDENTIFY POTENTIALLY OPTIMAL BOXES?

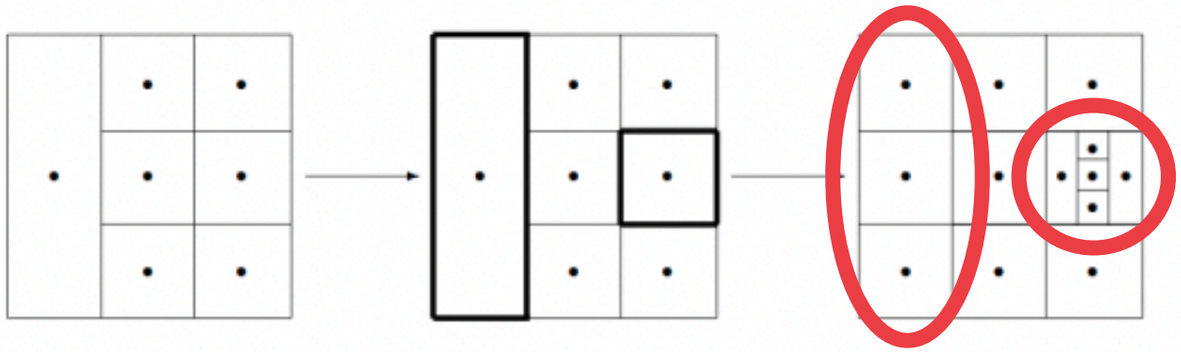
- **DEFINITION**: Let $\varepsilon > 0$ be a positive constant and let f_{min} be the current best function value.
  A hyper-rectangle j is said to be potentially optimal if there exists $K \geq 0$ such that:
  - $f(c_j) - Kd_j \leq f(c_i) - Kd_i$, for all $i = 1, 2, \ldots m$ (m being the current number of hyper-rectangles)
  - $f(c_j) - Kd_j \leq f_{min} - \varepsilon |f_{min}|$
  where c_j is the center of the hyper-rectangle j and d_j is the distance from the center of j to the corresponding vertices. A value of ε=0.0001 (more in general, the desired accuracy of the solution) is typically used to ensure that $f(c_j)$ exceeds f_{min} by some non-trivial amount.

- From the definition, we observe that a hyper-rectangle i is potentially optimal if:
  1. $f(c_i) \leq f(c_j)$ for all hyper-rectangles of equal size, $d_i = d_j$
  2. $d_i \geq d_k$ for all other hyper-rectangles k, and $f(c_i) \leq f(c_j)$ for all hyper-rectangles of equal size, $d_i = d_j$
  3. $d_i \leq d_k$ for all other hyper-rectangles k, and $f(c_i) = f_{min}$

DIVIDING RECTANGLES (DIRECT)

MAIN GIST OF THE ALGORITHM

- If the selected box is a (hyper)cube, i.e., all its edges have the same length, divide it as in the initial sampling and evaluate the objective function in all coordinate directions.

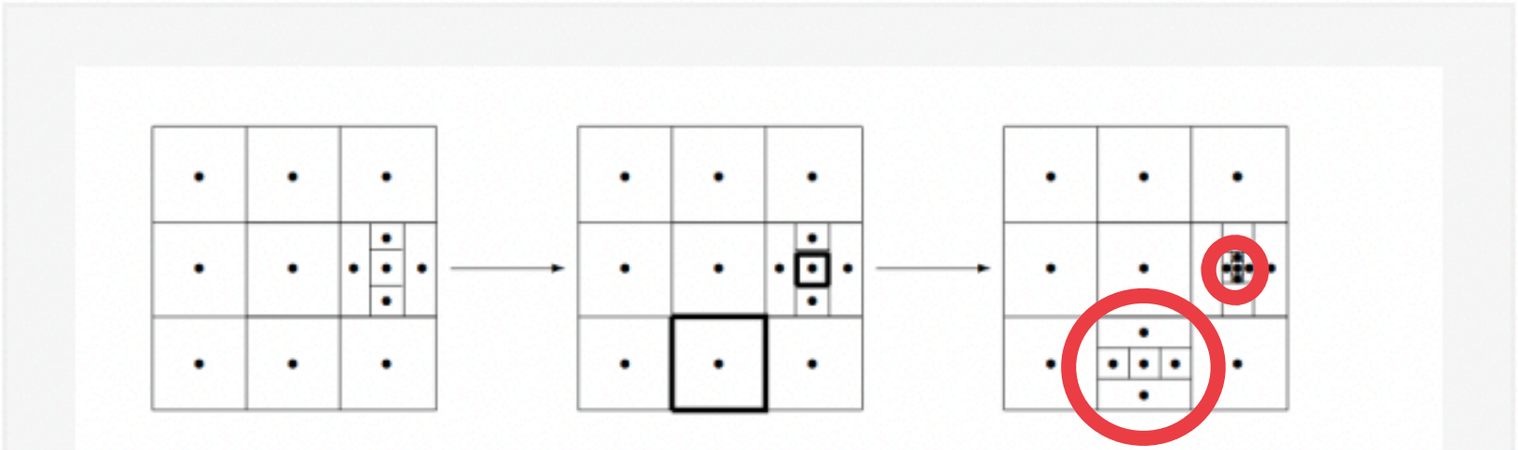


The central rectangle on the far right is potentially optimal. This rectangle is a cube, therefore division is identical to that of the initial sampling phase. The rectangle on the far left is also potentially optimal as it has a side which is longer than the sides of all other rectangles.

DIVIDING RECTANGLES (DIRECT)

MAIN GIST OF THE ALGORITHM

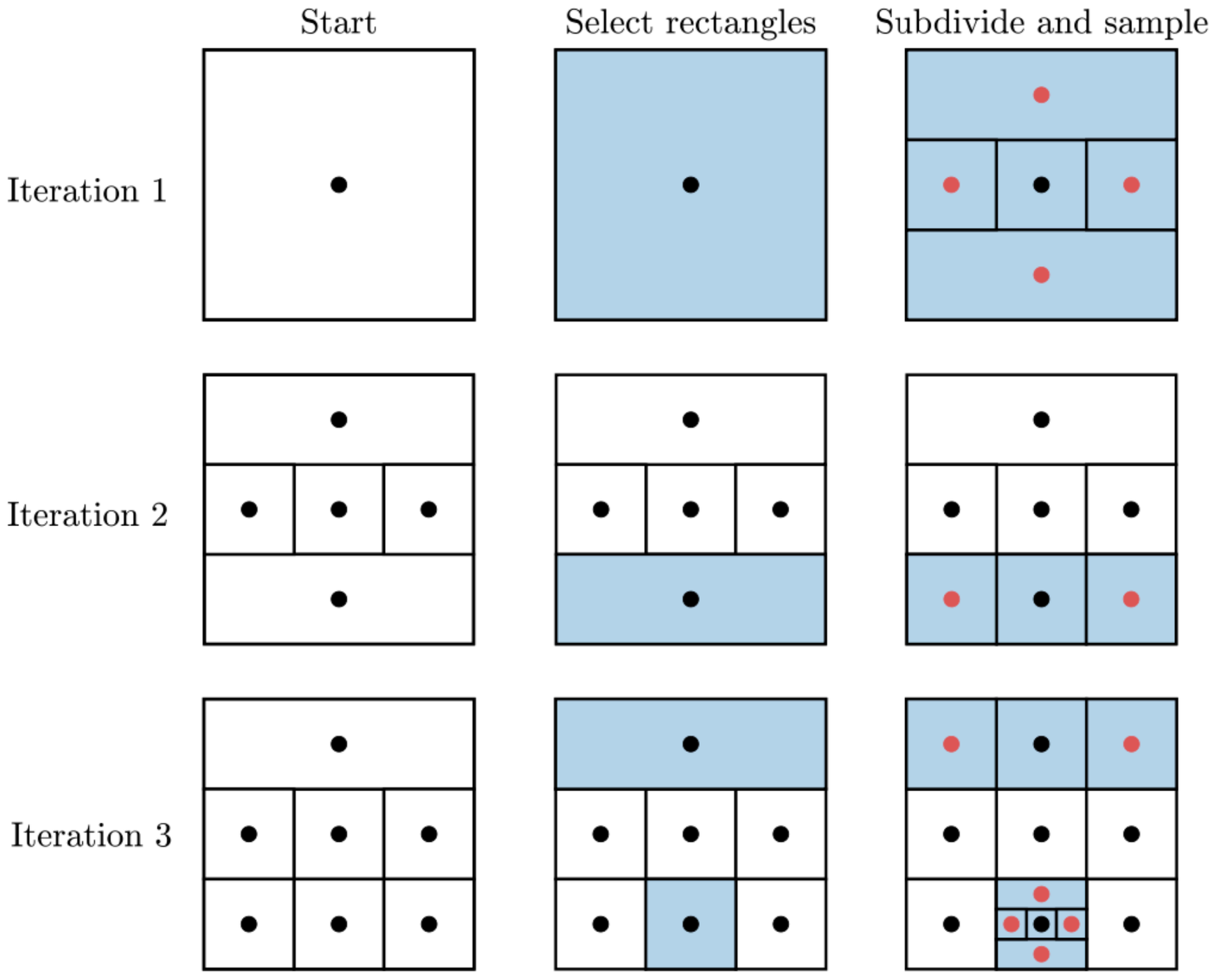
- This process of selecting potentially optimal rectangles, sampling, and dividing (with order determined by the objective function values returned during the sampling phase) is once again repeated, and continues for subsequent iterations until the stopping criterion is met.



The cube in the center of the far right rectangle is potentially optimal and is divided. In this case, the central cube on the bottom row, which has a side length that is greater than or equal to the side length of all other rectangles, is also potentially optimal and is therefore divided.

DIVIDING RECTANGLES (DIRECT)

MAIN GIST OF THE ALGORITHM



DIVIDING RECTANGLES (DIRECT)

PSEUDO-CODE

Given an objective function f and the design space $D = D_0$:

Step 1. Normalize the design space D to be the unit hypercube. Sample the center point c_i of this hypercube and evaluate $f(c_i)$. Initialize $f_{\min} = f(c_i)$, evaluation counter $m = 1$, and iteration counter $t = 0$.

Step 2. Identify the set S of potentially optimal boxes.

Step 3. Select any box $j \in S$.

Step 4. Divide the box j as follows:

(1) Identify the set I of dimensions with the maximum side length. Let δ equal one-third of this maximum side length.

(2) Sample the function at the points $c \pm \delta e_i$ for all $i \in I$, where c is the center of the box and e_i is the ith unit vector.

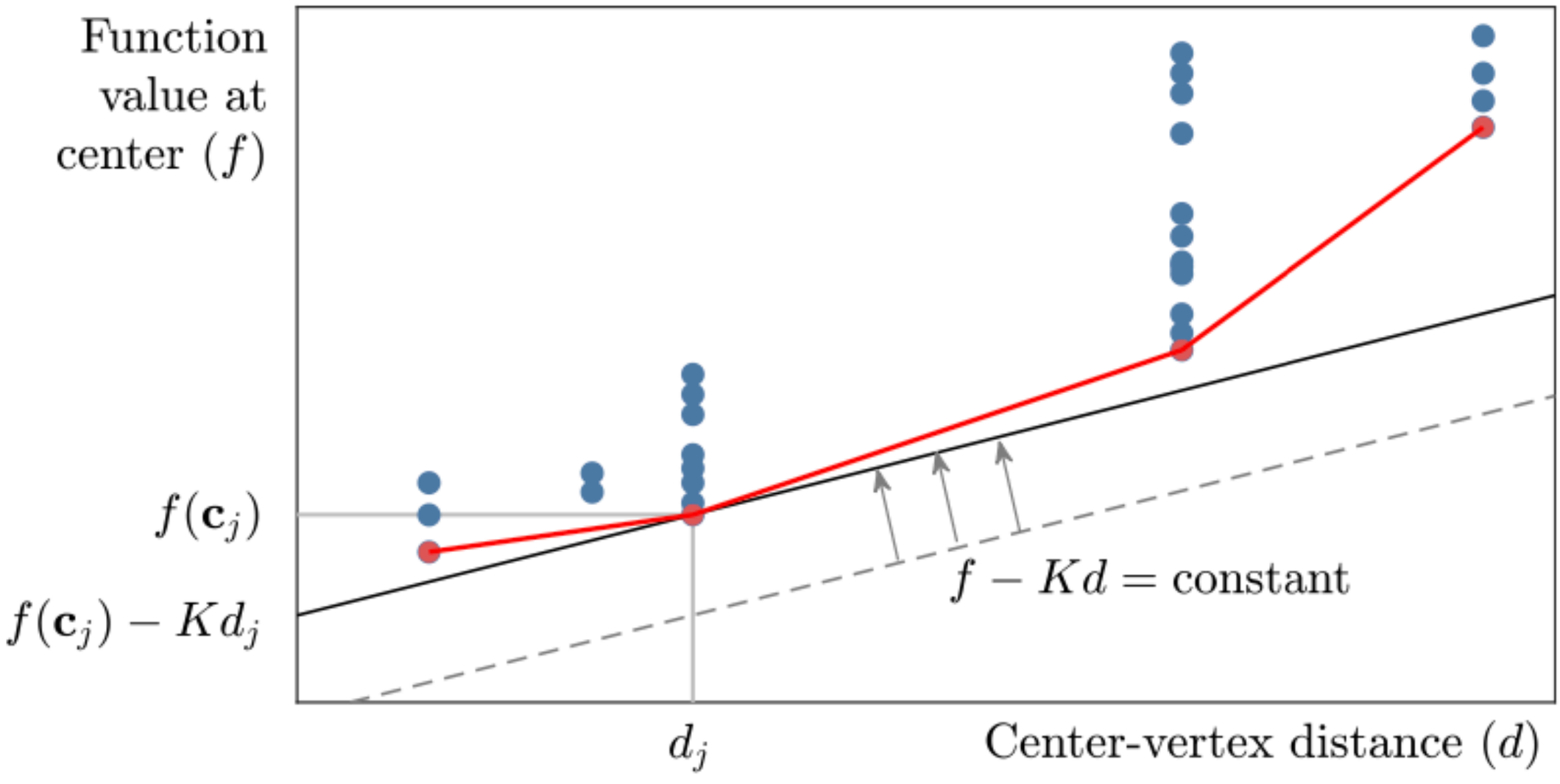
(3) Divide the box j containing c into thirds along the dimensions in I, starting with the dimension with the lowest value of $w_i = \min\{f(c + \delta e_i), f(c - \delta e_i)\}$, and continuing to the dimension with the highest w_i. Update $f_{\min}$ and m.

Step 5. Set $S = S - \{j\}$. If $S \neq \emptyset$ go to Step 3.

Step 6. Set $t = t + 1$. If iteration limit or evaluation limit has been reached, stop. Otherwise, go to Step 2.

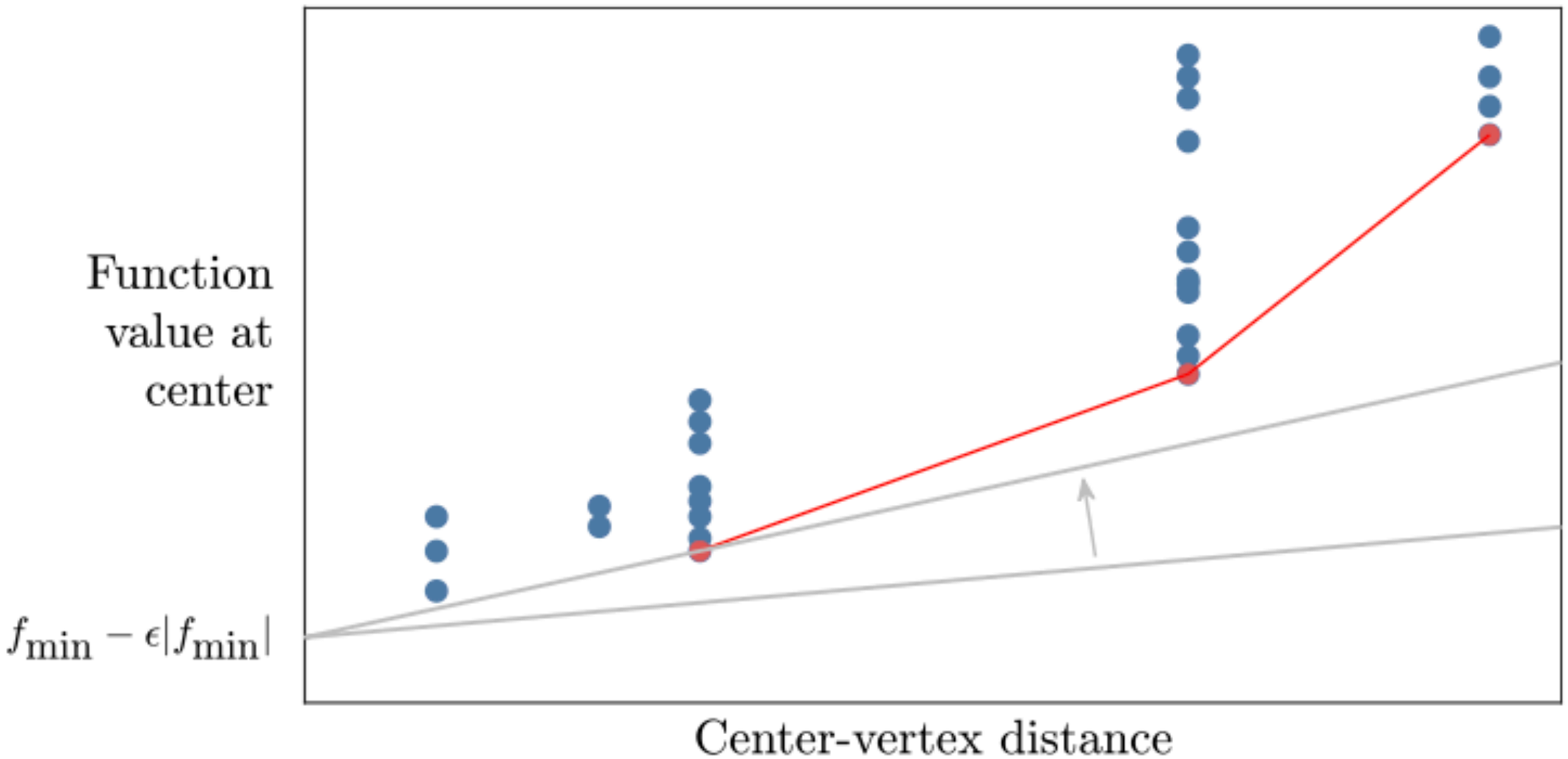
DIVIDING RECTANGLES (DIRECT)

GLOBAL CONVERGENCE PROPERTY



DIVIDING RECTANGLES (DIRECT)

GLOBAL CONVERGENCE PROPERTY



DIVIDING RECTANGLES (DIRECT)

ALGORITHMIC OPTIONS

Stopping rules

- Number of iterations or function evaluations.
- Minimum diameter: Terminate when the best potentially optimal box's diameter is less than this minimum diameter.
- Objective function convergence tolerance:

$$\tau_f = \frac{\tilde{f}_{\min} - f_{\min}}{1.0 + \tilde{f}_{\min}},$$

where $\tilde{f}_{\min}$ represents the previous computed minimum. The algorithm stops when τ_f becomes less than a user specified value.

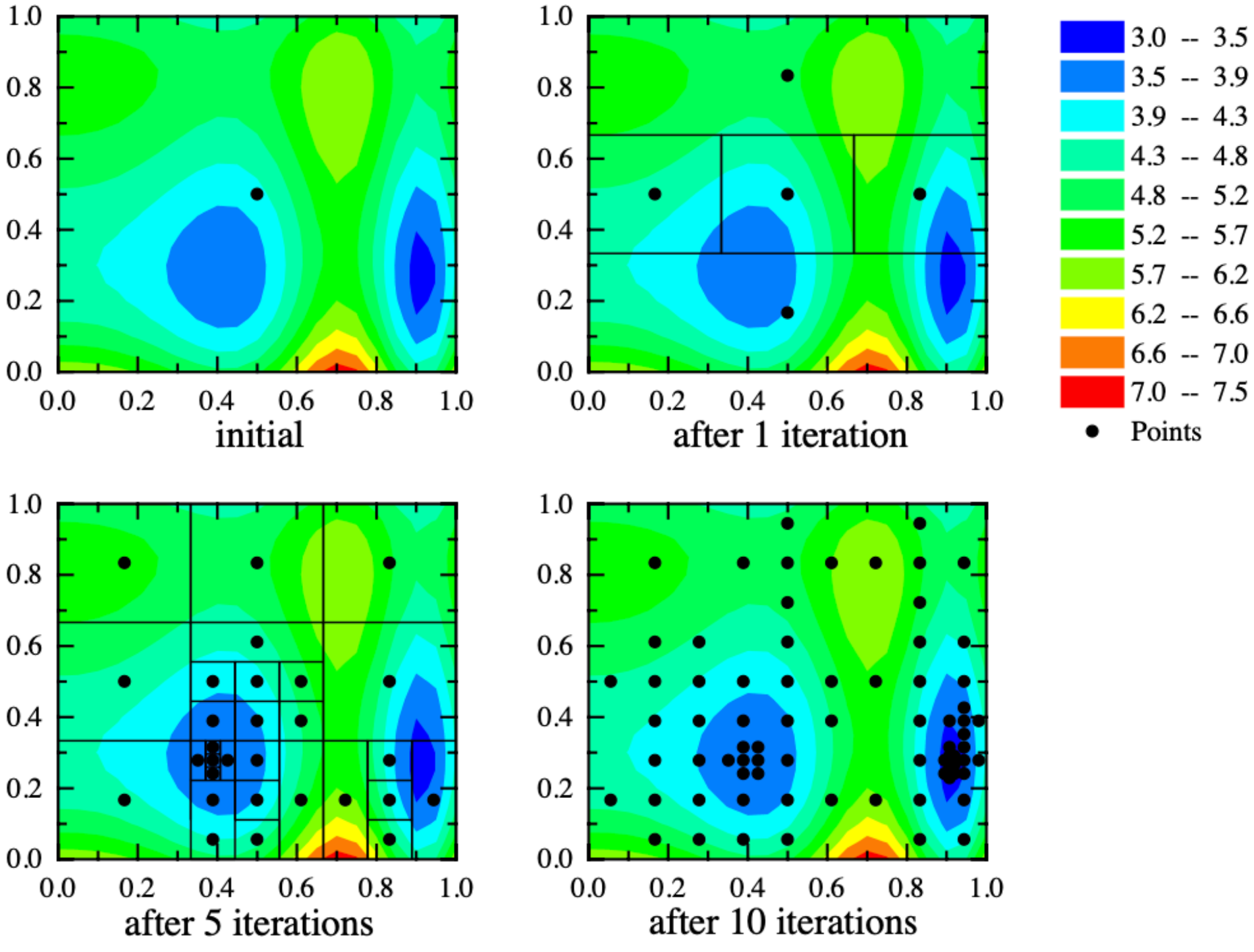
NOTE

As the iterations go to infinity, the size of the largest rectangle approaches zero. As a result, the points sampled by DIRECT will be "everywhere dense"; that is, for any point x in the initial hypercube and for any small $\delta > 0$, DIRECT is guaranteed to sample a point within a distance δ of x after some finite (possibly very large) number of iterations. If the objective function is continuous in the neighborhood of the global optimum x^*, then we are also guaranteed to eventually get within any given tolerance ε of the global minimum function value f^*.

This kind of "everywhere dense" convergence is not what one would ideally want; it would be much more desirable—and more efficient—if the algorithm only converged to the global minimum instead of converging to every point. Unfortunately, if the only thing one is willing to assume about the objective function is that it is continuous, then any deterministic algorithm that guarantees convergence to the global optimum must sample densely.

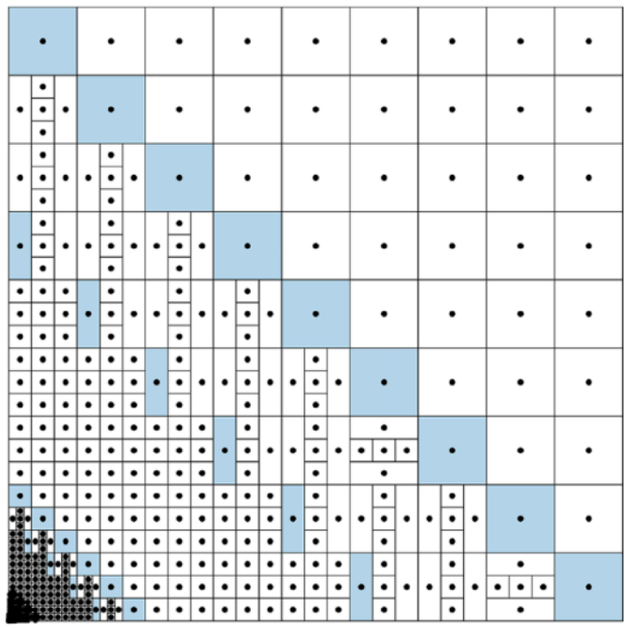
DIVIDING RECTANGLES (DIRECT)

THE ALGORITHM IN ACTION



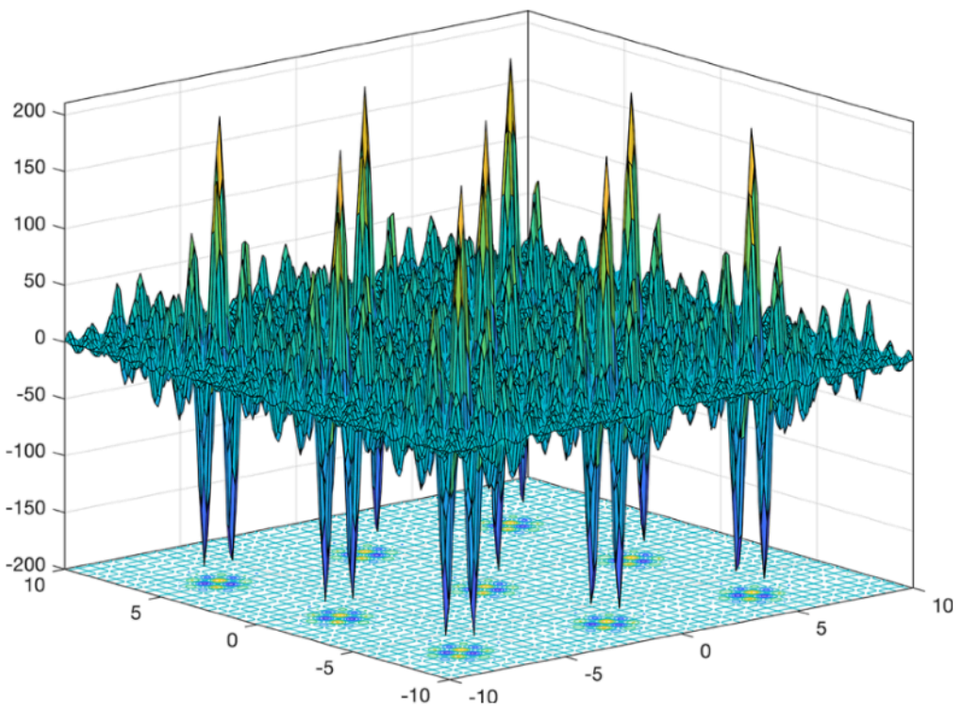
DIVIDING RECTANGLES (DIRECT)

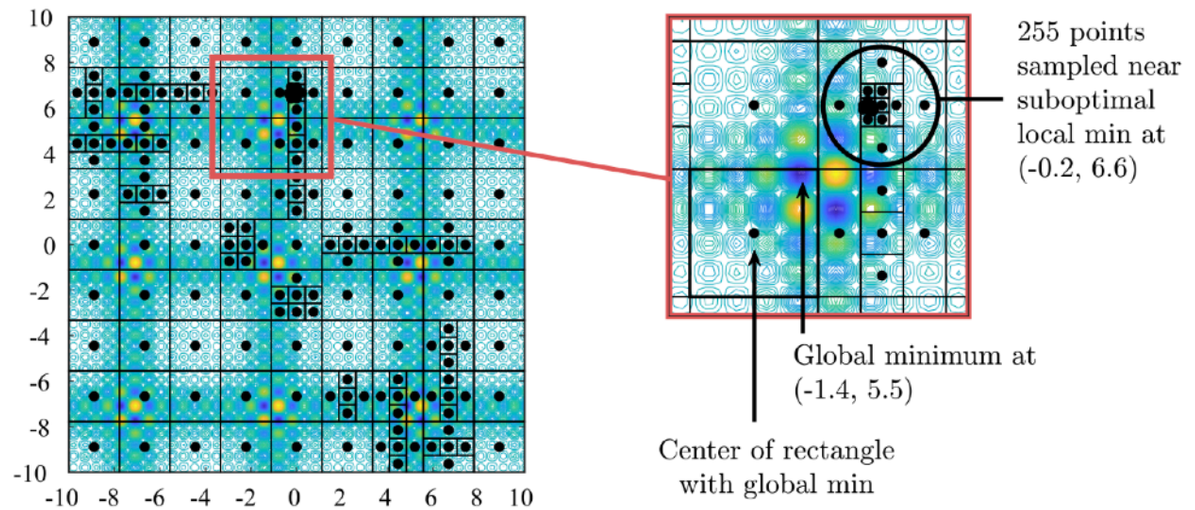
PROBLEMS (MANY MODERN VARIANTS TRY TO ADDRESS THEM!)



Selected rectangles (shaded) after 497 function evaluations when minimizing $f(x) = 1 + x_1 + x_2$

(many ties → too many rectangles are selected!)

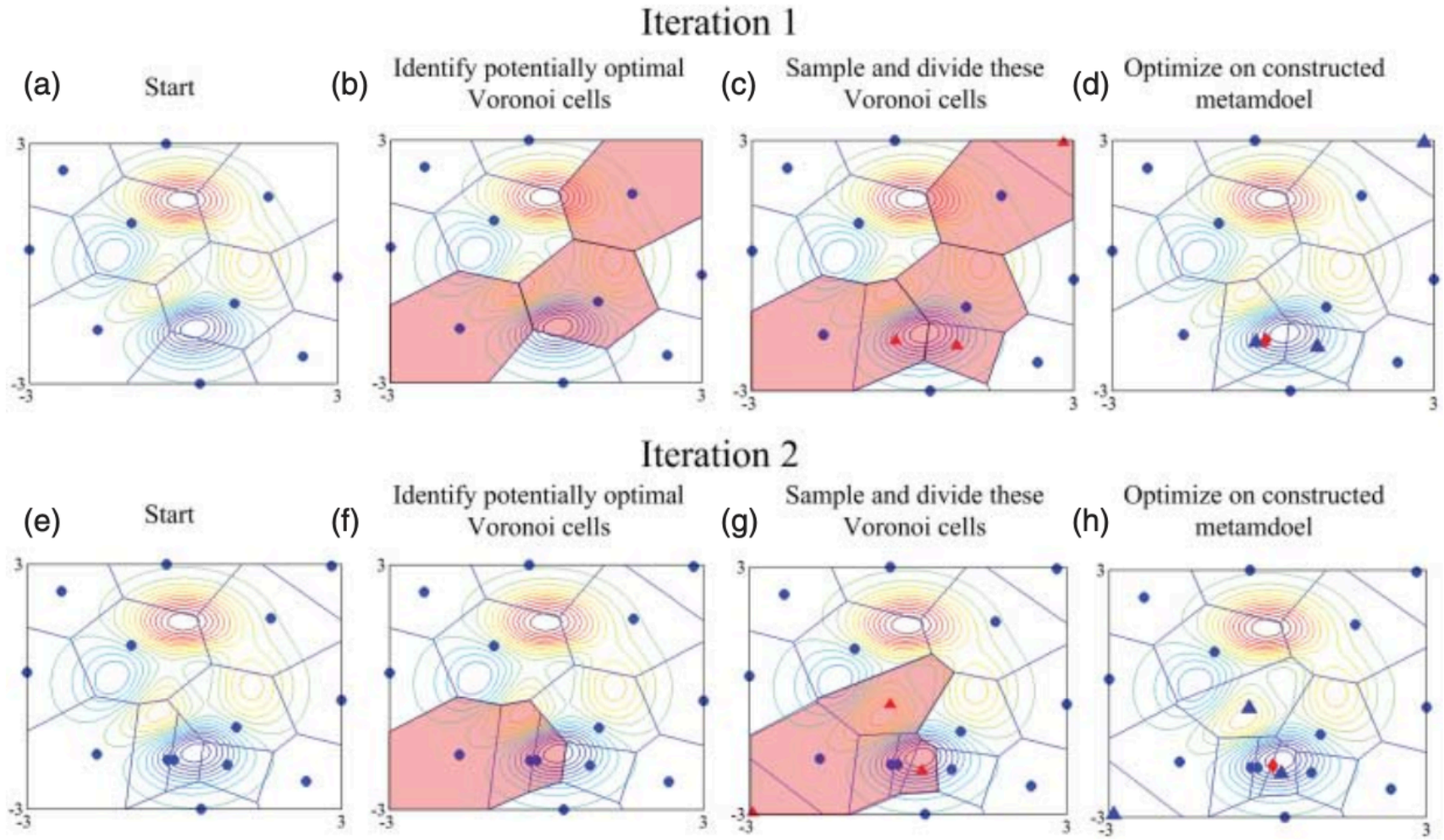


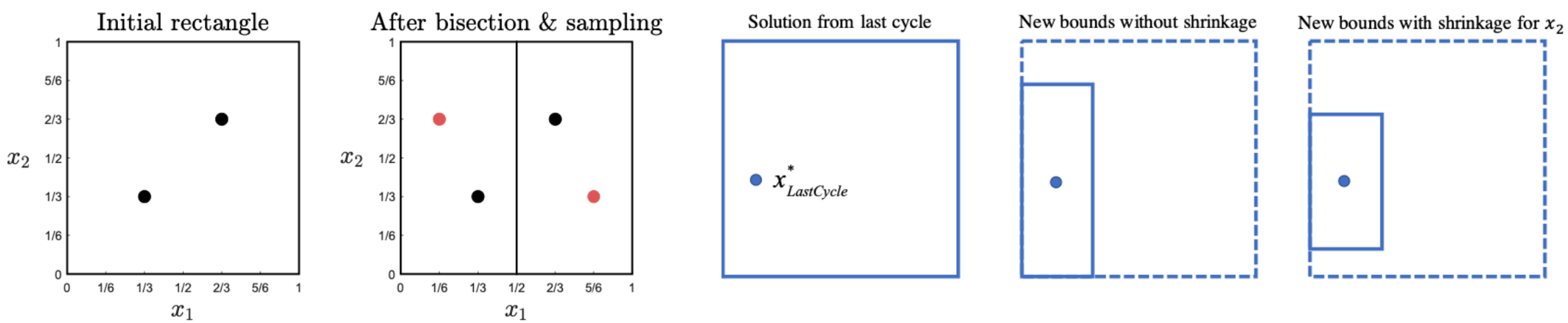


Too much time spent to refine local minima!

DIVIDING RECTANGLES (DIRECT)

SOME DIRECT VARIANTS: eDIRECT, BIRECT, HD-DIRECT, etc.





DIVIDING RECTANGLES (DIRECT)

DIRECT IN SCIPY.OPTIMIZE

scipy.optimize.direct

```
scipy.optimize.direct(func, bounds, *, args=(), eps=0.0001, maxfun=None,
maxiter=1000, locally_biased=True, f_min=- inf, f_min_rtol=0.0001, vol_tol=1e-16,
len_tol=1e-06, callback=None)                                          [source]
```

Finds the global minimum of a function using the DIRECT algorithm.

Parameters: **func** : *callable*

The objective function to be minimized. `func(x, *args) -> float` where `x` is an 1-D array with shape (n,) and `args` is a tuple of the fixed parameters needed to completely specify the function.

bounds : *sequence or* ***Bounds***

Bounds for variables. There are two ways to specify the bounds:

1. Instance of **`Bounds`** class.
2. `(min, max)` pairs for each element in `x`.

args : *tuple, optional*

Any additional fixed parameters needed to completely specify the objective function.

eps : *float, optional*

Minimal required difference of the objective function values between the current best hyperrectangle and the next potentially optimal hyperrectangle to be divided. In

DIVIDING RECTANGLES (DIRECT)

FURTHER READING

http://websites.umich.edu/~mdolaboratory/pdf/Jones2020a.pdf

This is a preprint of the following article, which is available at: http://mdolab.engin.umich.edu
Donald R. Jones and Joaquim R. R. A. Martins. The DIRECT Algorithm—25 Years Later. *Journal of Global Optimization*, 2020 (In press), doi: 10.1007/s10898-020-00952-6.

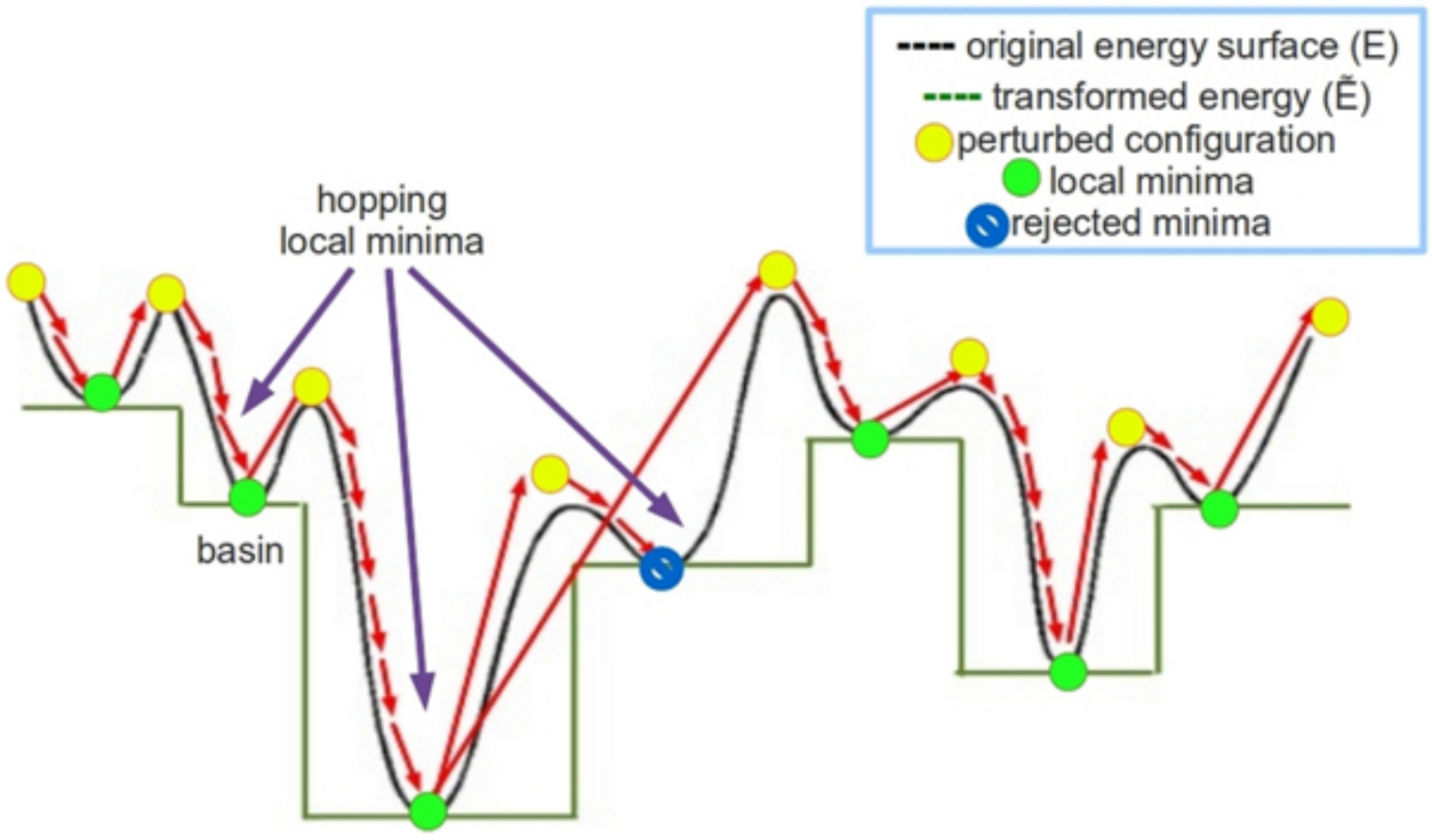
The DIRECT Algorithm—25 Years Later

Donald R. Jones
Joaquim R. R. A. Martins
Department of Aerospace Engineering, University of Michigan, Ann Arbor, MI, 48109

Abstract

Introduced in 1993, the DIRECT global optimization algorithm provided a fresh approach to minimizing a black-box function subject to lower and upper bounds on the variables. In contrast to the plethora of nature-inspired heuristics, DIRECT was *deterministic* and had only one hyperparameter (the desired accuracy). Moreover, the algorithm was simple, easy to implement, and usually performed well on low-dimensional problems (up to six-variables). Most importantly, DIRECT balanced local and global search (exploitation versus exploration) in a unique way: in each iteration, several points were sampled, some for global and some for local search. This approach eliminated the need for "tuning parameters" that set the balance between local and global search.

Basin hopping

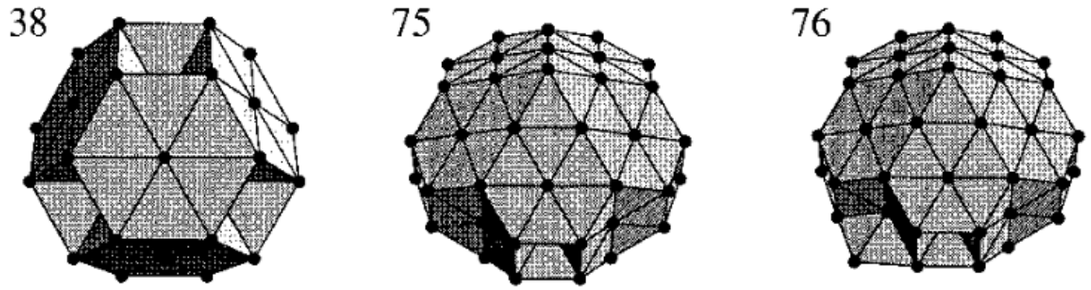


BASIN HOPPING

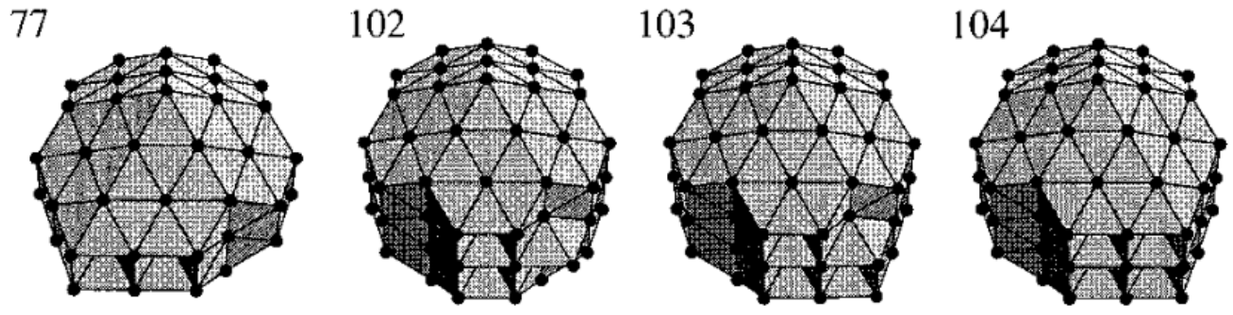
GENERAL INFORMATION

- Originally proposed by Wales & Doye in 1997 for use in the field of chemical physics: David J. Wales and Jonathan P. K. Doye, Global Optimization by Basin-Hopping and the Lowest Energy Structures of Lennard-Jones Clusters Containing up to 110 Atoms (1997) https://pubs.acs.org/doi/10.1021/jp970984n

- Mean feature: performs a series of local search runs, in between it "hops" around the problem space to explore different "basins" (i.e., areas around a minimum), to then find the optimum



- It works well for problems with "funnel-like, but rugged" landscapes (many minima separated by large barriers)



DEFINITION (basin, or "attraction basin"): *For the position* $\boldsymbol{x}_i^* \in \mathcal{X}$ *of an optimum,* $\text{basin}(\boldsymbol{x}_i^*) \subseteq \mathcal{X}$ *is the largest set of points such that for any starting point* $\boldsymbol{x} \in \text{basin}(\boldsymbol{x}_i^*)$ *the infinitely small step steepest descent algorithm will converge to* $\boldsymbol{x}_i^*$.

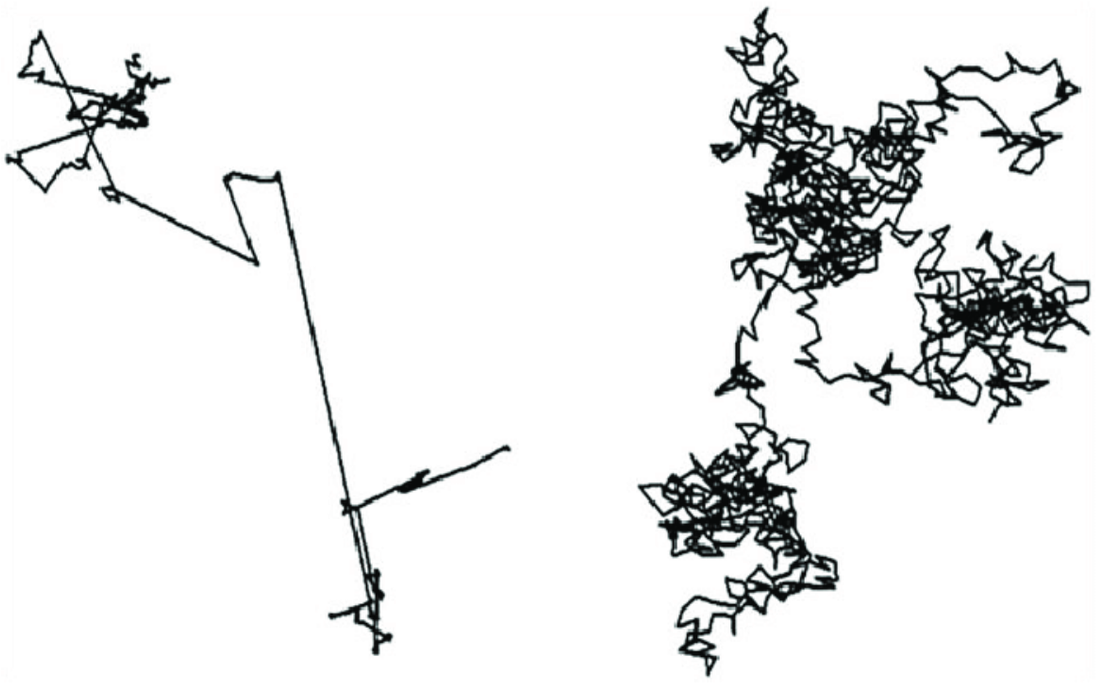
BASIN HOPPING

RELATED CONCEPTS *(we'll see most of them in the next lectures)*

- Although under a different name, BH is essentially a form of *Iterated Local Search* (ILS) with different (perturbed) starting points

- Also called **Monte Carlo Minimization** (MCM): in fact, it can be seen as an extension of the *Monte-Carlo sampling approach*

- Similar concepts in other areas of numerical optimization
  - *Simulated annealing*: it accepts worsening of the current solution with a decreasing probability
  - *Memetic algorithms*: genetic algorithms + local search
  - *Variable-size/adaptive mutations*, e.g. based on Lévy flight (it takes in most cases small random steps, with occasional big jumps)



Lévy flight (left)
vs
random walk (right)

BASIN HOPPING

MAIN GIST OF THE ALGORITHM

- Loop through the following steps:

1. Hopping: i.e., a perturbation of the solution, typically accomplished by random Monte-Carlo moves. This should facilitate the search algorithm to "jump" to new regions of the search space and possibly locate a new basin leading to a differing optimum.

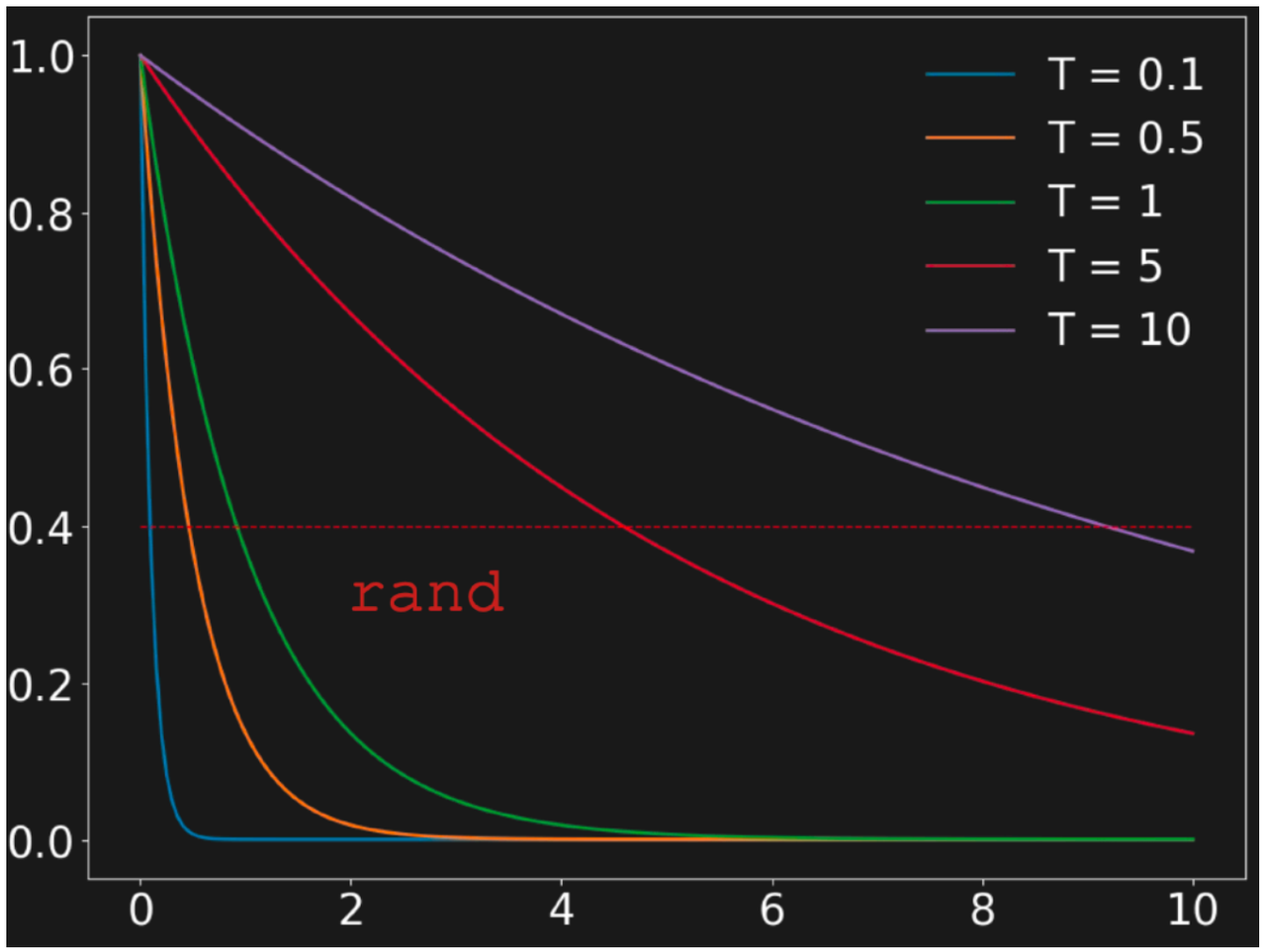
2. Local search: i.e., the perturbed solution is further refined by means of a LS algorithm (NOTE: in principle, any LS method can be used)

3. Acceptance/rejection of the new solution: i.e., decide if the new optimum should be kept as the basis for new random perturbations or discarded. This can be made in different ways. E.g. through the Metropolis criterion *(see next slide)*, threshold acceptance mechanisms, or statistic conditions.

BASIN HOPPING

METROPOLIS CRITERION

A stochastic decision function with a "temperature" variable *T*, a lot like Simulated Annealing. Temperature is adjusted as a function of the number of iterations of the algorithm and/or the quality of the solution. This facilitates arbitrary solutions to be accepted early in the run, when the temperature is "high", while it enforces a stricter policy of just accepting solutions of improved quality later on in the search, when the temperature is "low".



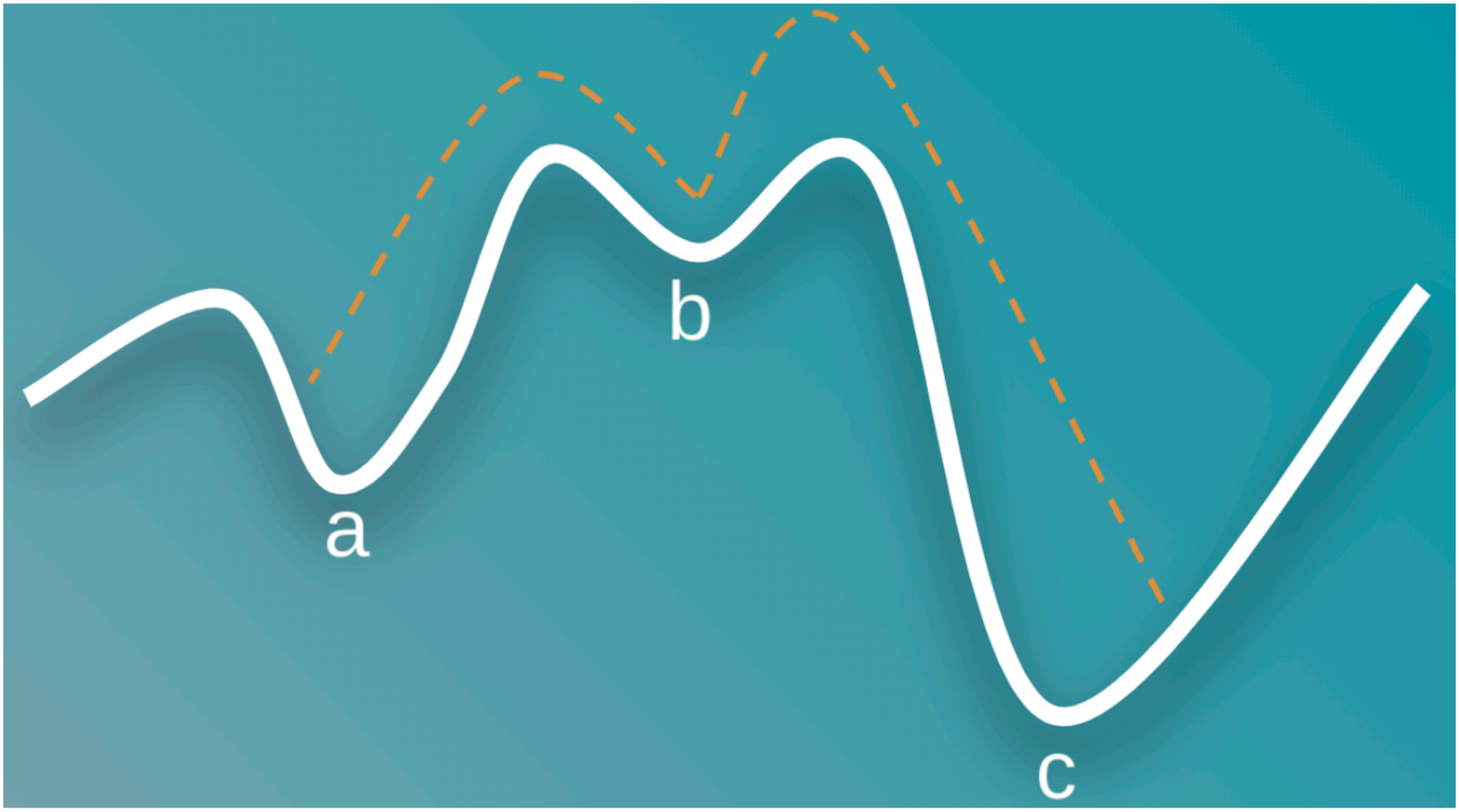
if $f(x_n) < f(x_{n-1})$
  accept x_n
else
  if $e^{-(f(x_n) - f(x_{n-1}))/T} \geq$ rand
    accept x_n
  else:
    reject x_n

Higher T: more likely to accept worse x_n

T → 0: the algorithm becomes more greedy

BASIN HOPPING

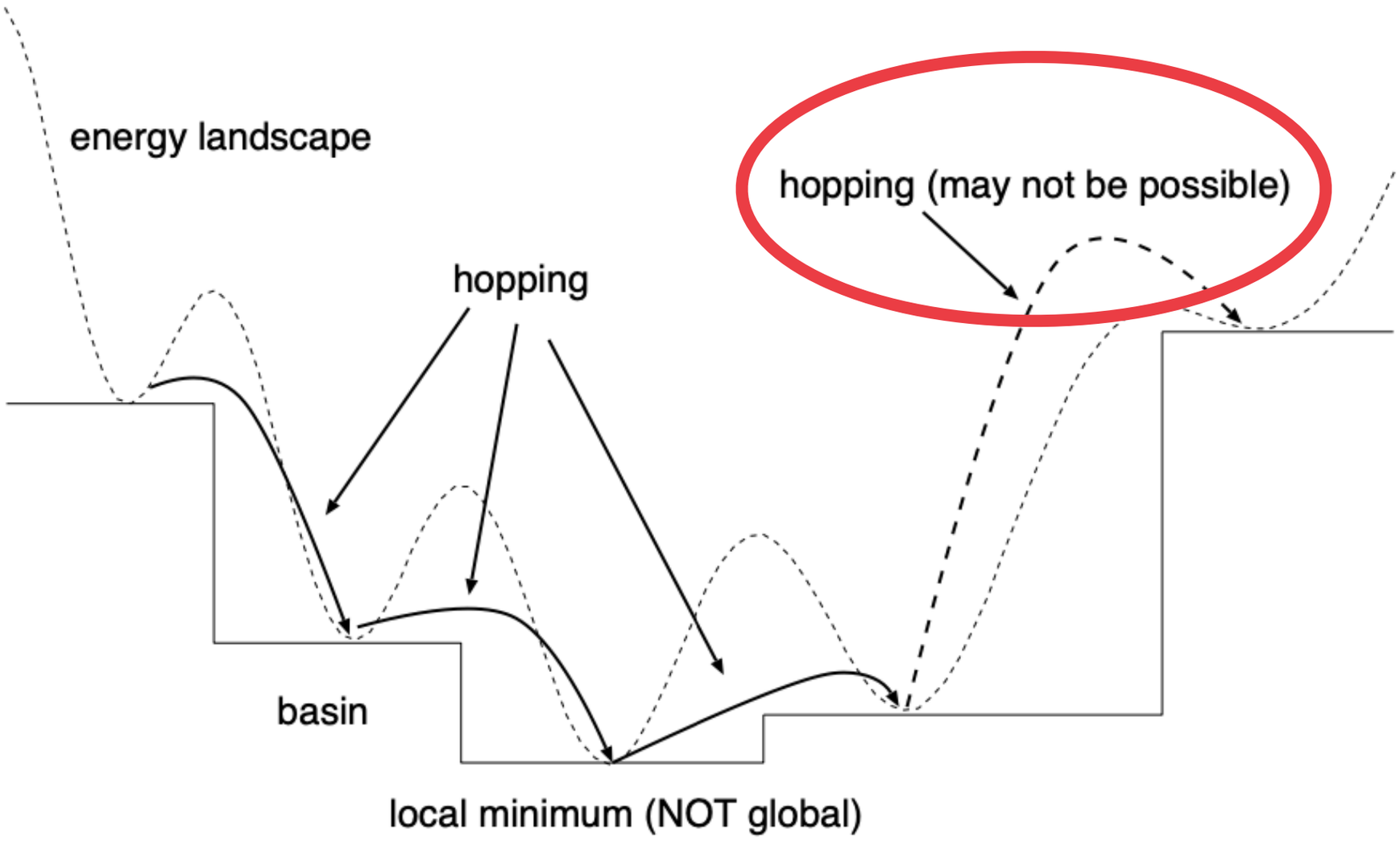
MAIN GIST OF THE ALGORITHM



BASIN HOPPING

MAIN GIST OF THE ALGORITHM

NOTE: In some cases, it may not be possible to move to a different basin by random hops!



BASIN HOPPING

BASIN HOPPING IN SCIPY.OPTIMIZE

scipy.optimize.basinhopping

```
scipy.optimize.basinhopping(func, x0, niter=100, T=1.0, stepsize=0.5,
minimizer_kwargs=None, take_step=None, accept_test=None, callback=None,
interval=50, disp=False, niter_success=None, seed=None, *, target_accept_rate=0.5,
stepwise_factor=0.9)                                                    [source]
```

Find the global minimum of a function using the basin-hopping algorithm.

Basin-hopping is a two-phase method that combines a global stepping algorithm with local minimization at each step. Designed to mimic the natural process of energy minimization of clusters of atoms, it works well for similar problems with "funnel-like, but rugged" energy landscapes [5].

As the step-taking, step acceptance, and minimization methods are all customizable, this function can also be used to implement other two-phase methods.

Parameters: **func** : *callable* ***f(x, *args)***

Function to be optimized. `args` can be passed as an optional item in the dict `minimizer_kwargs`

x0 : *array_like*

Initial guess.

niter : *integer, optional*

The number of basin-hopping iterations. There will be a total of `niter + 1` runs of the local minimizer.

The local search method. It is possible to use any method available in **scipy.optimize.minimize**.

BASIN HOPPING

BASIN HOPPING IN SCIPY.OPTIMIZE

scipy.optimize.minimize

`scipy.optimize.minimize(fun, x0, args=(), method=None, jac=None, hess=None, hessp=None, bounds=None, constraints=(), tol=None, callback=None, options=None)`

Minimization of scalar function of one or more variables. [source]

Parameters: **fun** : *callable*

The objective function to be minimized.

```
fun(x, *args) -> float
```

where `x` is a 1-D array with shape (n,) and `args` is a tuple of the fixed parameters needed to completely specify the function.

x0 : *ndarray, shape (n,)*

Initial guess. Array of real elements of size (n,), where `n` is the number of independent variables.

args : *tuple, optional*

Extra arguments passed to the objective function and its derivatives (*fun*, *jac* and *hess* functions).

method : *str or callable, optional*

Type of solver. Should be one of

- 'Nelder-Mead' (see here)
- 'Powell' (see here)
- 'CG' (see here)
- 'BFGS' (see here)
- 'Newton-CG' (see here)
- 'L-BFGS-B' (see here)
- 'TNC' (see here)
- 'COBYLA' (see here)
- 'SLSQP' (see here)
- 'trust-constr'(see here)
- 'dogleg' (see here)
- 'trust-ncg' (see here)
- 'trust-exact' (see here)
- 'trust-krylov' (see here)
- custom - a callable object (added in version 0.14.0), see below for description.

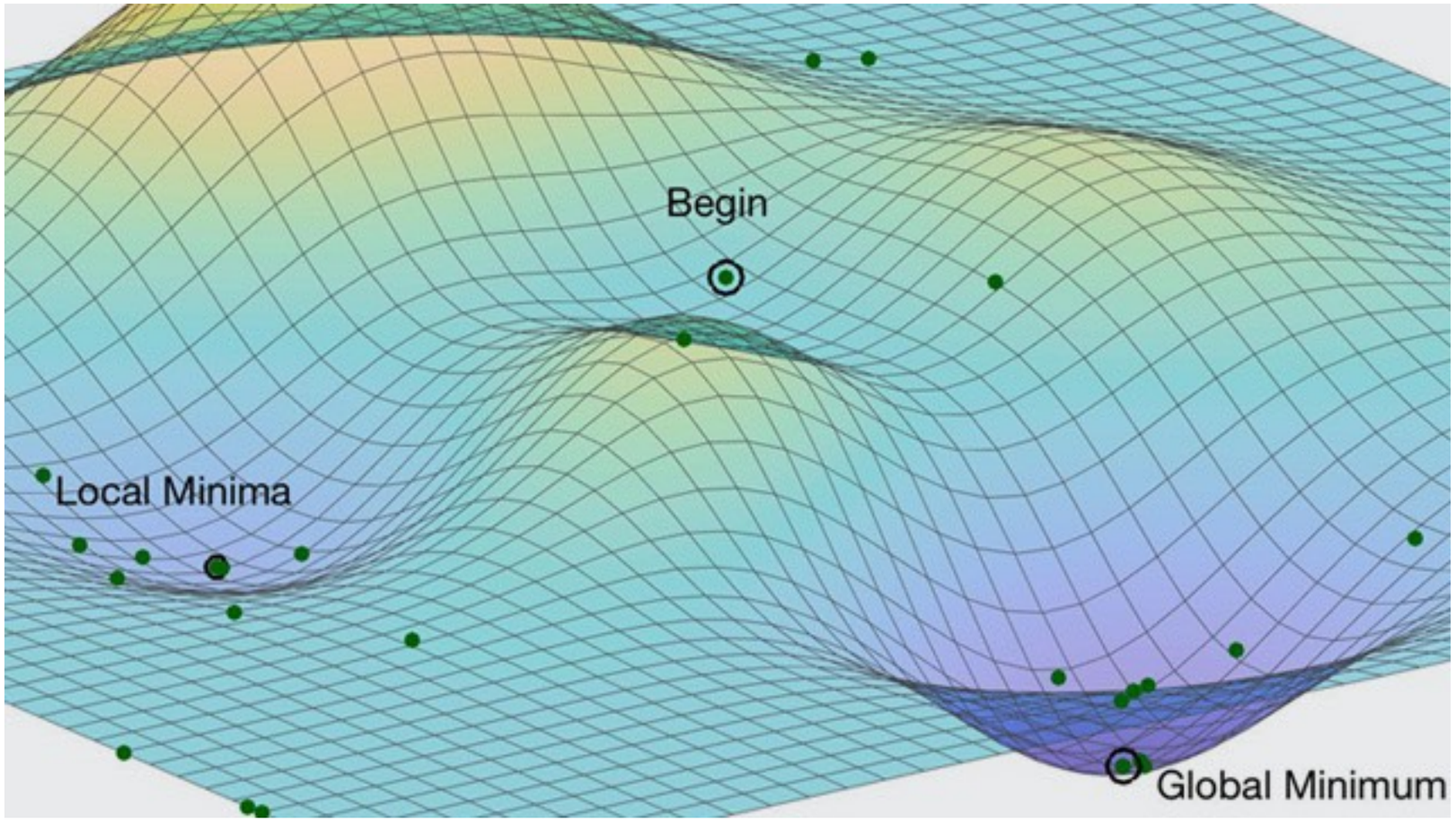
If not given, chosen to be one of `BFGS`, `L-BFGS-B`, `SLSQP`, depending on whether or not the problem has constraints or bounds.

By default *(we'll see something about these methods in the next lectures)*:

- BFGS (Broyden–Fletcher–Goldfarb–Shanno) algorithm: a quasi-Newton method, and its limited-memory variant L-BFGS (does not handle constraints).
- SLSQP: Sequential Least Squares Programming (handles constraints).

Other methods

(we'll see some of them in the rest of the course)



OTHER METHODS

THERE ARE SEVERAL OTHER OPTIONS FOR GLOBAL OPTIMIZATION

- **Metaheuristics/bio-inspired algorithms**: Genetic Algorithms, Swarm Intelligence methods, etc.
  - Usually, population-based: multiple solutions handled at a time, easily parallelizable
  - Don't have convergence guarantees, but practically work in several domains!

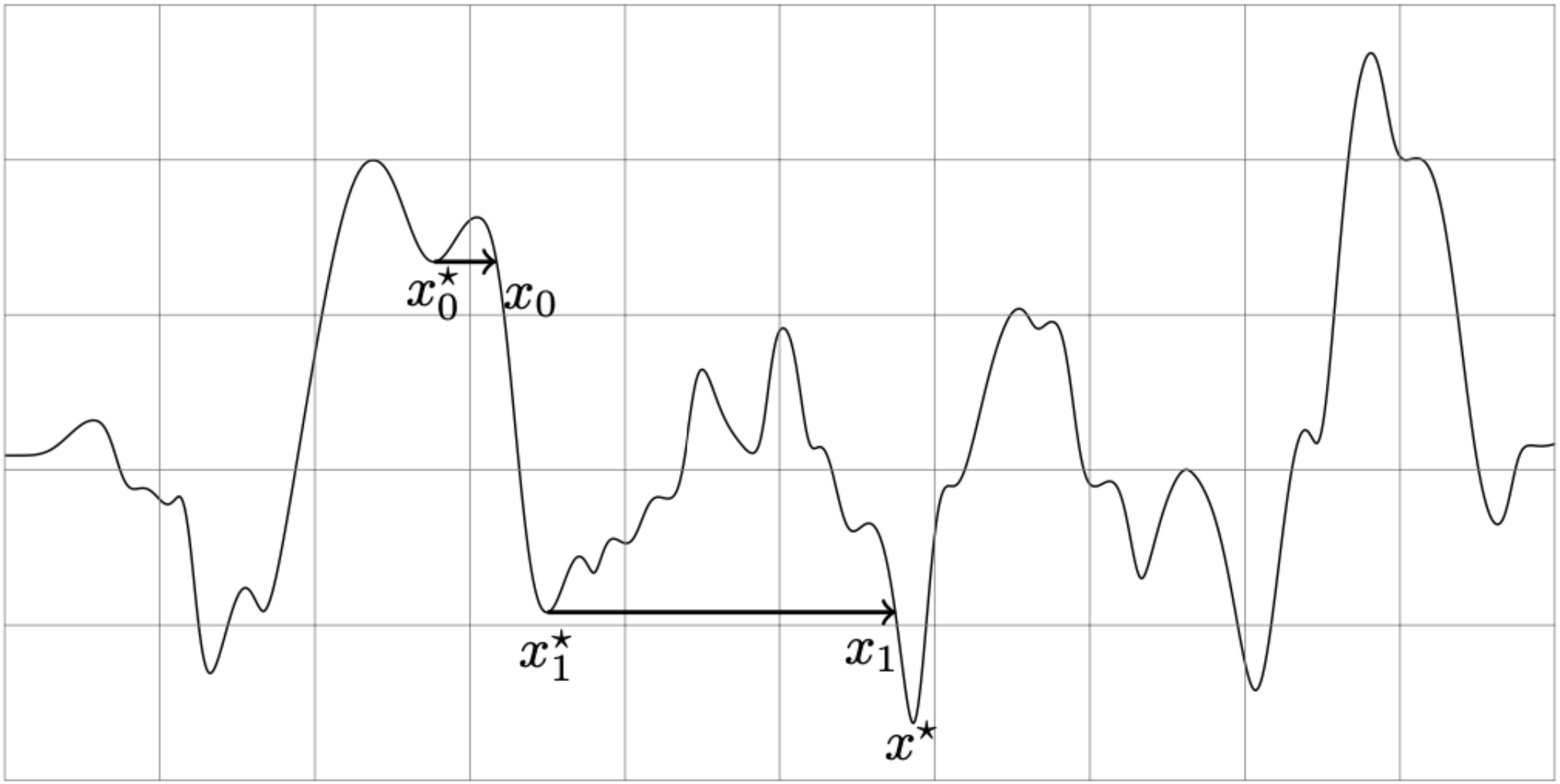
- **Simulated annealing**
  - Another simple metaheuristic that mimics the annealing process in metals.
  - Works on a single solution at a time, perturbed accordingly to a "temperature" cooling process.

- **Other kinds of Monte-Carlo sampling techniques**
  - E.g., stochastic tunneling derives an objective function non-linear transformation that facilitates "tunneling" from one local optimum to another (similar to quantum annealing)
  - Parallel tempering (a sort of ensemble of simulated annealing runs at different temperatures), based also on the work by Nobel Laureate Parisi

- **Bayesian optimization**
  - Especially suitable for expensive optimization (=each solution evaluation is computationally costly)
  - Effective for low dimensionalities (i.e., up to ~20 variables), e.g. in hyper-parameter optimization
  - Response-surface based method: it starts with a prior model, builds & updates a posterior of the objective function (usually based on Gaussian processes) to approximate the original function
  - The "acquisition function" decides where to sample the next solutions

OTHER METHODS

TUNNELING



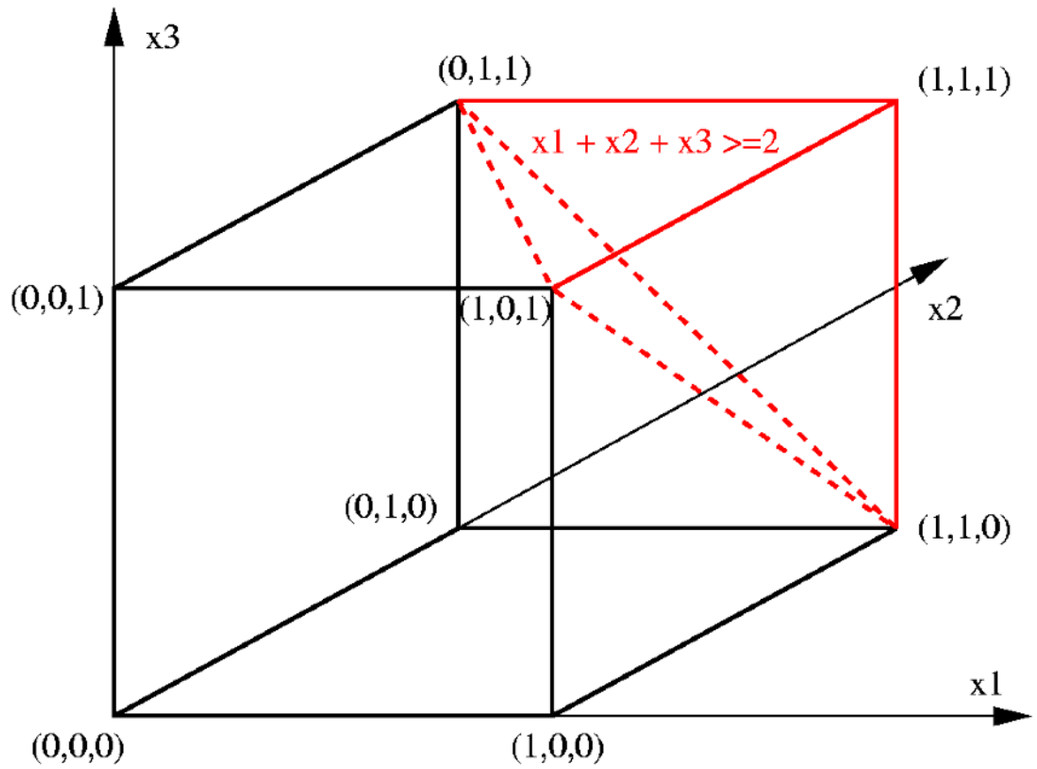
OTHER METHODS

THERE ARE SEVERAL OTHER OPTIONS FOR GLOBAL OPTIMIZATION

- **Cutting-plane, branch & bound**
  - Only for combinatorial / MILP problems

- **Tabu search**
  - Mostly for graph-based problems

- … and many others! (e.g., simplicial homology global optimization, available as scipy.optimize.shgo)



Questions?