OPTIMIZATION TECHNIQUES

Introduction - Grid search, random search and local search

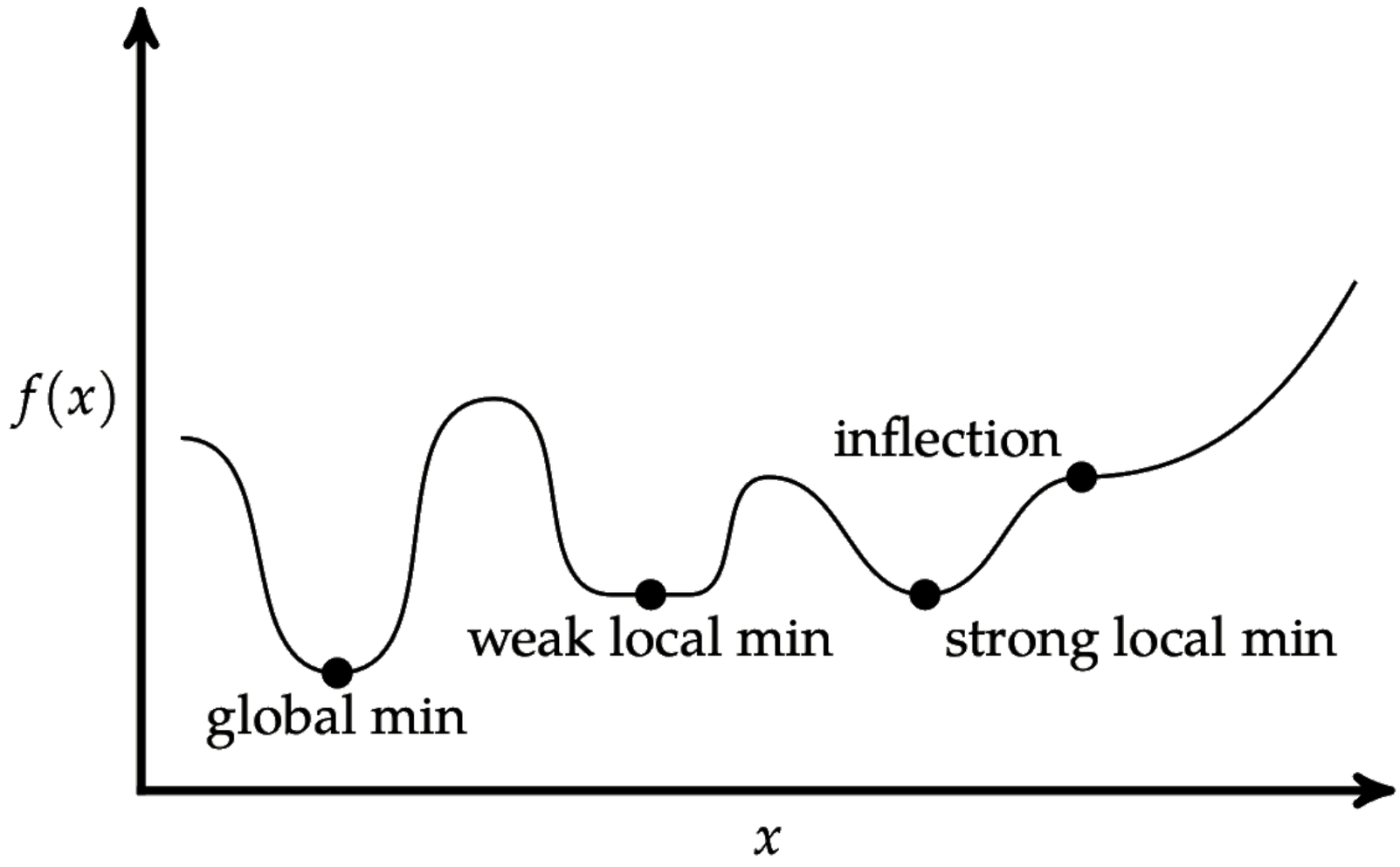
Prof. Giovanni Iacca

giovanni.iacca@unitn.it

UNIVERSITY OF TRENTO - Italy

Information Engineering
and Computer Science Department

Optimization



OPTIMIZATION

WHAT IS OPTIMIZATION?

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

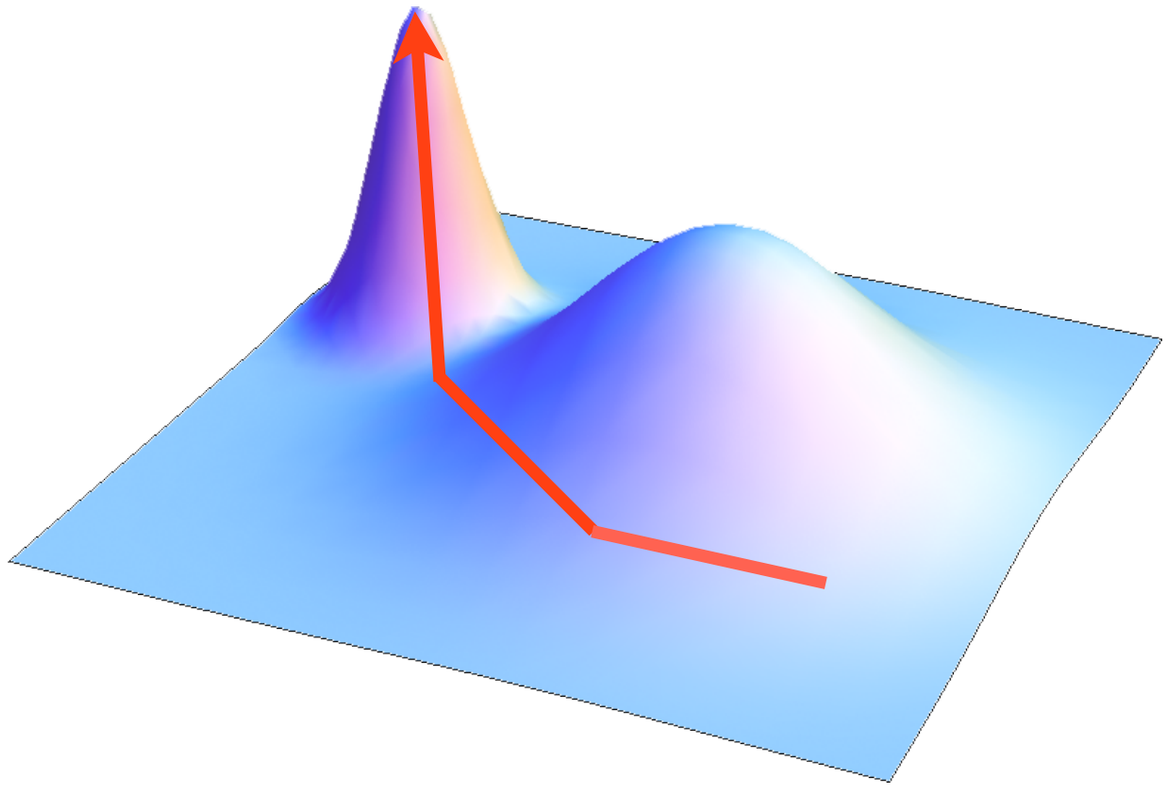
$$\text{find } x^* \ni f(x^*) = \min_{x \in D} \{f(x)\}$$

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

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

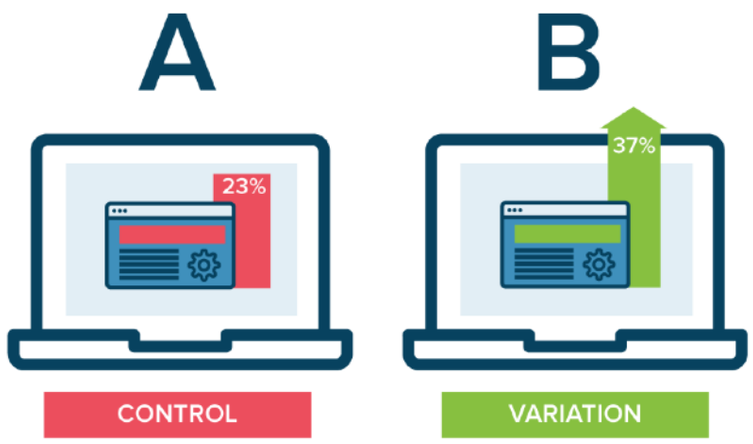
- Decision (design) variables:
  $x = [x(1), x(2), \ldots x(n)]$
- Objective function(s): $f(x)$
- Decision (design, search, solution) space: D
- Constraints: $g(x)$ and $h(x)$
- (Global/local) optimum: x^*

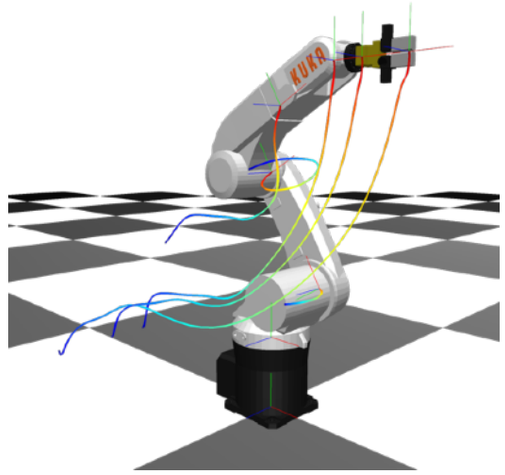
IMPORTANT NOTE
minimize $f(x)$ = maximize $-f(x)$

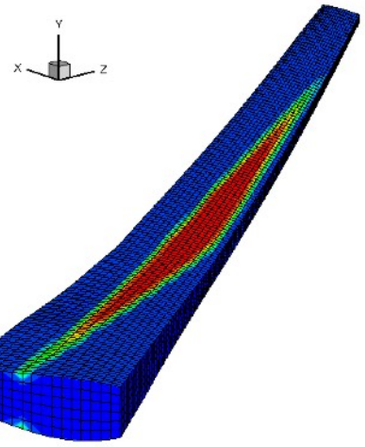


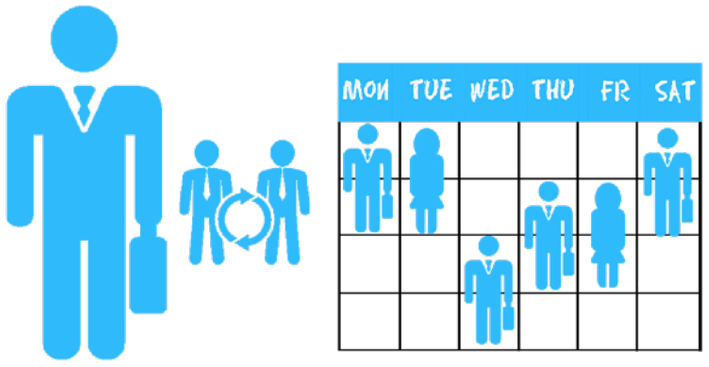
OPTIMIZATION

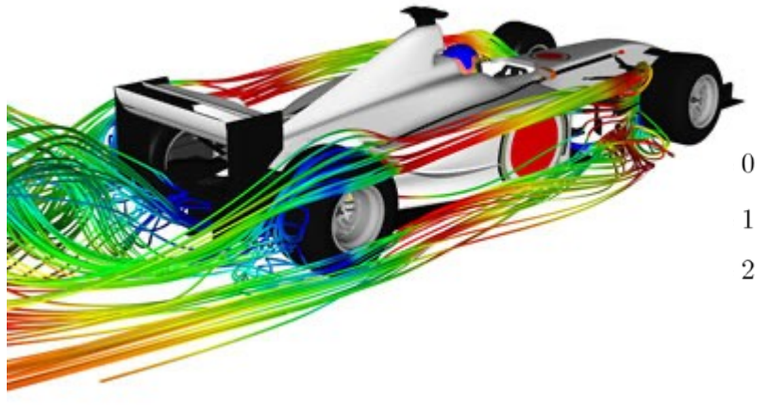
OPTIMIZATION IS “EVERYWHERE"

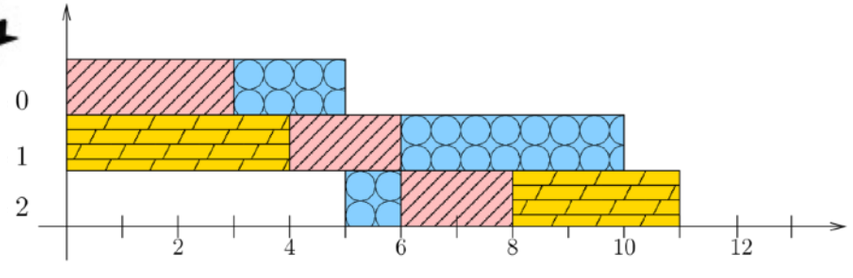


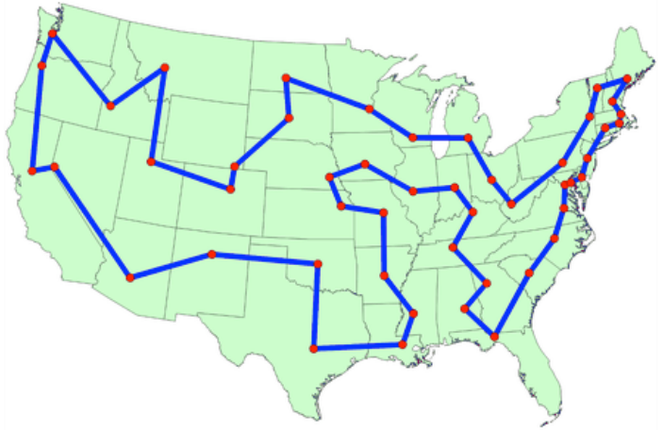






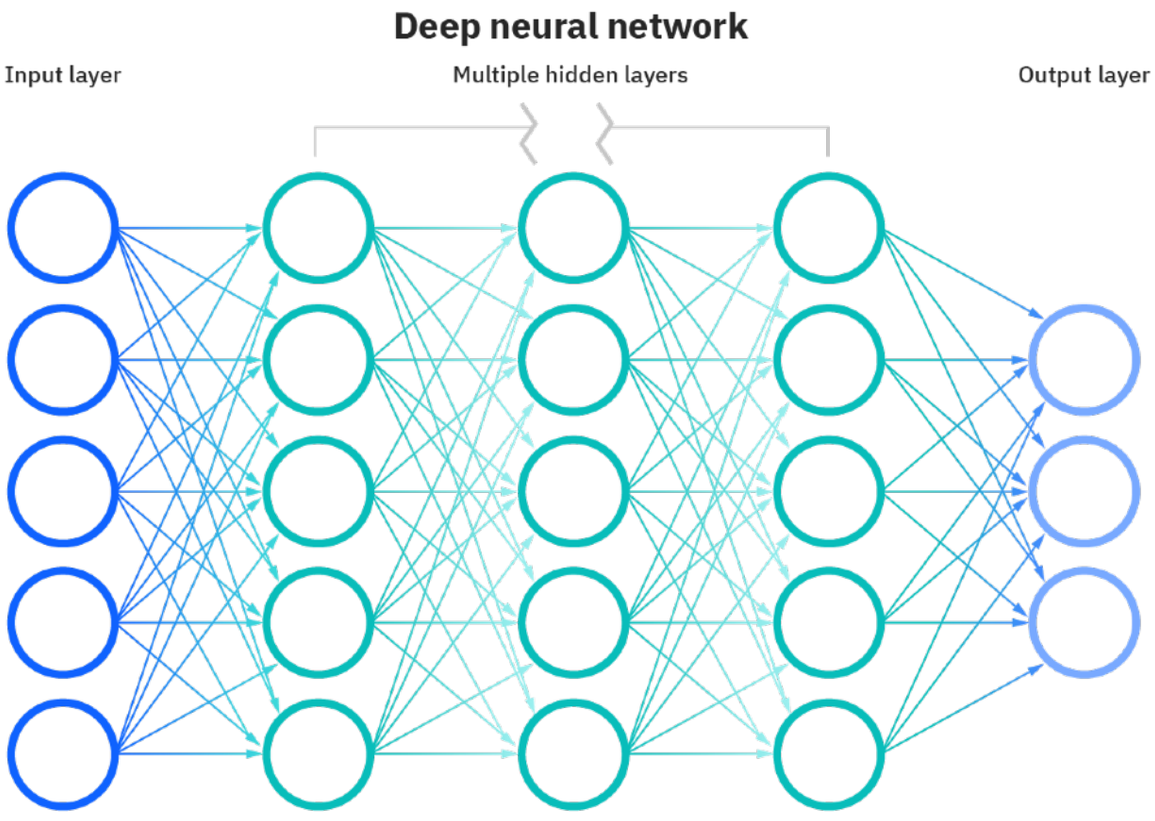


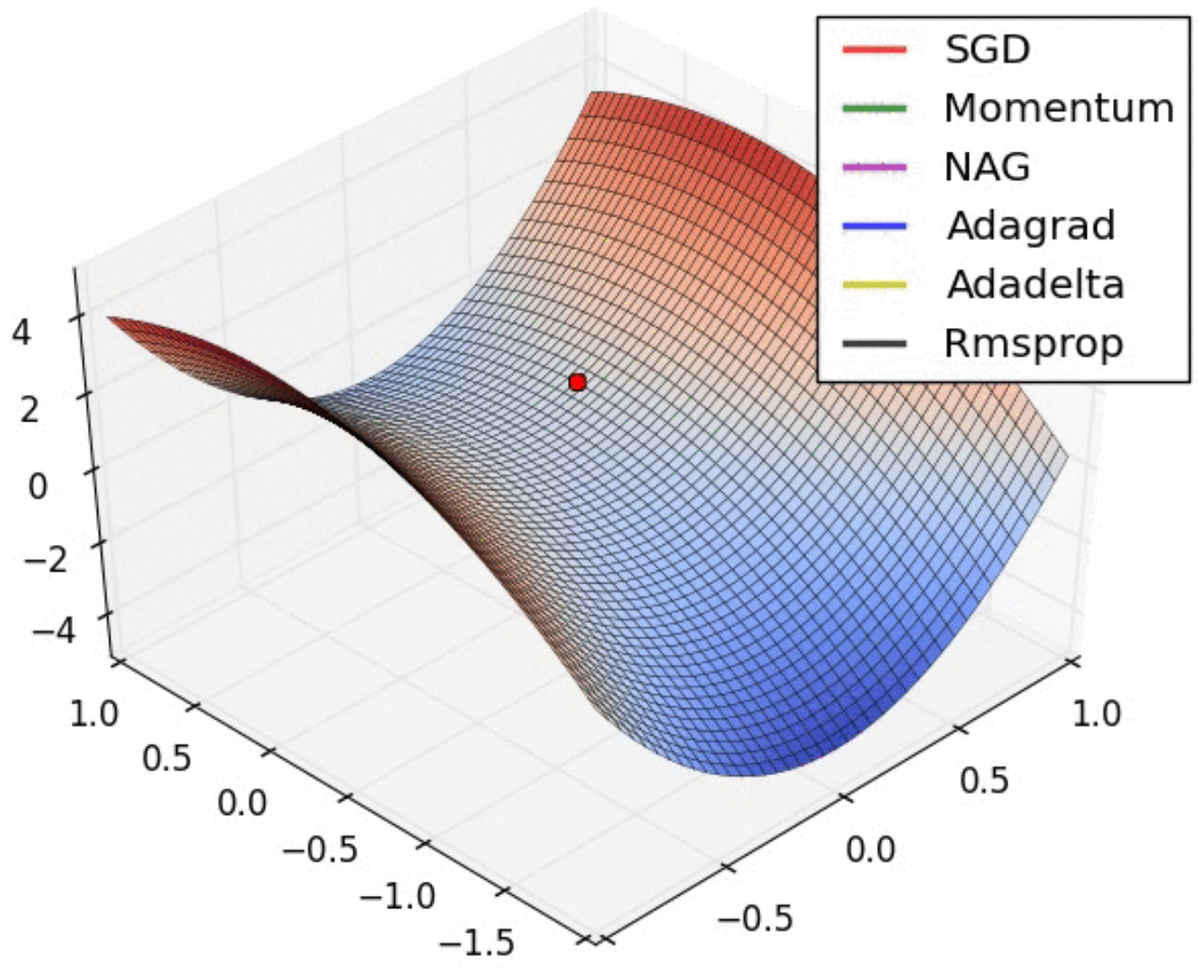




OPTIMIZATION

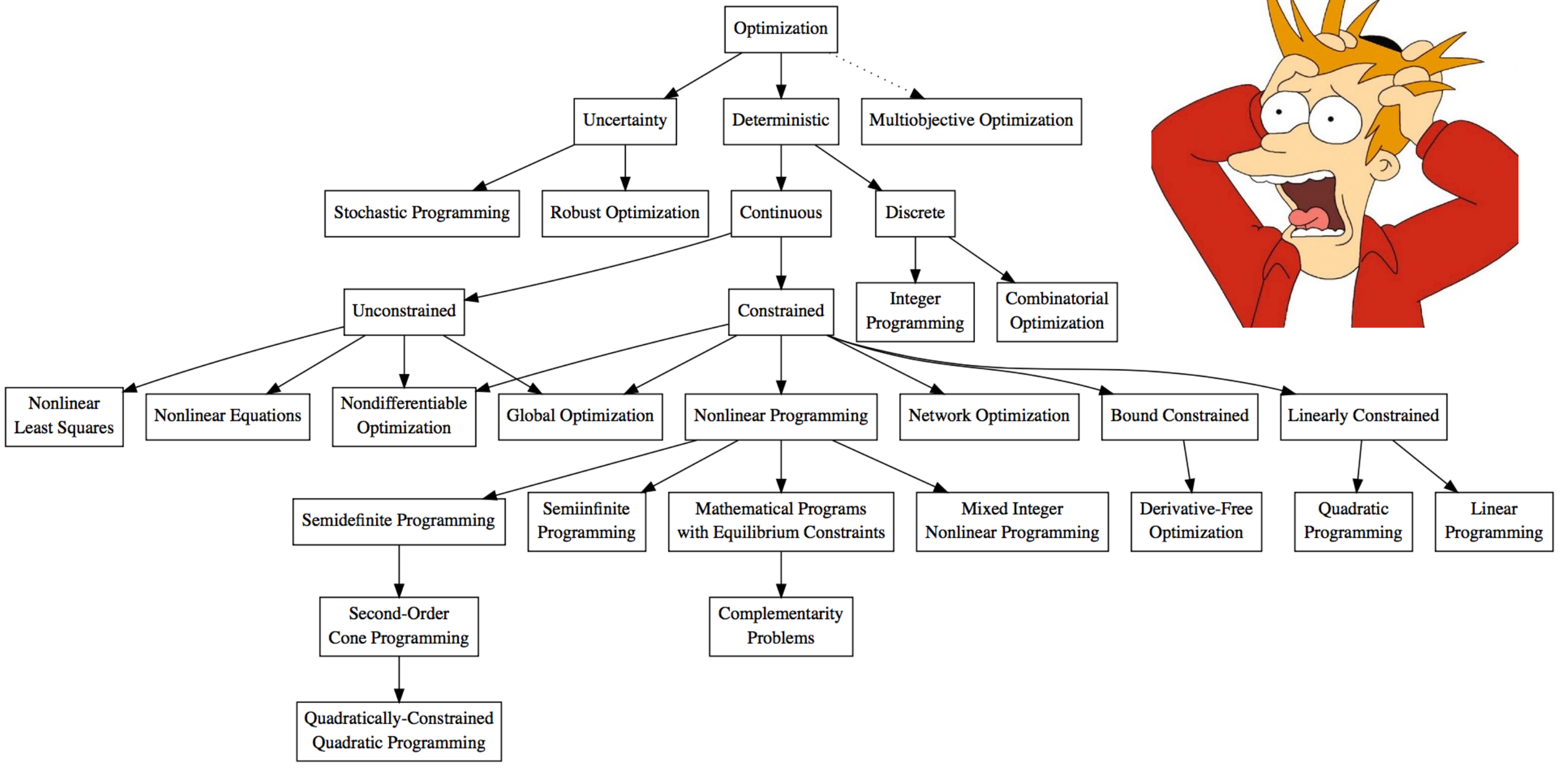
OPTIMIZATION IS “EVERYWHERE"





OPTIMIZATION

A *LOT* OF OPTIMIZATION PROBLEMS AND METHODS!



OPTIMIZATION

A *LOT* OF OPTIMIZATION PROBLEMS AND METHODS!

- **Continuous vs Combinatorial (Discrete) Optimization**
  Depending on continuous/discrete decision variables.

- **Linear vs Nonlinear Optimization**
  Depending on linear/nonlinear objective functions.

- **Single vs Multi-Objective Optimization**
  Depending on one or more (usually, 2) objective functions.

- **Constrained vs Unconstrained Optimization**
  Depending on the presence/absence of constraints (but, still subject to boundary constraints).

- **Stochastic/Dynamic vs Noiseless/Stationary Optimization**
  Depending on the presence of noise or time-dependency on any of the problem component (decision variable, objective function, constraints).

OPTIMIZATION

IT'S A HARD LIFE

- Optimization problems can be relatively "easy" to formulate, but very hard to solve, especially in complex applications with many variables. In fact, some features characterizing the problem can make it extremely challenging. For instance:
    - High **non-linearities** (optima are not on the constraint boundaries)
    - High **multimodality** (many local optima)
    - **Noisy objective function** (robust optimization is needed)
    - **Approximated objective function** (approximation errors must be accounted for)

- Computationally expensive problems
    - **Computationally expensive function** (e.g. long FEM/CFD simulations)
    - **Large-scale problems** ("needle in a haystack")
    - **Limited hardware** (drones, embedded systems, etc.)

OPTIMIZATION

ARE WE HUMANS ABLE TO OPTIMIZE?

- "Algorithm": a human being choses the points to evaluate.

- Practically considerations
  - Humans sometimes accumulate unique experience/domain knowledge
  - But:
    - Sow reaction time
    - Biases
    - We are bad at dealing with more than 1 or 2 dimensions (usually perform 1D search)

OPTIMIZATION

IT'S A HARD LIFE

That's why it's important:

1. To understand what kind of problems we are dealing with
   - What kind of objective function do we have it? Do we know its properties?
   - What kind of constraints do we have it (if any)?
   - What kind of decision variables do we have it (how many)?
   - Etc.
2. To choose the right optimization algorithm for that problem, e.g.:
   - Gradient-based or gradient-free?
   - Local or global search?
   - Single or multi-objective?
   - Constraint or unconstrained?
   - Do we have computational constraints?
   - Etc.

OPTIMIZATION

A WARNING

- No free lunch theorem by Wolpert and Macready (1997).

- Loosely speaking: there is no reason to prefer one algorithm over another, unless we know something regarding the probability distribution over the space of possible objective functions.

- In particular, if one algorithm performs better than another on one class of problems, it will perform worse on another class of problems.

- More rigorously, for a given pair of algorithms A and B:

$$\sum_f P(x_m|f, A) = \sum_f P(x_m|f, B)$$

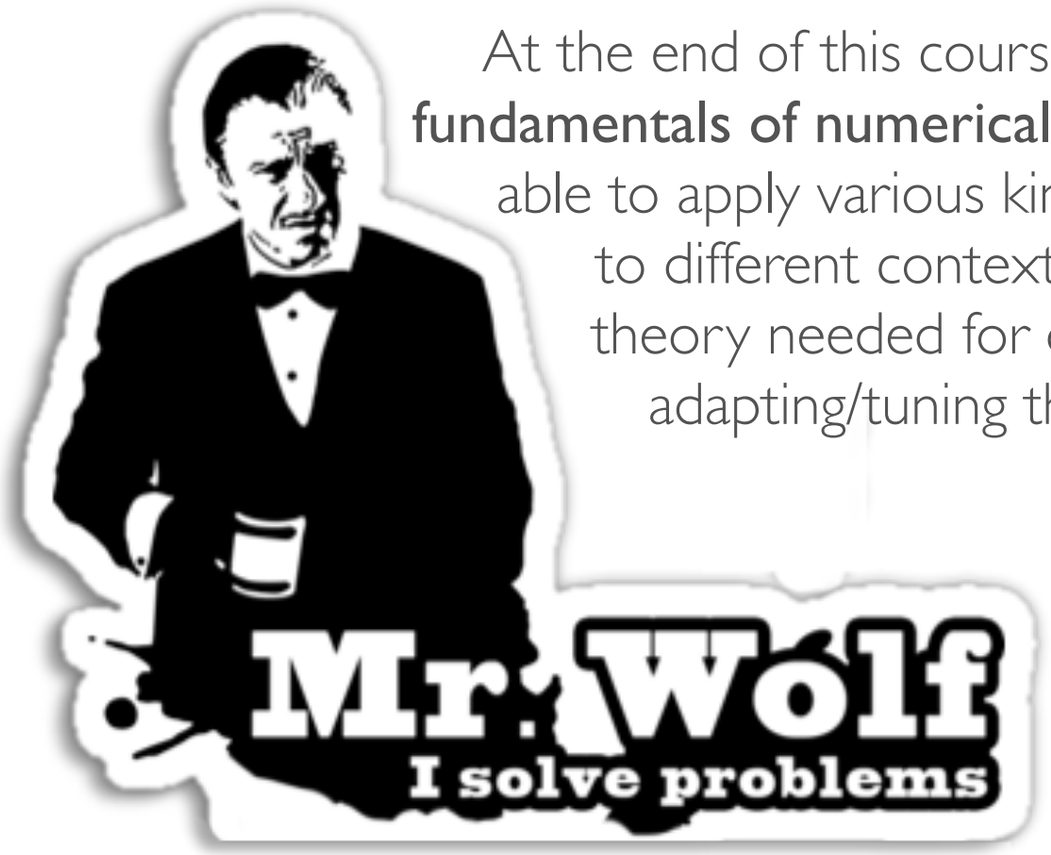
where $P(x_m|f,A)$ is the probability that algorithm A detects the optimal solution x_m for a generic objective function f and $P(x_m|f,B)$ is the analogue probability for algorithm B.

- The performance of every pair of algorithms over all possible problems is the same.

GOAL OF THIS COURSE

FROM THE SYLLABUS

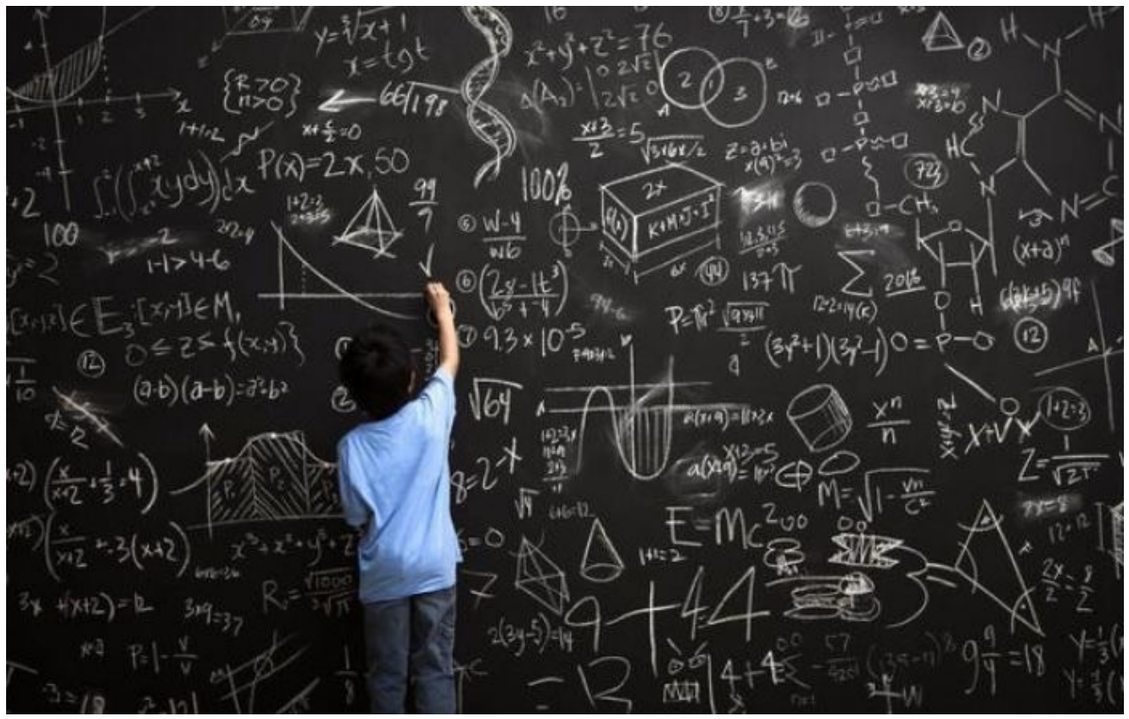
The goal of this course is to provide the **fundamentals of numerical optimization.** We will focus on **various kinds of optimization techniques**, ranging from continuous to discrete optimization methods, including global and local search methods, derivative-based and derivative-free algorithms, as well as traditional operational research techniques such as those used for solving integer programming, linear programming, and quadratic programming problems. We will address both **theoretical and practical considerations, complementing each lecture on the theory with a dedicated lab**.
During the labs, you will have the opportunity to test the various techniques on synthetic and real-world optimization problems.



At the end of this course, you will be familiar with the **fundamentals of numerical optimization**, and you will be able to apply various kinds of optimization algorithms to different contexts. You will also know the basic theory needed for developing new algorithms, or adapting/tuning them to handle new problems.

IMPORTANT NOTE
Numerical optimization is simply too broad to be studied in one single course… we will just "scratch the surface"!

Course organization

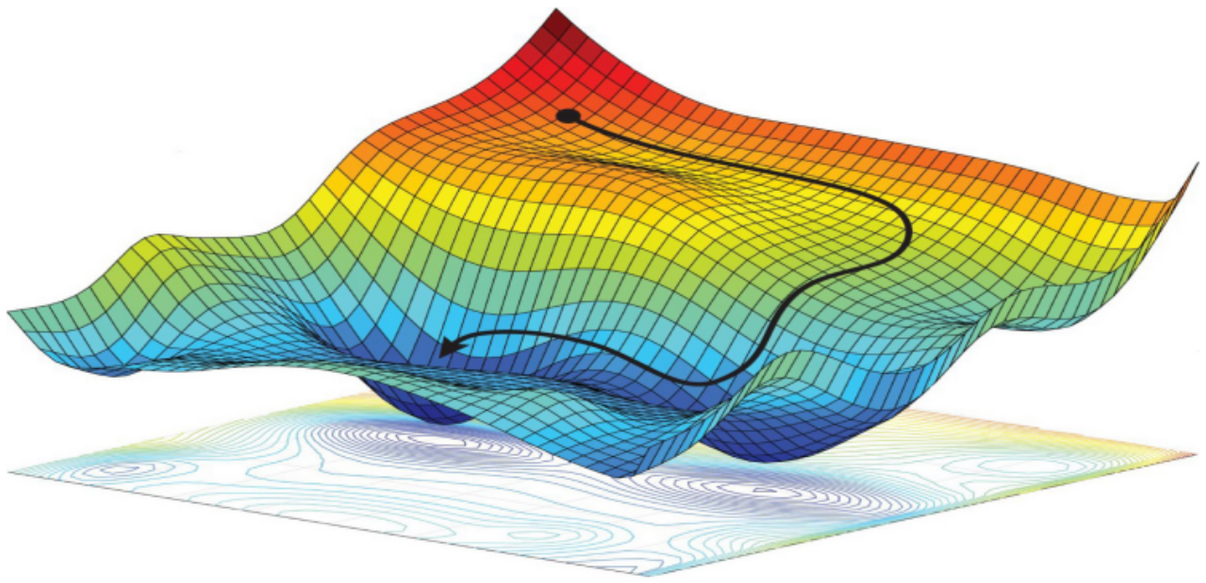


COURSE ORGANIZATION

TEACHING MODALITIES

- Standard classes
  - Slides
  - Additional resources online

- Labs
  - Experiments with existing Python libraries
  - Some programming skills will be required
  - Some statistics / maths background will be useful to analyze the results

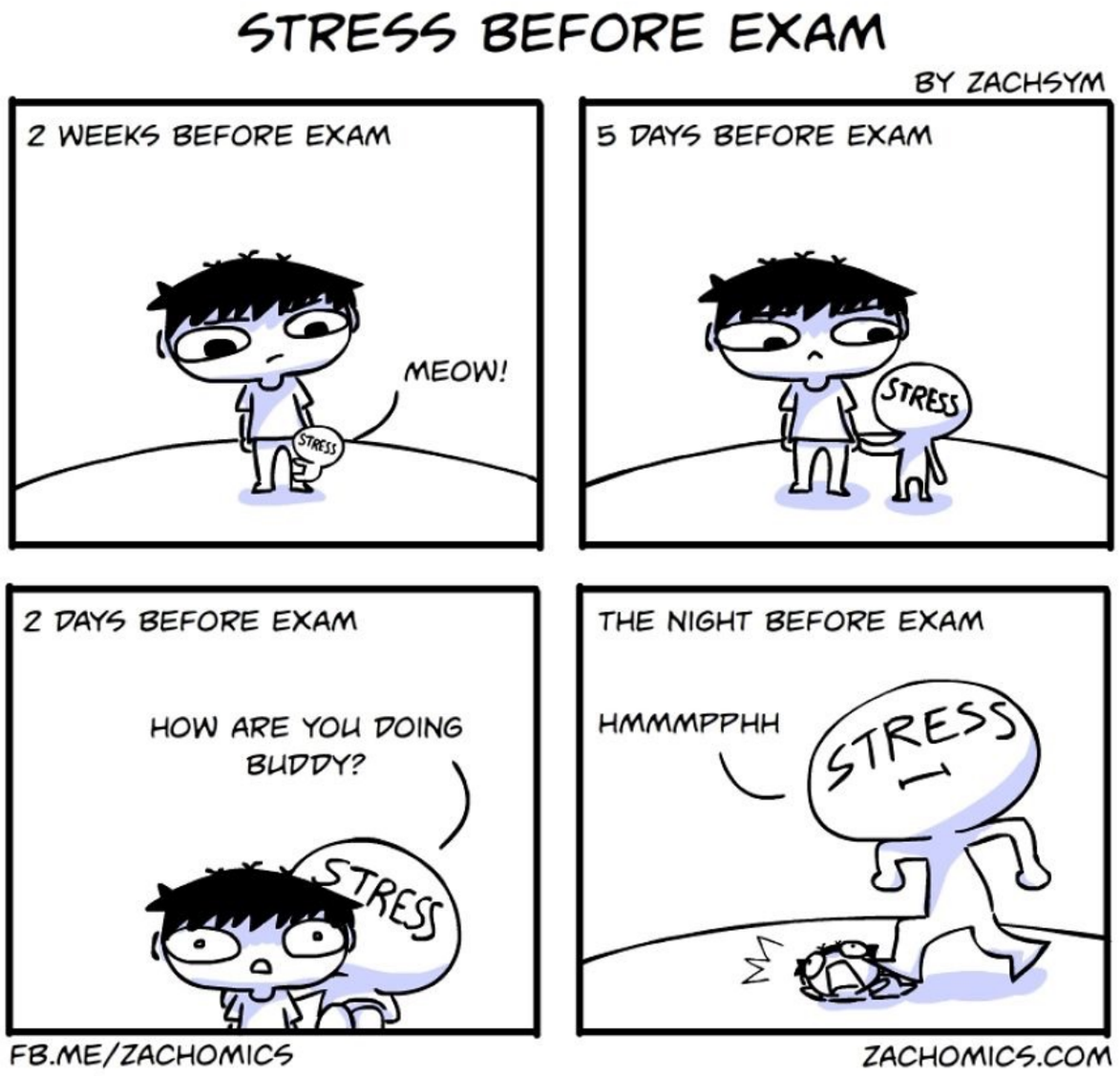


COURSE ORGANIZATION

EVALUATION MODALITIES

- Laboratory reports
  Following each laboratory, a short individual report (free format, e.g. ~1-2 pages notes, Python notebook etc.) must be prepared. While these reports must not be submitted before the oral exam, they are required to pass it.

- Oral exam
  During the oral exam, the laboratory reports as well as the theoretical topics of the course will be discussed. The final mark is calculated based on the mark obtained on the part of the oral exam on the theory (50%) and the quality of the lab reports (50%). The mark is given on a 30-point scale.



COURSE ORGANIZATION

PRACTICAL INFORMATION 2023-2024

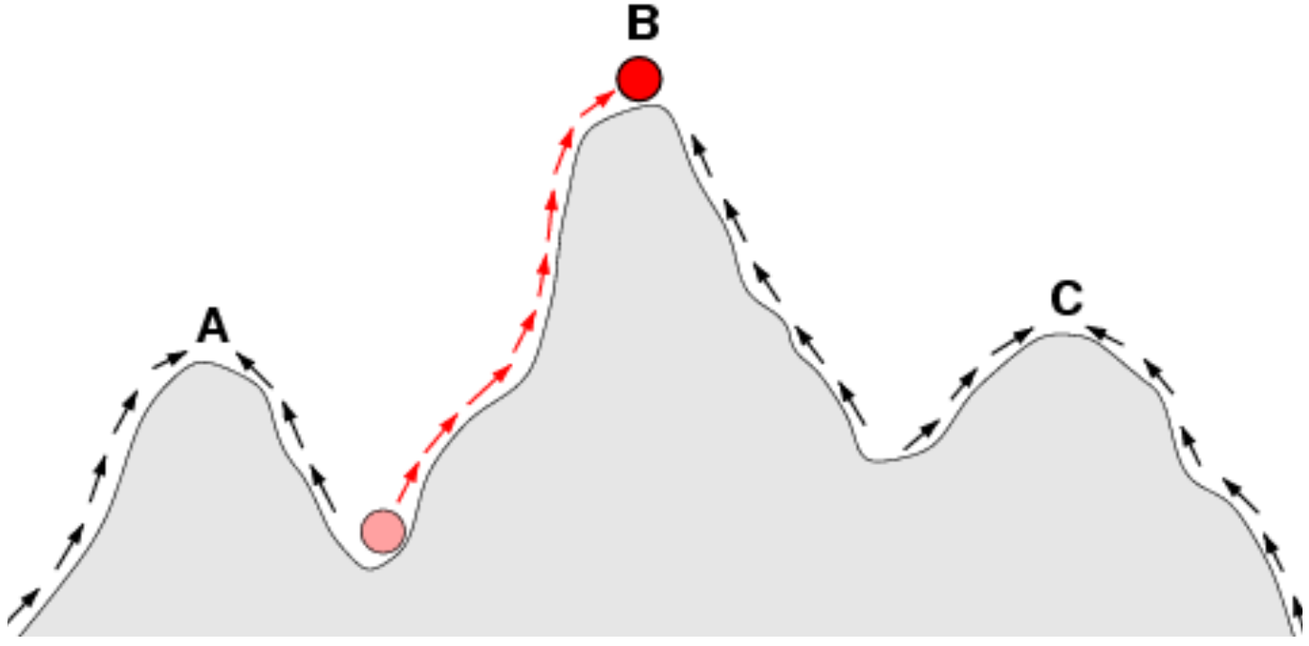
- 6 CFUs
- Lectures: theory every Mon at 9:30-11:30 (B112), labs every Fri at 9:30-11:30 (A202)
- Prerequisites: Linear algebra, calculus, basic knowledge of statistics and Python programming
- Recommended books:
  https://openmdao.org/a-new-free-text-book-on-mdo/
  https://intelligent-optimization.org/LIONbook/
- Instructor
  Prof. Giovanni Iacca, email: giovanni.iacca@unitn.it
  Web: https://sites.google.com/site/giovanniiacca/
  Office: DISI 121 (please send me an email if you need a meeting)
- Teaching assistants (for the labs)
  - Elia Cunegatti, email: elia.cunegatti@unitn.it
  - Chiara Camilla Rambaldi Migliore, email: cc.rambaldimigliore@unitn.it
  - Mátyás Vincze, email: mvincze@fbk.eu

COURSE ORGANIZATION

TOPICS

1. Intro + Grid search, random search, and local search (2h lecture + 2h lab)
2. Deterministic/stochastic global optimization (2h lecture + 2h lab)
3. Derivative-based optimization (2h lecture + 2h lab)
4. Variable neighborhood search (2h lecture + 2h lab)
5. Iterated local search and simulated annealing (2h lecture + 2h lab)
6. Bayesian optimization (2h lecture + 2h lab)
7. Genetics, evolution and nature-inspired analogies (2h lecture + 2h lab)
8. Multi-objective optimization (2h lecture + 2h lab)
9. Design of experiments (2h lecture + 2h lab)
10. Linear and quadratic programming (2h lecture + 2h lab)
11. Integer and dynamic programming (2h lecture + 2h lab)
12. Robust optimization (2h lecture + 2h lab)

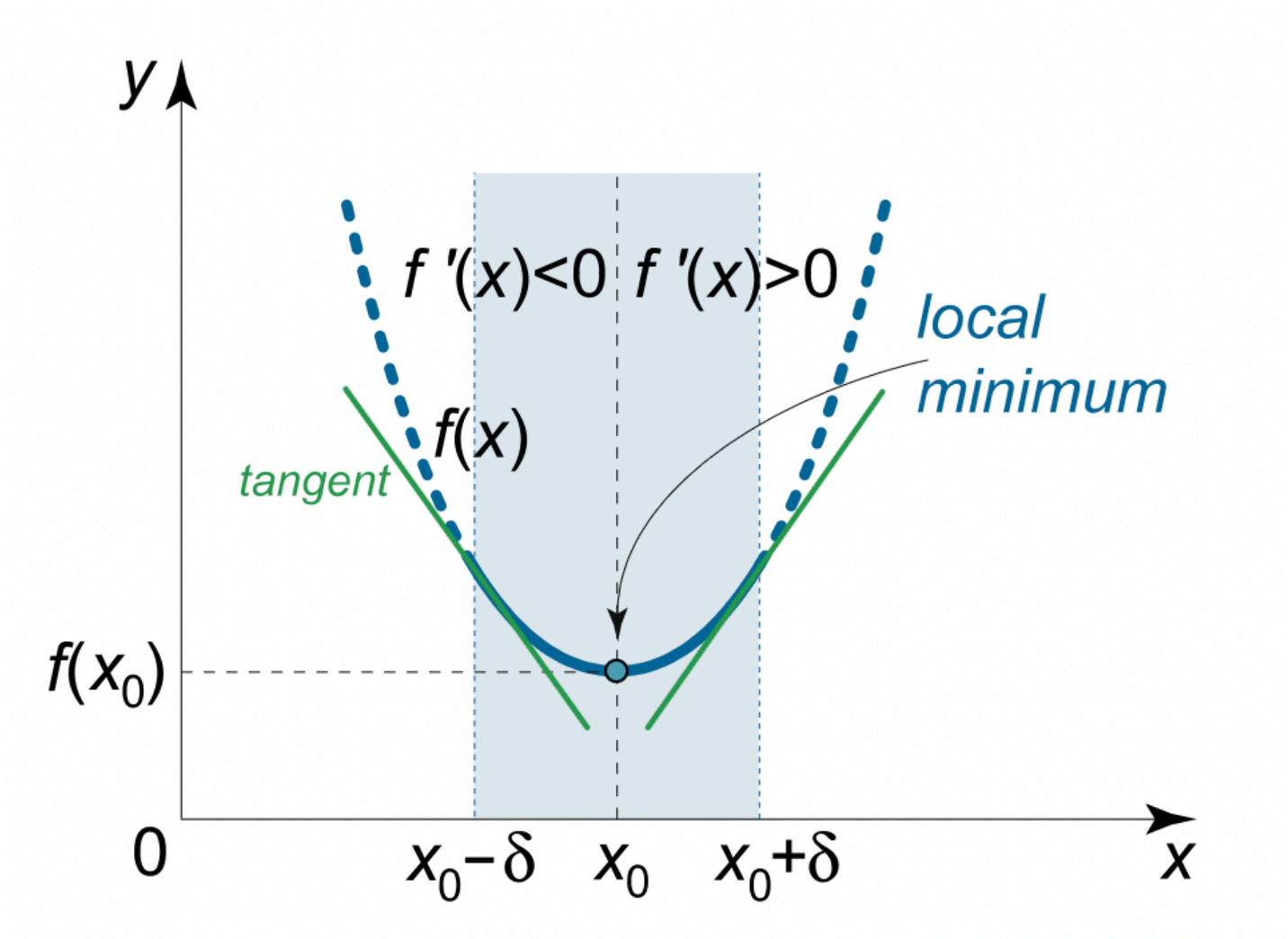
Let’s begin!



BASIC OPTIMIZATION ALGORITHMS

CLASSIC OPTIMIZATION APPROACHES

- **Analytical approach**: the function has an explicit analytical expression, derivable over all the variables (and, de facto, not highly multivariate) → **Calculus**



Example

$f(x) = (x-x0)^2$

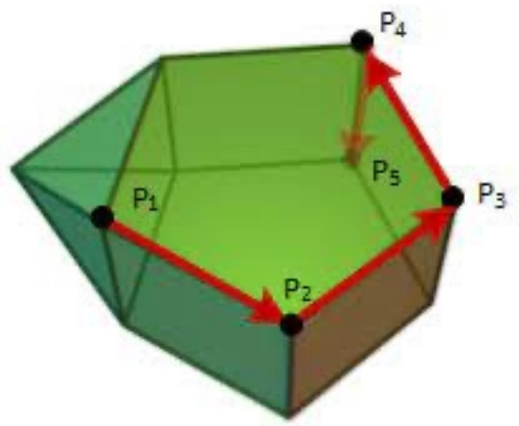
$f'(x) = 2(x-x0)$

$f'(x) = 0 \rightarrow x_{min}=x0$

BASIC OPTIMIZATION ALGORITHMS

CLASSIC OPTIMIZATION APPROACHES

- **Exact methods**: the function respects some specific hypotheses, e.g. it's a linear or quadratic problem. The method converges to the exact solution after a finite amount of steps of an iterative procedure.
  For instance, the simplex algorithm for linear programming *(we'll see it later in this course)*.

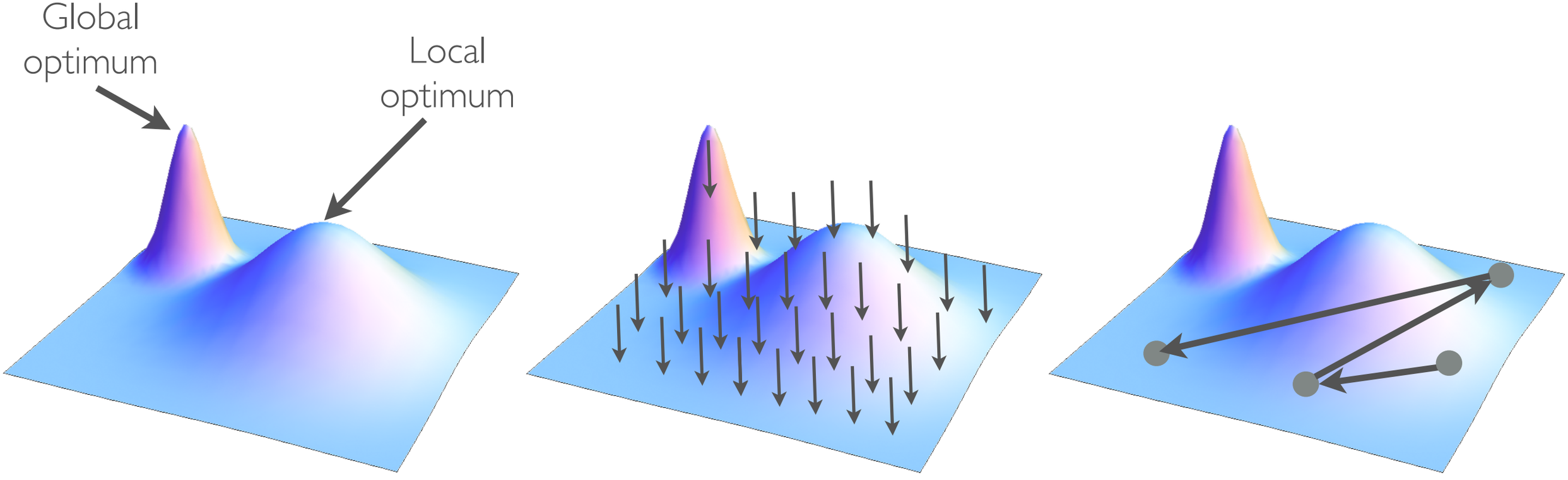


- **Approximate iterative methods**: the function respects some hypotheses and can be solved by applying an iterative procedure with an infinite number of steps. The application of the procedure for a finite amount of steps still leads to an approximation of the optimum.
  For instance, some kinds of gradient descent algorithms on some specific problems *(we'll see it later in this course)*.

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. *(we'll see them next week)*



BASIC OPTIMIZATION ALGORITHMS

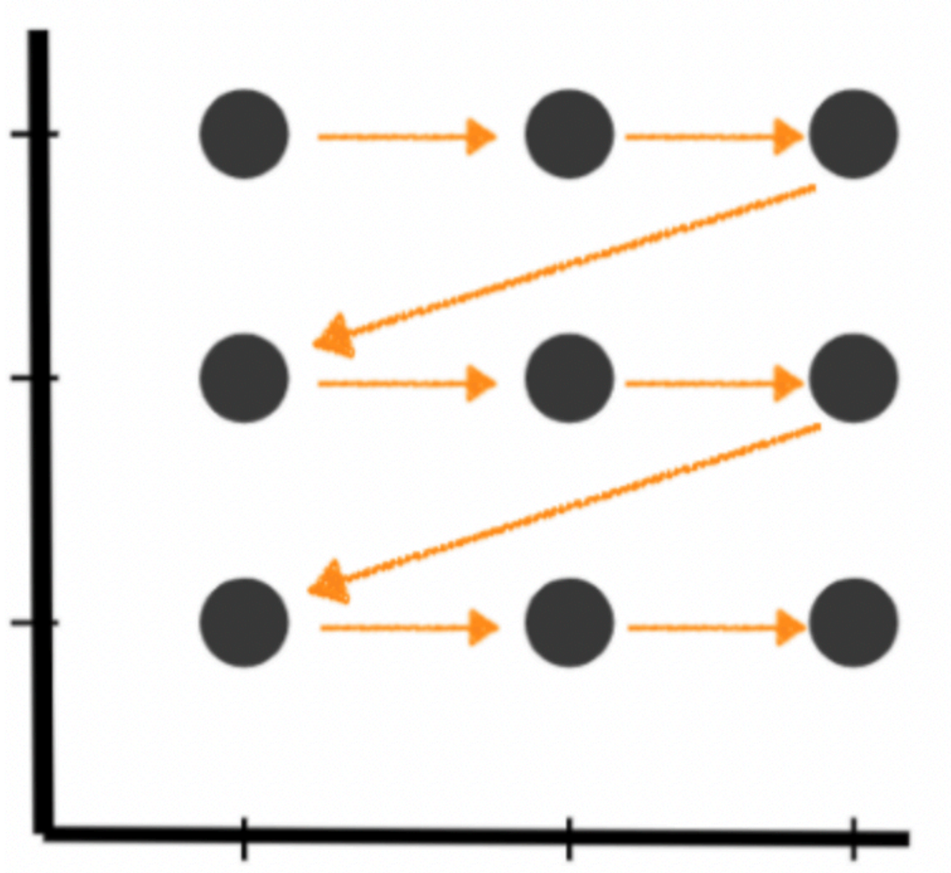
GRID SEARCH

- Typical example: finding the best set of hyper-parameters for a given Machine Learning model (e.g., a deep neural network) e.g. to optimize its accuracy.

- E.g., 2 hyper-parameters:
  learn_rate: [0.001, 0.01, 0.05]
  max_depth: [4, 6, 8, 10]
  → 12 combinations

- E.g., 4 hyper-parameters:
  learn_rate: [0.001, 0.01, 0.1, 0.2, 0.3, 0.4, 0.5],
  max_depth = [4, 6, 8, 10, 12, 15, 20, 25, 30]
  dropout_rate = [0.1, 0.15, 0.2, 0.25, 0.3]
  activation = ['tanh', 'relu']
  → 630 combinations

- Evaluations can be parallelized



BASIC OPTIMIZATION ALGORITHMS

GRID SEARCH

Problems

- (Exhaustive) grid search can be very expensive
  - Combinatorial explosion! The more parameters we have, the larger the grid.
  - What if every solution in the grid is computationally expensive to evaluate? E.g., every ML model should be trained and (cross)validated.

- What about continuous parameters?
  - We may easily miss the optimum!
  - We need to discretize the parameters (important to set the right tolerance).
  - Do we really know the right boundaries?

- Grid search is "uninformed": it does not use information from the search process to move towards the most promising search directions.

BASIC OPTIMIZATION ALGORITHMS

RANDOM SEARCH

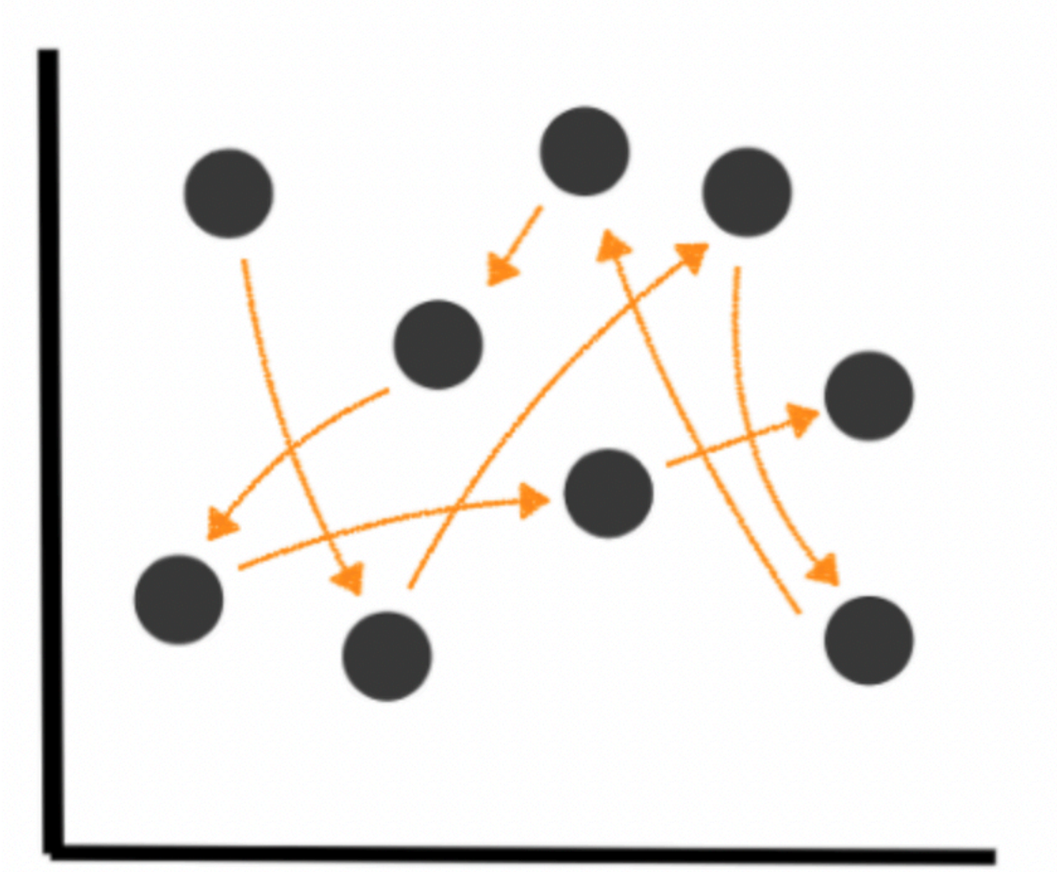
Main differences w.r.t. grid search

- You are unlikely to keep completely missing the "good area" for a long time when randomly picking new spots.

- A grid search may spend lots of time in a "bad area" as it covers exhaustively (however, this may happen also in random search).

- A sampling methodology is needed (e.g. uniform).

As for grid search:
- Also random search is "uninformed"
- Evaluations can be parallelized

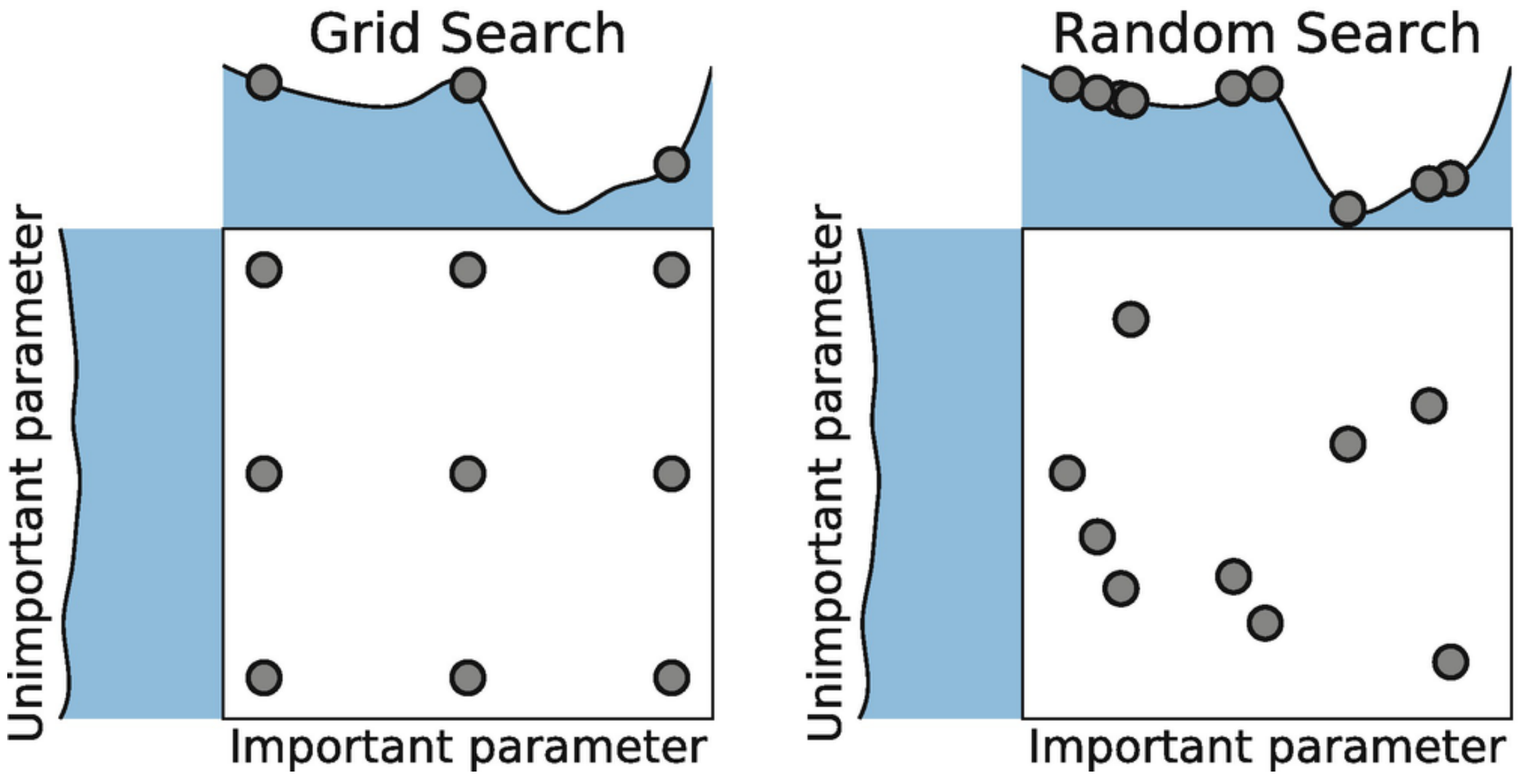


NOTE
For a fair comparison between algorithms (in this case random vs grid search), the same "budget", i.e., number of evaluated solutions, should be allotted.

BASIC OPTIMIZATION ALGORITHMS

RANDOM SEARCH

Main differences w.r.t. grid search



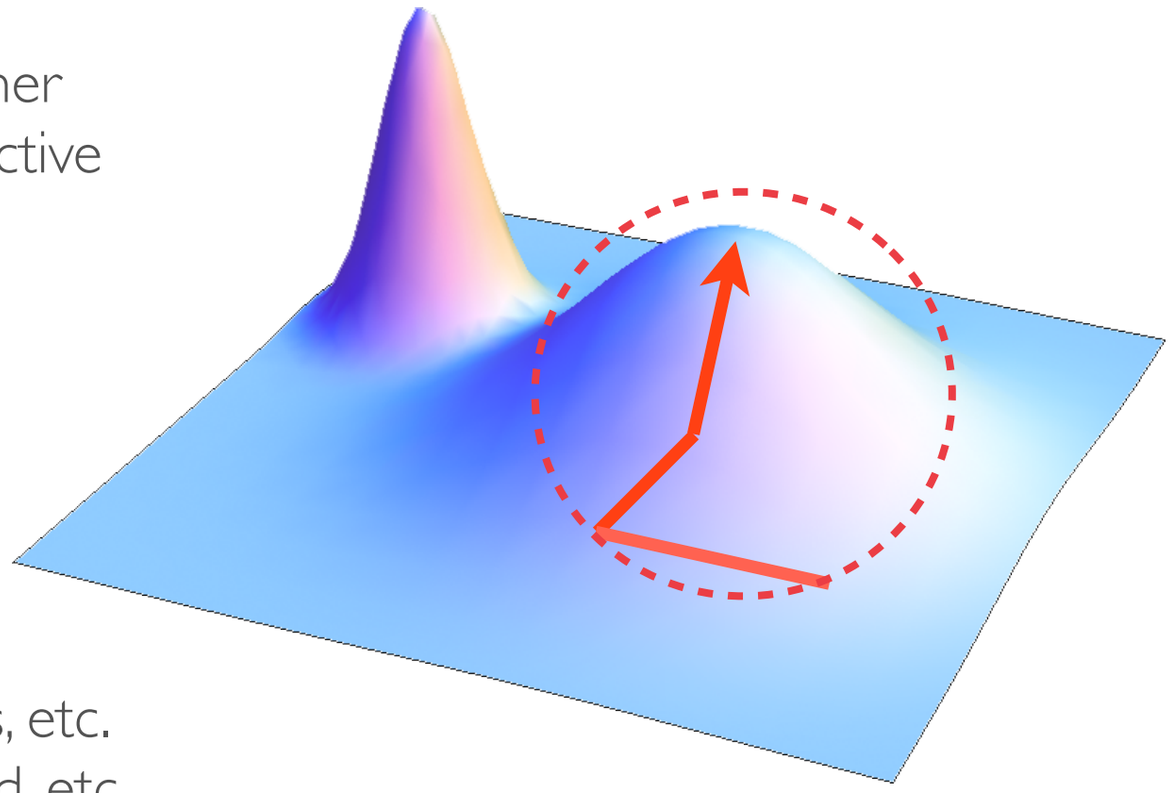
DEMO

BASIC OPTIMIZATION ALGORITHMS

LOCAL OPTIMIZATION: FIND THE LOCAL OPTIMUM

Classic (gradient-based) methods: use gradient or higher level derivates (or approximations thereof) of the objective function *(we'll see them later in this course)*:

- Gradient (steepest) descent
- Newton methods
- Quasi-Newton methods

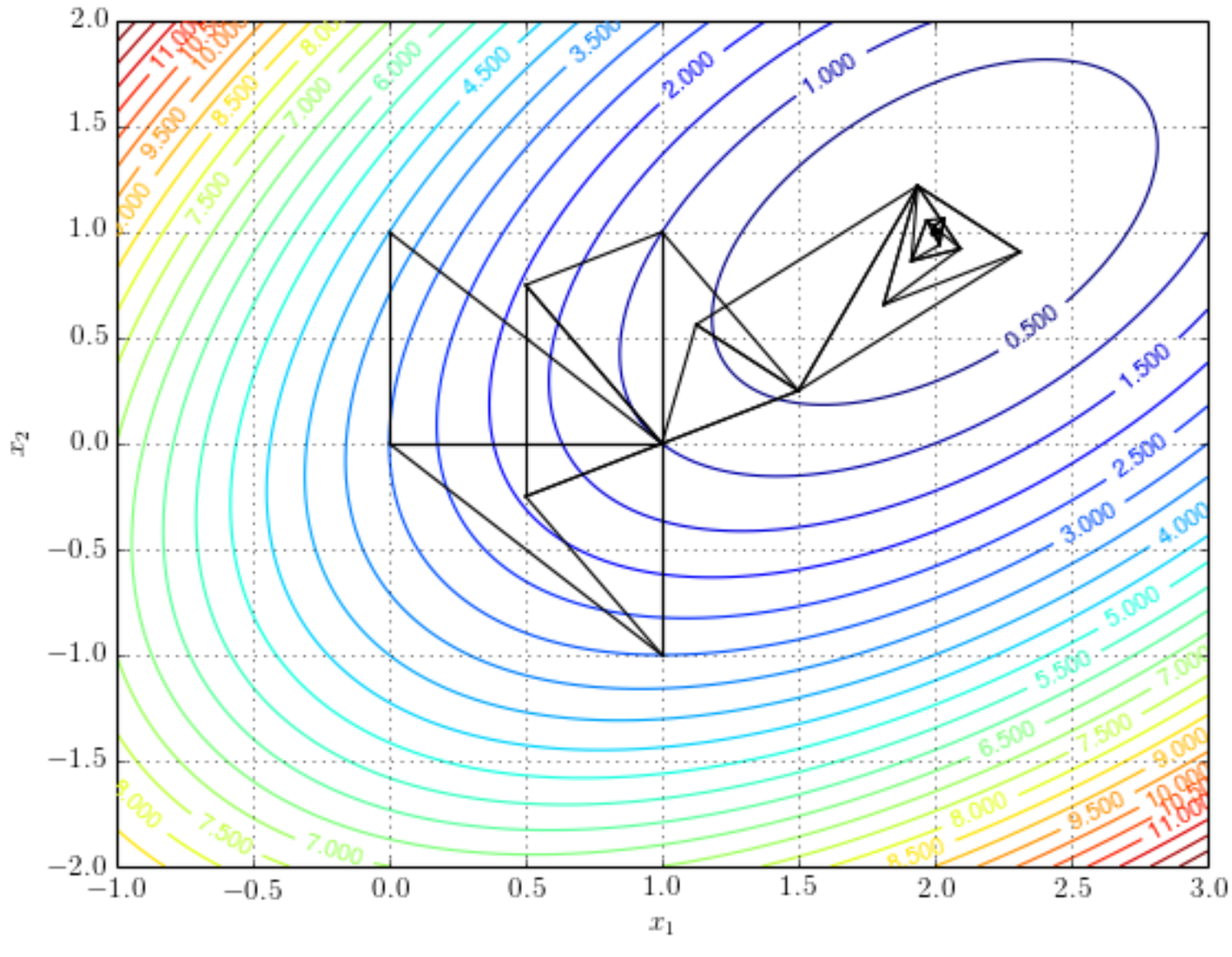


Gradient-free methods: do not use derivatives, but a **heuristic** method:

- Nelder-Mead, Powell, Rosenbrock, Hooke-Jeeves, etc.
- For 1-D problems: golden search, Brent's method, etc.

Heuristic (from the Greek ευρισκω heuriskō "I find, I discover"), also called "direct", "pattern search", or "generate and test" methods, are techniques designed for solving a problem (finding an approximate solution) when classic methods fail to find any exact solution, or are too slow to do that. This is achieved by trading optimality, completeness, accuracy, or precision for speed. Usually they are **greedy algorithms**.

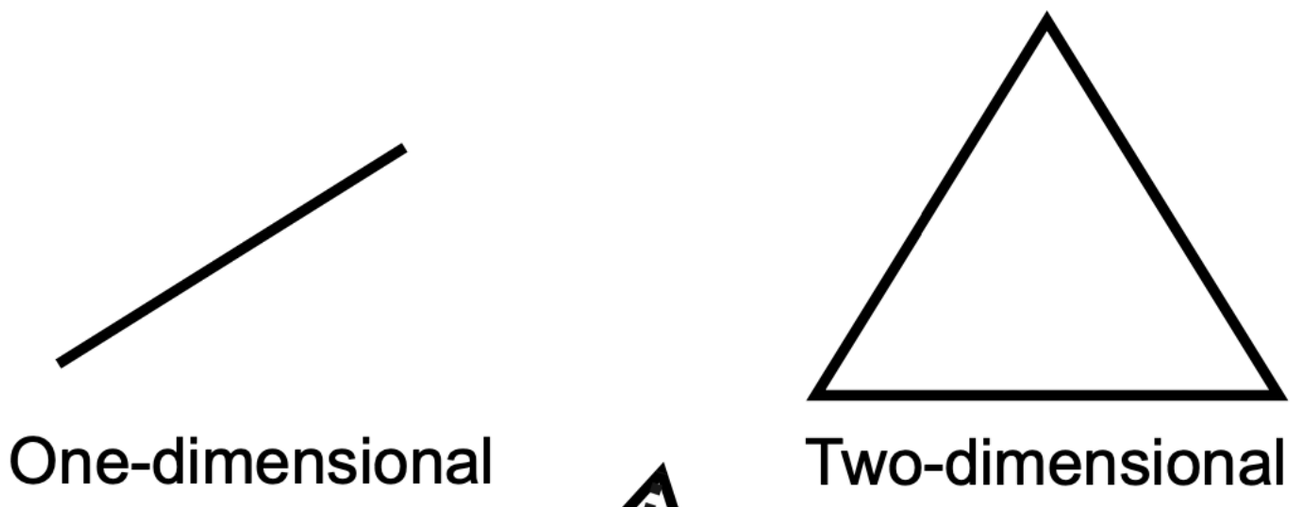
Nelder-Mead & Powell's methods

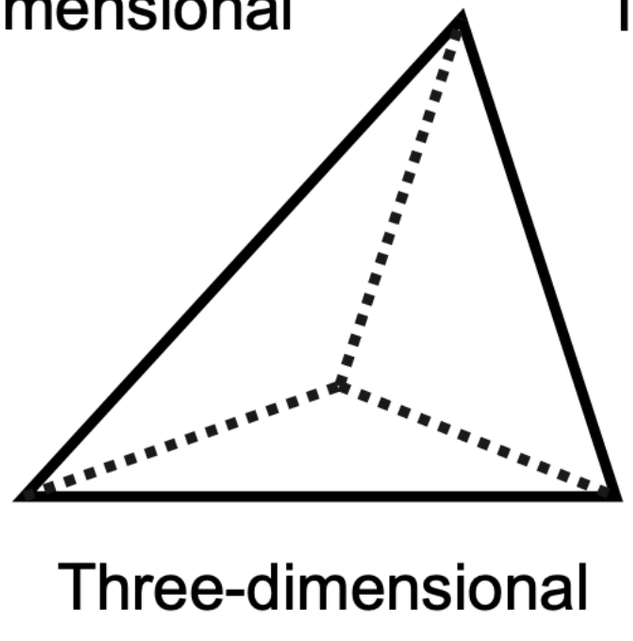


NELDER-MEAD SIMPLEX

DOWNHILL SIMPLEX METHOD BY NELDER AND MEAD (1965)

- Given a problem in N dimensions, start with N+1 points → Initial simplex.
  Simplex: Geometrical figure consisting, in N dimensions, of N+1 points (or vertices) and all their interconnecting line segments, polygonal faces, etc. (N = 2 → triangle, N = 3 → tetrahedron, etc.).



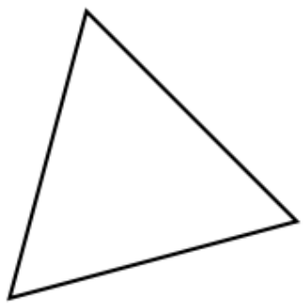


NOTE
Not to be confused with the simplex method for linear programming!

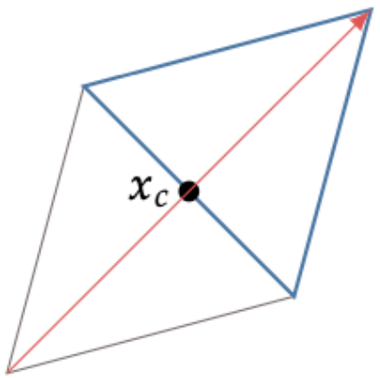
NELDER-MEAD SIMPLEX

THE ALGORITHM

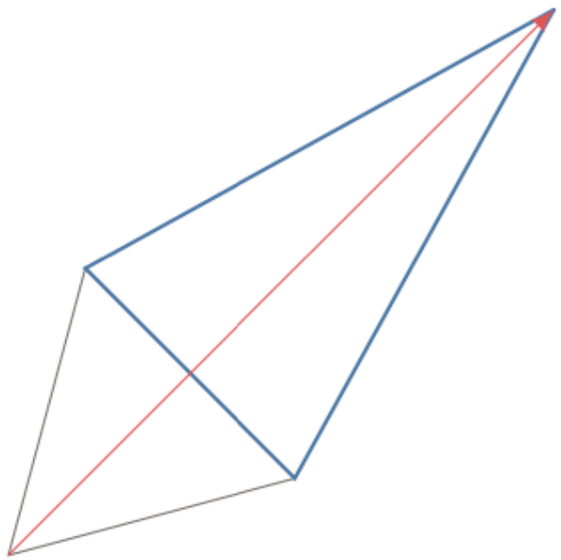
Main equation: $x = x_c + \alpha\left(x_c - x^{(n)}\right)$



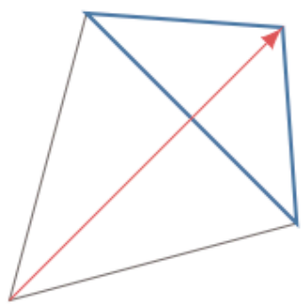
Initial simplex



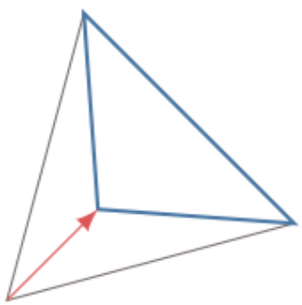
Reflection ($\alpha = 1$)



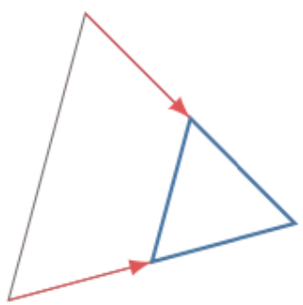
Expansion ($\alpha = 2$)



Outside contraction
($\alpha = 0.5$)



Inside contraction
($\alpha = -0.5$)



Shrink

NELDER-MEAD SIMPLEX

THE ALGORITHM

Inputs:

$x^{(0)}$: Starting point

τ_x: Simplex size tolerances

τ_f: Function value standard deviation tolerances

Outputs:

x^*: Optimal point

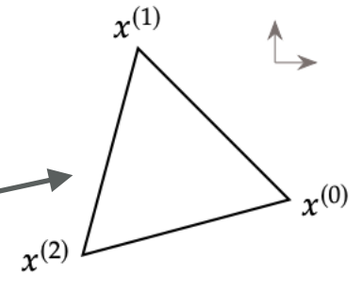
---

for $j = 1$ **to** n **do**
  $x^{(j)} = x^{(0)} + s^{(j)}$
end for

Create a simplex with edge length l
$s^{(j)}$ given by Eq. 7.2

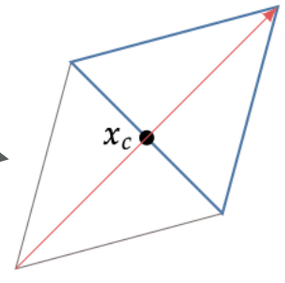
$$x^{(i)} = x^{(0)} + s^{(i)}, \tag{7.1}$$

$$s_j^{(i)} = \begin{cases} \frac{l}{n\sqrt{2}}\left(\sqrt{n+1}-1\right) + \frac{l}{\sqrt{2}}, & \text{if } j = i \\ \frac{l}{n\sqrt{2}}\left(\sqrt{n+1}-1\right), & \text{if } j \neq i\,. \end{cases} \tag{7.2}$$



while $\Delta_x > \tau_x$ or $\Delta_f > \tau_f$ **do**
  Sort $\left\{x^{(0)}, \ldots, x^{(n-1)}, x^{(n)}\right\}$
  $x_c = \frac{1}{n}\sum_{i=0}^{n-1} x^{(i)}$
  $x_r = x_c + \left(x_c - x^{(n)}\right)$

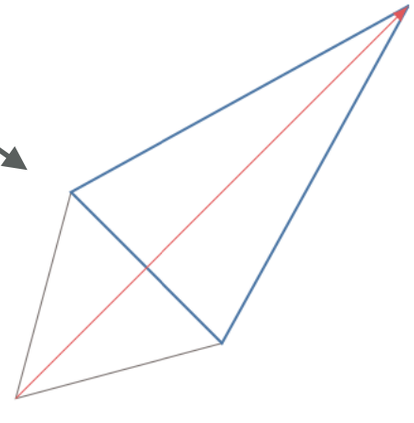
Simplex size (Eq. 7.6) and standard deviation (Eq. 7.7)
Order from the lowest (best) to the highest $f(x^{(j)})$
The centroid excluding the worst point $x^{(n)}$ (Eq. 7.4)
Reflection, Eq. 7.3 with $\alpha = 1$



  if $f(x_r) < f(x^{(0)})$ **then**
    $x_e = x_c + 2\left(x_c - x^{(n)}\right)$
    if $f(x_e) < f(x^{(0)})$ **then**
      $x^{(n)} = x_e$
    else
      $x^{(n)} = x_r$
    end if

Is reflected point is better than the best?
Expansion, Eq. 7.3 with $\alpha = 2$
Is expanded point better than the best?
Accept expansion and replace worst point

Accept reflection



NELDER-MEAD SIMPLEX

THE ALGORITHM

else if $f(x_r) \leq f(x^{(n-1)})$ **then**
  $x^{(n)} = x_r$
else
  if $f(x_r) > f(x^{(n)})$ **then**
    $x_{ic} = x_c - 0.5\left(x_c - x^{(n)}\right)$
    if $f(x_{ic}) < f(x^{(n)})$ **then**
      $x^{(n)} = x_{ic}$
    else
      for $j = 1$ **to** n **do**
        $x^{(j)} = x^{(0)} + 0.5\left(x^{(j)} - x^{(0)}\right)$
      end for
    end if
  else
    $x_{oc} = x_c + 0.5\left(x_c - x^{(n)}\right)$
    if $f(x_{oc}) < f(x_r)$ **then**
      $x^{(n)} = x_{oc}$
    else
      for $j = 1$ **to** n **do**
        $x^{(j)} = x^{(0)} + 0.5\left(x^{(j)} - x^{(0)}\right)$
      end for
    end if
  end if
end if
end while

Is reflected better than second worst?
Accept reflected point

Is reflected point worse than the worst?
Inside contraction, Eq. 7.3 with $\alpha = -0.5$

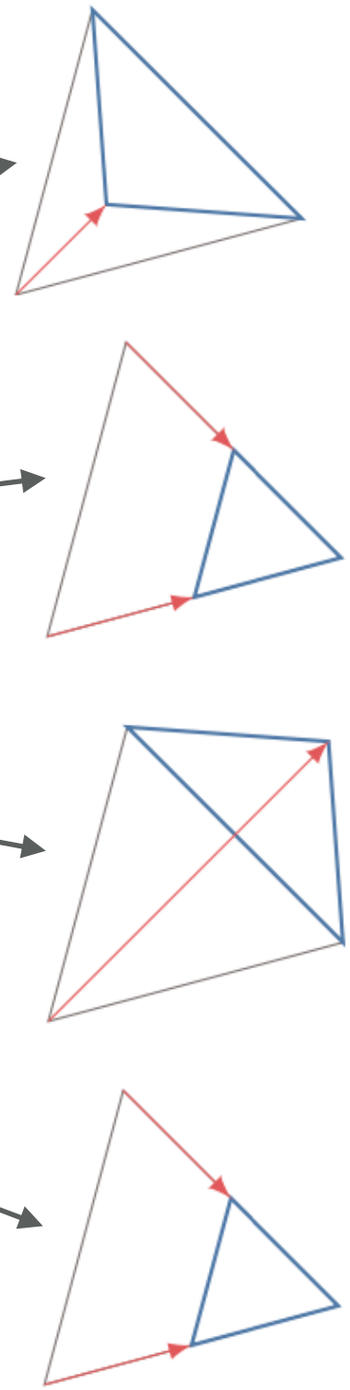
Inside contraction better than worst?
Accept inside contraction

Shrink, Eq. 7.5 with $\gamma = 0.5$

Outside contraction, Eq. 7.3 with $\alpha = 0.5$

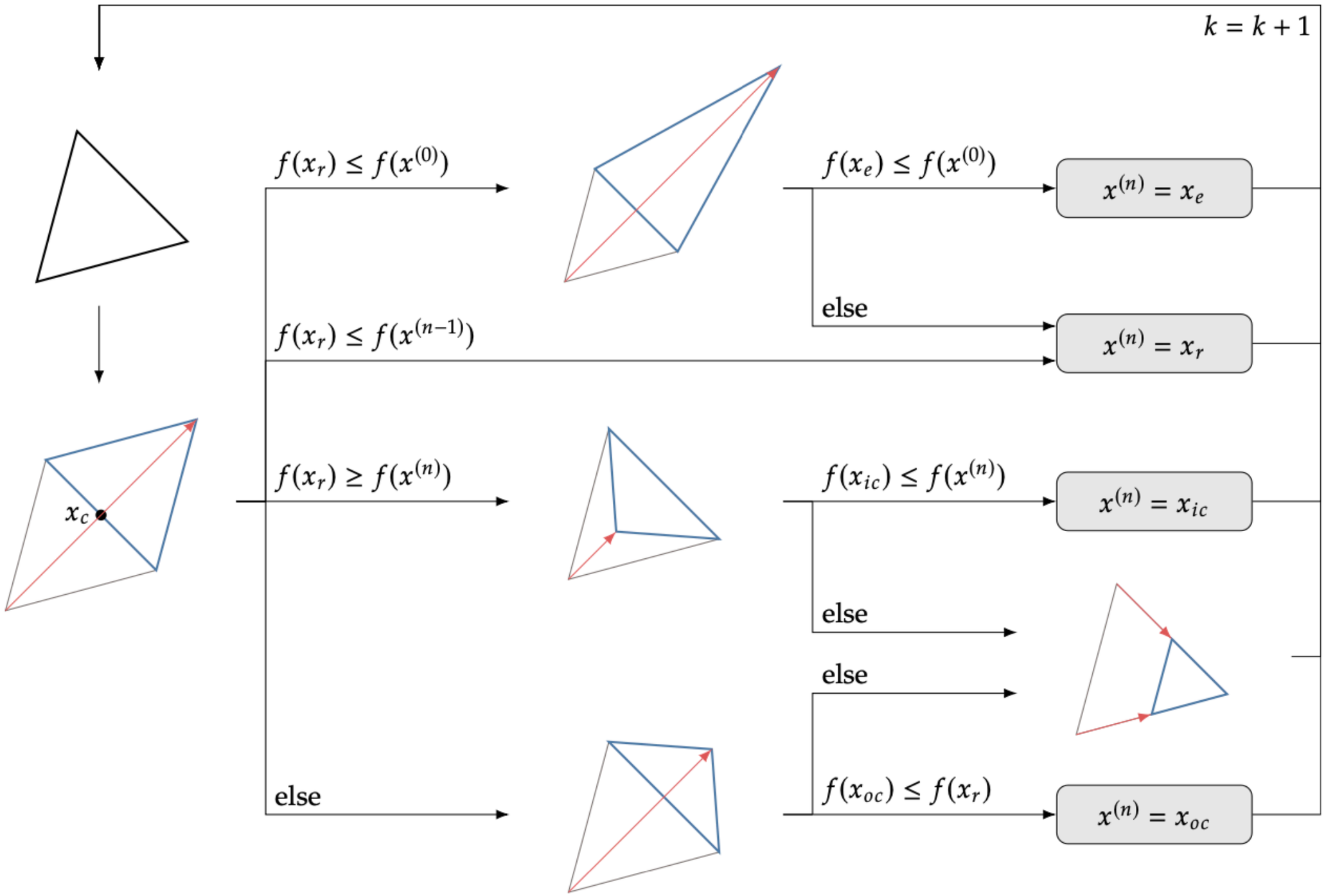
Is contraction better than reflection?
Accept outside contraction

Shrink, Eq. 7.5 with $\gamma = 0.5$



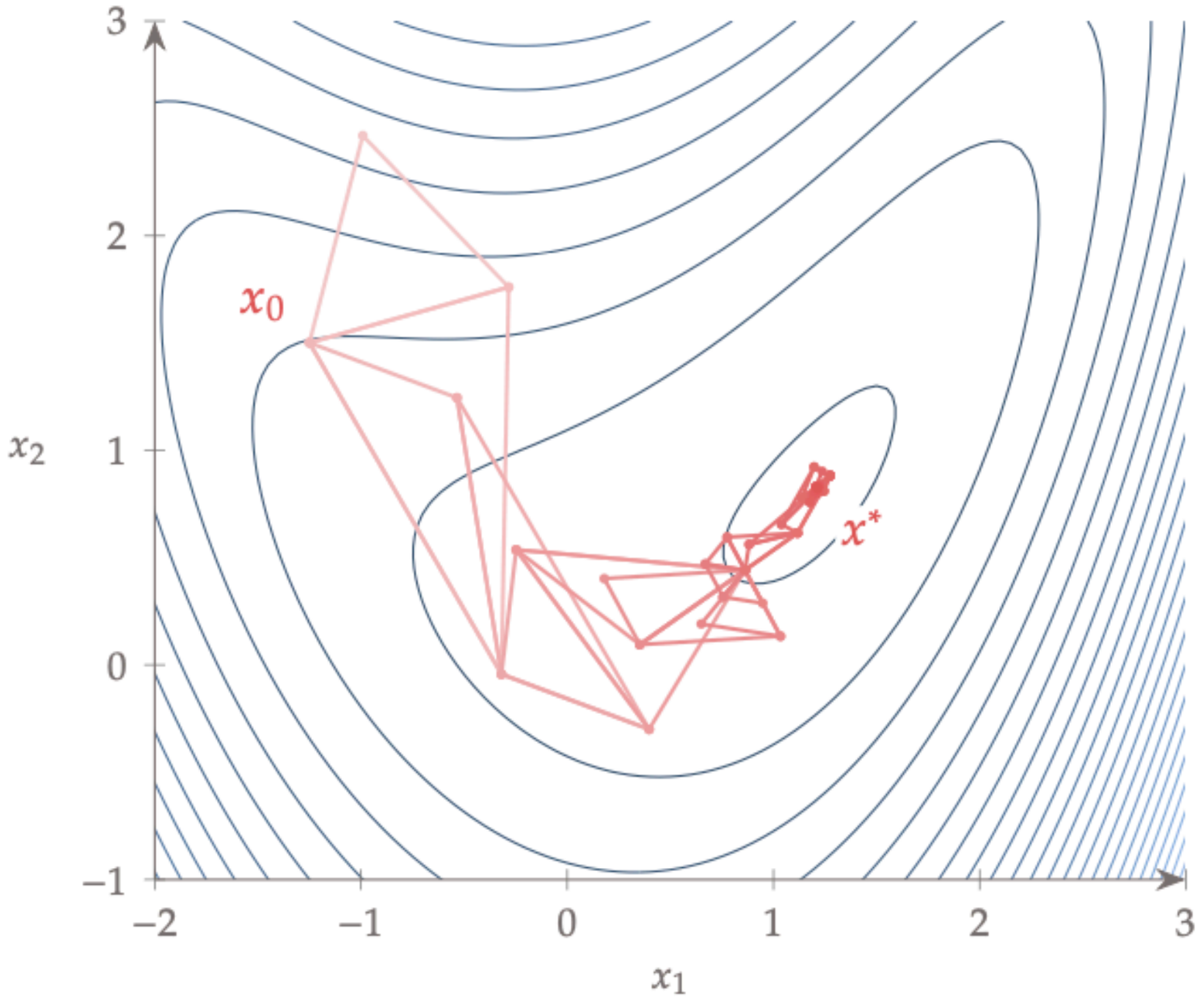
NELDER-MEAD SIMPLEX

THE ALGORITHM



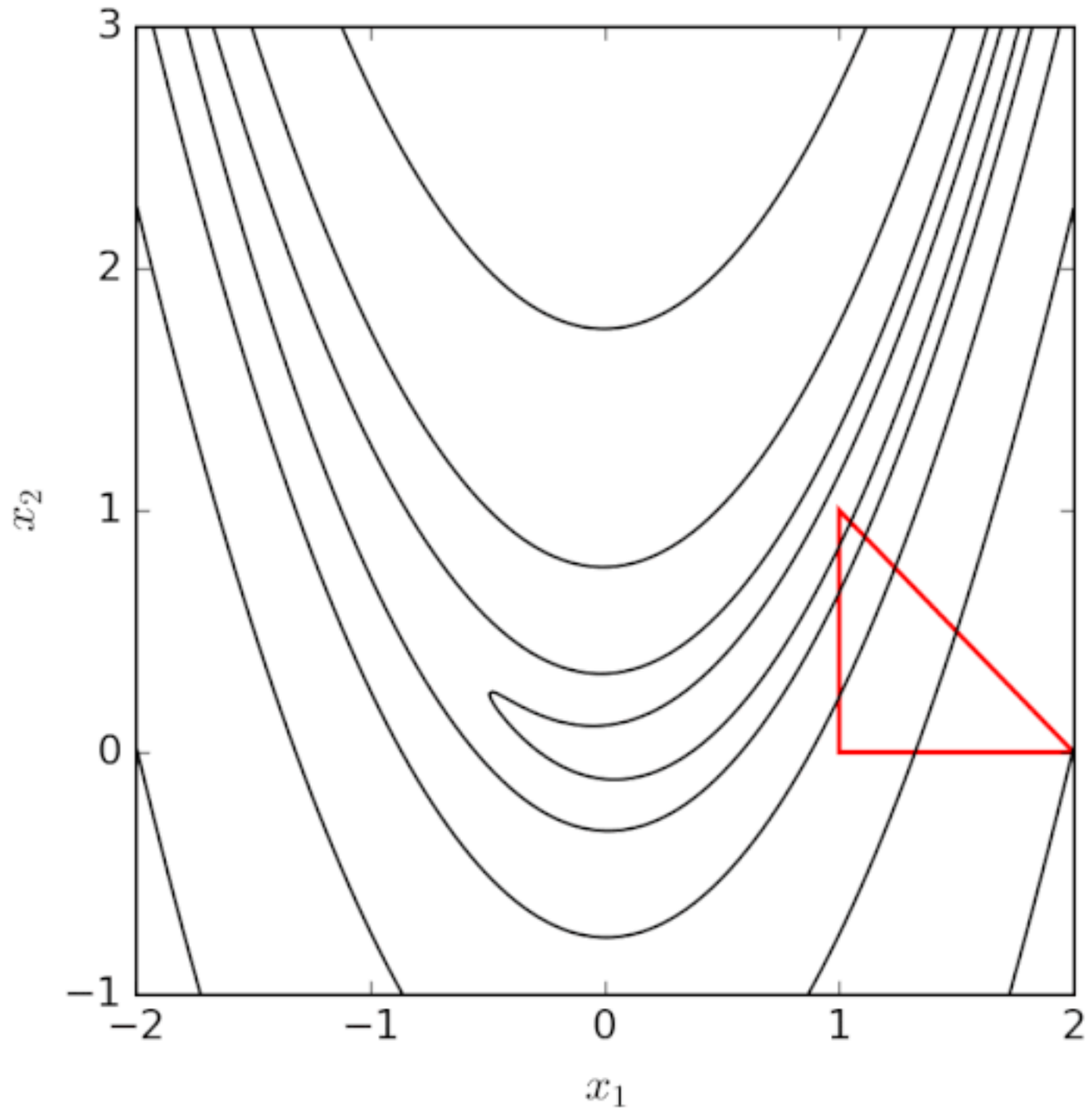
NELDER-MEAD SIMPLEX

THE ALGORITHM IN ACTION



NELDER-MEAD SIMPLEX

THE ALGORITHM IN ACTION



NELDER-MEAD SIMPLEX

PRACTICAL CONSIDERATIONS

- Relatively robust to initial solutions
- Not very robust to noise
- Extensively used
  - Can deal with non-differentiable functions, and even with some discontinuities
  - Cannot directly handle constraints (unless a penalty method is used)
- Often considered as a baseline method in literature on optimization
- However, requires many evaluations of *f(x)* compared to other methods
- No parallel evaluation (the algorithm is intrinsically sequential)

DEFINITIONS

Sequential algorithm: The point at which *f(x)* is evaluated depends on the results of all past evaluations. Evaluations of *f(x)* have to be carried out sequentially.

Parallelizable algorithm: Evaluations of *f(x)* are (at least partially) independent and can be carried out in parallel. See grid and random search.

NELDER-MEAD SIMPLEX

HOW TO USE IT IN PYTHON

NOTE: Nelder-Mead method is also used e.g. in MATLAB's fminsearch and other packages.

```
from scipy.optimize import fmin
```

SciPy.org Sponsored By ENTHOUGHT

Scipy.org | Docs | SciPy v1.2.3 Reference Guide | Optimization and Root Finding (**scipy.optimize**)

scipy.optimize.fmin

scipy.optimize.fmin(*func*, *x0*, *args=()*, *xtol=0.0001*, *ftol=0.0001*, *maxiter=None*, *maxfun=None*, *full_output=0*, *disp=1*, *retall=0*, *callback=None*, *initial_simplex=None*) [source]

Minimize a function using the downhill simplex algorithm.

This algorithm only uses function values, not derivatives or second derivatives.

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

The objective function to be minimized.

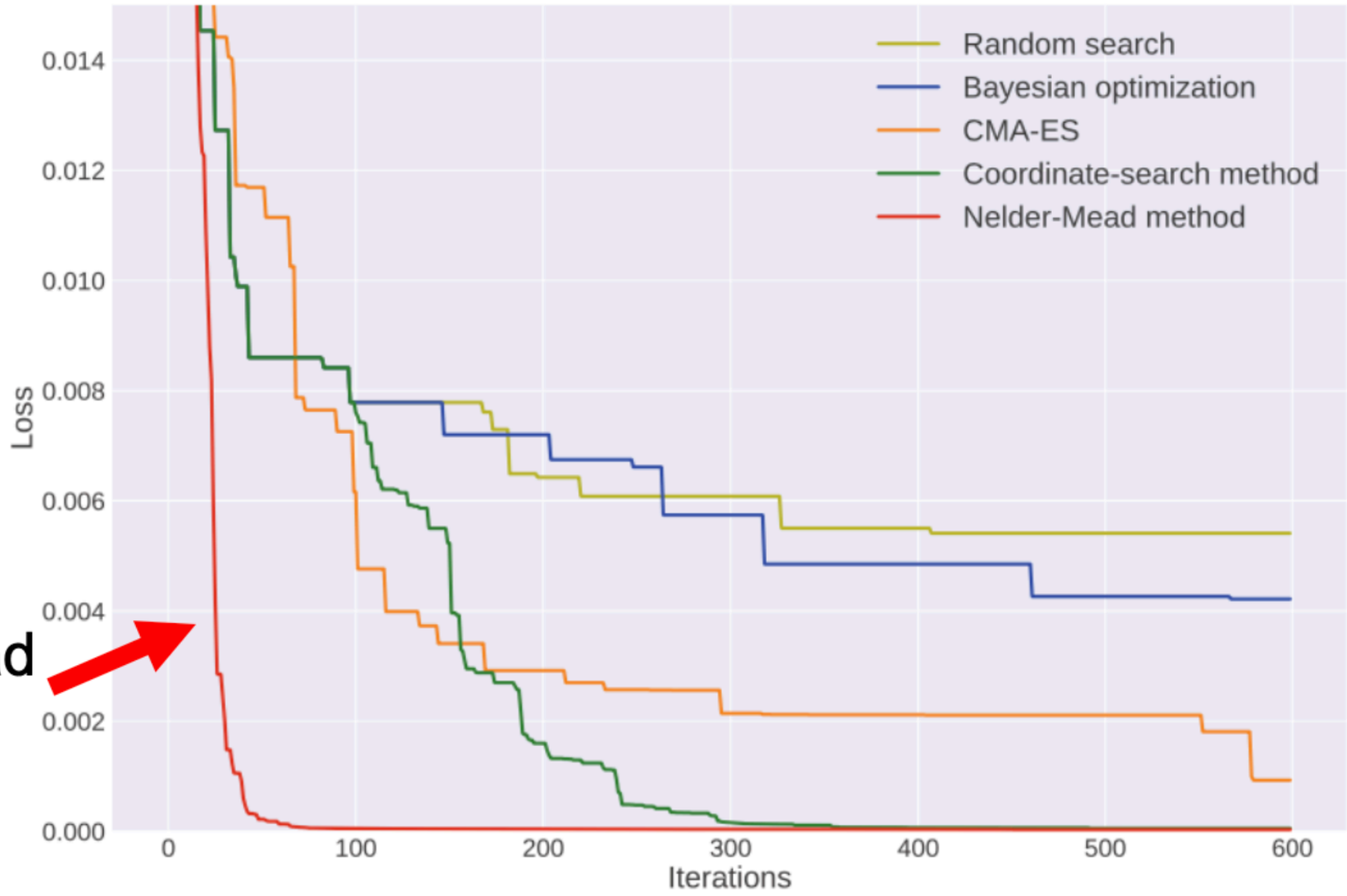
x0 : *ndarray*

Initial guess.

NELDER-MEAD SIMPLEX

TO GIVE YOU AN IDEA OF HOW EFFECTIVE IT CAN BE…

The Nelder-Mead (NM) method is superior to other HPO methods [Ozaki et al. 2017].



Nelder-Mead Method

Mean loss of all executions for each method per iteration (LeNet)

POWELL'S METHOD

THE ALGORITHM (POWELL 1964)

- If we start at point x in an N-dimensional space, and proceed in a certain direction **n**, then any function of N variables $f(x)$ can be minimized along the direction **n** by one-dimensional methods.

- But: efficiency depends on how the next direction **n** is chosen.

- Powell's Method provides a set of N mutually conjugate directions.

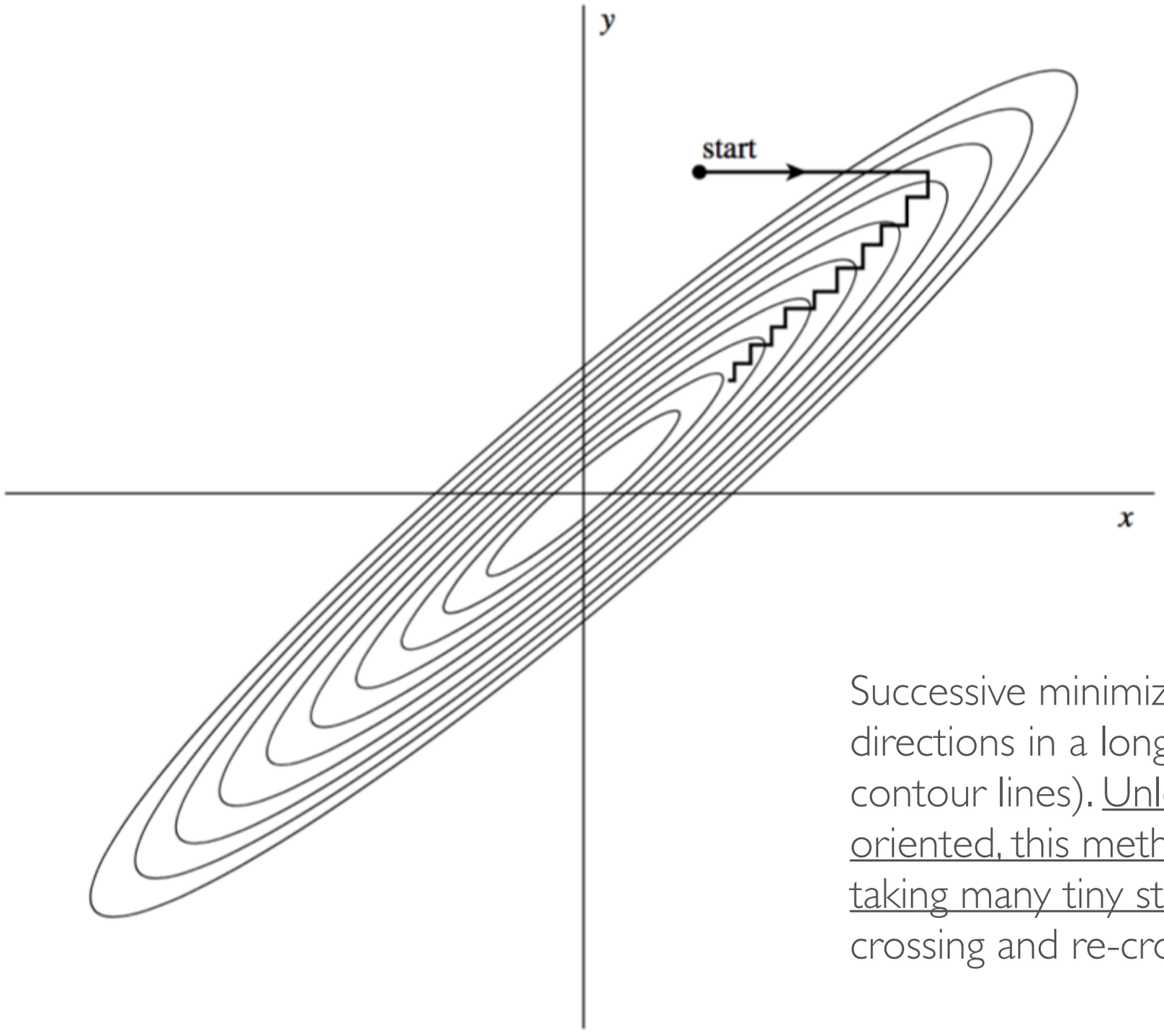
- Conjugate directions: "non-interfering" directions, with the special property that minimization along one is not "spoiled" by subsequent minimization along another, so that interminable cycling through the set of directions can be avoided

- Two vectors **u** and **v** are conjugate with respect to **Q** (or **Q**-orthogonal) if $\mathbf{u}^T\mathbf{Q}\mathbf{v} = 0$ (in case of optimization, **Q** is essentially the Hessian matrix, i.e., the matrix of 2nd order derivatives)
  - NOTE: this does not mean that the Powell's method used derivatives!

- Use this set of conjugate directions to efficiently perform line minimization
  - E.g., it reaches the minimum after $N(N+1)$ line minimizations if $f(x)$ is quadratic (this property makes the algorithm "quadratically convergent")

POWELL'S METHOD

GRAPHICAL EXPLANATION



Successive minimizations along coordinate directions in a long, narrow "valley" (shown as contour lines). Unless the valley is optimally oriented, this method is extremely inefficient, taking many tiny steps to get to the minimum, crossing and re-crossing the principal axis.

POWELL'S METHOD

THE ORIGINAL ALGORITHM

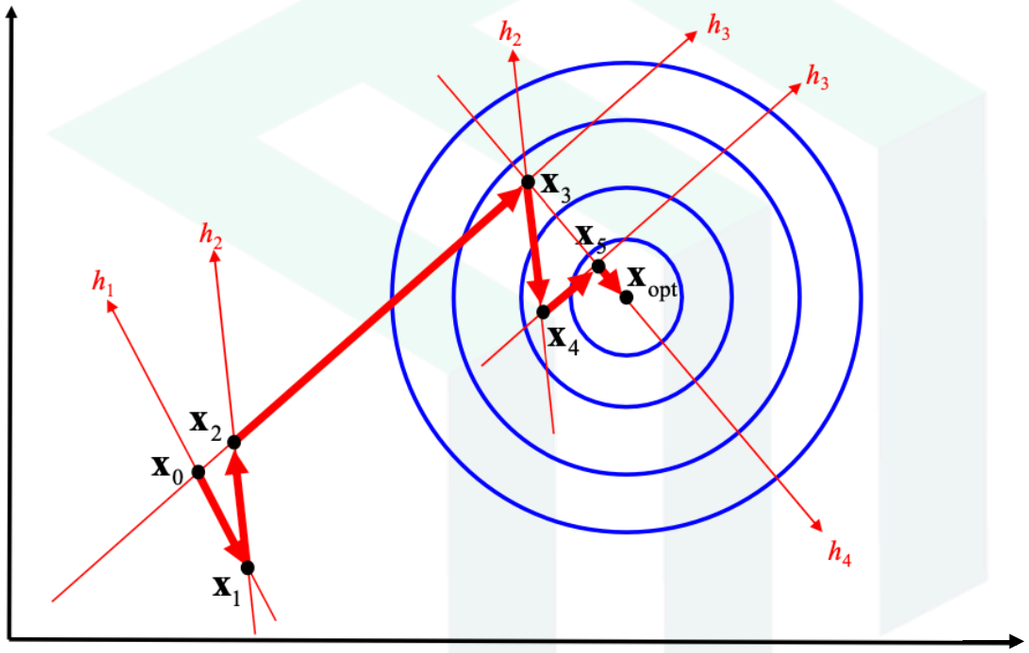
Initialize the set of directions $\mathbf{u}_i$ to the basis vectors: $\mathbf{u}_i = \mathbf{e}_i$, $i = 0, ..., N-1$.

Repeat following sequence of steps until function stops decreasing:

1. Save your starting position as $\mathbf{P}_0$.
2. For $i = 0, ..., N-1$, move $\mathbf{P}_i$ to the minimum along direction $\mathbf{u}_i$ and call this point $\mathbf{P}_{i+1}$.
3. For $i = 0, ..., N-2$, set $\mathbf{u}_i \leftarrow \mathbf{u}_{i+1}$.
4. Set $\mathbf{u}_{N-1} \leftarrow \mathbf{P}_{N-1} - \mathbf{P}_0$.
5. Move $\mathbf{P}_{N-1}$to the minimum along direction $\mathbf{u}_{N-1}$ and call this point $\mathbf{P}_0$.

POWELL'S METHOD

GRAPHICAL EXPLANATION



1. Pick a starting point $\mathbf{x}_0$ and two different starting directions h_1 and h_2.
2. Starting at $\mathbf{x}_0$, perform a 1D optimization along h_1 to find extremum $\mathbf{x}_1$.
3. Starting at $\mathbf{x}_1$, perform a 1D optimization along h_2 to find extremum $\mathbf{x}_2$.
4. Define h_3 to be in the direction connecting $\mathbf{x}_0$ to $\mathbf{x}_2$.
5. Starting at $\mathbf{x}_2$, perform a 1D optimization along h_3 to find extremum $\mathbf{x}_3$.
6. Starting at $\mathbf{x}_3$, perform a 1D optimization along h_2 to find extremum $\mathbf{x}_4$.
7. Starting at $\mathbf{x}_4$, perform a 1D optimization along h_3 to find extremum $\mathbf{x}_5$.
8. Define h_4 to be in the direction connecting $\mathbf{x}_3$ to $\mathbf{x}_5$.
9. Staring at $\mathbf{x}_5$, perform a 1D optimization along h_4 to find extremum $\mathbf{x}_{opt}$.

This last 1D optimization is guaranteed to find the maximum of a quadratic because Powell showed that h_3 and h_4 are both conjugate directions.

POWELL'S METHOD

CORRECTED VERSION

Problem: throwing away, at each stage, $\mathbf{u}_0$ in favor of $\mathbf{P}_{N-1} - \mathbf{P}_0$ tends to produce sets of directions that "fold up on each other" and become linearly dependent.

Solutions:

1. Reinitialize the set of directions $\mathbf{u}_i$ to the basis vectors $\mathbf{e}_i$ after every N or $N+1$ iterations of the basic procedure.

2. Reset the set of directions to the columns of any orthogonal matrix.

3. Still take $\mathbf{P}_{N-1} - \mathbf{P}_0$ as new direction discarding the old direction along which the function $f(\cdot)$ made its *largest decrease*.

Questions?