Iterative Algorithms

Matteo Fasiolo

Search Problems

Previously, we considered search problems of the type:

$$\{i^*\} = \underset{i \in \mathcal{F}}{\operatorname{argmin}} f(i)$$

where:

- $\mathcal{F}$ defines a **search space**,
- f defines the **search criterion**.

Local Search Algorithms

Local search algorithms is the name of **a set of search algorithms** where:

1. the algorithm starts from a candidate solution a problem.
2. It searches for a better solution that is close to the current solution.
3. If it find a better solution, it searches for an even better solution in its neighbourhood.
4. Keep iterating until a certain stopping criterion is satisfied.

For local algorithms to work, the local problem must provide **enough information** to the original problem.

Greedy Algorithms

A greedy algorithm considers a sequence subproblems.

It find the optimal solution to the subproblem, and then it revises the subproblem.

In our algorithm for playing tic-tac-toe the subproblem was the next turn.

If the subproblems are local, then we can say that an algorithm is local and greedy.

Example, hill-climbing:

$$x^*_{t+1} = \operatorname*{argmax}_{x \in [x^*_t - \epsilon, x^*_t + \epsilon]} f(x),$$

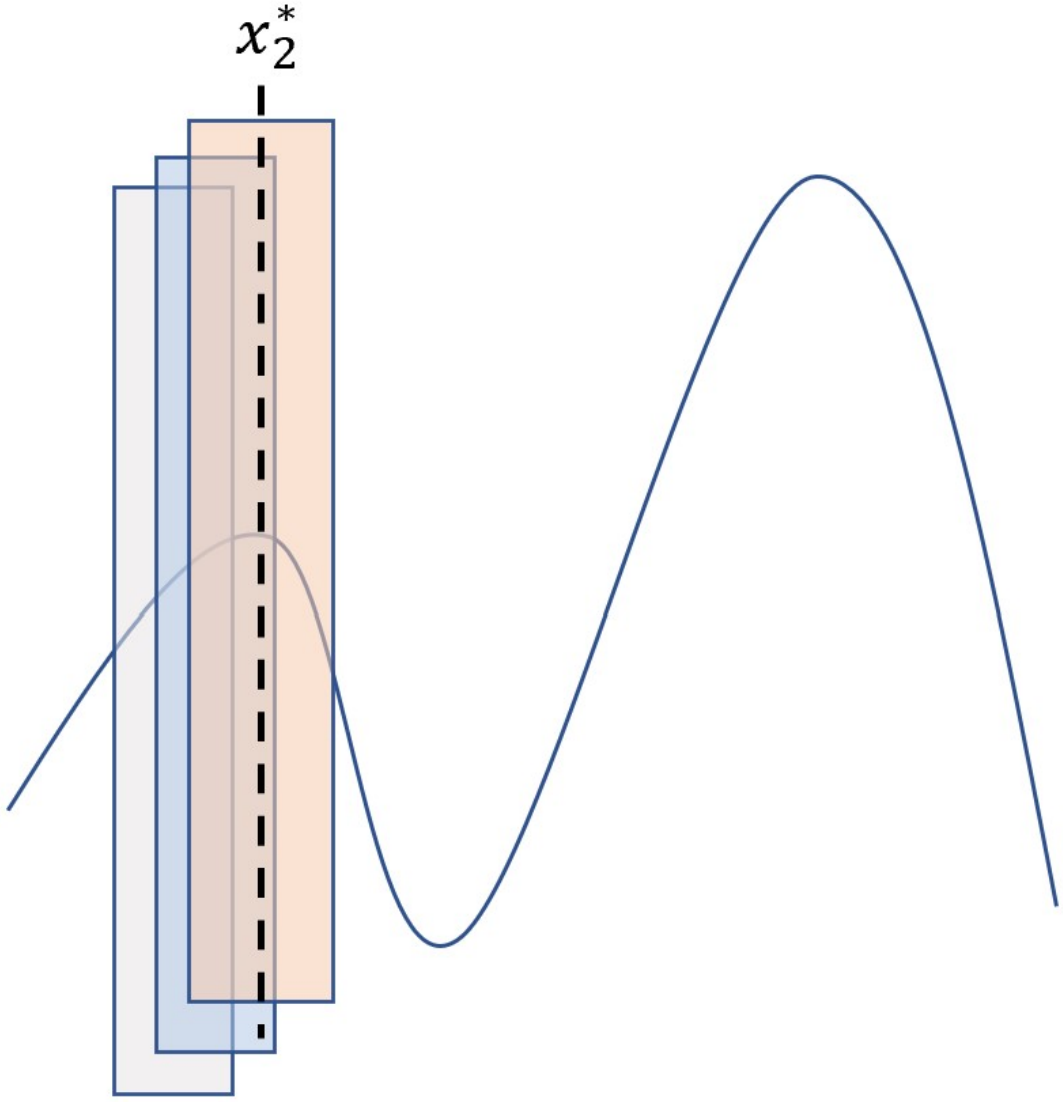
Limitations

Greedy/Local algorithms are myopic: **They may not lead to the global optimal solution.**

Hill-climbing does not find the global maximum if the search space of each subproblem is too small.

TicTacToe does not anticipate opponent moves leading to a suboptimal move.

Local optimum



Algorithm stuck at x_2^*!

A Suboptimal Move

```
X O *
X X O
* * *

AI plays...
X O O
X X O
* * *
```

It does not anticipate your opponent's move (X checkmate!)

Iterative Algorithms

The hill climbing algorithm is also an iterative algorithm:

- It starts searching from an initial value x_0.
- It solves a sequence of search problems:

$$x^*_{t+1} = \underset{x \in [x^*_t - \epsilon, x^*_t + \epsilon]}{\text{argmax}} \; f(x),$$

  where $t+1$-th problem is derived from the t-th
- It terminates when x^*_{t+1} converges.

Iterative algorithms seek to approximate the solution to a numeric problem by **successive improvements**.

- At each iteration, it revises the current approximation, so that it is closer to the true solution.
- The algorithm stops when a stopping criteria is met.

Iterative algorithms **do not have to be greedy or local** when improving your approximation.

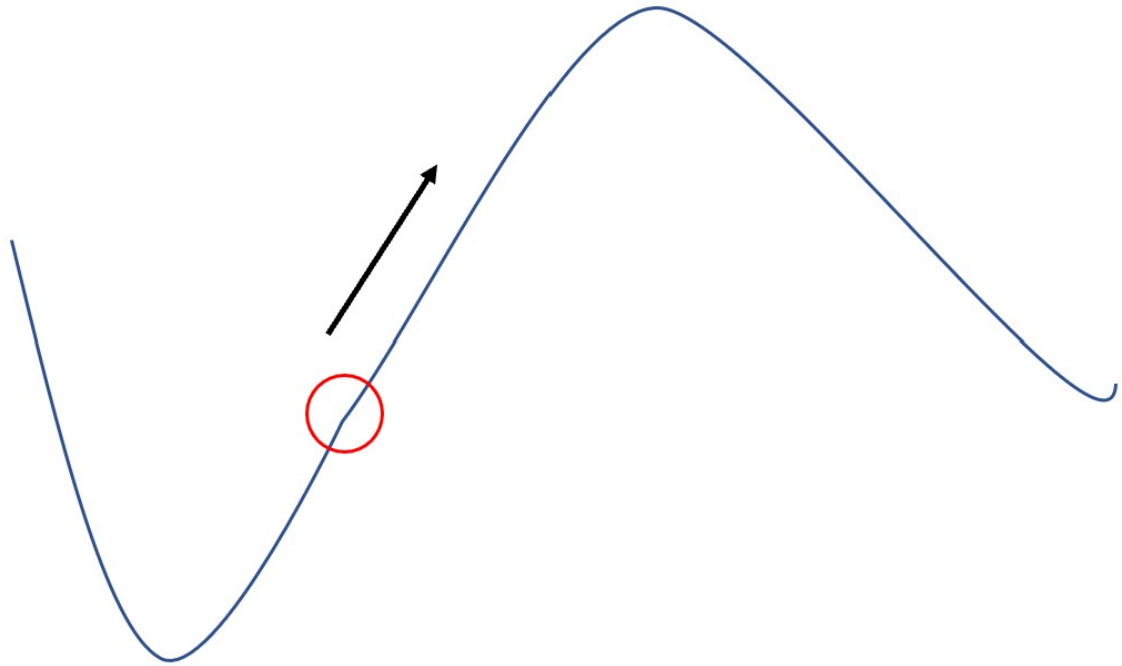
It is fine even if the **approximation** is random at each iteration.

Now we will see an iterative algorithm called

- **steepest ascent (or descent)**
- **gradient ascent (or descent)**

The Slope of a Function

From the previous example, one can see our hill climbing algorithm is quite naive. It does not take into account the **slope** of $f(x)$.



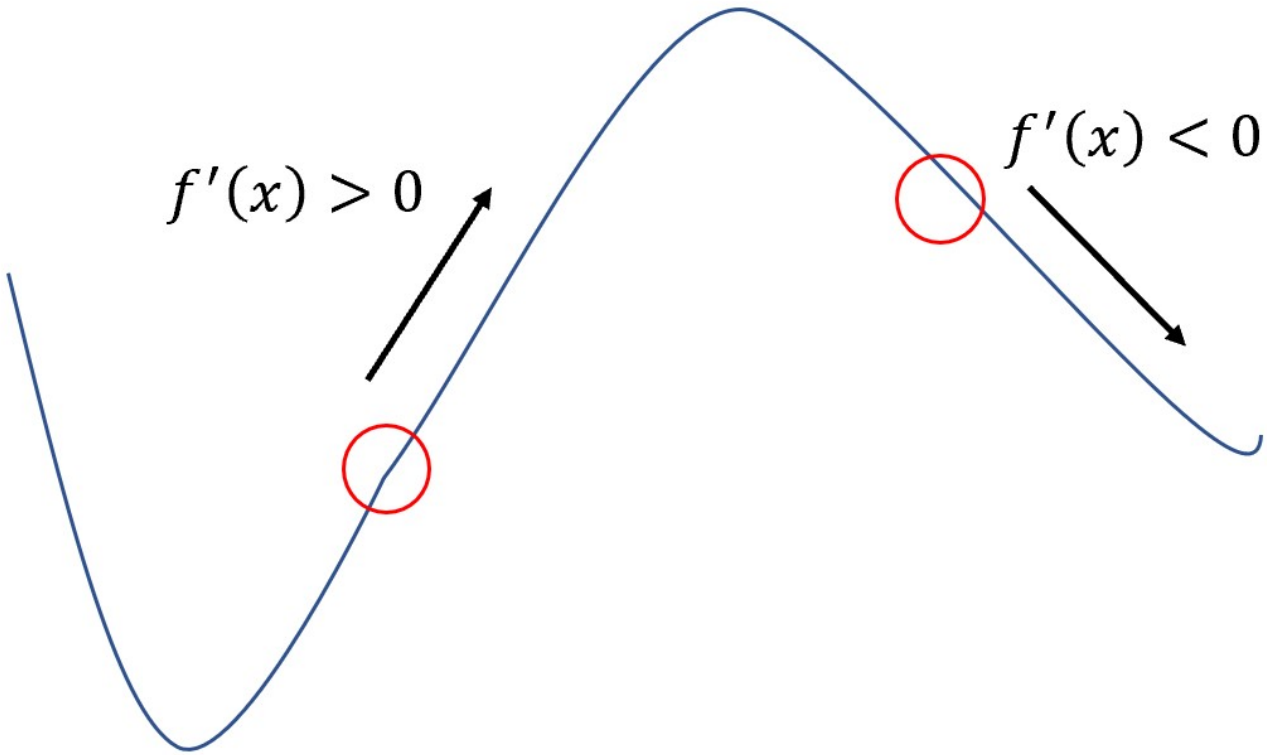
For **a differentiable function**, its slope indicates the direction you should go to maximize/minimize your function.

- To maximize, climb up the slope.
- To minimize, slide down the slope.

What is the slope of a differentiable function?

- Its first derivative:

$$f'(x) = \frac{df(x)}{dx}$$



Gradient Ascent Algorithm

Gradient Ascent is an iterative algorithm for solving the following search problem:

$$x^* = \underset{x \in \mathcal{X}}{\text{argmax}}\, f(x),$$

where $\mathcal{X}$ is the search space, e.g., $\mathbb{R}$.

It starts with a random guess x_0.

At iteration t, it revise the current guess x_t by using the derivative (gradient) information $f'(x_t)$.

Repeat until stopping criteria is met.

Pseudo Code: Gradient Ascent

1. initialize x_0 with a random (or educated) guess
2. For $t = 0$ to T
   a. $x_{t+1} \leftarrow x_t + \epsilon f'(x_t)$.
      - ϵ is the **step size**, a small number, say 0.1.
   b. If $|x_{t+1} - x_t| < \eta$, stop the loop.
      - η is an extremely small number, say 10^{-5}.
3. x_T is your approximation to x^*.

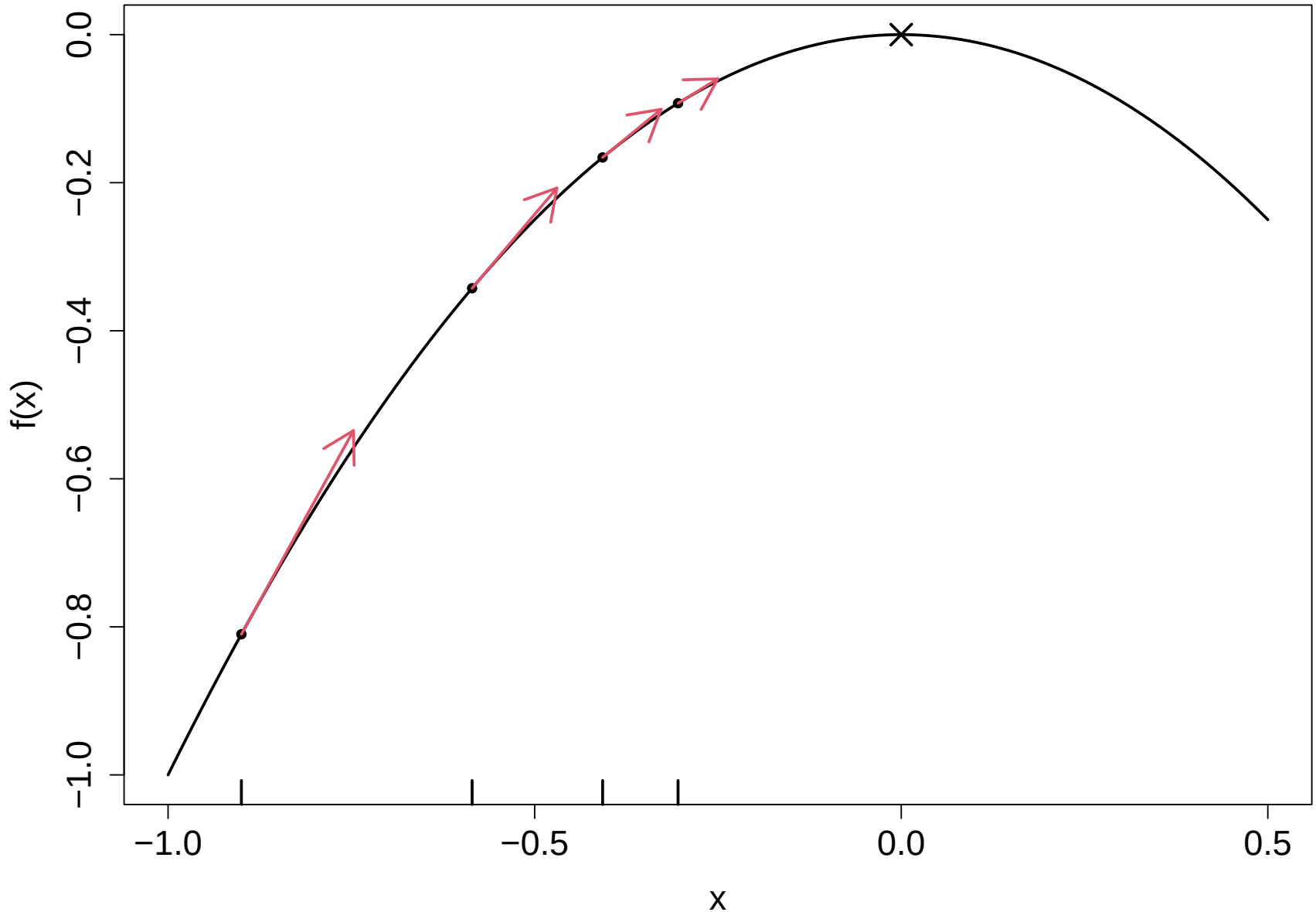
Note in b. we could also check whether $f'(x_t) \approx 0$.

In principle we should check that $f''(x_T) > 0$ to be sure you have reached a maximum.

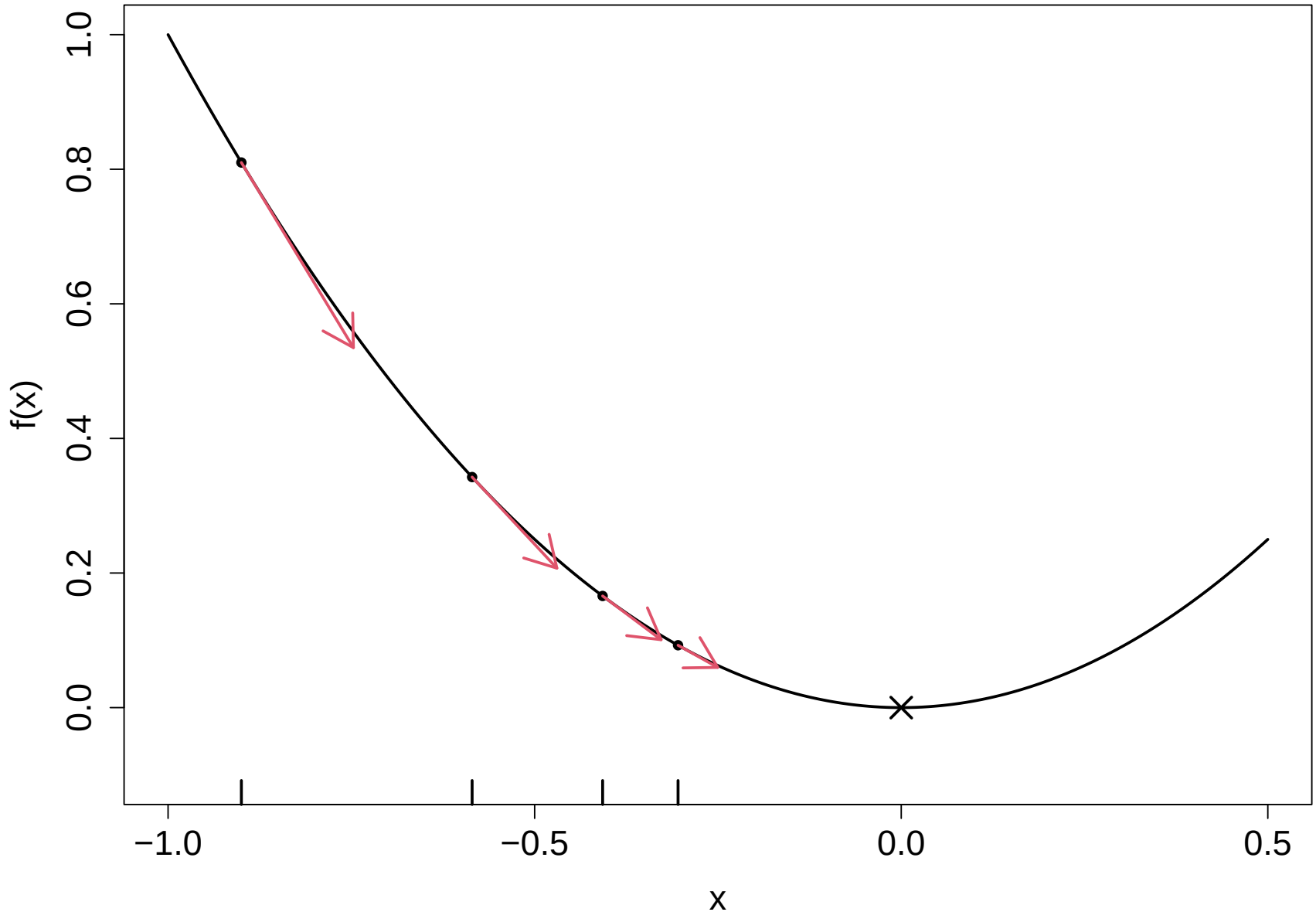
```
double x0 = -4; //initial guess
int T = 10000; //maximum iteration
double epsilon = 0.1; // step size
double eta = 1e-5; // stopping threshold

double xt = x0;
for(int t=0; t<T; t++){
    // gradient ascent!
    double xt1 = xt + epsilon*df(xt);
    // do we stop?
    if (fabs(xt1 - xt) < eta){
        xt = xt1;
        break;
    }
    xt = xt1;
}
```

No need to search a grid!

```
f(-4.000000) = -0.368995)
f(-3.999771) = -0.368995)
f(-3.999537) = -0.368994)
...
f(2.539099) = 1.760173)
f(2.539110) = 1.760173)
f(2.539120) = 1.760173)
f(2.539130) = 1.760173)
f(2.539140) = 1.760173)
```


0.0
−0.2
−0.4
−0.6
−0.8
−1.0
f(x)
−1.0
−0.5
0.0
0.5
x

Note that $x^* = \underset{x \in \mathcal{X}}{\text{argmax}} f(x) = \underset{x \in \mathcal{X}}{\text{argmin}} - f(x)$ (steepest descent).



Local Optimum

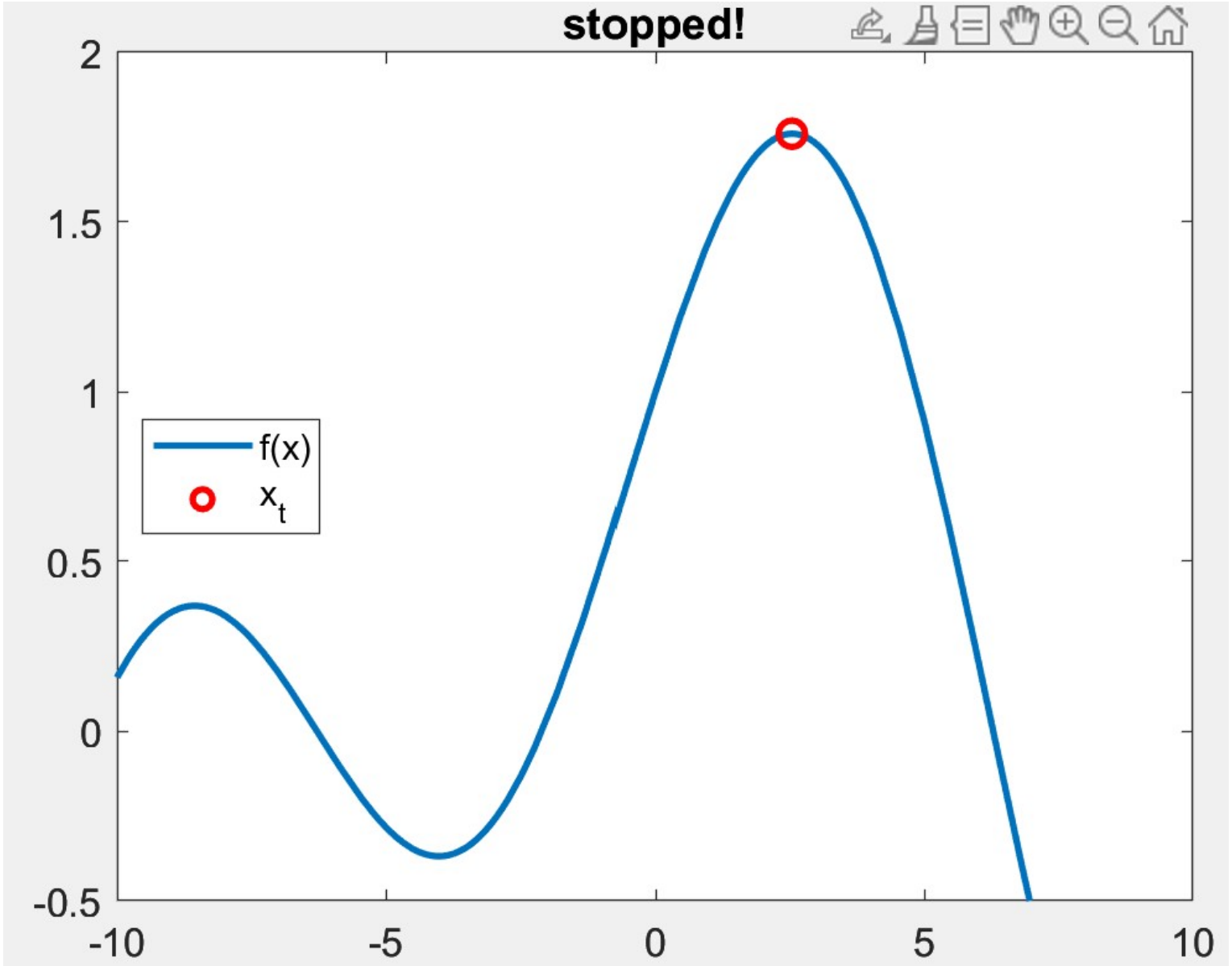
Relative to Hill Climbing, gradient descent uses more information about the problem.

Like Hill Climbing algorithm, gradient descent is not guaranteed to return the global maximum solution.

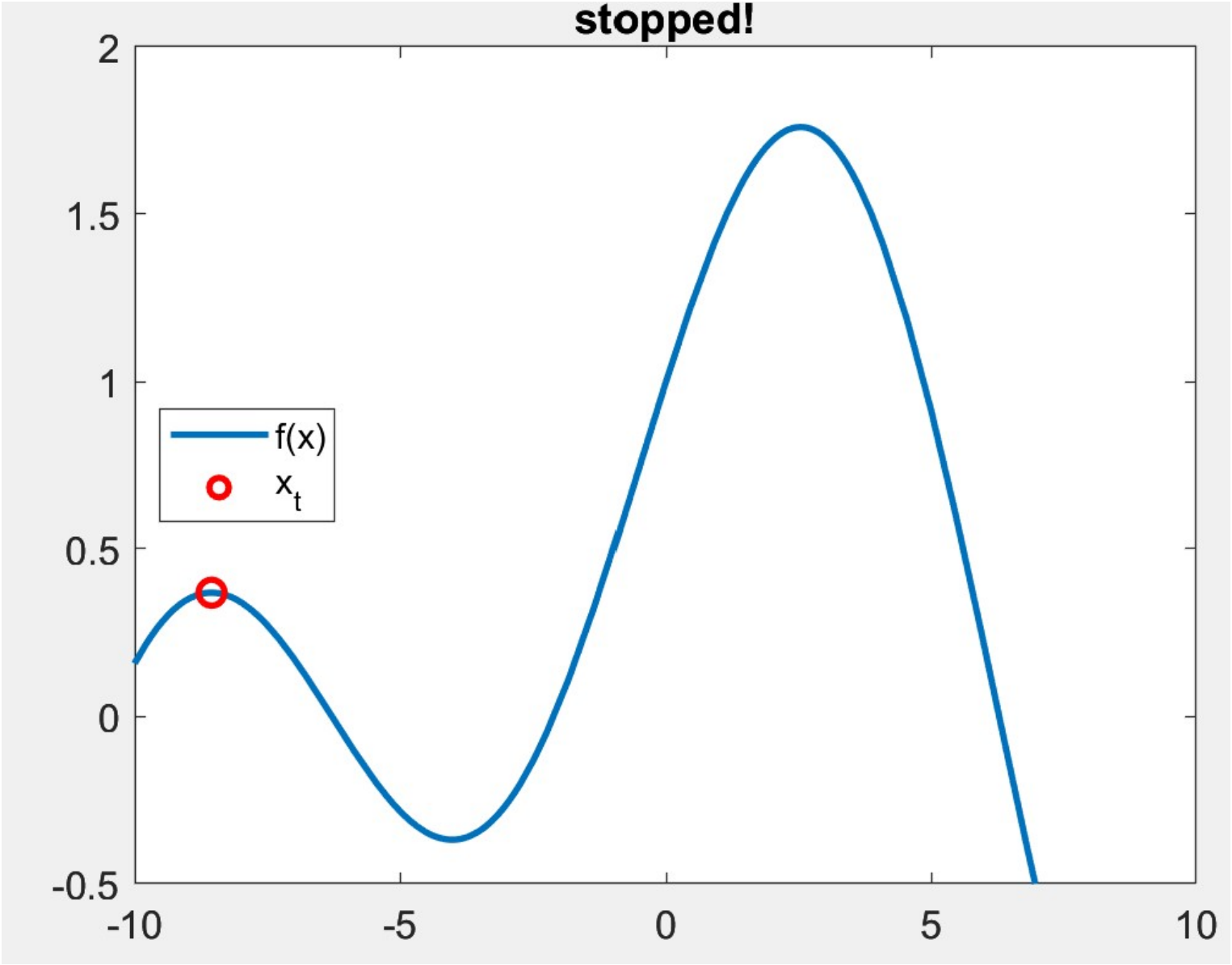
Like Hill Climbing algorithm, the gradient ascent is also not aware of the global structure of $f(x)$.

So it may get stuck at a **local optimum**.

$x_0 = -3$:



$x_0 = -5$:



Gradient Ascent for Multivariate Functions

The idea of gradient ascent easily extends to multivariate functions.

$$x^* = \underset{x \in \mathcal{X}}{\text{argmax}}\, f(x_1, x_2),$$

where, e.g, $\mathcal{X} := \mathbb{R}^2$.

The gradient of a function is defined as

$$\nabla f(x_1, x_2) := \begin{bmatrix} \partial f(x_1, x_2)/\partial x_1 \\ \partial f(x_1, x_2)/\partial x_2 \end{bmatrix}$$

The gradient of a multivariate function is a **vector** that points to the direction where f increases the fastest.

Pseudo Code: Gradient Ascent (bivariate)

1. initialize x_0 with a random guess
2. For $t = 0$ to T
   - $x_{t+1} \leftarrow x_t + \epsilon \nabla f(x_1, x_2)|_{(x_1,x_2)=x_t}$.
     - ϵ is the **step size**, a small number, say 0.1.
   - If $\|x_{t+1} - x_t\| < \eta$, stop the loop.
     - η is a extremely small number, say 10^{-5}.
3. x_T is your approximation to x^*.

Is gradient ascent greedy?

Assume that we are at x_0 and that want to move to $x_0 + \delta$.

The step has to be small $||\delta|| = \epsilon$ (e.g. $\epsilon = 10^{-5}$).

At x_0, we can Taylor expand a differentiable function around it:

$$f(x_0 + \delta) = f(x_0) + \nabla f(x_0)^T \delta + \cdots$$

If we discard all higher order term we get approximation

$$f(x_0 + \delta) \approx \tilde{f}(x_0 + \delta) = f(x_0) + \nabla f(x_0)^T \delta$$

And $\tilde{f}(x_0 + \delta)$ is maximal at

$$\delta = \epsilon \frac{\nabla f(x_0)}{||\nabla f(x_0)||}.$$

Gradient ascent is greedy: we are getting the maximum gain for a small displacement.

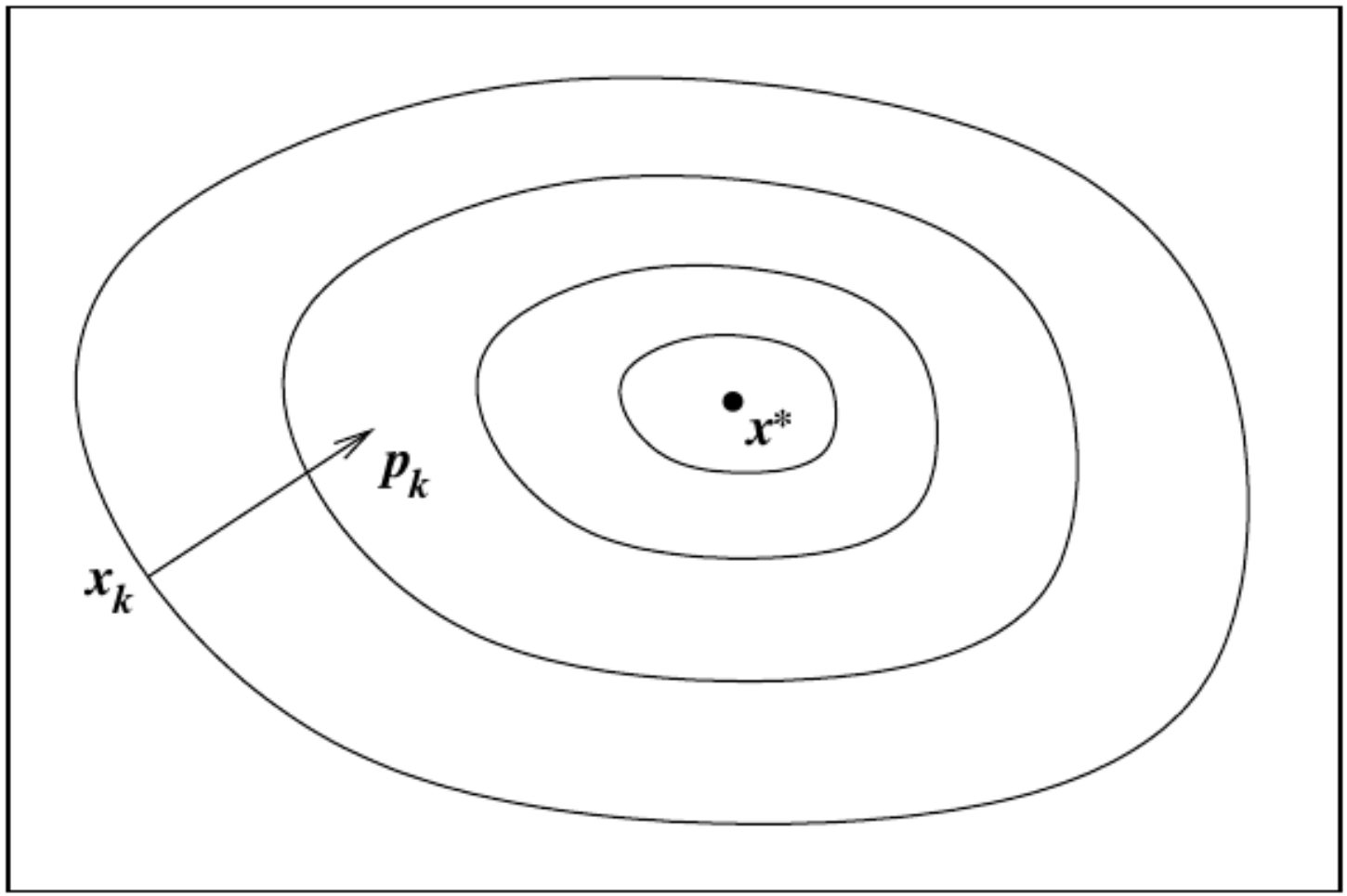


Image from Nocedal and Wright (1999).

Steepest ascent is myopic

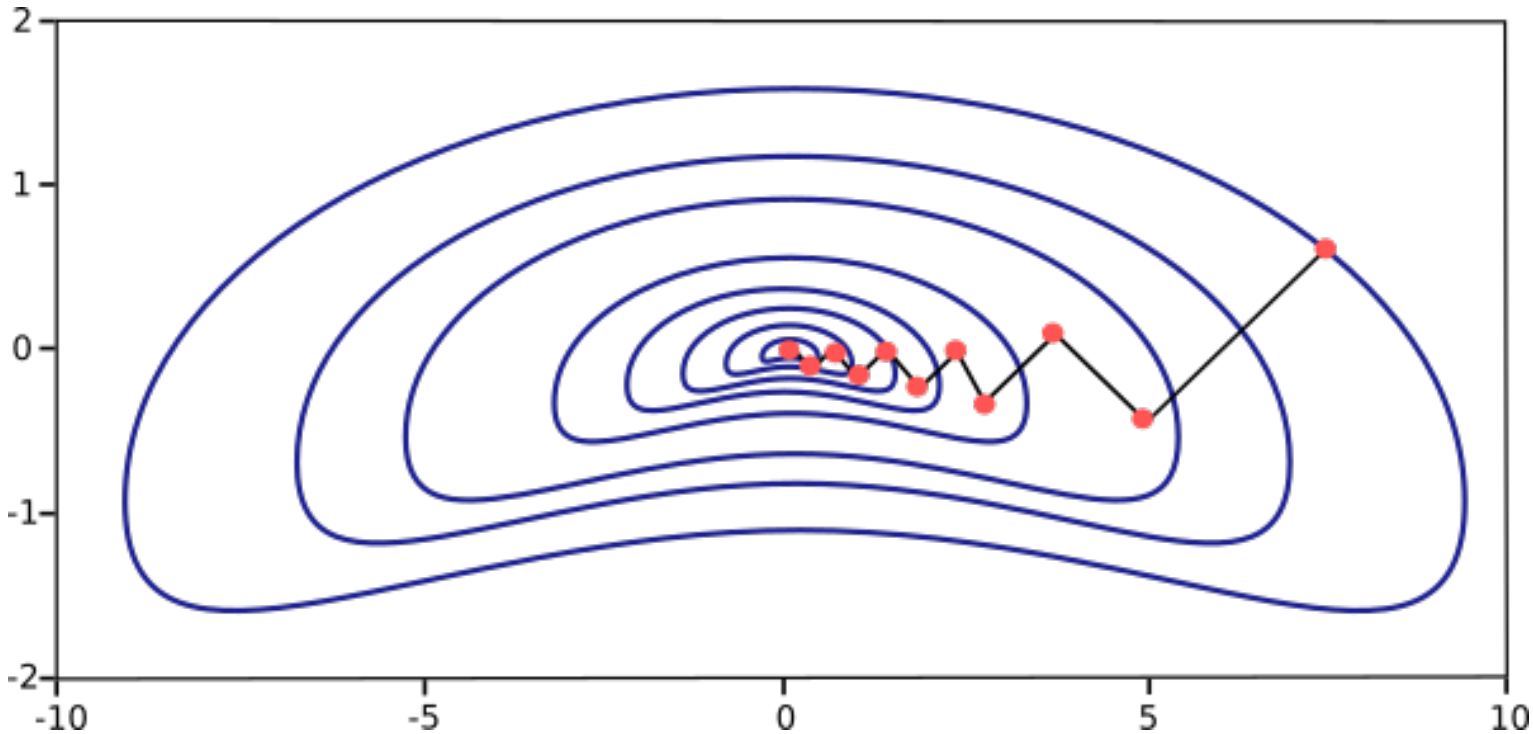


Image from https://www.neuraldesigner.com/.

Newton-Raphson's algorithm uses 1st and 2nd derivatives (the Hessian matrix) and it is less myopic.

Gradient Ascent vs. Hill Climbing

Similarities:

- They both search for the maximum of $f(x)$.
- They are both iterative algorithms.
- They both use local structure of function $f(x)$.
- They are both greedy.
- They both may be stuck at a local optimum.

Dissimilarities:

- Hill climbing evaluates $f(x), x \in [x_t - \epsilon, x_t + \epsilon]$ while gradient ascent evaluates $f'(x_t)$.
- **Gradient ascent cannot be applied to non-differentiable functions** (e.g. $f(x) = |x|$), but Hill climbing can.

Time Complexity: amount of computer time it takes to run an algorithm as a function of the size of the problem it is solving.

During tutorial on sorting, we worked on algorithms that take $\frac{\text{len}(\text{len}+1)-2}{2}$ steps to sort a sequence of length len.

We say that algorithms is "of order" len^2 or $O(\text{len}^2)$ because, for large len, the terms proportional to len^2 is the largest component of the cost.

So if the sorting algorithm takes t seconds to run on a 1000 elements array, it will take $4t$ seconds to run on a 2000 elements array.

What about Gradient Ascent vs. Hill Climbing?

Assume that:

- Evaluating $f(x)$ and $f'(x)$ takes a unit time (e.g, 1 second).
- Both algorithm run for T steps before stop.
- There are total G grid points in the hill climbing search interval $[x_t - \epsilon, x_t + \epsilon]$.

Hill Climbing: $O(GT)$

Gradient Ascent: $O(T)$

NOTE you need to be able to quantify time complexity for the exam.

Curse of Dimensionality

Hill climbing algorithm can also be generalized to multivariate function.

However, the number of grid points G grows exponentially with dimensionality of your search space.

- Number of unit length intervals in $[0, 10]$: 10.
- Number of unit squares in $[0, 10]^2$: 100.
- Number of unit cubes in $[0, 10]^3$: 1000.

Time complexity of Hill Climbing grows exponentially with dimensionality of your search space!

Gradient Ascent does not have such a problem, assuming the gradient of f can be easily evaluated.

Conclusion

Iterative algorithms seek to approximate the solution to a numeric problem by **successive improvements**.

Gradient ascent solves the following problem:

$$x^* = \operatorname*{argmax}_{x \in \mathcal{X}} f(x)$$

Gradient ascent revises the current guess x_t by using the derivative (gradient) information $f'(x_t)$.

Gradient ascent may also be stuck at the local optimum.

Gradient ascent easily generalizes to multivariate functions without suffering from the curse of dimensionality.

Homework 1.

Here we revisit hill-climbing in an object-oriented context.

1. Open `lab_steep_template.cpp`
2. Implement the hill-climbing algorithm according to the skeleton code.
3. Verify your implementation with the example output provided in the code comments.
4. By changing `x0`, could you get hill-climbing stuck on one of the local optimums?
   - By changing `epsilon` (and keeping `x0`), could the algorithm recover the global optimal solution again?

Homework 2 (submit)

1. Write a new public method `void GAsolve()` in `Problem` class:
   - It finds the maximizer of $f(x)$ using gradient ascent algorithm.
2. Call `GAsolve()` in main (use the same `Problem` object).
3. Using the same `x_0` and `epsilon`, do `GAsolve` and `solve` produce the exact same solution?
   - If not, how different are they?
   - Which one produces a better solution?
4. Change the initial guess, observe gradient ascent produces a locally optimal solution.
5. Check how gradient ascent works for very large and very small `epsilon`.

Homework 3

1. Now, write a new class:

```
class NewProblem: public Problem
{
  // a new function to be maximized
  double f(double x)
  {
    return -x*x;
  }
  public:
  // constructor for the new class
  NewProblem(double eps, double initial_guess) :
     Problem(eps, initial_guess) {}
};
```

and add the following two lines of code in `main`

```
NewProblem p2(.5, -4);
p2.solve();//what do we maximize? Did you expect this?
```

2. Change the function `f` in the `Problem` class:

```
double f(double x) // f(x),
{
    return sin(x / 2) + cos(x / 4);
}
```

to

```
virtual double f(double x) // adding "virtual"
{
    return sin(x / 2) + cos(x / 4);
}
```

A parent's `virtual f` method is overridden by the child's `f` method.

For a very clear example, see the first answer here.

Can you do the same for steepest ascent?