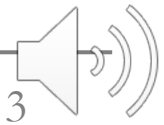
# Hill climbing

- Hill climbing is related to gradient ascent, but it doesn't require you to know the strength of the gradient or even its direction: you just iteratively test new candidate solutions in the region of your current candidate, and adopt the new ones if they're better.

- This enables you to climb up the hill until you reach a local optimum.

**Hill-Climbing**

1: S ← some initial candidate solution . *The Initialization Procedure*
2: **repeat**
3: R ← Tweak(Copy(S)) . *The Modification Procedure*
4: **if** Quality(R) > Quality(S) **then** . *The Assessment and Selection Procedures*
5: S ← R
6: **until** S is the ideal solution or we have run out of time
7: **return** S



# Hill climbing

We can make this algorithm a little more aggressive: create n "tweaks" to a candidate solution all at one time, and then possibly adopt the best one.

This modified algorithm is called **Steepest Ascent Hill-Climbing**, because by sampling all around the original candidate solution and then picking the best, we're essentially sampling the gradient and marching straight up it.

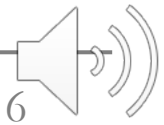
# Steepest Ascent Hill-Climbing

**Steepest Ascent Hill-Climbing**

1: n number of tweaks desired to sample the gradient
2: S ← some initial candidate solution .
3: **repeat**
4: R ← Tweak(Copy(S)) .
5: **for** n − 1 times **do**
6: W ← Tweak(Copy(S))
7: **if** Quality(W) > Quality(R) **then**
8: R ← W
9: **if** Quality(R) > Quality(S) **then.** S ← R
10: **until** S is the ideal solution or we have run out of time
11: **return** S

# Steepest Ascent Hill-Climbing with Replacement

1: n number of tweaks desired to sample the gradient
2: S ← some initial candidate solution .
3: Best ← S
4: **repeat**
5: R ← Tweak(Copy(S)) .
6: **for** n − 1 times **do**
7: W ← Tweak(Copy(S))
8: **if** Quality(W) > Quality(R) **then**
9: R ← W
*10:* *S* ← *R*
*11:* **if** Quality(R) > Quality(*Best*) **then.** *Best* ← R
12: **until** *Best* is the ideal solution or we have run out of time
13: **return** *Best*



# Solution Representation

What might a candidate solution look like?
It could be a vector;
or an arbitrary-length list of objects;
or an unordered set or collection of objects;
or a tree;
or a graph.
Or any combination of these.

Whatever seems to be appropriate to your problem. If you can create the four functions above in a reasonable fashion, that will be fine.

One simple and common representation for candidate solutions is a fixed-length vector of real-valued numbers.

# Exploration vs Exploitation

If the random noise added is very small, then Hill-Climbing will march right up a local hill and be unable to make the jump to the next hill because the bound is too small for it to jump that far.

Further, the rate at which it climbs the hill will be bounded by its small size.

On the other hand, if the the random noise added is large, then Hill-Climbing will bounce around a lot and when it is near the top of a hill, it will have a difficult time converging to the peak, as most of its moves will be so large as to overshoot the peak.

Thus small sizes of the bound of noise move slowly and get caught in local optima;

large sizes on the bound of noise bounce around too frenetically and cannot converge rapidly to finesse the very top of peaks.

# Exploration vs Exploitation

Exploration versus Exploitation

Optimization algorithms which make largely local improvements are exploiting the local gradient, and algorithms which mostly wander about randomly are thought to explore the space.

As a rule of thumb: you'd like to use a highly exploitative algorithm (it's fastest), but the "uglier" the space, the more you will have no choice but to use a more explorative algorithm.

# Single-State Global Optimization Algorithms

A global optimization algorithm is one which, if we run it long enough, will eventually find the global optimum

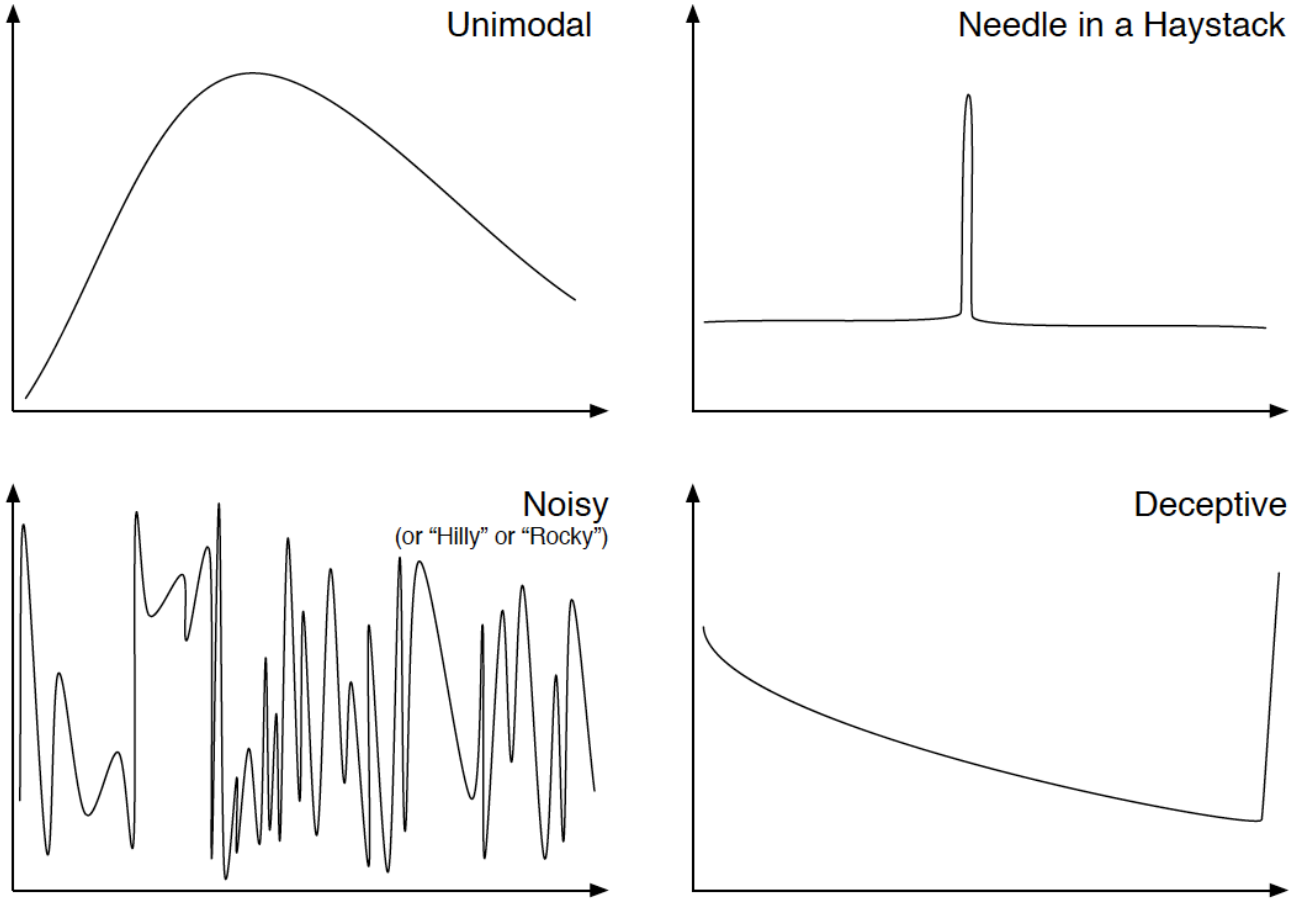
There are many ways to construct a global optimization algorithm.

The simplest one is: **Random Search**.

Random Search is the extreme in exploration (and global optimization); in contrast, Hill-Climbing may be viewed as the extreme in exploitation (and local optimization).

There are ways to achieve reasonable exploitation and still have a global algorithm: as for example Hill-Climbing with Random Restarts taht is a combination of the two approches

# Single-State Global Optimization Algorithms



# Single-State Global Optimization Algorithms

Adjust the **Modification** procedure Tweak occasionally makes large, random changes.
*Why this is Global ?* If you run the algorithm long enough, this randomness will cause Tweak to eventually try every possible solution.
*Exploration vs. Exploitation:* The more large, random changes, the more exploration.

Adjust the **Selection** procedure Change the algorithm so that you can go down hills at least some of the time.
*Why this is Global ?* If you run the algorithm long enough, you'll go down enough hills that you'll eventually find the right hill to go up.
*Exploration vs. Exploitation*: The more often you go down hills, the more exploration.

**Jump to Something New** Every once in a while start from a new location.
*Why this is Global ?* If you try enough new locations, eventually you'll hit a hill which has the highest peak.
*Exploration vs. Exploitation*: The more frequently you restart, the more exploration.

**Use a Large Sample** Try many candidate solutions in parallel.
*Why this is Global ?* With enough parallel candidate solutions, one of them is bound to be on the highest peak.
*Exploration vs. Exploitation:* More parallel candidate solutions, more exploration.