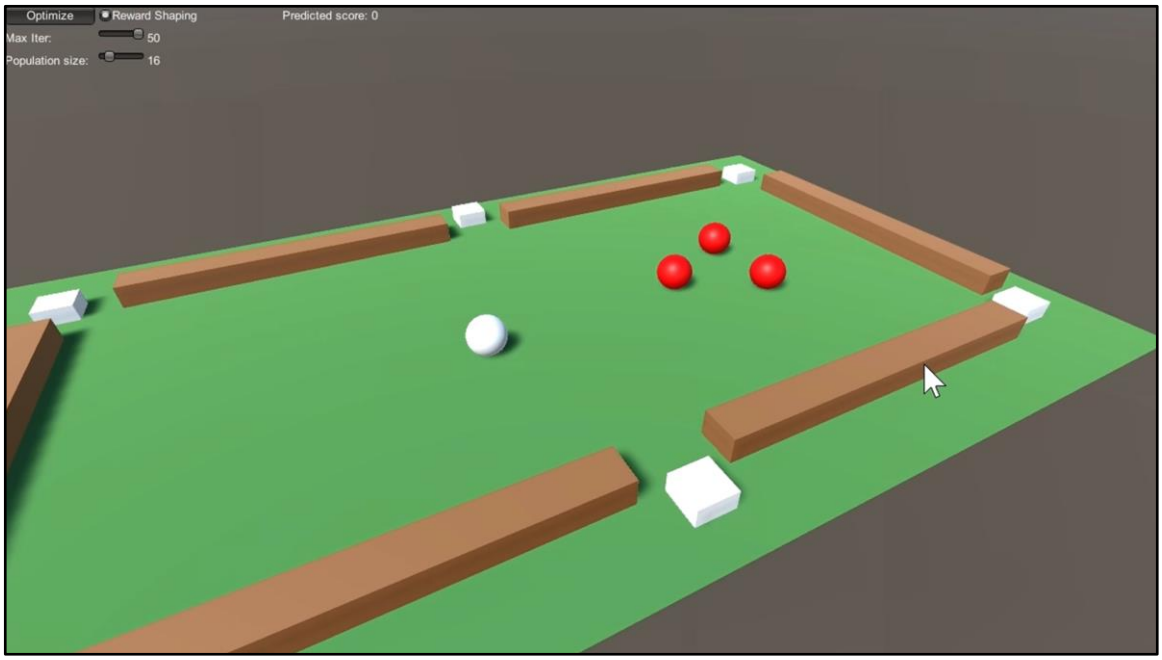
What every game developer should know about

# Mathematical Optimization (Part 1)

Intelligent Computational Media, spring 2019
Aalto University
Prof. Perttu Hämäläinen
perttu.hamalainen@aalto.fi

This lecture was originally done for the 2018 course Computational Intelligence in Games, but the content can be applied to any interactive software.




In this lecture, we will learn to, e.g., how to implement AI that can do trick shots in pool / billiards in Unity.

This project is included as a simple example in the course repository.

# Contents

- Optimization: the what and why
- Gradient-based optimization
- Population-based optimization
- Approach: A visual introduction. Some math included for self-study, but will be skipped to save time.

## What is mathematical optimization?

- Optimizing code is just one optimization problem
- In general: find parameters **x** that minimize or maximize some objective function $f(\mathbf{x})$
- We denote vectors of variables with boldface, i.e., $\mathbf{x}=[x_1,x_2,...x_N]$

In maximization, $f(\mathbf{x})$ is often called a *fitness function*. In reinforcement learning, we talk about reward functions and value functions.

## Why does it matter?

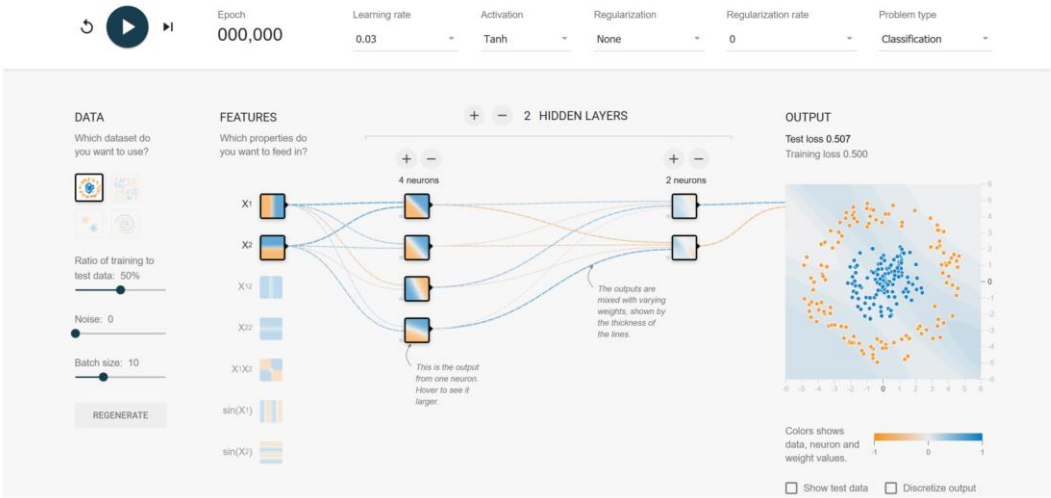
- Game Design is optimization: For example, maximize *enjoyment*(**x**)
- A/B testing is a simple optimization method
- Game playing is (usually) optimization => optimization algorithms can be used for game playing and testing
- More generally, AI = problem solving = optimization (e.g., adjust neural network parameters to minimize some loss function, or find a gameplay strategy that maximizes the probability of winning)

That is, we treat the game experience goals as a function of game design parameters **x**, which one can also think of as coordinates of a design space.

Obviously, representing all possible designs as a parameter vector is not possible in practice, but mathematical optimization can be used for subproblems that are well-defined enough. In fact, as we shall discuss later, A/B testing is a simple optimization method for such problems.

# Optimizing actions

# Optimizing neural network parameters



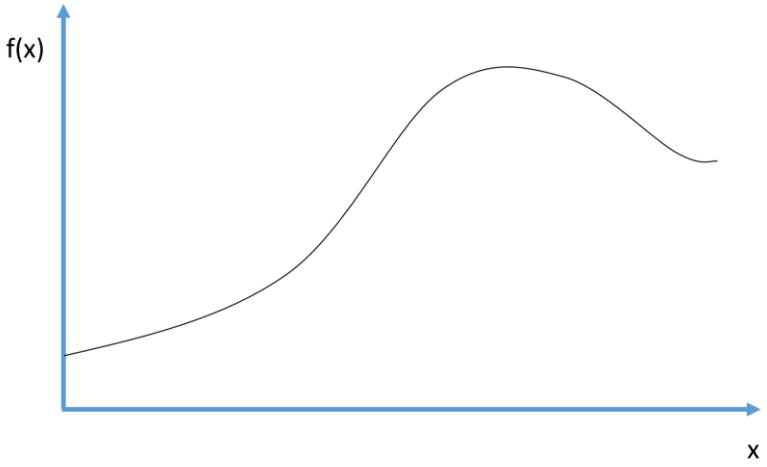
## The simplest optimization procedure

1. Initialize **x** randomly or based on some initial guess
2. Try some new $\mathbf{x}_{new}$ (typically near the current **x**)
3. If $f(\mathbf{x}_{new}) > f(\mathbf{x})$, set $\mathbf{x} = \mathbf{x}_{new}$ //assuming $f()$ is to be maximized
4. Repeat steps 2 & 3

This is similar to *simulated annealing* with specific parameters

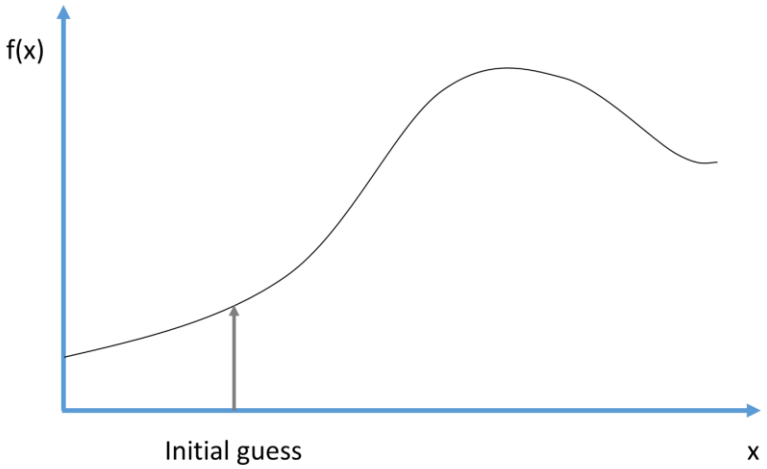
https://en.wikipedia.org/wiki/Simulated_annealing

## The simplest optimization procedure



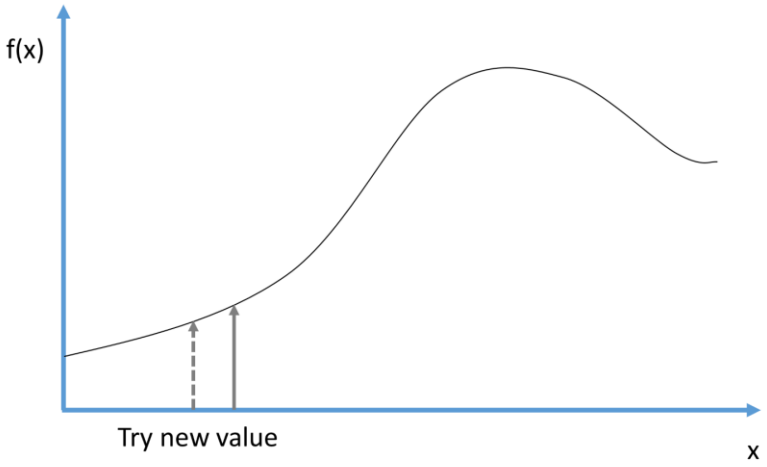

Video of rosenbrock this way

## The simplest optimization procedure



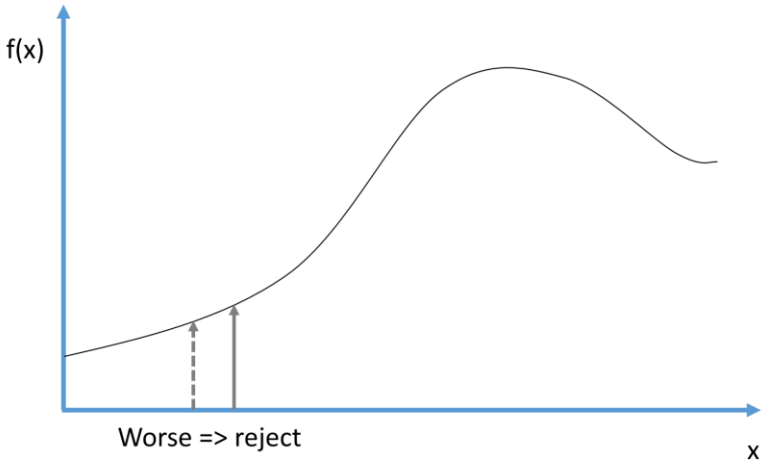

Video of rosenbrock this way

## The simplest optimization procedure



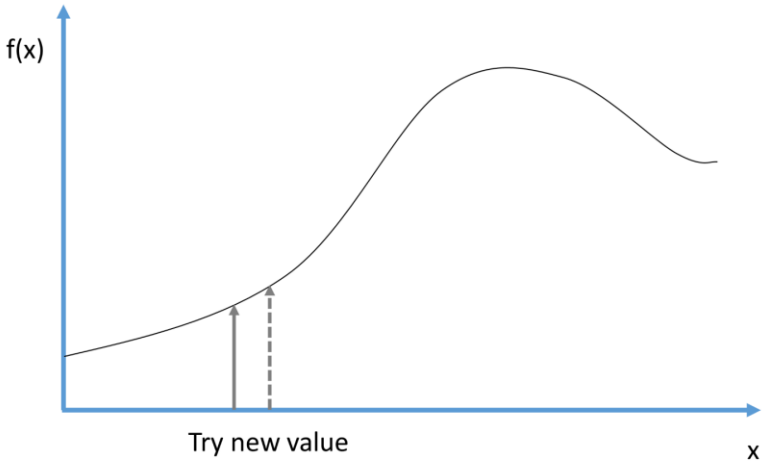

Video of rosenbrock this way

## The simplest optimization procedure



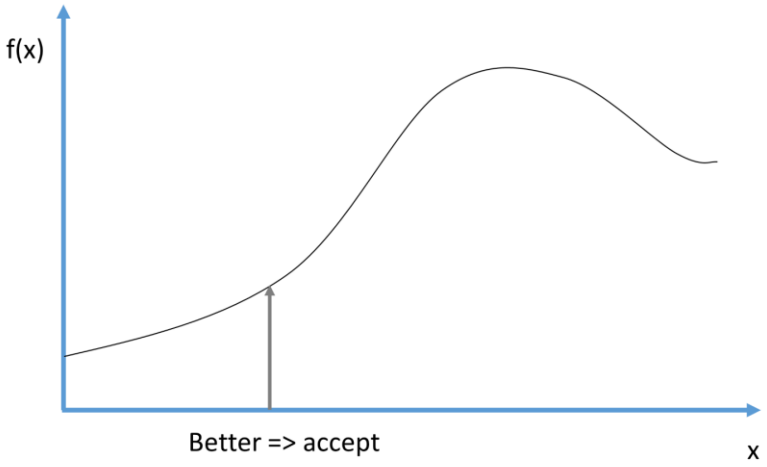

Video of rosenbrock this way

## The simplest optimization procedure



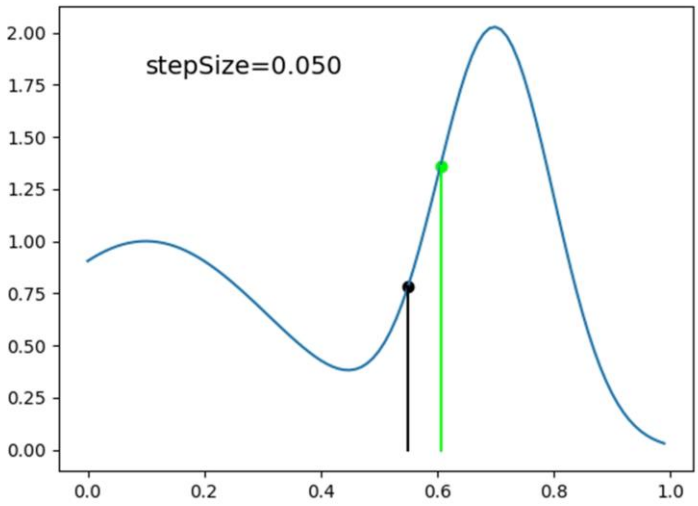

Video of rosenbrock this way

## The simplest optimization procedure




Video of rosenbrock this way

# The simplest optimization procedure



# A/B testing

- The simple optimization procedure where
  - **x** denotes some game design parameters (e.g., monetization strategy)
  - $f(\mathbf{x})$ is evaluated through deploying the game to some players, and measuring impact, e.g., monetization
  - $\mathbf{x}_{new}$ is selected by a human designer based on some hypotheses of player preferences and behavior (typically more efficient than algorithms that don't understand player psychology)

## The simplest optimization procedure

1. Initialize **x** randomly or based on some initial guess
2. Try some new $\mathbf{x}_{new}$ (typically near the current **x**)
3. If $f(\mathbf{x}_{new}) > f(\mathbf{x})$, set $\mathbf{x} = \mathbf{x}_{new}$ //assuming $f()$ is to be maximized
4. Repeat steps 2 & 3

Recap

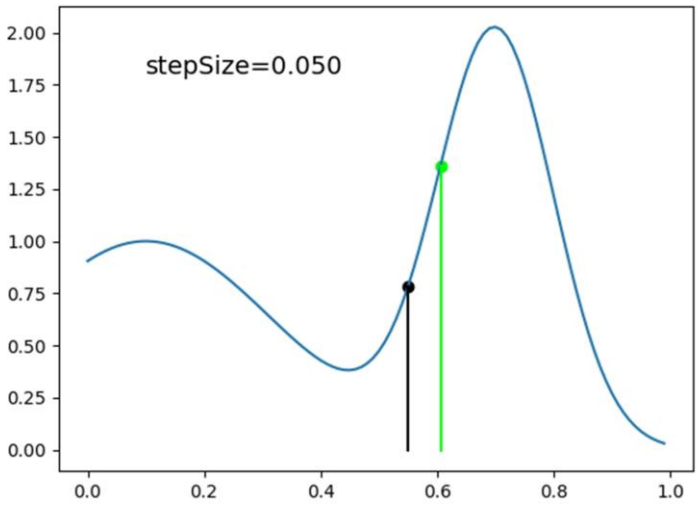
# The simplest optimization procedure

1. Initialize **x** randomly or based on some initial guess
2. Try some new $\mathbf{x}_{new}$ (typically near the current **x**)
3. If $f(\mathbf{x}_{new}) > f(\mathbf{x})$, set $\mathbf{x} = \mathbf{x}_{new}$ //assuming $f()$ is to be maximized
4. Repeat steps 2 & 3

Modifications:

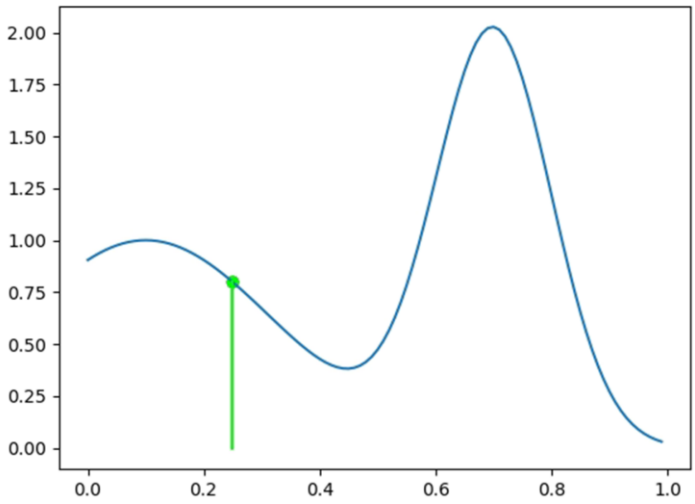
- Generic: How to select $\mathbf{x}_{new}$?
- Generic: How to adjust algorithm parameters?
- Problem-specific: How to initialize?
- Problem-specific: How to modify $f(\mathbf{x})$ or parameterize **x** such that optimization is easier?

## Sampling $\mathbf{x}_{new}$ from a normal distribution



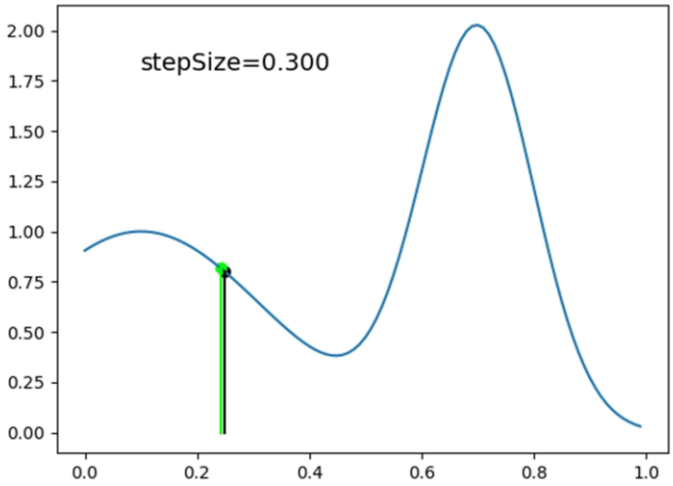

Here is an implementation of the procedure where $\mathbf{x}_{new}$ is simply sampled from a normal distribution centered at the current **x**

## Bad initialization => getting stuck in local optimum

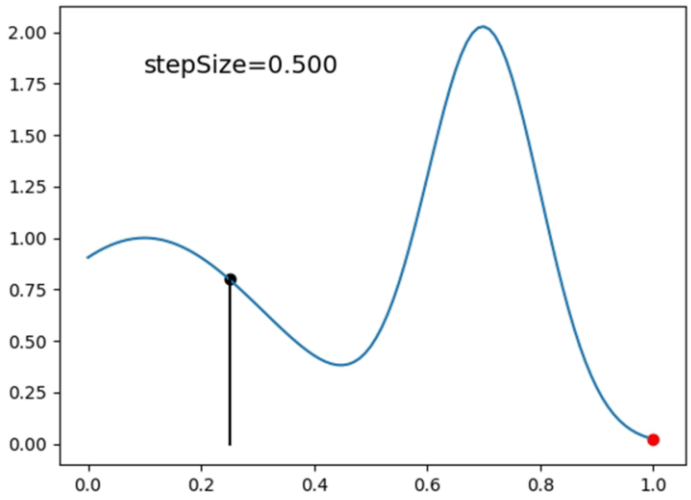



For

Large sampling variance => recovery from bad init, but slow convergence

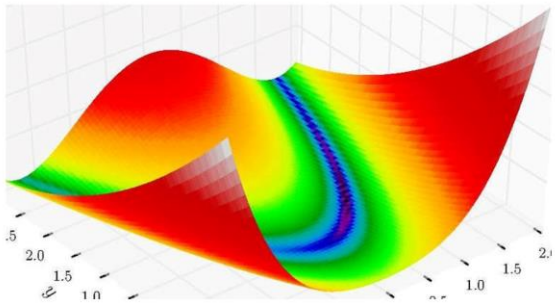


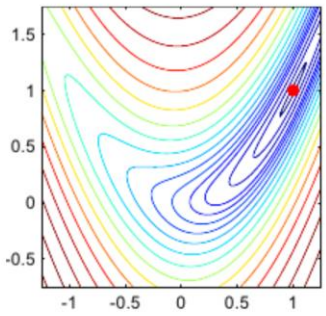
Decaying variance => recovery from bad init, better final convergence



# Curse of dimensionality

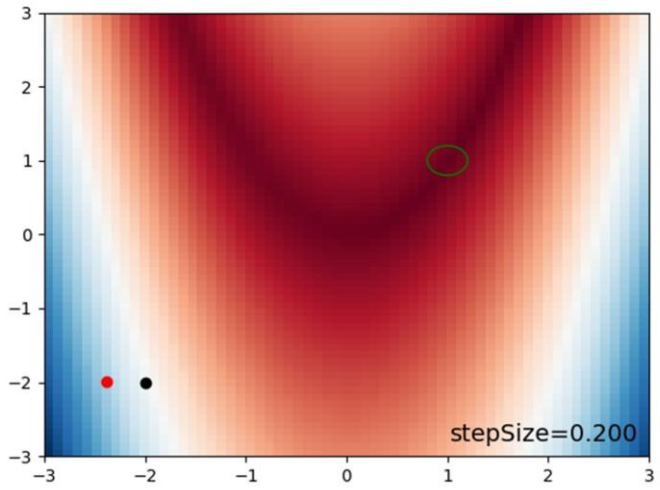
- Problem: in high-dimensional problems, random search is bad at finding the direction of improvements
- A good test function: Rosenbrock




The

# Progress slows down in the Rosenbrock valley



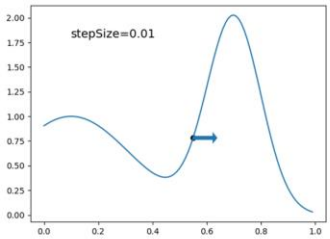
## Curse of dimensionality

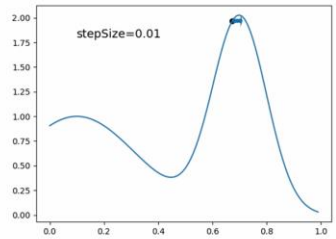
- Problem: in high-dimensional problems, random search is bad at finding the direction of improvements
- 2D Rosenbrock: a narrow valley
- 3D Rosenbrock: a narrow ”tunnel”
- The search space grows exponentially with dimensionality! (e.g., if one tries even just 2 values for each $x_1$, $x_2$, ..., $x_N$, there's $2^N$ possible combinations.

A ”valley” is a bad metaphor. Better to think of ”fog” in 3D-space, where the goal is to find the densest point. The fog is sparse except along a 3D curve - density grows slowly along the curve, decreases rapidly everywhere outside the curve. Finding the curve direction through random sampling requires exponentially more samples as the number of dimensions grows.

# Utilizing gradient information

- Finding the direction of improvement => use *gradient* information
- Gradient is *the multivariate generalization of the derivative*
- A vector that points to direction of steepest ascent of f(**x**) in the space of **x**. Denoted ∇f(**x**).
- Length proportional to steepness



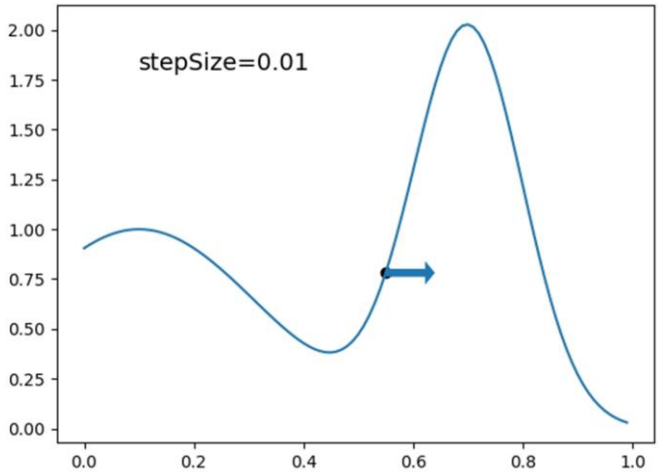



## Computing the gradient

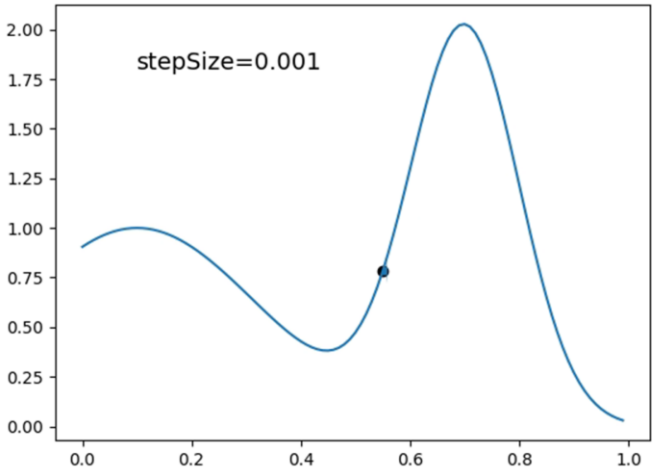
- Analytically (TensorFlow etc. do this automatically)
- Finite differences: measure the change of $f(\mathbf{x})$ when $\mathbf{x}$ perturbed along each coordinate axis
- $g_i(\mathbf{x}) = [f(\mathbf{x}) - f(\mathbf{x}+k\mathbf{u}_i)] \,/\, k$, where $\mathbf{u}_i$ is a unit vector pointing along the $i$:th coordinate axis, $k$ is a small number
- For N dimensions, this requires N+1 evaluations of $f(\mathbf{x})$

Analytical is better especially if the $f(\mathbf{x})$ evaluations are costly to compute

Gradient ascend: $x_{new} = x + stepSize * \nabla f(x)$



# Smaller stepsize: no oscillations, but slow convergence



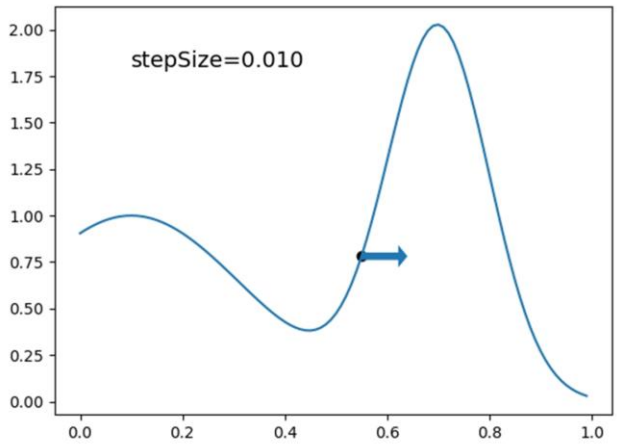
## Speeding up convergence

- Step size decay
- Gradient clipping
- Momentum
- Cyclic coordinate descend (decomposition into subproblems)
- Use curvature information
- Preconditioning

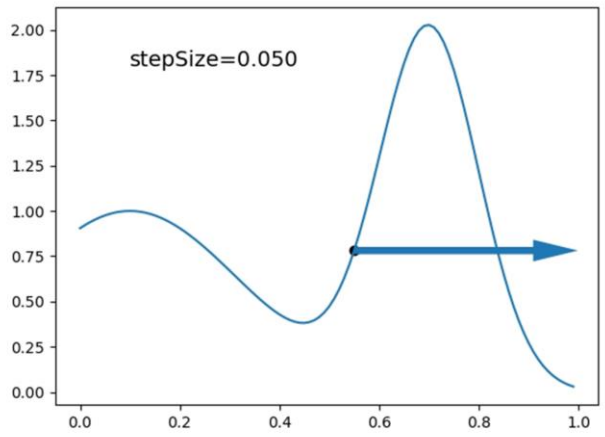
Gradient clipping is good if one can decide reasonable upper and lower limits for step size)

Ensuring at least a minimum step size is good if one can tolerate some error in the result

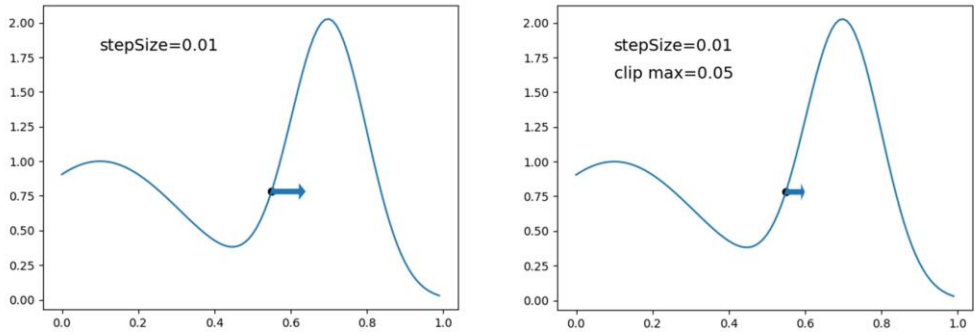
# Step size decay



# Step size decay

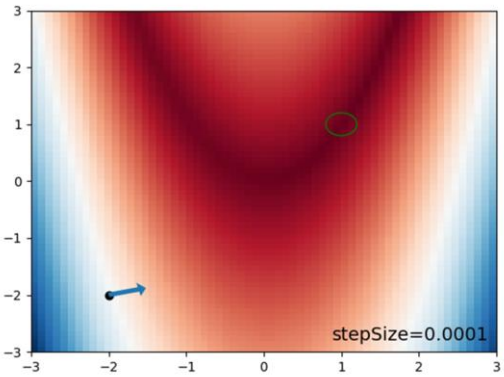


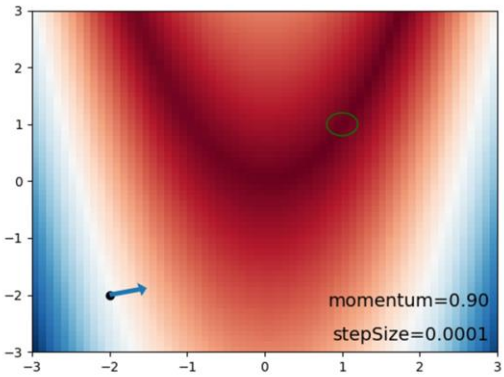
# Gradient clipping




Gradient clipping may be useful especially if discontinuities...

# Momentum

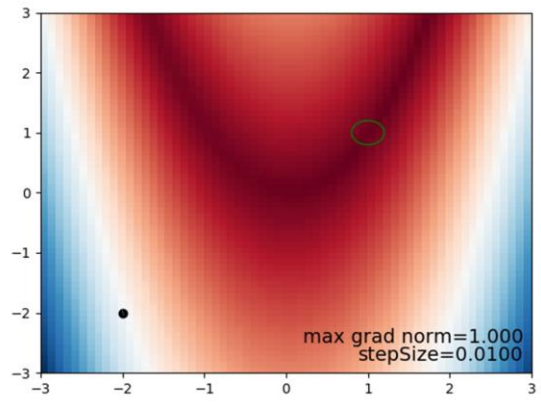


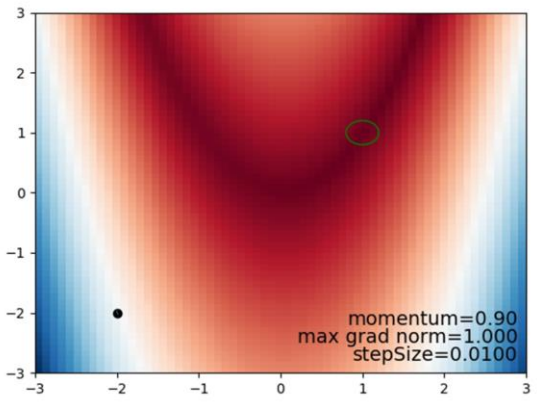



# Momentum

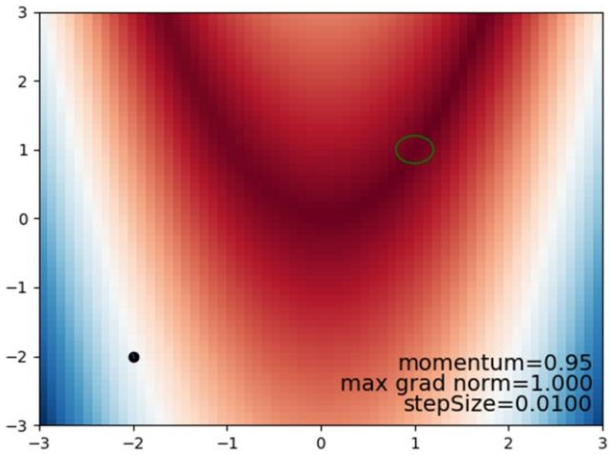
- Gradient interpreted as acceleration instead of velocity
- $\mathbf{v}$ = momentum*$\mathbf{v}$ + $\nabla f(\mathbf{x})$
- $\mathbf{x}_{new}$ = -stepSize * $\mathbf{v}$
- 1-momentum = friction, damping of velocity between updates
- Can be used with gradient clipping

# Momentum

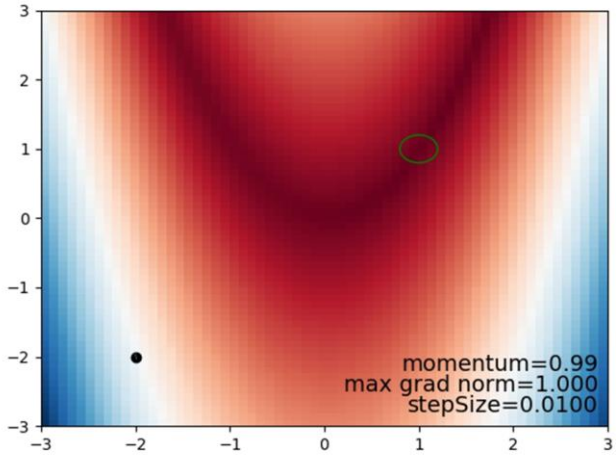

max grad norm=1.000
stepSize=0.0100


momentum=0.90
max grad norm=1.000
stepSize=0.0100

# Momentum



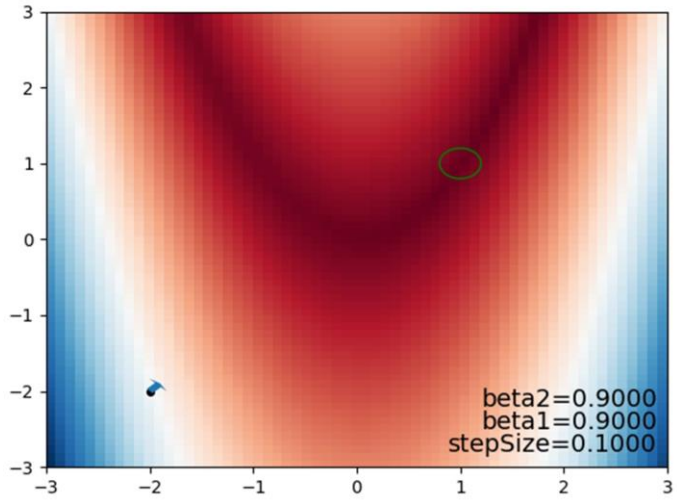
# Momentum




https://distill.pub/2017/momentum/

A great online paper with informative interactive visualizations and also mathematical analysis of momentum

# Adam: modern momentum method



Published as a conference paper at ICLR 2015

# ADAM: A METHOD FOR STOCHASTIC OPTIMIZATION

**Diederik P. Kingma***
University of Amsterdam
dpkingma@uva.nl

**Jimmy Lei Ba***
University of Toronto
jimmy@psi.utoronto.ca

## ABSTRACT

We introduce *Adam*, an algorithm for first-order gradient-based optimization of stochastic objective functions, based on adaptive estimates of lower-order moments. The method is straightforward to implement, is computationally efficient, has little memory requirements, is invariant to diagonal rescaling of the gradients, and is well suited for problems that are large in terms of data and/or parameters. The method is also appropriate for non-stationary objectives and problems with very noisy and/or sparse gradients. The hyper-parameters have intuitive interpretations and typically require little tuning. Some connections to related algorithms, on which *Adam* was inspired, are discussed. We also analyze the theoretical convergence properties of the algorithm and provide a regret bound on the convergence rate that is comparable to the best known results under the online convex optimization framework. Empirical results demonstrate that Adam works well in practice and compares favorably to other stochastic optimization methods. Finally, we discuss *AdaMax*, a variant of *Adam* based on the infinity norm.

Adam has become a method many researchers try first. Only need to adjust the step size, defaults usually work for the other parameters

https://arxiv.org/pdf/1412.6980v8.pdf

**Algorithm 1:** *Adam*, our proposed algorithm for stochastic optimization. See section 2 for details, and for a slightly more efficient (but less clear) order of computation. $g_t^2$ indicates the elementwise square $g_t \odot g_t$. Good default settings for the tested machine learning problems are $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$. All operations on vectors are element-wise. With $\beta_1^t$ and $\beta_2^t$ we denote $\beta_1$ and $\beta_2$ to the power $t$.

**Require:** $\alpha$: Stepsize
**Require:** $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
**Require:** $f(\theta)$: Stochastic objective function with parameters $\theta$
**Require:** $\theta_0$: Initial parameter vector
  $m_0 \leftarrow 0$ (Initialize 1$^{\text{st}}$ moment vector)
  $v_0 \leftarrow 0$ (Initialize 2$^{\text{nd}}$ moment vector)
  $t \leftarrow 0$ (Initialize timestep)
  **while** $\theta_t$ not converged **do**
    $t \leftarrow t + 1$
    $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep $t$)
    $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
    $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
    $\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
    $\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
    $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$ (Update parameters)
  **end while**
  **return** $\theta_t$ (Resulting parameters)

**while** $\theta_t$ not converged **do**

$t \leftarrow t + 1$

$g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$

$m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$

$v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$

$\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$

$\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$

$\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$

**end while**

**while** $\theta_t$ not converged **do**

$t \leftarrow t + 1$

$g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$

$m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$

$v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$

$\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$

$\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$

$\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$

**end while**

t is simply an iteration counter variable

**while** $\theta_t$ not converged **do**

$t \leftarrow t + 1$

$g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$

$m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$

$v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$

$\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$

$\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$

$\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$

**end while**

**g** denotes the gradient vector

**while** $\theta_t$ not converged **do**

$$t \leftarrow t + 1$$

$$g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$$

$$m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$$

$$v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$$

$$\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$$

$$\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$$

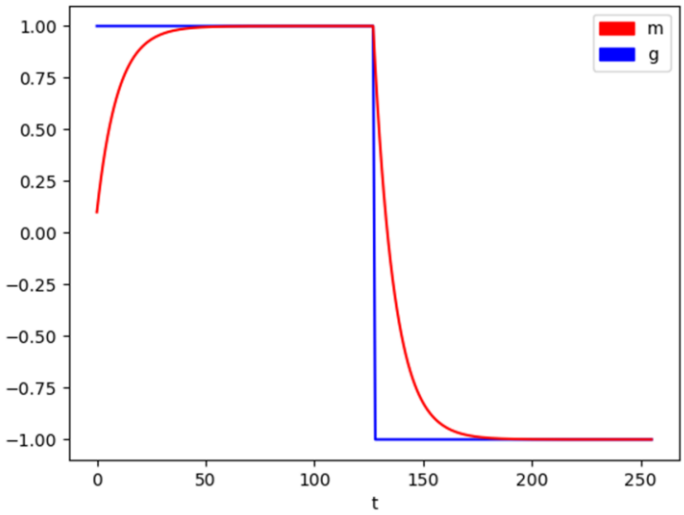
$$\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$$

**end while**

$\mathbf{m}_t$ is an exponentially smoothed version of the gradient. Note that beta1 + (1-beta1)=1, i.e., for stationary input, the output asymptotically approaches the input.

This kind of exponential smoothing is very common in engineering, good to learn to notice the pattern.

# Exponential smoothing, beta1=0.9



**while** $\theta_t$ not converged **do**

$t \leftarrow t + 1$

$g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$

$m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$

$v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$

$\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$

$\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$

$\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$

**end while**

$\mathbf{v}_t$ is similar, but using squared gradient values, i.e., sqrt($\mathbf{v}_t$) is a smoothed estimate of per-variable gradient magnitude.

**while** $\theta_t$ not converged **do**

$t \leftarrow t + 1$

$g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$

$m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$

$v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$

$\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$

$\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$

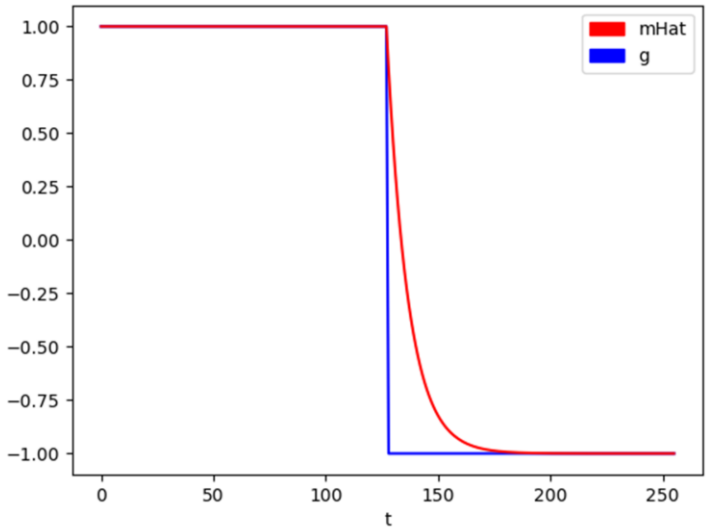
$\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$

**end while**

These lines just compute an ”unbiased” version of $\mathbf{m}_t$ and $\mathbf{v}_t$.

Initially, when t is small, m^hat is updated more instantaneously updated.

## Bias correction




The goal of the bias correction is to initially adapt the smoother output immediately, and then slow down. This way, the initial value of m does not have an effect on the results.

**while** $\theta_t$ not converged **do**

$$t \leftarrow t + 1$$
$$g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$$
$$m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$$
$$v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$$
$$\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$$
$$\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$$
$$\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$$

**end while**

This last line is the secret sauce of Adam, so let's investigate is closer.

**while** $\theta_t$ not converged **do**

Change of parameter values

$$\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$$

**end while**

This is what you should focus on, i.e., the change in parameter values for each iteration

**while** $\theta_t$ not converged **do**

Step size

$$\theta_t \leftarrow \theta_{t-1} - \boxed{\alpha} \cdot \hat{m}_t / (\sqrt{\hat{v}_t} + \epsilon)$$

**end while**

The first term is just the step size

**while** $\theta_t$ not converged **do**

Smoothed gradient

$$\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$$

**end while**

This is the smoothed gradient, the hat denoting the bias correction. So far, this is just gradient descend that averages the gradient over many iterations. The gradient is smoothed also when using momentum, but the Adam's difference to momentum is that progress does not keep accelerating if the gradient is constant. Because beta1+(1-beta1)=1, smoother output does not keep increasing infinitely with constant input.

**while** $\theta_t$ not converged **do**

Per-variable scaling

$$\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / \left(\sqrt{\widehat{v}_t} + \epsilon\right)$$

**end while**

The final component amounts to a per-variable scale for the smoothed gradient. The epsilon can be neglected, it's just a small constant to prevent division by zero.

**while** $\theta_t$ not converged **do**

Per-variable scaling

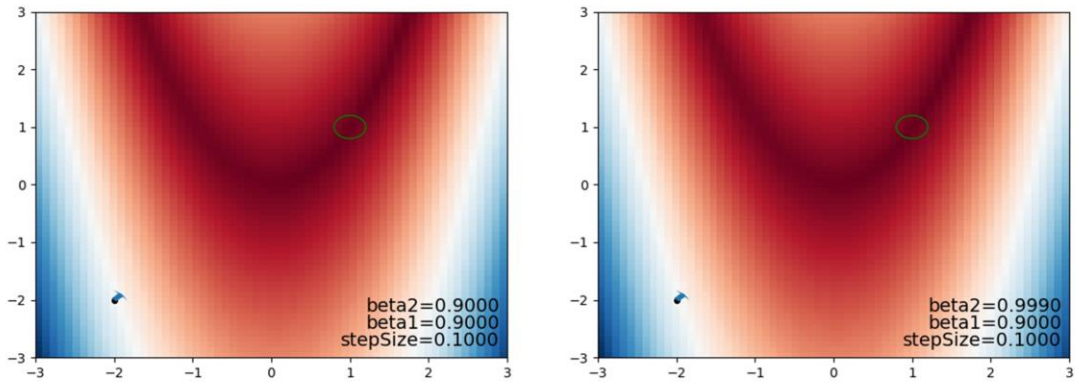
$$\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / \left(\sqrt{\widehat{v}_t} + \epsilon\right)$$

**end while**

As $\mathbf{v}_t$ is the smoothed squared gradient and the hat denotes bias-correction, per-element square root of $\mathbf{v}_t$ is simply the magnitude of each gradient variable averaged over many time steps.

What this means is that Adam step size is approximately invariant to the objective function scale and gradient scale – this means that it should move as fast in Rosenbrock valley as when far from the valley where the gradient is high. That is, the effect is similar but more sophisticated to gradient clipping. This also makes it easier to adjust alpha, as it needs less tweaking when the objective function is altered.

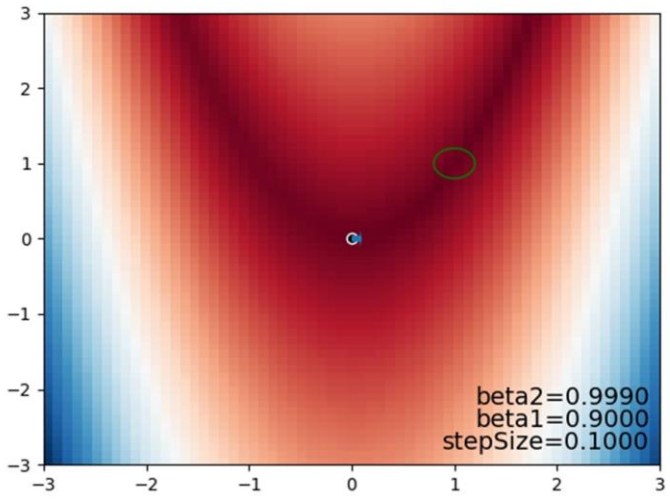
# Adam: modern momentum method




The beta2 defines how slowly the gradient magnitude estimate adapts. The default recommended value is 0.999, which is however for training very large neural networks with many iterations and a small time step.

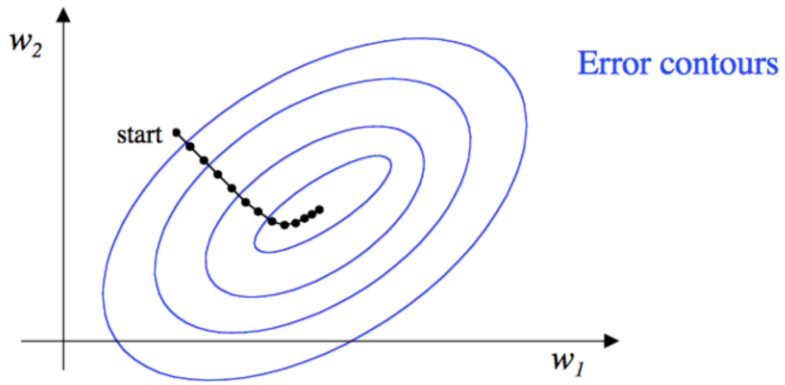
In this kind of a simple function where we want to reach the optimum in just a few hundred iterations with a large step size, beta2 must be decreased, especially if the initialization is far from the optimum. Here, the gradient magnitude is initially very high, which means the gradient is scaled down aggressively. Thus, when Adam reaches the Rosenbrock valley, it stops until the scaling adapts. With the default large beta2 on the right, this takes a long time.

# Adam: modern momentum method




However, if gradient magnitude at the initial point is already close to gradient near the optimum, the default large beta2=0.999 works fine.

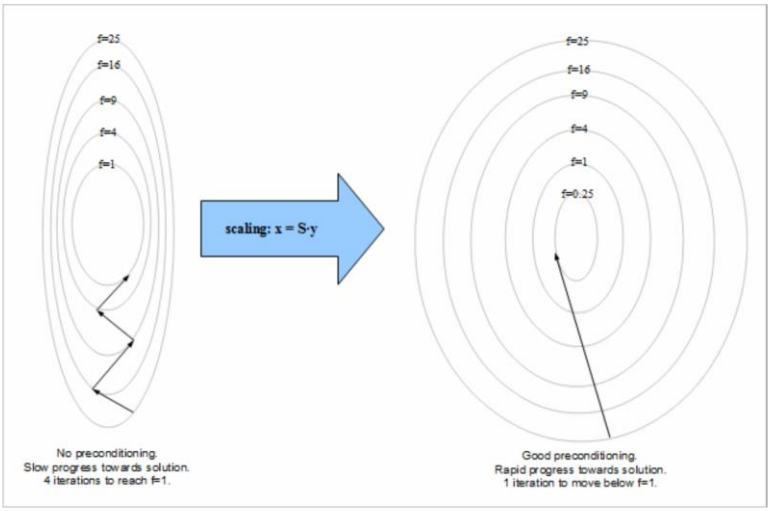
## Problem: Gradient does not often point to the optimum (even in convex optimization)




In general, the fundamental problem plaguing Adam and other gradient-based methods in Rosenbrock and many other optimization problems is that the gradient does not point towards the optimum.

Gradient is perpendicular to the isocontours.

# Solution: Preconditioning




Preconditioning means transforming the search space in a way that makes the gradient point to the optimum. Here, an example preconditioner. In practice, hard to find suitable transforms.

Adam's per-variable scaling acts as a preconditioner. The function on the left is more sensitive to changes in x than changes in y, which makes the adaptive scaling result in smaller steps along x axis than y axis, effectively transforming the function contours to be more spherical and correcting the gradient direction towards the optimum instead of towards the y axis.

# Curvature

- Gradient descend: form a 1st order Taylor expansion, i.e., linear approximation of $f(\mathbf{x})$, use that to determine search direction
- Newton's method: a 2nd order model using 2nd derivatives
- If $f(\mathbf{x})$ is quadratic, model fits perfectly and directly gives the optimum

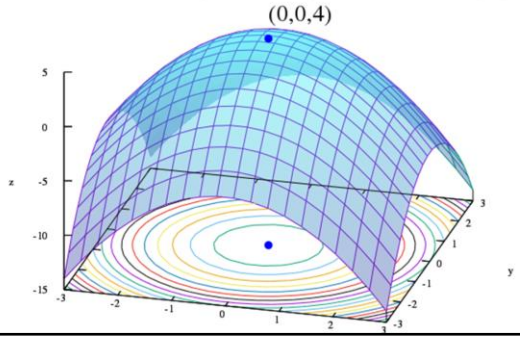


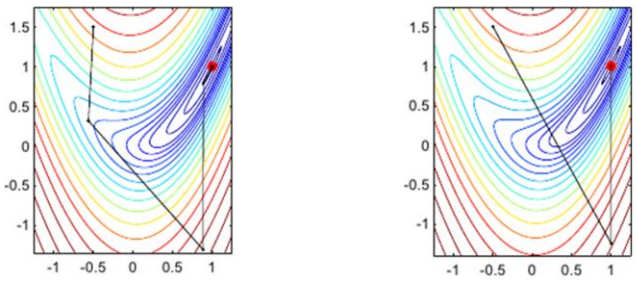

Image by IkamusumeFan - Own work, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=42043175

An alternative to preconditioning is to use curvature info. On a quadratic $f(\mathbf{x})=\mathbf{x}'A\mathbf{x}+\mathbf{x}'\mathbf{b}$, Newton's method finds the minimum in one step.

The image shows a quadratic surface, which is fully defined by the first and second derivatives (gradient vector and Hessian matrix) at any point. Thus, the gradient and Hessian are enough to find the maximum or minimum.

## Curvature

- Newton's method requires the Hessian matrix of second derivatives, requires N^2 function evaluations and memory
- BFGS, L-BFGS: approximate Hessian indirectly
- If $f(\mathbf{x})$ is sum of squares: Gauss-Newton, Levenberg-Marquardt




Images: http://www.brnt.eu/phd/node10.html

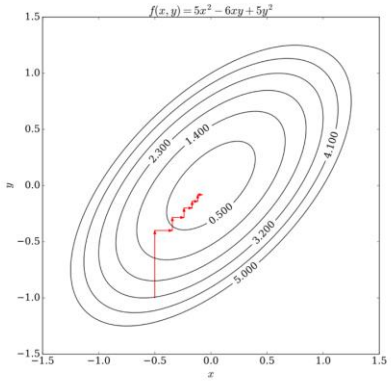
Left: Newton's method on Rosenbrock

Right: Gauss-Newton on Rosenbrock (possible, since the function can be expressed as sum of squares, see http://www.brnt.eu/phd/node10.html)

Levenberg-Marquardt is a regularized or "trust region" version of Gauss-Newton. Both methods approximate the Hessian matrix as $\mathbf{H}=2\mathbf{J}'\mathbf{J}$, where $\mathbf{J}$ is the Jacobian matrix of first partial derivatives. They can also be thought of as *multiobjective gradient descend*: $f(\mathbf{x})$ is assumed to be of form $f(\mathbf{x})=\sum_i(r_i^2(\mathbf{x}))$, and the $i$:th row of $\mathbf{J}$ corresponds to the gradient vector $\nabla r_i(\mathbf{x})$. The multiple objectives and gradient vectors give the algorithms more information to use at each iteration, leading to faster convergence.

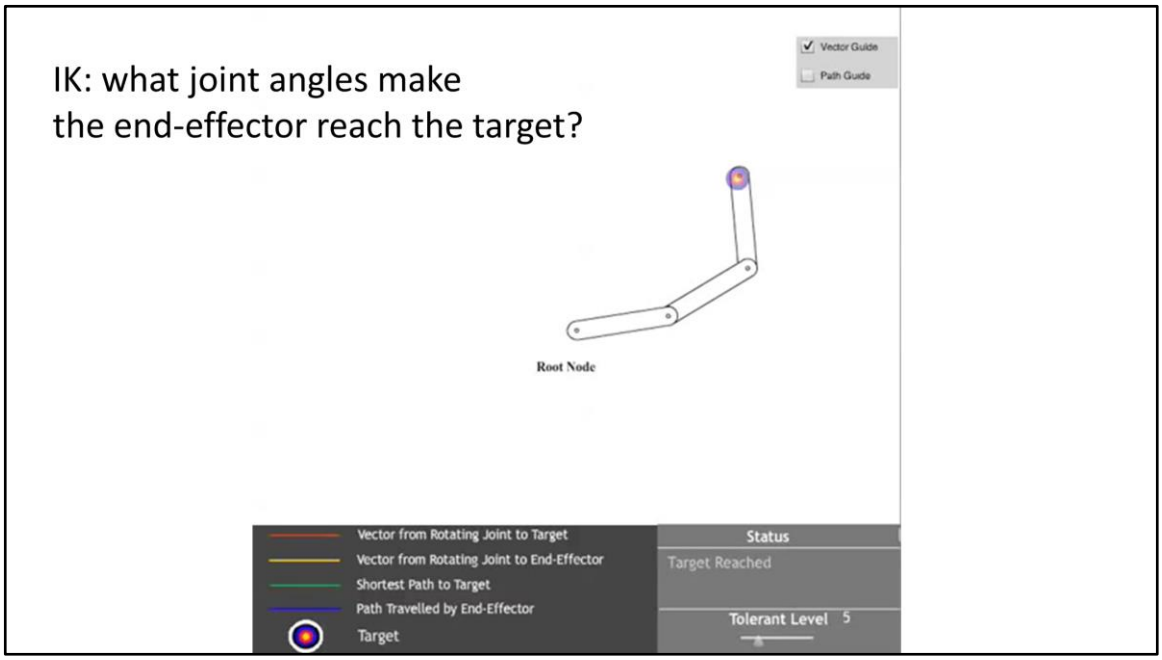
## Cyclic coordinate descend

- Move along one axis at a time until at optimum
- A simple and efficient optimization method for inverse kinematics (IK)




The ”until at optimum” can be implemented iteratively in gradient descend manner.

Sometimes, however, it’s possible to use curvature or otherwise solve the optimum for a single variable at a time, for example, in inverse kinematics.

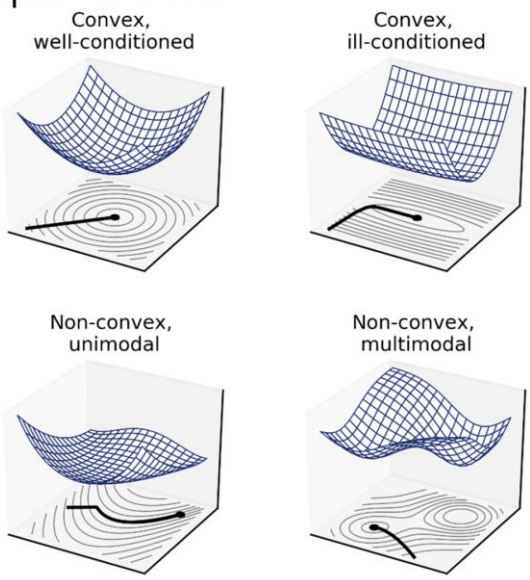



https://www.youtube.com/watch?v=MvuO9ZHGr6k

CCD is particularly suitable for IK because the optimal rotation for a single axis can be solved with basic trigonometry in a single step (just rotate by the angle between the joint-effector and joint-target vectors), although in this video the algorithm takes many small steps until switching to another joint

Note that as IK objective function is typically of the sum-of-squares form (sum of squared differences between x,y,z coordinates of the end effector and target), Gauss-Newton is another common optimization method, usually called ”Jacobian Pseudo-Inverse” in the animation literature. However, CCD is easier to implement as no matrix operations are needed.

# Difficulty of optimization

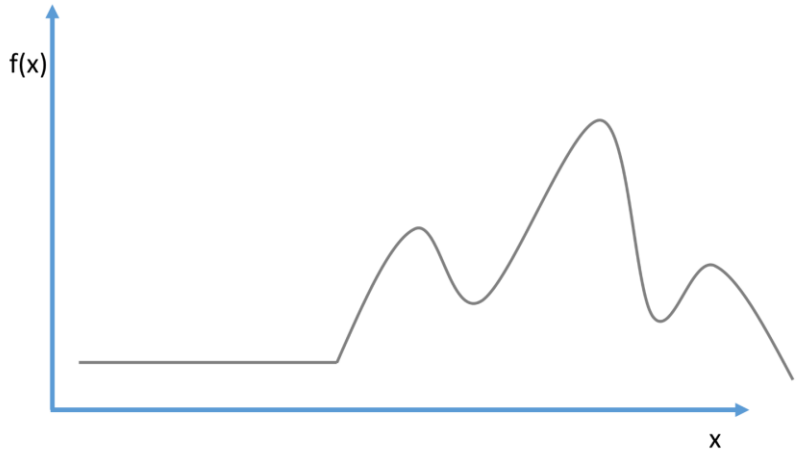



Four classes of optimization problems, with difficulty growing from left to right and top to bottom. The black trajectories show the progress of gradient descend from some initialization.

The convex vs. non-convex name comes from the shape of the isocontours.

Gradient-based methods work well on convex problems, and often also on non-convex ones. However, multimodality makes things even more difficult.

## Real-life problems: multimodality, absence of gradients

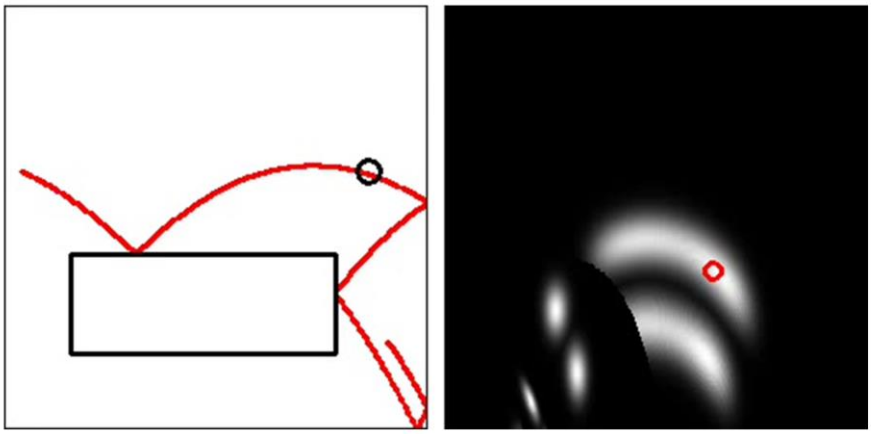



Indeed, in real-life problems, one often encounters both multimodality and flat regions. Gradient info does not help if in the wrong peak or at a flat region

## Optimization and game design

- A/B testing does not use gradients or momentum – direction of improvement is guessed by the designer
- Evaluating $f(\mathbf{x})$ with players is costly
- If $f(\mathbf{x})$ can be evaluated with AI players, computational optimization becomes feasible
- Even then, $f(\mathbf{x})$ may be noisy, have sharp edges etc. that gradien-based methods can't handle

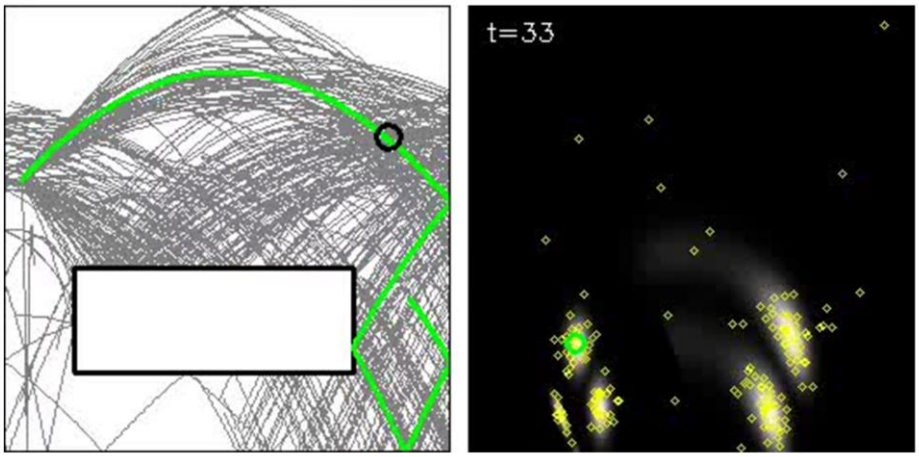
## Real-life problems: multimodality, absence of gradients



One reason why one encounters these problems in game development problems is physics, in particular contacts.

Note that even in the somewhat smooth white ridges, there's jagged microstructure caused by the discretization of the physics simulation. This makes optimization difficult for gradient-based methods, especially without momentum.

## Population-based methods to the rescue




Fortunately, there are many so-called population-based optimization methods that do not need derivatives at all. Instead, they operate on sets or populations of samples or ”particles”, i.e., random vectors of optimized parameters $\mathbf{x}^{(i)}$, where the superscript typically denotes sample index in the population. The objective function $f()$ is evaluated for each $\mathbf{x}^{(i)}$, and the function values are used to inform how to evolve the population.

In the figure & video, each sample corresponds to a trajectory on the left, and a point in the speed,angle –space on the right. The green trajectory corresponds to the green circle.

As shown on the video, the sampling distribution can be adapted such that the samples gravitate towards function peaks, even if there are multiple peaks that are shifting over time. Here, this is because the target is moved between iterations. The target is denoted by the black circle on the left.

Source: Hämäläinen, P., Eriksson, S., Tanskanen, E., Kyrki, V., & Lehtinen, J. (2014). Online motion synthesis using sequential monte carlo. *ACM Transactions on Graphics (TOG)*, *33*(4), 51.

# Multimodal vs unimodal

- Multimodal: find many peaks/valleys of $f(\mathbf{x})$
- Used for 1) presenting user with choices, 2) robust tracking of changing landscape
- High-dimensional problems => exponentially many peaks => finding all not feasible
- A good unimodal population-based technique first explores many peaks, gradually zooms in on the most prominent one

## CMA-ES (Hansen & Ostenmeier 2001)

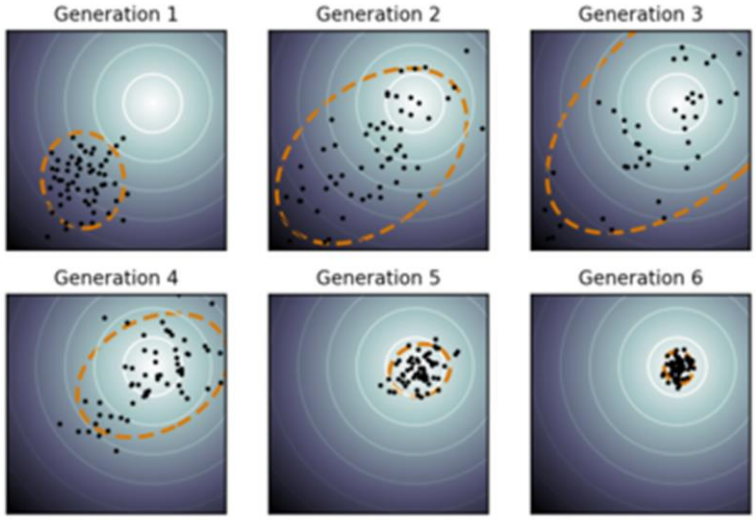



Image source: https://en.wikipedia.org/wiki/CMA-ES

For many, the unimodal method of choice is CMA-ES. It and its recent variants work up to thousands or even millions of optimized variables.

The name stands for Covariance-Matrix Adaptation Evolution Strategy.

The inventor says: ” The CMA-ES is typically applied to unconstrained or bounded constraint optimization problems, and search space dimensions between three and a hundred. The method should be applied, if derivative based methods, e.g. quasi-Newton BFGS or conjugate gradient, (supposedly) fail due to a rugged search landscape (e.g. discontinuities, sharp bends or ridges, noise, local optima, outliers). If second order derivative based methods are successful, they are usually faster than the CMA-ES: on purely convex-quadratic functions, $f(x)=x^{T}Hx$, BFGS (Matlabs function fminunc) is typically faster by a factor of about ten (in terms of number of objective function evaluations needed to reach a target function value, assuming that gradients are not available). On the most simple quadratic function $f(x)=||x||^{2}=x^{T}x$ BFGS is faster by a factor of about 30.

Similar to quasi-Newton methods (but not inspired by them), the CMA-ES is a **second order** approach estimating a positive definite matrix within an iterative procedure (more precisely: a covariance matrix, that is, *on convex-quadratic functions*, closely related to the inverse Hessian). This makes the method feasible on non-separable and/or badly conditioned problems. In contrast to quasi-Newton methods, the CMA-ES does not use or approximate gradients and does not even presume or require their existence. This makes the method feasible on **non-smooth** and even non-continuous problems, as well as on multimodal and/or noisy problems. It turns out to be a particularly reliable and highly competitive evolutionary algorithm for local optimization (Hansen & Ostermeier 2001) and, surprising at first sight, also for global optimization (Hansen & Kern 2004, CEC 2005, Hansen 2009 in BBOB-2009). "

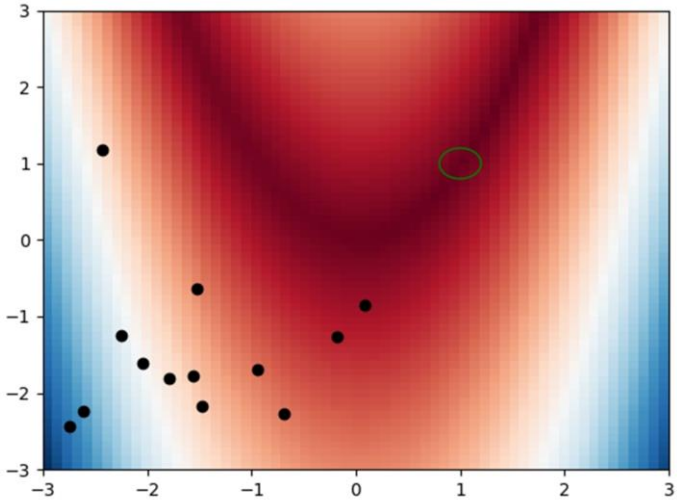
Source: https://www.lri.fr/~hansen/cmaesintro.html

In a way, we go back to the beginning, where we randomly sampled around the current solution. However, in CMA-ES, we produce multiple samples per iteration and perform statistical analysis that helps us determine good sampling directions for the next iteration. This is crucial for high-dimensional difficult problems one encounters in practice. CMA-ES solidly outperforms more naive approaches like Simulated Annealing with spherical Gaussian mutations.

## Population-based methods to the rescue

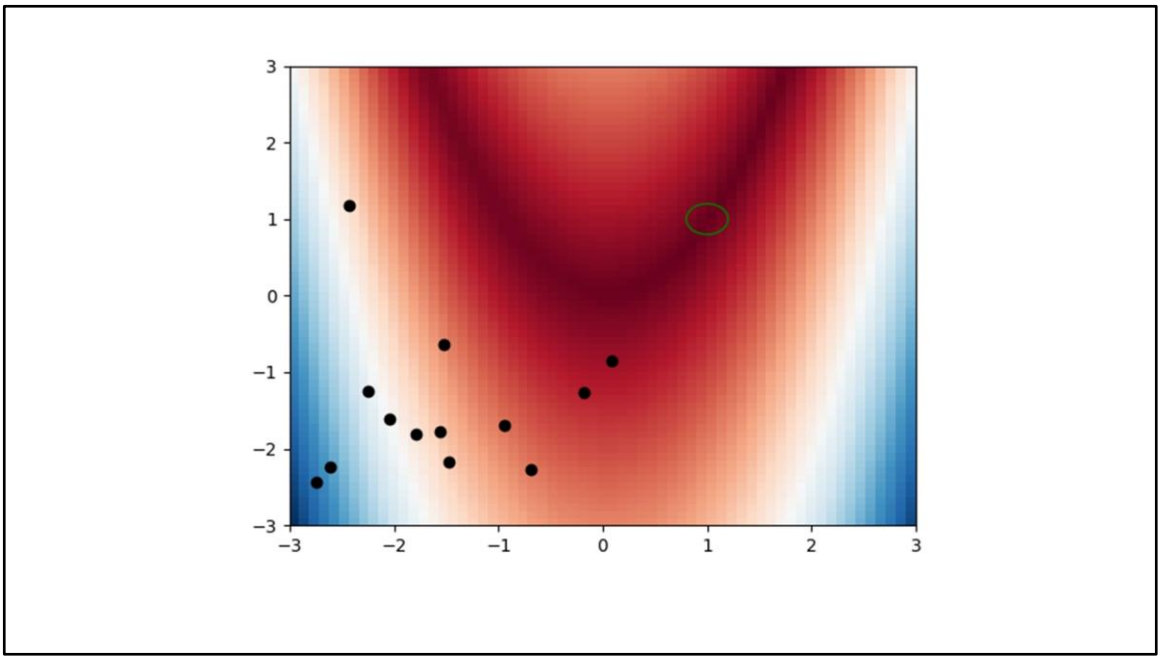
- Initialize population
- Repeat: Mutate, Recombine, Prune
- Historical: Genetic algorithms, Simulated Annealing
- Modern: parameteric evolution strategies: CMA-ES and its variants such as LM-MA-ES
- Benefit: Investigate multiple locations at once, gradually narrow down the search

# CMA-ES

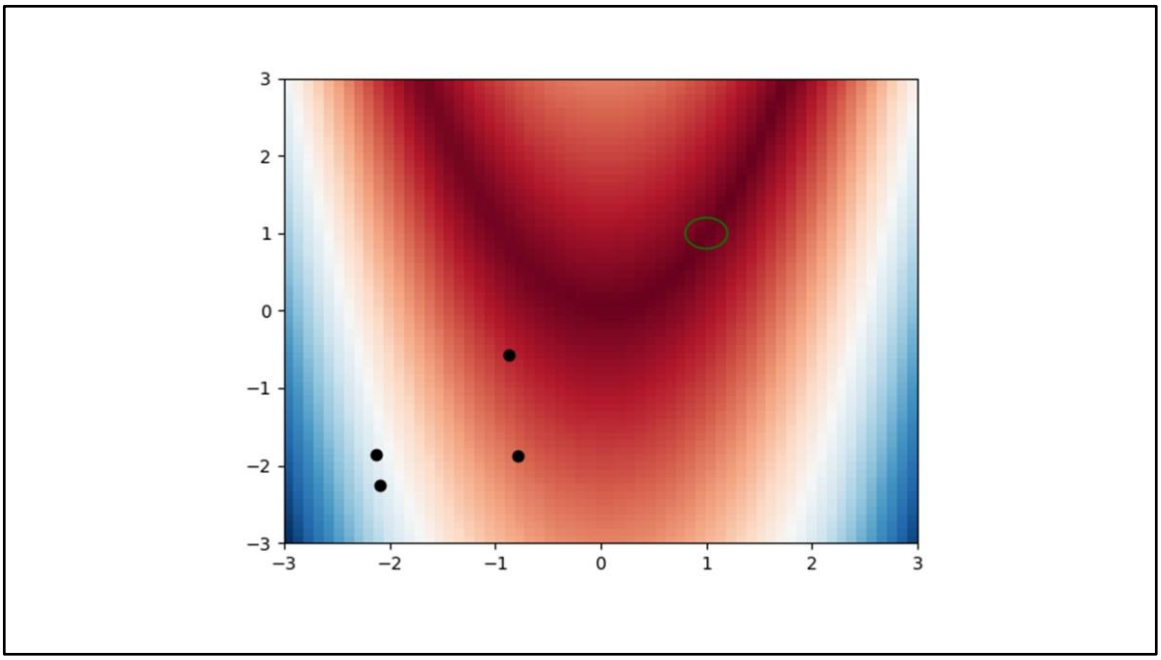


# CMA-ES

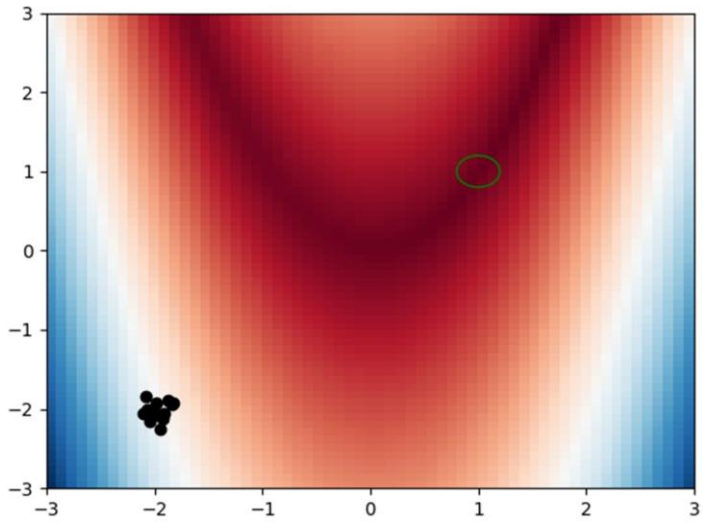
- $\mathbf{x}_{new}$ are sampled by perturbing current x along directions that have previously resulted in improvement
  - Assumes some local smoothness of $f(\mathbf{x})$
  - A bit like importance-sampling: noise of gradient estimate is reduced event with a small sampling budget per step
- Overall sampling variance is adapted – if we keep consistently finding better $f(\mathbf{x})$ in the same direction, we're probably advancing along a smooth region => can try to sample further ahead
  - Conversely, if the direction of advancement is constantly changing, we are probably zigzagging back and forth around an optima => must reduce the search radius to find the exact peak or valley
  - Somewhat similar to momentum & step-size adaptation




Again

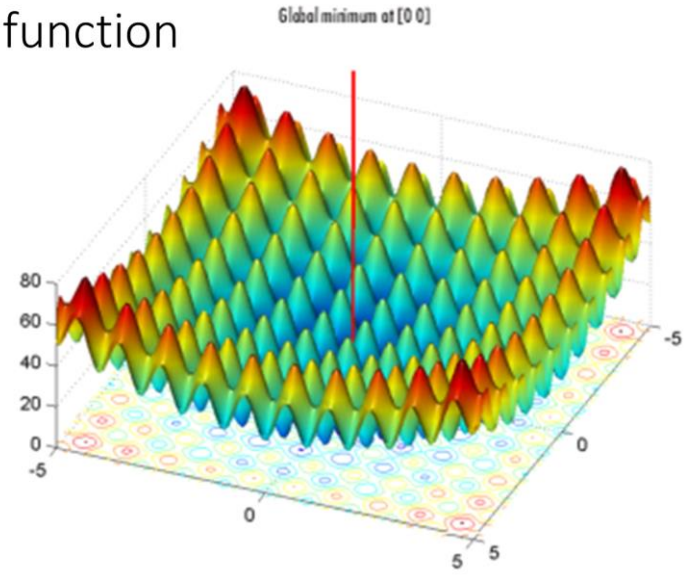



Smaller population (6)

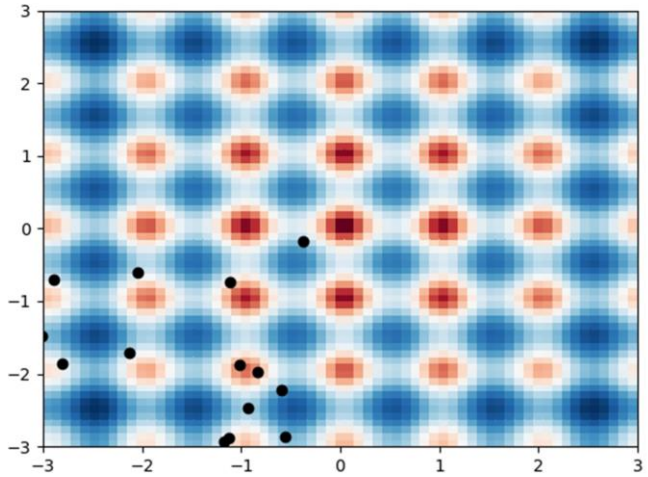



Usually, the initial sampling variance should be large enough such that the true optimum is within a few standard deviations of the mean. However, if the function landscape is smooth, CMA-ES recovers well from bad initialization.

# Rastrigin function

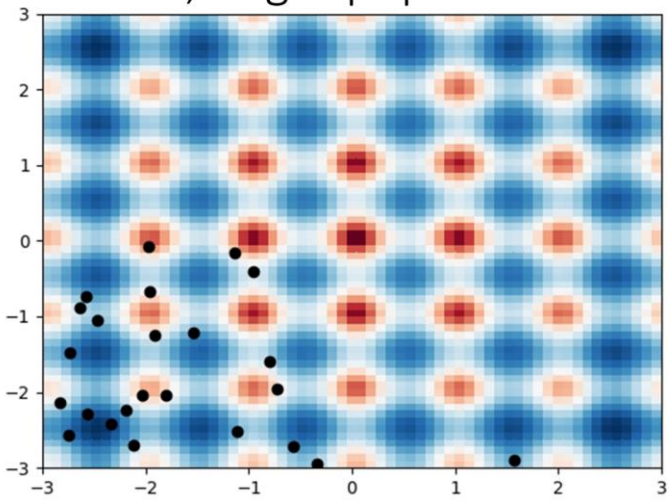


## Rastrigin function




Population size 16

## Rastrigin function, larger population size




Population size 32

# The CMA Evolution Strategy: A Tutorial

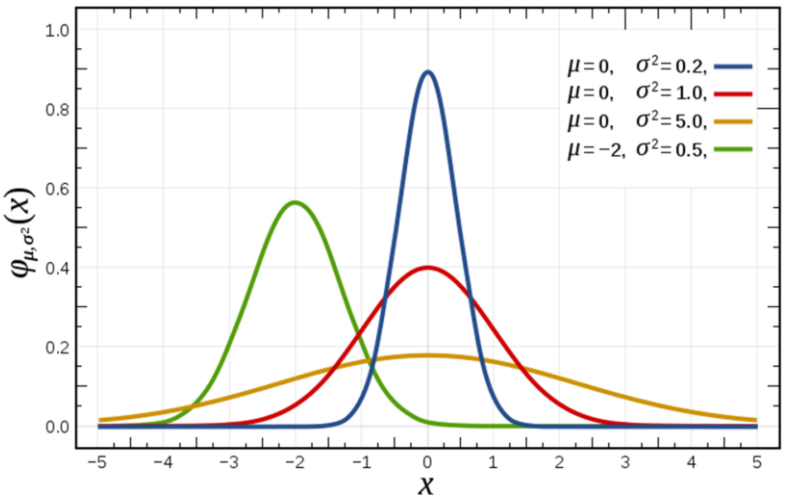
Nikolaus Hansen
Inria
Research centre Saclay–Île-de-France
Université Paris-Saclay, LRI

## Contents



This is the paper to read if one wants to understand CMA better

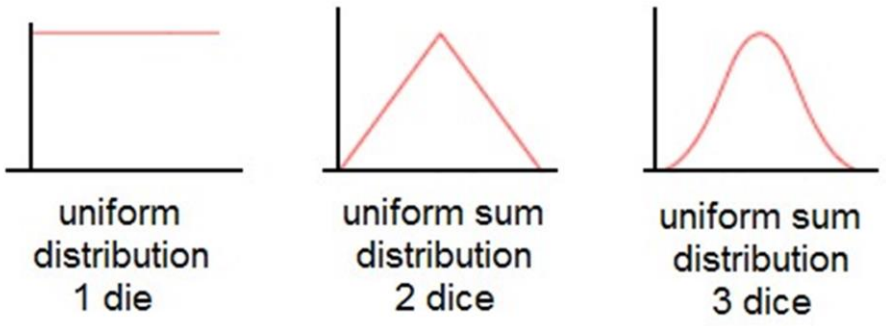
# Preliminaries: normal (Gaussian) distribution




https://en.wikipedia.org/wiki/Normal_distribution

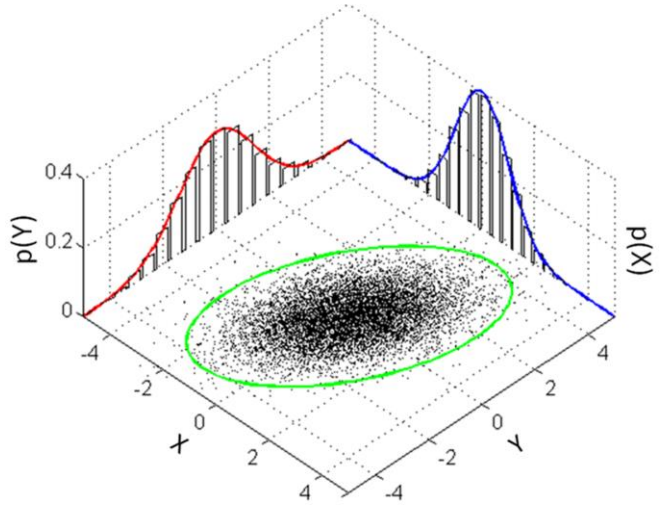
Defined by a mean μ and standard deviation σ, or variance $\sigma^2$

## Preliminaries: normal (Gaussian) distribution




Relevant to games, as the sum of many random numbers such as dice throw results are (close to) normally distributed. This property is mathematically defined as the Central Limit Theorem.

## Preliminaries: normal (Gaussian) distribution



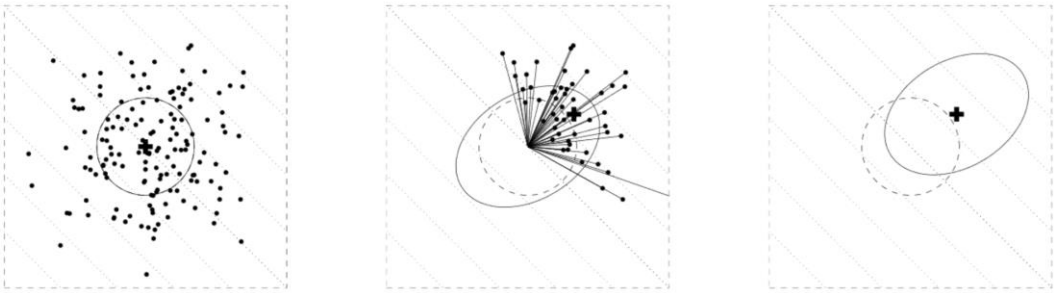

https://en.wikipedia.org/wiki/Multivariate_normal_distribution

With more variables, the distribution is defined with a mean vector **μ** of length N and a covariance matrix **Σ** of size N-by-N. The region with 1 standard deviation is usually illustrated as an ellipse, or a hyperellipsoid in high-dimensional cases. The shape and scale of the ellipse is defined by the covariance matrix, centered at the mean.

In terms of linear algebra, the axes of the ellipse equal the eigenvectors of the covariance matrix.

Both the marginal and conditional distributions of single variables are one-dimensional normal distributions, which comes handy in probabilistic inference and derivations, which we do not go into on this course.

## CMA-ES in a nutshell



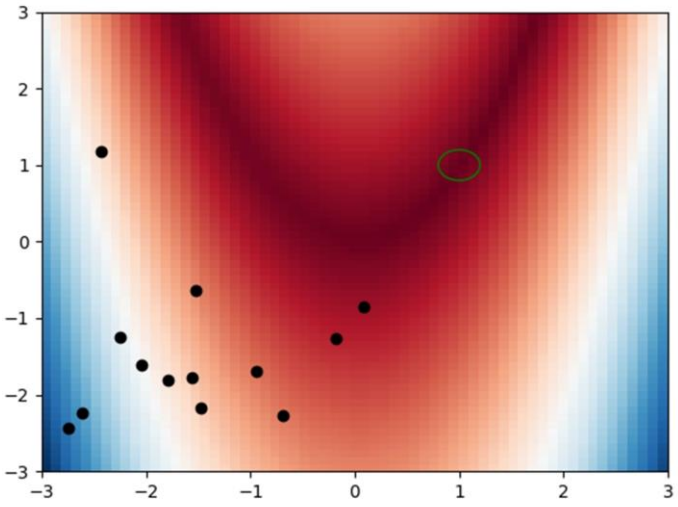
Hansen's illustration of CMA-ES on a $f(\mathbf{x})$ that grows towards the top-right corner.

Left: initial normal distribution (standard deviation denoted by the circle) and samples drawn from it.

Middle: samples are sorted based on $f(\mathbf{x})$ and 50% of worst samples are pruned. The cross marks the mean of the non-pruned elite samples. The circle marks the normal distribution that approximates the elite samples if the mean is not yet moved from the original sampling mean.

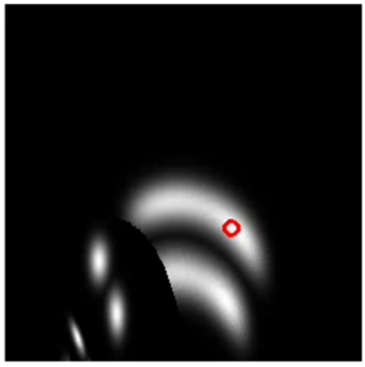
Right: the next iteration's sampling distribution uses the new normal distribution, centered at the mean of the elite samples.

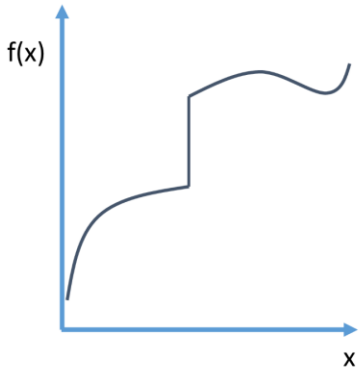
In practice, CMA does not update the mean and covariance fully at each iteration, but smooths the changes over many iterations. This allows the method to work robustly even with a small population size.

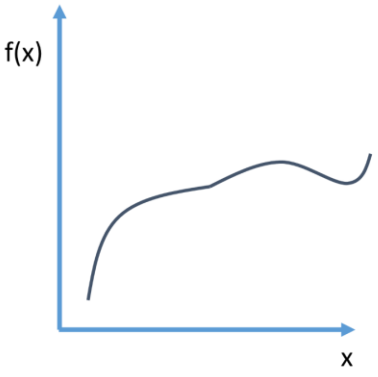



Let’s again see this in action over many iterations.

## Invariance to order-preserving transformations






A nice thing about CMA-ES is that it only cares about the rank of samples when sorted by objective function value. This means that the two functions are equal from the point of view of CMA-ES, and discontinuities with infinite gradients do not cause problems, unlike with gradient-based methods (momentum methods are particularly sensitive to these).

## CMA-ES has been reinvented many times

- Cross-Entropy Method
- Estimation of Multivariate Normal Algorithm (EMNA)
- However, there are also differences and CMA-ES is more sophisticated and robust

EMNA is a special case of what is known as the family of Estimation of Distribution Algorithm.

# Simplify Your Covariance Matrix Adaptation Evolution Strategy

Hans-Georg Beyer and Bernhard Sendhoff *Senior Member, IEEE*

***Abstract*—The standard Covariance Matrix Adaptation Evolution Strategy (CMA-ES) comprises two evolution paths, one for the learning of the mutation strength and one for the rank-1 update of the covariance matrix. In this paper it is shown that one can approximately transform this algorithm in such a manner that one of the evolution paths and the covariance matrix itself disappear. That is, the covariance update and the covariance matrix square root operations are no longer needed in this novel so-called Matrix Adaptation (MA) ES. The MA-ES performs nearly as well as the original CMA-ES. This is shown by empirical investigations considering the evolution dynamics and the empirical expected runtime on a set of standard test functions. Furthermore, it is shown that the MA-ES can be used as search engine in a Bi-Population (BiPop) ES. The resulting BiPop-MA-ES is benchmarked using the BBOB COCO framework and compared with the performance of the CMA-ES-v3.61 production code. It is shown that this new BiPop-MA-ES – while algorithmically simpler – performs nearly equally well as the CMA-ES-v3.61 code.**

***Index Terms*—Matrix Adaptation Evolution Strategies, Black Box Optimization Benchmarking.**

call the algorithm's peculiarities, such as the covariance matrix adaptation, the evolution path and the cumulative step-size adaptation (CSA) into question. Notably in [4] a first attempt has been made to replace the CSA mutation strength control by the classical mutative self-adaptation. While getting rid of any kind of evolution path statistics and thus yielding a much simpler strategy[1], the resulting rank-$\mu$ update strategy, the so-called Covariance Matrix Self-Adaptation CMSA (CMSA-ES) does not fully reach the performance of the original CMA-ES in the case of small population sizes. Furthermore, some of the reported performance advantages in [4] were due to a wrongly implemented stalling of the covariance matrix update in the CMA-ES implementation used.

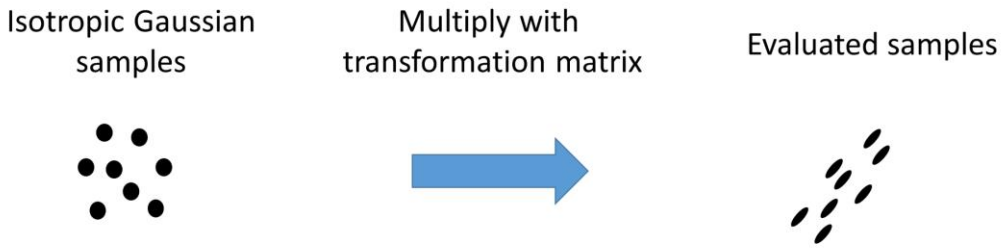
Meanwhile, our theoretical understanding of the evolution dynamics taking place in CMSA- and CMA-ES has advanced. Basically two reasons for the superior CMA-ES performance can be identified in the case of small population sizes:

- *Concentrate evolution along a predicted direction.*

CMA-ES has been popular for many years. In 2017, an interesting simplified version called MA-ES was introduced.

## MA-ES (Beyer and Sendhoff 2017)

- Original CMA-ES memory requirements is $O(N^2)$, computation cost $O(N^3)$
- MA-ES is a simplification that performs roughly as well (Beyer & Sendhoff 2017), $O(N^2 \log(N))$




N is the number of optimized variables, i.e., problem dimensionality. Note that the log(N) complexity only applies if one uses the recommended population size, i.e., 4+floor(3*log(N))

MAES main difference: instead of estimating covariance matrix and computing its inverse, simply use a transformation matrix that rotates and scales the population of samples that is originally drawn from an isotropic Gaussian distribution. The matrix is defined such that it can be updated without matrix inversion, only using matrix-vector multiplication and addition.

In effect, the transformation matrix equals the matrix of eigenvectors of the covariance matrix, but it turns out the eigenvectors can be inferred directly, and computing and inverting the covariance matrix is redundant.

# Limited-Memory Matrix Adaptation for Large Scale Black-box Optimization

**Ilya Loshchilov**
Research Group on Machine Learning
for Automated Algorithm Design
University of Freiburg, Germany
ilya.loshchilov@gmail.com

**Tobias Glasmachers**
Institut für Neuroinformatik
Ruhr-Universität Bochum, Germany
tobias.glasmachers@ini.rub.de

**Hans-Georg Beyer**
Research Center Process and Product Engineering
Vorarlberg University of Applied Sciences, Dornbirn, Austria
hans-georg.beyer@fhv.at

## Abstract

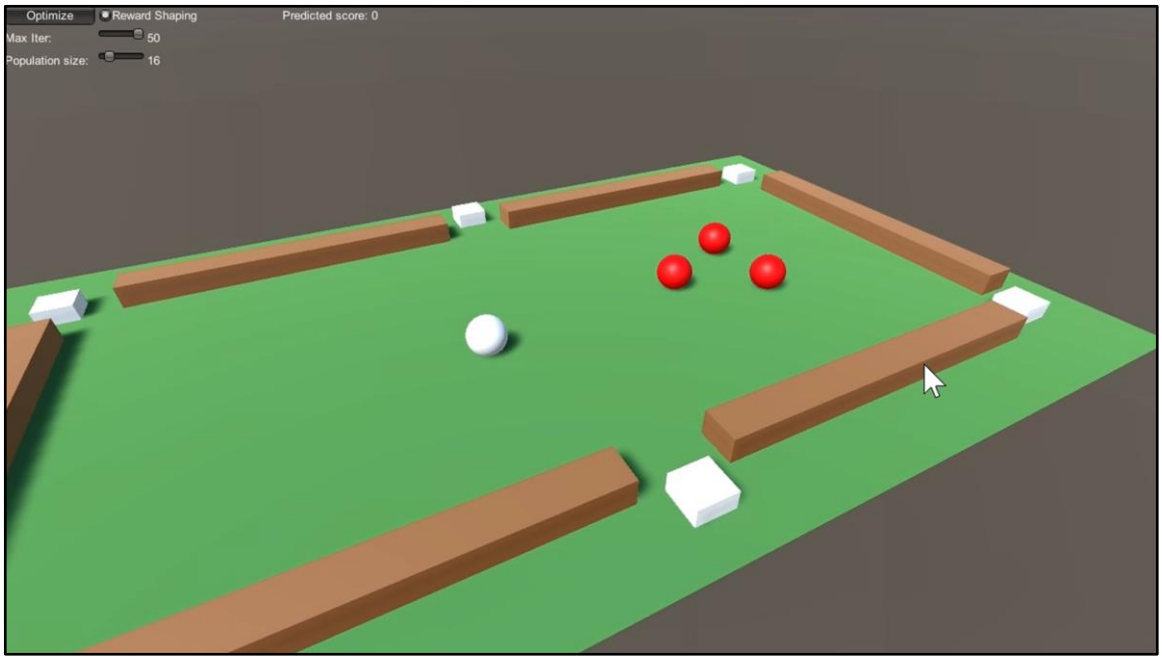
The Covariance Matrix Adaptation Evolution Strategy (CMA-ES) is a popular method to deal with nonconvex and/or stochastic optimization problems when the gradient information is not available. Being based on the CMA-ES, the recently proposed Matrix Adaptation Evolution Strategy (MA-ES) provides a rather surprising result that the covariance matrix and all associated operations (e.g., potentially unstable eigendecomposition) can be replaced in the CMA-ES by a updated transformation matrix without any loss of performance. In order to further simplify MA-ES and reduce its $\mathcal{O}(n^2)$ time and storage complexity to $\mathcal{O}(n \log(n))$, we present the Limited-Memory Matrix Adaptation Evolution Strategy (LM-MA-ES) for efficient zeroth order large-scale optimization. The algorithm demonstrates state-of-the-art performance on a set of established large-scale benchmarks. We explore the algorithm on the problem of generating adversarial inputs for a (non-smooth) random forest classifier, demonstrating a surprising vulnerability of the classifier.

https://arxiv.org/abs/1705.06693

## LM-MA-ES (Loschilov et al. 2017)

- Faster MA-ES, also limited memory variant (LM-MA-ES), which scales to millions of variables, O($N$ log($N$))
- LM-MA-ES works even in generating adversarial images, something that had previously only been done with gradient-based optimization
- Unity implementation provided in the course repository
- Loschilov recommends: for N<100, use MA-ES, otherwise try LM-MA-ES

LM-MA-ES uses a low-rank transformation matrix represented as a set of vectors, with the default number of vectors proportional to log(N).




Finally, we get back to this teaser, which is a simple demo project of using MAES in Unity.

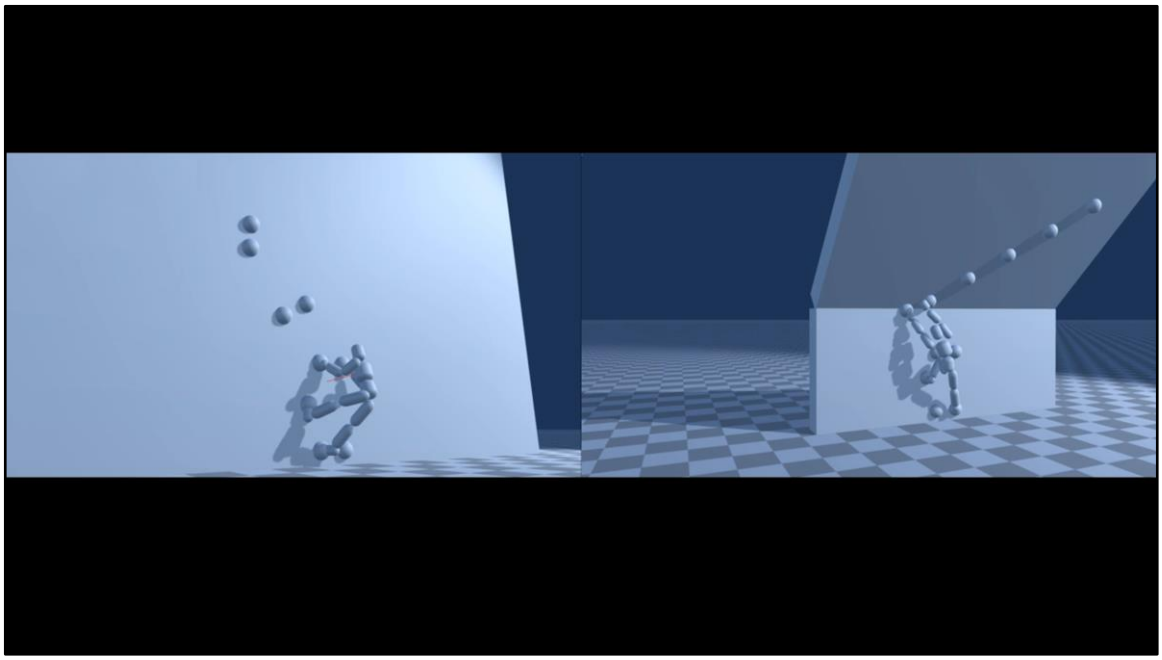
## MAES Unity interface

```
public interface IMAES
{
    int recommendedPopulationSize(int dim);
    void init(int dim, int populationSize, double[] initialMean, float init
    void generateSamples(OptimizationSample[] samples, int nInitialGuesses
    double update(OptimizationSample[] samples);
    double[] getBest();
    double getBestObjectiveFuncValue();
    void optimize(Func<double[], int, double> objectiveFunc, int maxIter);
}
```

Plain C# implementation, no math or machine learning libraries needed.

To let you experiment with CMA-ES or more specifically, the MA-ES variant, I've ported Ilya Loschilov's code to Unity and C#.

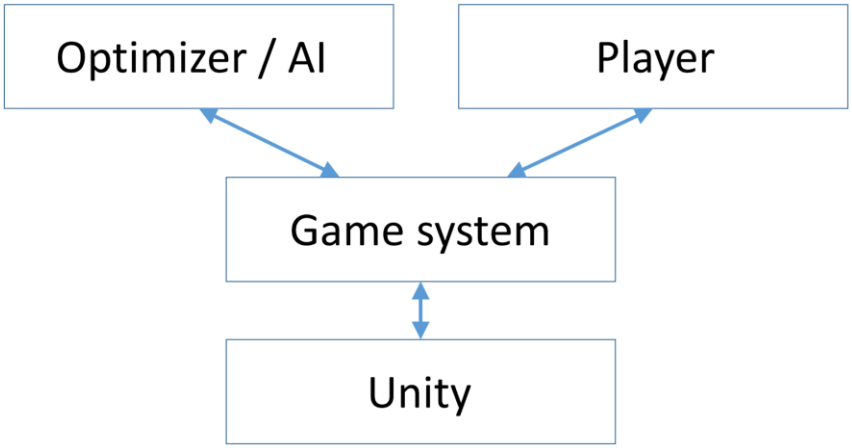
The MAES code does not require any fancy math libraries and it even works with the old .NET 3.5. It's just one C# file of few hundred lines. Thus, no penalty on executable size or portability.



The billiards case is of course only two-dimensional. However, we've successfully used CMA-ES in much more difficult cases previously.

In this climber paper, we utilized CMA-ES as the body controller that executes climbing actions planned using a graph search approach (more about that in the next lecture).

## Design pattern: AI and Player as game system controllers

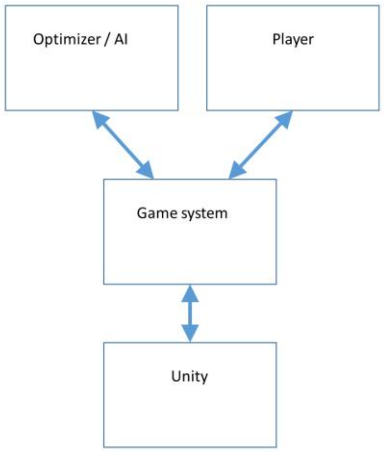



Some remarks about software architecture design for AI or optimization.

Typically, it makes sense to adopt an architecture where the game system is separated from its manipulations and visualizations. Both player and AI often need the same game system API, but use it in a different manner and provide different GUI and visualizations for the player or developer.

## Game system

- Implement actions
- Simulate actions speculatively (for evaluating $f(\mathbf{x})$ without affecting system state)
- Save & restore state (needed for the above)
- The speculative simulation may also be useful for the player!




The most important functionality needed by both AI and player


Here’s an example of using speculative actions and their debug visualization for the player. Although this might make the game too trivial, it may be useful during development and testing.

More about this:

# IntelligentPool code walkthrough...

Let’s take a look at the project code (note to self: duplicate screen...)

GameSystem.cs: the game system
Player.cs: the human interface
ShotOptimization.cs: the optimizer and its UI

## Reward shaping

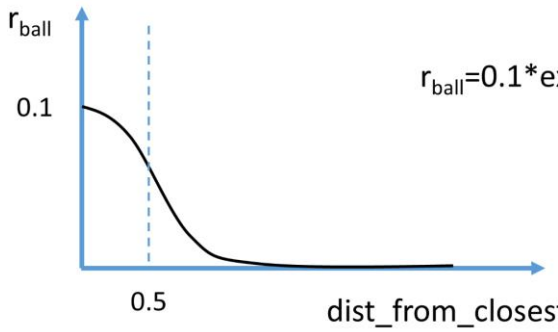
- Simply giving a +1 score for each pocketed ball => mostly flat $f(\mathbf{x})$
- Reward shaping, proxy objective: give a small bonus for balls that are close to pockets at the end of the shot

Often, the objective function is simply too difficult to optimize, for example, due to large flat regions that provide no information about where to explore next. In this case, one may have to resort to using a proxy objective function that is not strictly correct, but much easier to optimize. In reinforcement learning, this is known as ”reward shaping”.

In the pool case, one can try to make the optimizer favor shots where at least some balls end up close to the pockets. After such shots have been found, there’s a high probability that small perturbations of **x** result in pocketing balls.

## Reward shaping

- Simply giving a +1 score for each pocketed ball => mostly flat $f(\mathbf{x})$
- Reward shaping, proxy objective: give a small bonus for balls that are close to pockets at the end of the shot




$r_{ball}=0.1*\exp(-0.5*dist\_from\_closest^2/0.5^2)$

0.1 = a constant so much smaller than 1 that pocketing more balls is always better, no matter the bonus

0.5 = a tolerance value. If dist_from_closest is larger than this, $r_{ball}$ decreases rapidly towards zero. One can also think of this as a standard deviation of a normal distribution with a similar (unnormalized) probability density function.

Some form of reward shaping or proxy rewards or objective functions are widely used to make optimization and machine learning problems easier for algorithms. A bit similar to preconditioning, but if done wrong, can actually lead the algoritms converge somewhere else than the real optimum.

## Parallel computing

- Population-based methods are usually trivially parallelizable
- Threading overhead negligible if $f(\mathbf{x})$ evaluation is costly
- However, physics simulators typically not thread-safe
- Unity: physics simulation automatically parallelized if groups of objects not interacting (not used in the billiards demo)

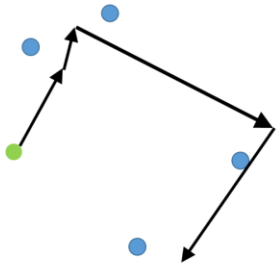
Parallelizing would require multiple copies of the table and balls.

## Discrete-valued optimization

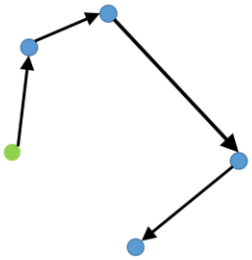
- The examples so far assume continuous-valued **x**
- CMA-ES and gradient-based methods don't work on discrete/combinatorial problems (e.g., typical game puzzles)
- Approach 1: Brute-forcing (try all combinations)
- Approach 2: Parameterize as continuous-valued problem
- Approach 3: Discrete optimization methods, e.g., PBIL, Genetic Algorithms, Simulated Annealing

Won't go into details on approach 3, but will help you if you have a discrete optimization problem in your projects.

## Parameterizing a combinatorial problem as continuous-valued: the traveling salesman



City sequence as continous-valued x,y coordinates



For evaluating $f(\mathbf{x})$, map coordinates to closest cities

The function landscape will have flat gradient regions, but CMA-ES still works much better than brute-forcing. It will naturally produce behavior where the algorithm first gets clusters of cities right, and then iterates within the clusters.

The point here is that similar coordinates (i.e., values for **x**) yield similar objective function values, i.e., there's structure that optimization methods can utilize. For example, this means that if some **x** is particularly good, more good values can be found in the vicinity.

```
#Choosing an optimization algorithm
if N>100:
  algorithm=LM-MA-ES
else:
  algorithm=MA-ES

#The following can be ignored if (LM-)MA-ES gives good enough results
if gradient can be computed:
  if f(x) is smooth and unimodal:
    if no optimization or linear algebra libraries, or limited memory:
      algorithm=Adam
    else:
      if f(x) is formulated as a sum of squares:
        algorithm=Levenberg-Marquardt  #i.e., regularized Gauss-Newton
      else:
        if Hessian matrix can be computed:
          algorithm=Newton
        else:
          algorithm=L-BFGS
```

To conclude, here's my heuristics for choosing optimization algorithm.

N denotes the dimensionality of **x**, and we assume continuous-valued **x**.

Adam is widely used in optimizing neural network models, because the models are so large that other methods need too much memory.

For discrete-valued problems, I would first try to think of a way to convert it to a continuous-valued one, and if that fails, try PBIL, which is the combinatorial equivalent of CMA-ES.

# Exercise: CMA-ES for Abstract Neural Adversarial Images

- **x** contains the parameters of a painting composed of primitives like rectangles (parameters: corner coordinates, color)
- $f(\mathbf{x})$ is the target class probability of a convolutional image classifier network, similar to what we trained in the day 1 exercise.
- Bonus: optimize the adversarial images pixel-by-pixel using Tensorflow's Adam
- Jupyter notebooks: CMA-ES_Art.ipynb, AdversarialMNIST.ipynb