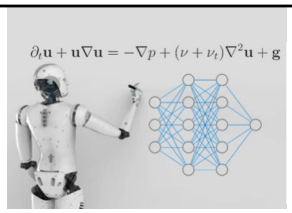


Hands on Machine Learning
for Fluid Dynamics

7 – 11 February 2022

von KARMAN INSTITUTE
FOR FLUID DYNAMICS

Lecture 6
Bio-Inspired Optimization: Genetic Algorithms and Particle Swarms

Miguel Alfonso Mendez
mendez@vki.ac.be

Slide 1

Python Files for this Lecture

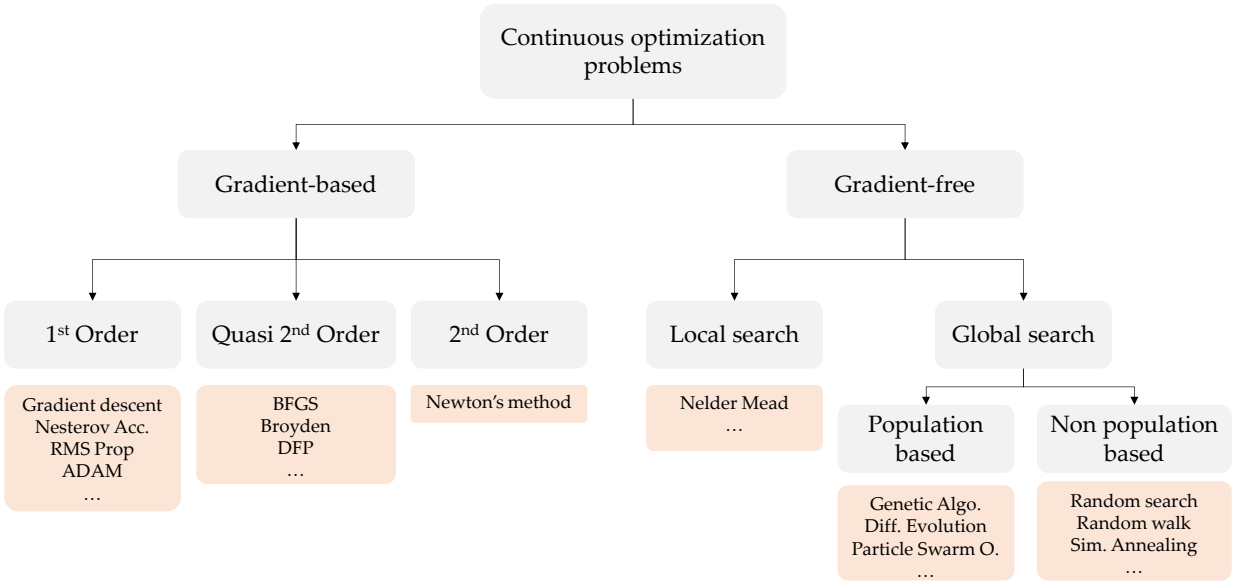
VKI Lecture Series and Events - Site for Participants > Hands on Machine Learning for Fluid Dynamics > Python_Codes > **Lecture_6**

Name	Modified	Modified By	Created	Created By	Description
0_Fail_O_HARMO.py	A few seconds ago	Miguel Alfonso Mendez	A few seconds ago	Miguel Alfonso Mendez	
0_Fail_O_ROSEN.py	A few seconds ago	Miguel Alfonso Mendez	A few seconds ago	Miguel Alfonso Mendez	
1_Function_Definitions.py	A few seconds ago	Miguel Alfonso Mendez	A few seconds ago	Miguel Alfonso Mendez	
2a_Rosenbrock_function_Random_Search.py	A few seconds ago	Miguel Alfonso Mendez	A few seconds ago	Miguel Alfonso Mendez	
2b_Rosenbrock_function_Simulated_Anneali...	A few seconds ago	Miguel Alfonso Mendez	A few seconds ago	Miguel Alfonso Mendez	
3a_Harmo_function_Random_Search.py	A few seconds ago	Miguel Alfonso Mendez	A few seconds ago	Miguel Alfonso Mendez	
3b_Harmo_function_Simulated_Annealing.py	A few seconds ago	Miguel Alfonso Mendez	A few seconds ago	Miguel Alfonso Mendez	
4a_Rosenbrock_function_Genetic_Algorithm...	A few seconds ago	Miguel Alfonso Mendez	A few seconds ago	Miguel Alfonso Mendez	
4b_Harmo_function_Genetic_Algorithms.py	A few seconds ago	Miguel Alfonso Mendez	A few seconds ago	Miguel Alfonso Mendez	
5a_Rosenbrock_function_Particle_Swarm.py	A few seconds ago	Miguel Alfonso Mendez	A few seconds ago	Miguel Alfonso Mendez	
5b_Harmo_function_Particle_Swarm.py	A few seconds ago	Miguel Alfonso Mendez	A few seconds ago	Miguel Alfonso Mendez	
GA_Functions.py	A few seconds ago	Miguel Alfonso Mendez	A few seconds ago	Miguel Alfonso Mendez	
PSO_Functions.py	A few seconds ago	Miguel Alfonso Mendez	A few seconds ago	Miguel Alfonso Mendez	

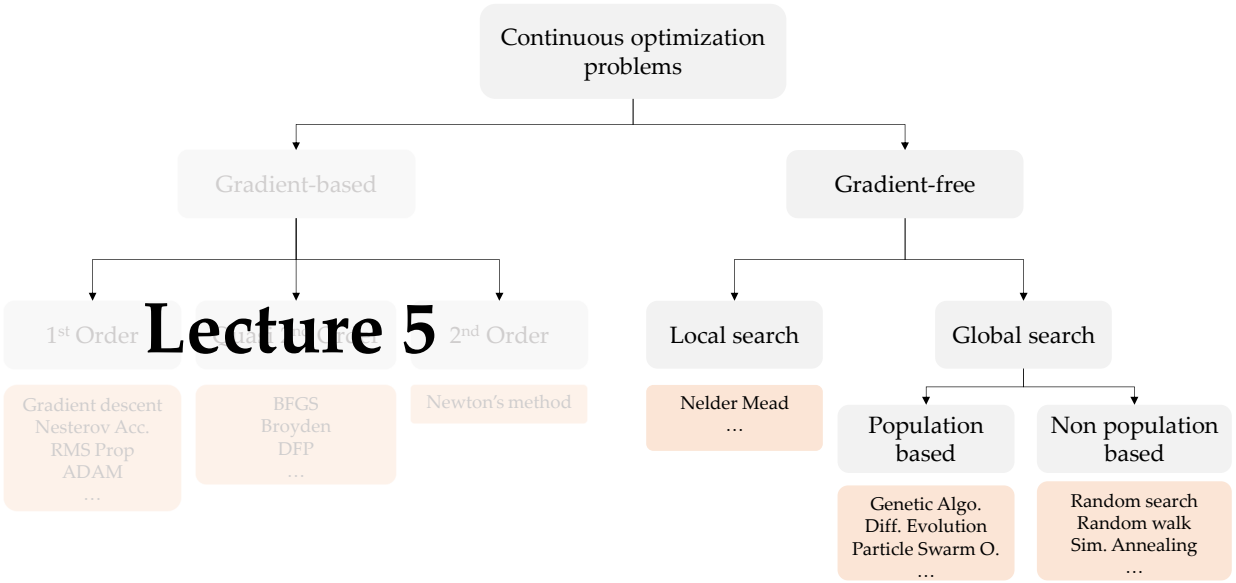
Table of Contents

Table of Contents

Optimization methods



Optimization methods



Lecture 5

The General Framework

We are in the field of **metaheuristic,** dealing with procedures to find, generate, or select a heuristic (partial search algorithm) that may provide a sufficiently good solution to an optimization problem.

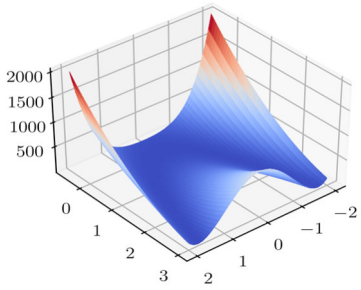
When you should consider these tools:

1. You are not interested in guarantees for optimality. You are ok with 'an improvement'
2. You do not know much about the process (cost function) you seek to optimize. You can only 'sample it'.
3. You want to 'explore' the domain of your cost function (the search space), and you might not even know 'from where to start'
4. Your cost function is not expensive

Slide 7

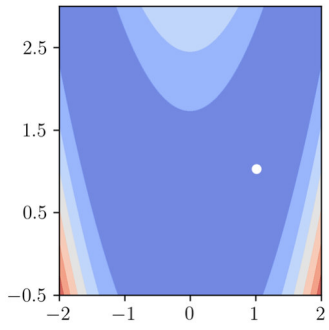
Test 1 in this Lecture

```
#%% Rosenbrock function definition
def rosenbrock(X):  # The Function to Minimize
    """Function to be minimized"""  # new
    x = X[0]; y = X[1]
    C= 100*(y-x**2)**2+(1-x)**2.
    return C
```



This is Rosenbrock function you have seen in the previous lecture. We keep the search space in the domain [-2,2]x [-0.5,3]

We have one global minimum in a valley (1,1), at
Would an optimizer be able to get there?



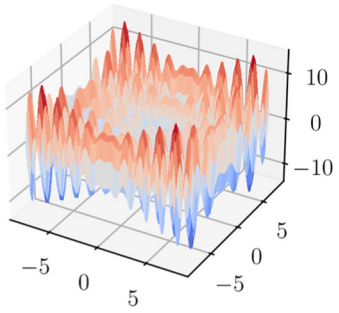
Challenge

Although the function is smooth, the minimum is in a valley. Finding the valley is easy, moving there to find the minima is not easy!

Slide 8

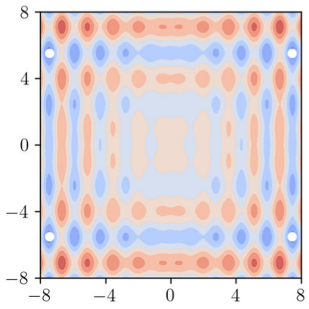
Test 2 in this Lecture

```
#%% Harmonic function definition
def harmo(X):  # The Function to Minimize
    """Function to be minimized"""  # new
    C=X[0]*np.sin(4*X[0])+1.1*X[1]*np.sin(2*X[1])
    return C
```



Let's limit the search space in the domain [-8,8]x [-8,8]

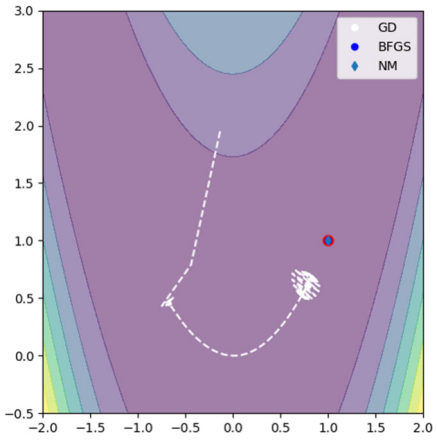
We have 4 global minima at the corners.
Would an optimizer be able to get there?



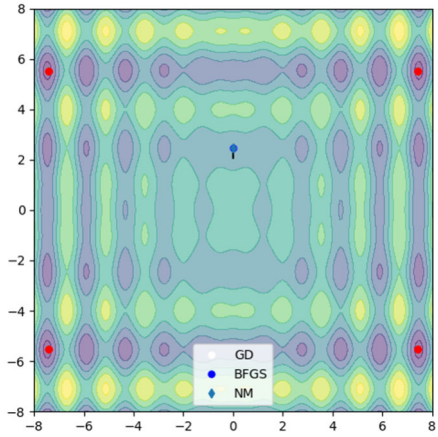
Challenge
Too many local minima, nearly impossible to escape from there using gradient information

Slide 9

An illustrative failure



Ok, but with some difficulties:
The valley does not offer enough gradient to a plaing GD, but more sophisticated tools handle it.



Complete Failure:
the result depends on where we start. We get trapped easily with any gradient-based tool.

Slide 10

Table of Contents

The Naïve Random Search

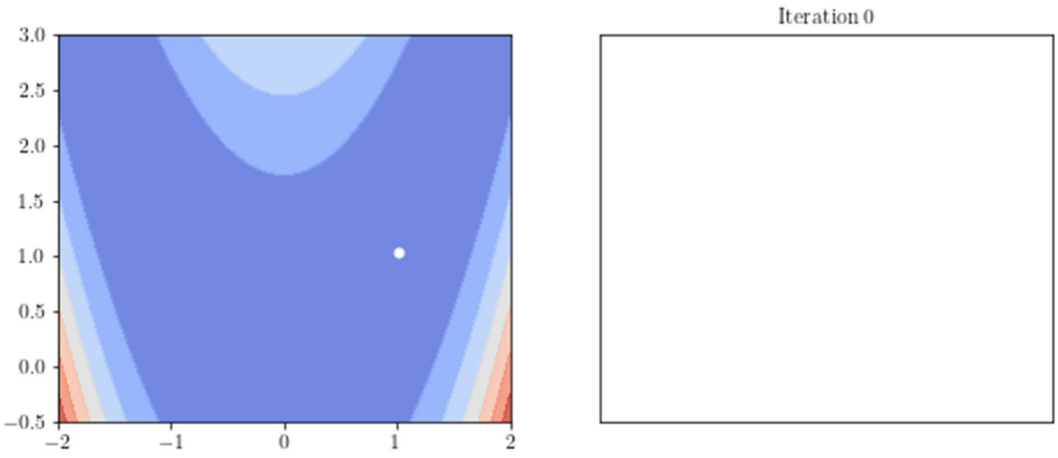
```
def Random_S(func,X0,X_Bounds,n_iter):
  # Perform random search to minimize func, starting from X0
  # and running for n_iterations within the domain X_Bounds
  # The output is the full story of X_k, func(X_k):
  n_dim=np.shape(X_Bounds)[0]
  # Initialize the next vector
  X_next=np.zeros(n_dim)
  # Initialize the new vector
  X_best=X0;
  # Prepare the story of attempt and evaluations
  X_k,Cost_k =np.zeros((n_dim,n_iter+1)),np.zeros(n_iter+1)
  # Run the first evaluation
  X_k[:,0]=X_best
  Cost_k[0]=func(X_best)
```

The simplest stochastic and gradient free approach is the naïve random search algorithm. Here a function to minimize a function using random search.

The idea is as simple as you can think of: you take a random candidate and evaluate it. If this is better that what you currently have, you take it; else you stay where you are!

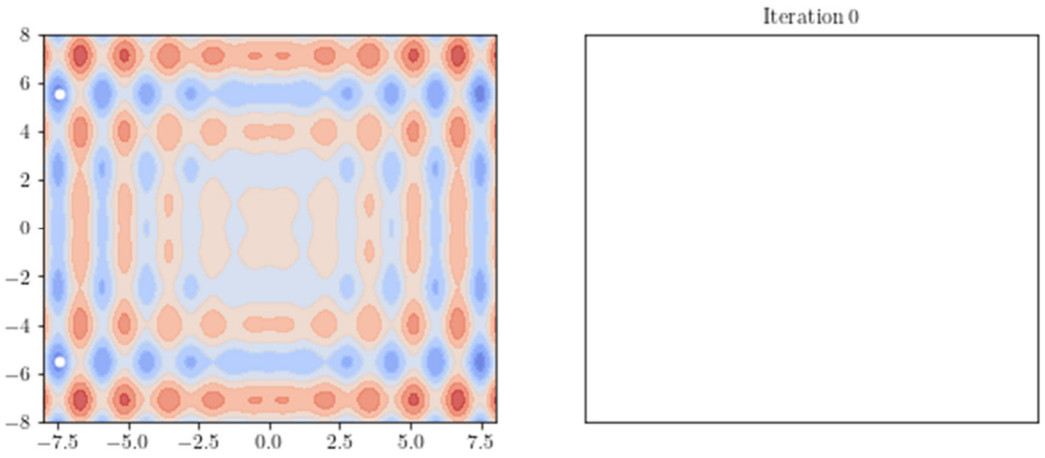
```
# Loop over the iterations
for k in range(1,n_iter):
  # Pick a random vector in the given range
  for j in range(n_dim):
     X_next[j]=np.random.uniform(low=X_Bounds[j][0],\
                                 high=X_Bounds[j][1])
  # Evaluate function in the new point
  F_next=func(X_next)
  # If the next is better, replace the new
  if F_next<Cost_k[k-1]:
   # Go for the new option
   X_k[:,k]=X_next
   Cost_k[k]=F_next
  else:
   # Stay where you are
   X_k[:,k]=X_k[:,k-1]
   Cost_k[k]=Cost_k[k-1]
return X_k,Cost_k
```

Random Search on Rosenbrock's Function



Slide 13

Random Search on Harmonic Function



Slide 14

Simulated Annealing

A simple modification leads to the popular simulated annealing.

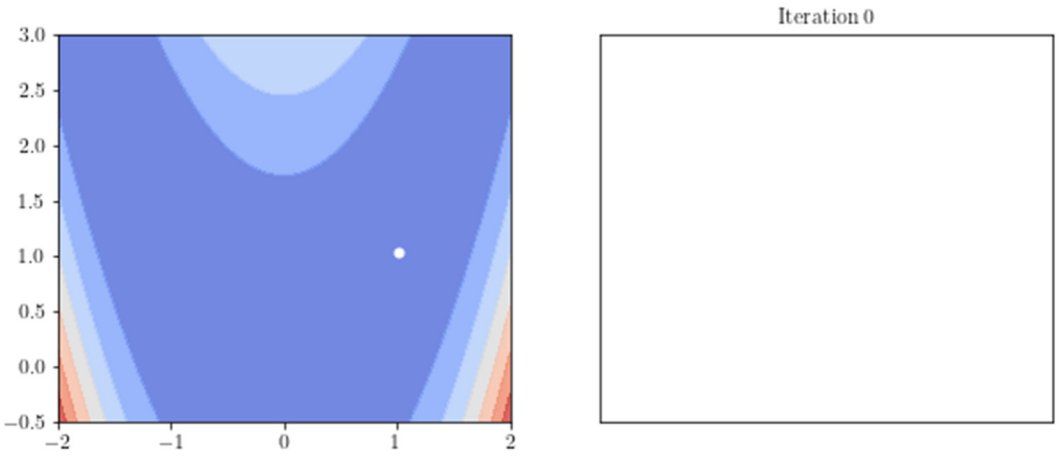
This algorithm is inspired by the annealing process in metallurgy, where a metal is heated quickly to high temperatures and then cooled slowly.

- The first portion of the code is as before, except for two new parameters controlling the cooling.
- We add a probability of exploration: depending on a 'Temperature' parameter, we might be willing to accept a 'worse'.

If the temperature is low, this is a random search with not much exploration.

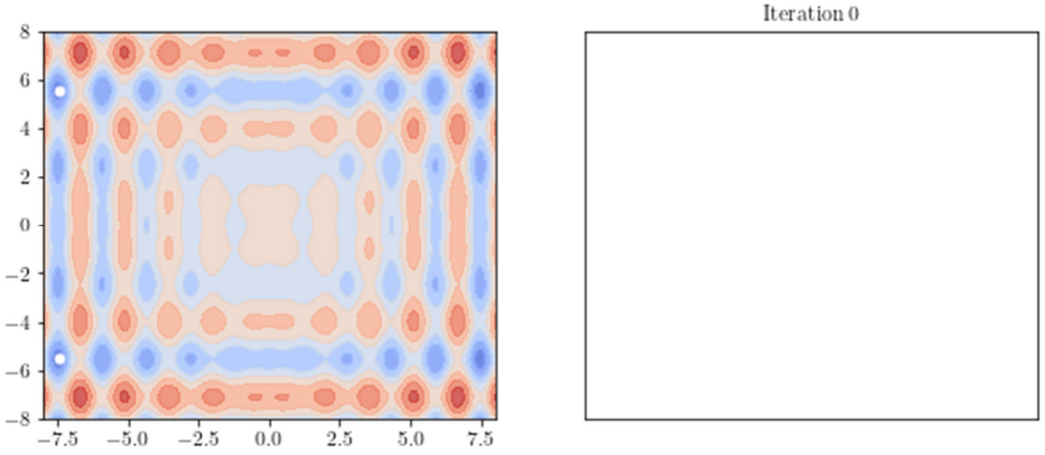
```
# Loop over the iterations
for k in range(1,n_iter):
  # Pick a random vector in the given range
  for j in range(n_dim):
     X_next[j]=np.random.uniform(low=X_Bounds[j][0],\
                                 high=X_Bounds[j][1])
  # Evaluate function in the new point
  F_next=func(X_next)
  # Check if this solution is better (if so note it)
  if F_next<F_best:
     X_best=X_next
     print('Change the current Best to X='+str(X_best))
  # Get 'Temperature'
  T=t_I/(k**a+1)
  # Get the difference between current and best (could be 0!)
  diff=np.abs(F_next-F_best)/F_best
  # Compute the metropolis acceptance
  Metropolis=np.exp(-diff/T)
  Rand_Change=np.random.rand()<Metropolis
  # Tell us if the change occurred randomly
  if Rand_Change:
     print('Random Change! High Temperature!')
  # We might do a replacement even if it's 'Bad'
  if F_next<Cost_k[k-1] or Rand_Change:
   # Go for the new option
   X_k[:,k]=X_next
   Cost_k[k]=F_next
  else:
   # Stay where you are
   X_k[:,k]=X_k[:,k-1]
   Cost_k[k]=Cost_k[k-1]
```

Simulated Annealing on Rosenbrock's Function



Slide 16

Simulated Annealing on Harmonic Function



Slide 17

Table of Contents

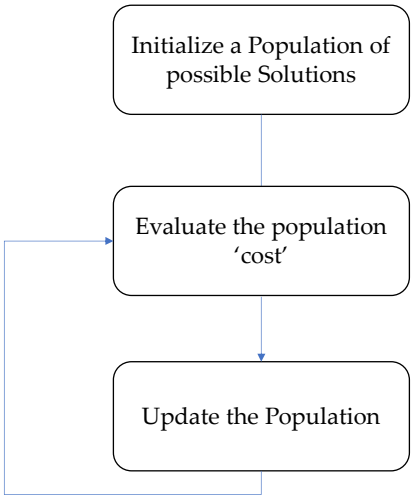
A Stochastic, Population based Approach

We will follow a 'population-based approach'

```
{
    Initialize population;
    Evaluate population;
    while Termination Criteria Not Satisfied
    {
        Update the population;
        Evaluate the population;
    }
}
```

This is the general architecture for all population-based methods. **Only the update** is different (based on different heuristics!)

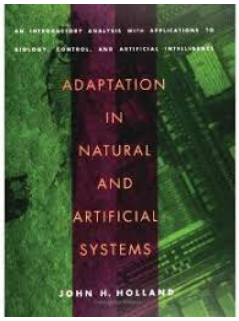
Note: the 'Evaluation' is the most expensive operation but… it can be parallelized very easily

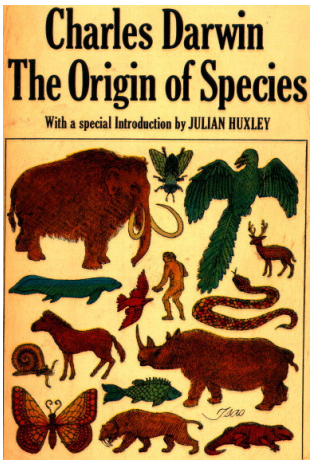


Evolution Strategies and Genetic Algorithms

Genetic Algorithm were mostly developed in the 60-70's. The main idea is to imitate the mechanisms of biological evolution and in particular the idea of 'survival of the fittest'.

One of the fathers of this idea was John Henry Holland who published his book in 1975

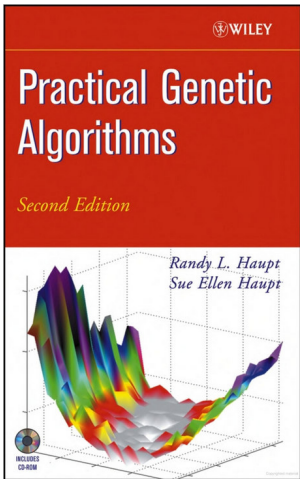


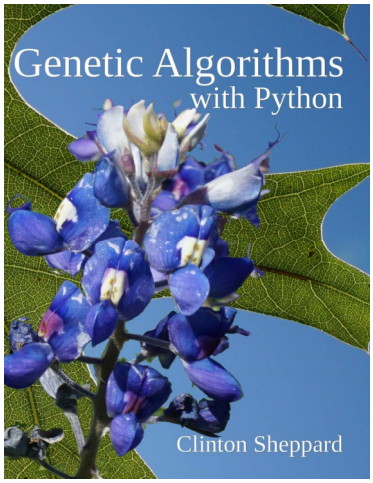


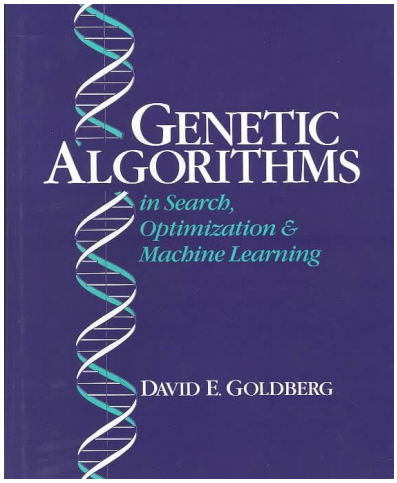
•*Adaptation in Natural and Artificial Systems* (1975, MIT Press)

Slide 20

Some Good References







Some Fancy Terminology

Individuals: Every member of our population, i.e., 'a possible solution' of our optimization. Practically this will be a vector in the domain of the cost function.

Fitness and Selection: The value of the cost function for each individual will define its 'fitness'. The idea is that the 'fittest' individuals have better 'traits' than the other and their traits should propagate and mix towards the next generation. The fittest individuals also have the highest chance of mating…!

Genetic Operations

Cross-Over When two individuals' mate, their 'traits' are combined

Mutation Some random mutation occurs between one generation and the other. This is how the algorithm 'explores'.

Elitism Some of the best individuals will 'copy themselves' to the next generation. This is how the algorithm 'exploits'.

Off-spring. Well… not much to expect here… that's just the updated population!

The GA in Practice

The GA codes used in this work are taken from [Mendez et al]

In this work, the GA was used to solve an inverse problem: calibrate a model from data.

The provided GA_Functions.py contains four functions with the main steps:

```
#%% Import the key functions
from GA_Functions import Initialize_POP
from GA_Functions import Evaluate_POP
from GA_Functions import Update_POP
from GA_Functions import GA
```

Slide 23

The GA in Practice

```
#%% 4. Main GA Function Definition (non parallel code)
def GA(Func,X_Bounds,n_p=100,N_ITER=100,
       n_G=0.5,sigma_I_r=6,mu_I=0.3,mu_F=0.05,
       p_M=0.5,n_E=0.05):
    """Genetic Algorithm Optimization of a function Func.

    # Initialize Population
    print('Initializing Population...')
    X_V=Initialize_POP(n_p,X_Bounds,n_G=0.5,sigma_I_r=6)
    # Prepare Some Stats
    Err_Best=np.zeros((N_ITER,1))
    Err_Mean=np.zeros((N_ITER,1))
    print('Preparing the loop...')
    for k in range(N_ITER):
      Err_1=Evaluate_POP(X_V,Func) # Evaluate Error
      X_V=Update_POP(X_V,Err_1,X_Bounds,k,N_ITER,\
                     mu_I=mu_I,mu_F=mu_F,p_M=p_M,n_E=n_E)
      Err_Best[k]=np.min(Err_1); Err_Mean[k]=np.mean(Err_1)
    # Finally give the answer
    Index=Err_1.argmin();
    X_S=X_V[:,Index]
    X_U=np.std(X_V,axis=1)
    print('Optimization finished')
    return X_S, X_U, X_V
```

Call to the main Function

Initialize

Evaluate

Update

Population Evolution

Final Statistics

Slide 24

The GA Initialization

```
#%% 1. Initialize Population
def Initialize_POP(n_p,X_Bounds,n_G=0.5,sigma_I_r=6):
  """Initialize Population.

    n_f=len(X_Bounds)
   #Generate an Initial Population
     #%% Half distributed with Gaussian pdf
    N_Gau_pop=int(n_G*n_p) # number of entries with Gaussian pdf
    X_G=np.zeros((n_f,N_Gau_pop))
    Mean_X=np.zeros((n_f,1))
    Sigma_X=Mean_X
    for j in range(n_f):
     Mean_X=(X_Bounds[j][1]+X_Bounds[j][0])/2
     Sigma_X=abs(X_Bounds[j][1]-X_Bounds[j][0])/sigma_I_r
     X_G[j,:]=np.random.normal(Mean_X,Sigma_X, N_Gau_pop)
     #%% Half Uniformly distributed
    n_U=n_p-N_Gau_pop
    X_U=np.zeros((n_f,n_U))
    for j in range(n_f):
     X_U[j,:]=np.random.uniform(X_Bounds[j][1],X_Bounds[j][0], n_U)
    #%% Prepare Initial Population
    X_V=np.concatenate([X_G, X_U],axis=1)
    return X_V
```

Call to the Function

Portion of the population distributed as a Gaussian

Portion of the population uniformly distributed

Slide 25

The GA Evaluation

```
#%% 2 Evaluate Population
def Evaluate_POP(X_V,Func):
   """Evaluate a population of candidates.
```

Call to the Function

```
  n_f,n_p=X_V.shape; # Number of features and Pop size
  Err_1=np.zeros((n_p,1)) # Cost Function
  for k in range(n_p): # To be parallelized
   Err_1[k]=Func(X_V[:,k])
  return Err_1
```

Population evaluation

This is obviously the most expensive step, but also the one which could be more easily parallelized!

The GA Update (1)

```
#%% 3. Update Population
def Update_POP(X_V,Err_1,X_Bounds,n_I,N_ITER,mu_I=0.3,mu_F=0.05,p_M=0.5,n_E=0.05):
    """Update Population.
```

```
 # Optional: Introduce an update bar
 progress(n_I,N_ITER)
 print("\n Best:  %s Mean %s" % (np.min(Err_1), np.mean(Err_1)))
 #%% Sort the Population and bring forward elitism and mutations
 n_f,n_p=X_V.shape; # Number of features and Pop size
 index=Err_1.argsort(axis=0) # Sorted index
 # Mutated Elements
 alpha=1/N_ITER*np.log(mu_F/mu_I) # Exp Coefficient
 Mut=mu_I*np.exp(alpha*n_I) # Number of mutate elements (float)
 N_M=int(np.round(Mut*n_p)) # Number of mutate elements (int)
 N_E=int((n_p-N_M)*n_E) # Number of Elite Elements
 N_C=int(n_p-N_M-N_E) # Number of Cross-over Elements
 print(" Elites:%s Mutated:%s Cross-over:%s" % (N_E, N_M, N_C))
```

(1) Find out how the genetic operations will be distributed

Slide 27

The GA Update (2)

```
#%% Perform Genetic Operations
# 1. Elitism-----------------------------------------------------------------
X_V_E=X_V[:,index[0:N_E,0]]
# 2. Mutations ---------------------------------------------------------------
# Number of chromosomes that will mutate
P_M=int(p_M*n_f)
# We mutate over the best n_M Individuals
# Take randomly the chromosomes that will mutate
X_V_M=np.zeros((n_f,N_M))
for m in range(N_M):
 X_V_M[:,m]=X_V[:,index[m,0]] # Take the Best N_M
 print('Mutation ' +str(m))
 for mm in range(P_M):
  Ind_M=np.random.randint(0,n_f)
  print('Change entry ' +str(Ind_M))
  X_V_M[mm,m]=np.random.uniform(X_Bounds[Ind_M][1],X_Bounds[Ind_M][0], 1)
# 3. Cross-Over   ------------------------------------------------------------
X_V_C=np.zeros((n_f,N_C))
for k in range(0,N_C):
 SEL=np.random.triangular(0, 0, N_C, 2)
 for j in range(0,n_f):
  a=np.random.uniform(0,1,1)
  X_V_C[:,k]  = a*X_V[:,index[int(SEL[0]),0]]  +(1-a)*X_V[:,index[int(SEL[1]),0]]
```

(2) Elitism

(3) Mutation

(4) Cross-Over

Slide 28

The GA Update (3)

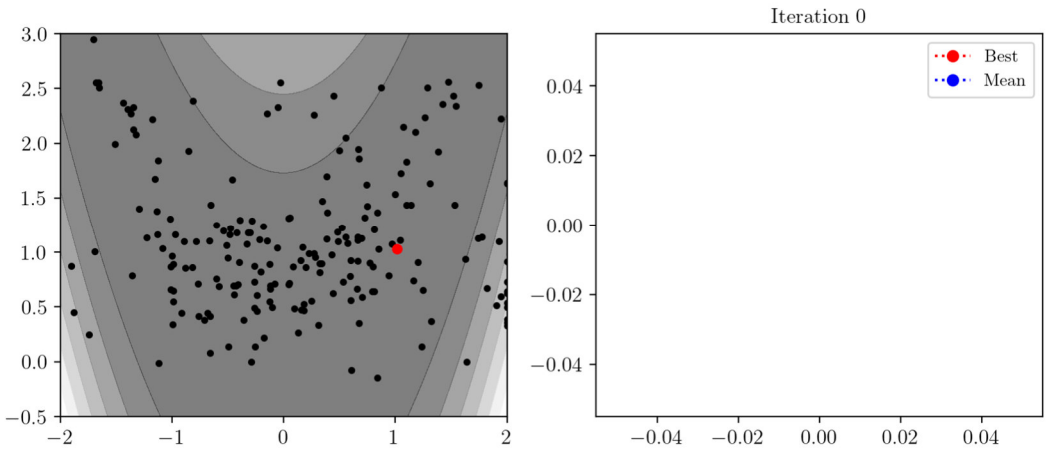
```
#%% Final Concatenation + cleaning
X_V_n=np.concatenate([X_V_C, X_V_E,X_V_M],axis=1)
for j in range(0,n_f):
 mask1=X_V_n[j,:] < X_Bounds[j][0]
 X_V_n[j,mask1]=X_Bounds[j][0]
 mask2=X_V_n[j,:] > X_Bounds[j][1]
 X_V_n[j,mask2]=X_Bounds[j][1]
return X_V_n
```

(5) Mount the full population and cap on the boundaries

The rest of the code contains a function for creating an animated gif

Slide 29

GA Results on the Rosenbrock Function



GA Results on the Harmonic Function

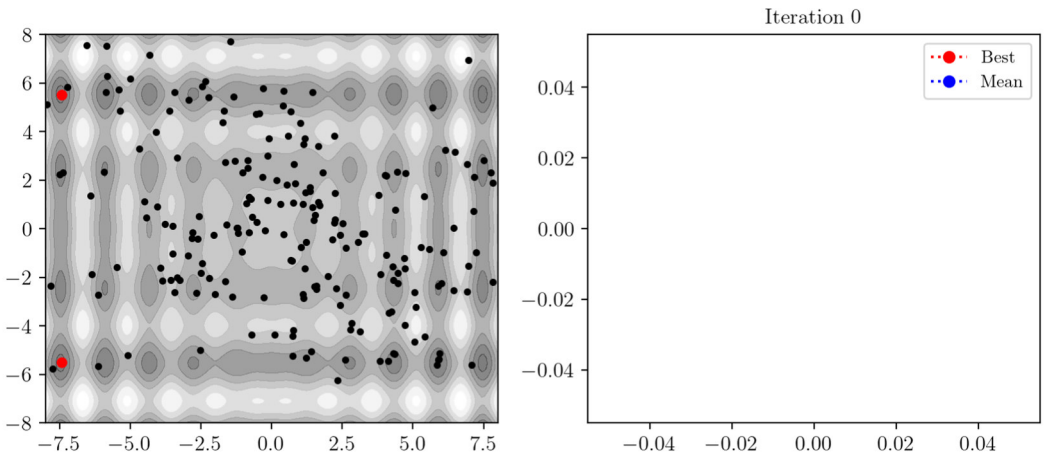


Table of Contents

Particle Swarm Optimization

Particle swarm optimization is a more recent optimization tool developed in 1995.
The main idea is to imitate the notion of 'swarm intelligence', defined as the '**emergent collective intelligence of groups of simple agents**'

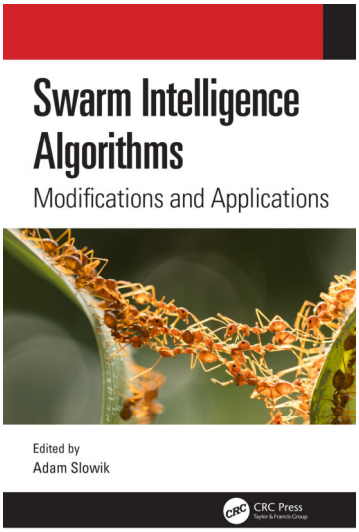
Dr. James Kennedy

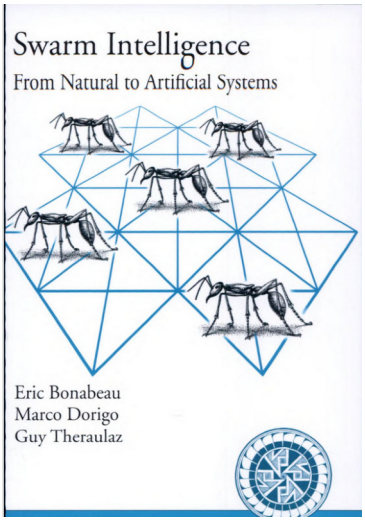
Dr. Russell Eberhart

J. Kennedy and R.C. Eberhart, "Particle swarm optimization," in IEEE Int. Conf.on Neural Networks., Perth, Australia, vol. 4, 1995, pp. 1942-1948.

33

Some Good References





34

Idea and Related Terminology

Instead of individuals, we now have **particles** and instead of a population we now have a **swarm**. Every particle moves in our search space at a certain **velocity.**

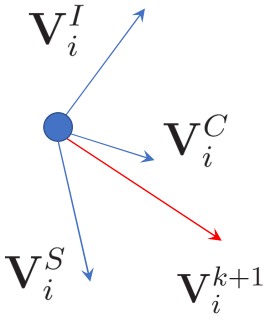
The motion of a particle depends on its own memory (cognitive term) and the motion of the swarm (social term). The position of a particle i at step k is:

$$\mathbf{X}_i^{k+1} = \mathbf{X}_i^k + \mathbf{V}_i^k$$

The velocity of a particle i at step k depends on three terms:

$$\mathbf{V}_i^{k+1} = w^k \mathbf{V}_i^k + c_c R_1(\tilde{\mathbf{X}}_i^k - \mathbf{X}_i^k) + c_s R_2(\tilde{\mathbf{X}}^k - \mathbf{X}_i^k)$$

Inertia $\mathbf{V}_i^I$ — Cognitive term $\mathbf{V}_i^C$ — Social term $\mathbf{V}_i^S$



User defined parameters
Random number

Best location ever visited by the particle

Best location within the swarm

35

General Architecture

```
#%% 4. Main GA Function Definition (non parallel code)
def PSO(Func,X_Bounds,n_p=100,N_ITER=100,
        n_G=0.5,sigma_I_r=6,w_I=0.0001,w_F=0.0001,c_c=0.1,c_s=0.3):
    """Particle Swarm Optimization of a function Func.
```

Call to the main Function

```
# Initialize Population
print('Initializing Population...')
X_V,V_P=Initialize_POP(n_p,X_Bounds,n_G=0.1,sigma_I_r=6,I_V=0)
# Prepare Some Stats
Err_Best=np.zeros((N_ITER,1)) # Best Cost Function Value
Err_Mean=np.zeros((N_ITER,1)) # Mean of the Cost Function
Err_1=np.zeros((n_p,1)) # Initialization of the Errors
print('Preparing the loop...')
for k in range(N_ITER):
  Err_1=Evaluate_POP(X_V,Func) # Evaluate Error
  if k==0:
   Err_B_V=Err_1 # Best error obtained by a particle
   X_B_V=X_V # Best location ever visited by a particle
  X_V_n, V_P_n, X_B_V_n, Err_B_V_n=Update_POP(X_V,V_P,X_B_V,Err_B_V,Err_1,\
           X_Bounds,k,N_ITER,w_I,w_F,c_c,c_s)
  Err_Best[k]=np.min(Err_1); Err_Mean[k]=np.mean(Err_1)
```

Initialize (position+ velocity)

At the first iteration there is no memory

Update

Final Statistics

Initialization

Initial Velocity Parameter

```
def Initialize_POP(n_p,X_Bounds,n_G=0.5,sigma_I_r=6,I_V=0.1):
  """Initialize Population.
```

```
    n_f=len(X_Bounds)
   #Generate an Initial Population
     #%% Half Initially distributed with Gaussian pdf
    N_Gau_pop=int(n_G*n_p) # number of entries with Gaussian pdf
    X_G=np.zeros((n_f,N_Gau_pop))
    Mean_X=np.zeros((n_f,1))
    Sigma_X=Mean_X
    for j in range(n_f):
     Mean_X=(X_Bounds[j][1]+X_Bounds[j][0])/2
     Sigma_X=abs(X_Bounds[j][1]-X_Bounds[j][0])/sigma_I_r
     X_G[j,:]=np.random.normal(Mean_X,Sigma_X, N_Gau_pop)
     #%% Half Initially Uniformly distributed
    n_U=n_p-N_Gau_pop
    # Initialize Uniformly Distributed Positions and Velocities
    X_U=np.zeros((n_f,n_U)); V_P=np.zeros((n_f,n_p))

    for j in range(n_f):
     X_U[j,:]=np.random.uniform(X_Bounds[j][1],X_Bounds[j][0], n_U)
     V_MAX=np.abs(X_Bounds[j][1]-X_Bounds[j][0])*I_V
     V_P[j,:]=np.random.uniform(-V_MAX,V_MAX, n_p)
    #%% Prepare Initial Population
    X_V=np.concatenate([X_G, X_U],axis=1)
```

Portion of the population distributed as a Gaussian

Portion of the population uniformly distributed

Initial Velocity

37

Update

```
def Update_POP(X_V,V_P,X_B_V,Err_B_V,Err_1,
               X_Bounds,n_I,N_ITER,w_I=0.3,w_F=0.05,c_c=2,c_s=2):
    """Update Population.

    n_f,n_p=X_V.shape; # Number of features and Pop size
    # Coefficient for the Inertia reduction
    alpha=1/N_ITER*np.log(w_F/w_I) # Exp Coefficient
    w=w_I*np.exp(alpha*n_I) # Actual Inertia Coefficient

    #%% Find the Global BEST (for SOCIAL TERM)
    ID=np.where(Err_1 == Err_1.min())
    X_BEST=X_V[:,ID[0]] # If you have more than one, take the first

    #%% Store the BEST for each particle (for Cognitive TERM)
    for k in range(n_p):
     # Define the Particle BEST
     if Err_1[k]<Err_B_V[k]:
       # Assign best error and best position in items that are overwritten
       Err_P_Best=Err_1[k]; X_P_BEST=X_B_V[:,k]
       Err_B_V[k]=Err_P_Best # Update the Best Error
       X_B_V[:,k]=X_P_BEST # Update the Best Location

    # Define Random numbers for Cognitive and Social Terms
    R1=np.random.uniform(0,1, (n_f,n_p))
    R2=np.random.uniform(0,1, (n_f,n_p))

    X_V_n=X_V+V_P # Update of the Population
    V_P_n=w*V_P+c_c*R1*(X_B_V-X_V)+c_s*R2*(X_BEST-X_V)
```

Exponentially decreasing inertia

The current best

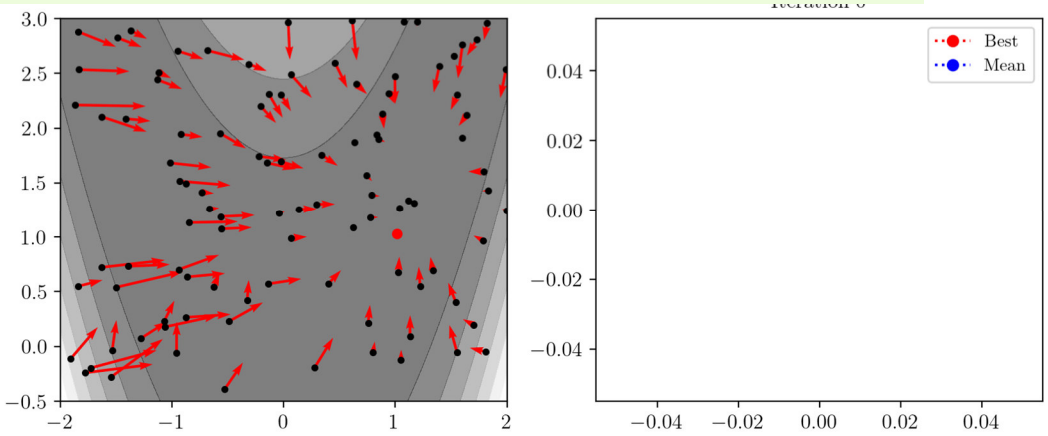
Keep track of the best point ever visited by the particle

Define random numbers for the update

Position and Velocity Update

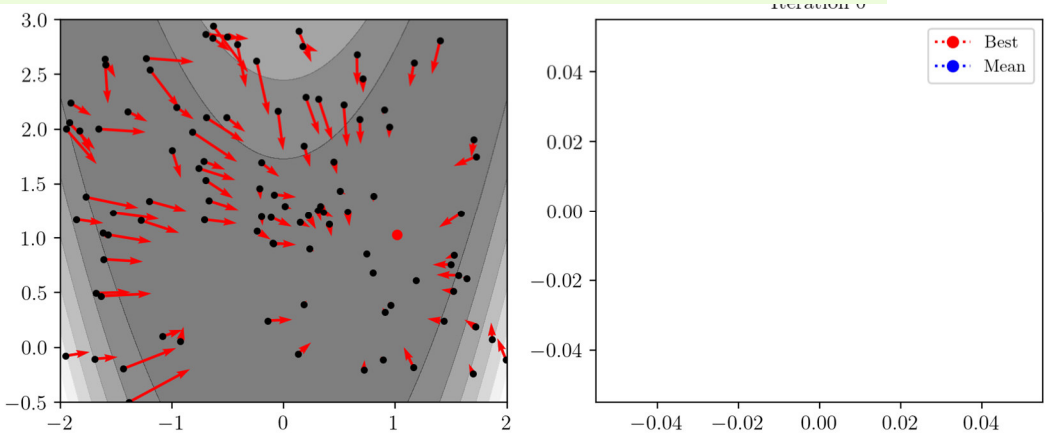
A Non-Collaborative Example

```
# Example with non collaborative particles
X_S, X_U, X_V=Anim_COMP(Func,X_Bounds,n_p=100,N_ITER=100,\
                 n_G=0.5,sigma_I_r=6,w_I=0.1,w_F=0.01,c_c=0.8,c_s=0.02,\
                 x_1m=-2,x_1M=2,x_2m=-0.5,x_2M=3,\
                     npoints=200,Name_Video='Rosen_PSO_EG.gif')
```

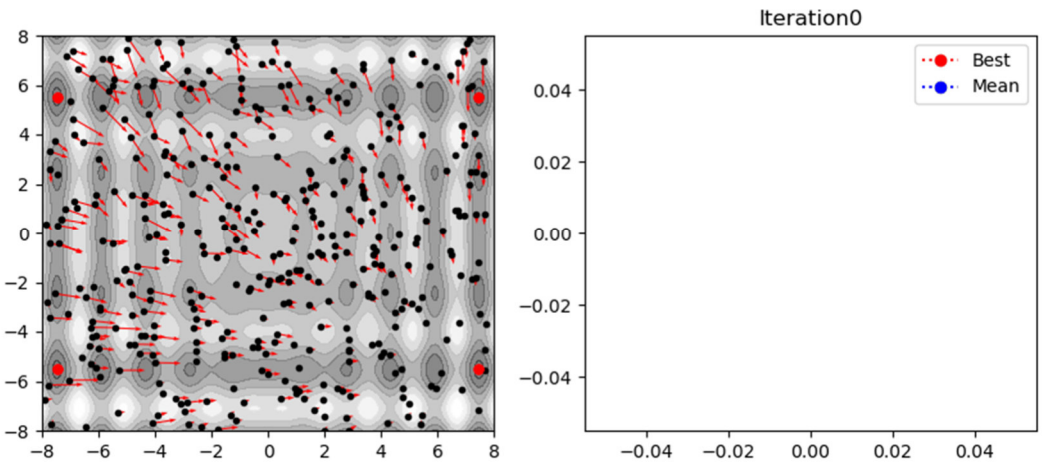


Good Teamwork!

```
# Example with collaborative particles
X_S, X_U, X_V=Anim_COMP(Func,X_Bounds,n_p=100,N_ITER=100,\
                n_G=0.5,sigma_I_r=6,w_I=0.1,w_F=0.01,c_c=0.02,c_s=0.5,\
                x_1m=-2,x_1M=2,x_2m=-0.5,x_2M=3,\
                    npoints=200,Name_Video='Rosen_PSO_TW.gif')
```



PSO Results on Harmonic Function (ex)



41

Take Home Messages

Genetic and Swarm Intelligence algorithms are particularly interesting because…

1. Extremely simple coding (almost no mathematical background required!)
2. They can be easily parallelized
3. They are 'global optimizers' and can handle extremely complex cost functions
4. Easy generalization and no extra cost at higher dimensions

On the other hand, you should consider that…

1. Very Poor 'Sample Efficiency': many cost function evaluations are needed
2. The good setting of hyperparameters is strongly test case dependent
3. Performances are highly sensitive to hyperparameters (collaboration vs egoism!)
4. You might be looking for one good solution… not a population/swarm of solutions!

The end!

Slide 43