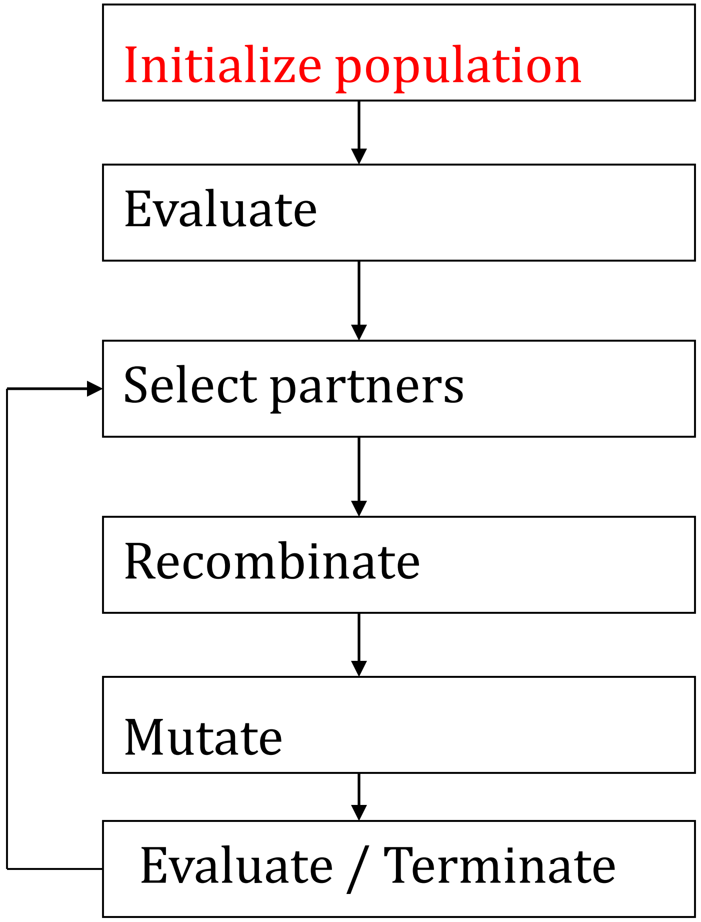
Evolutionary Strategies ES

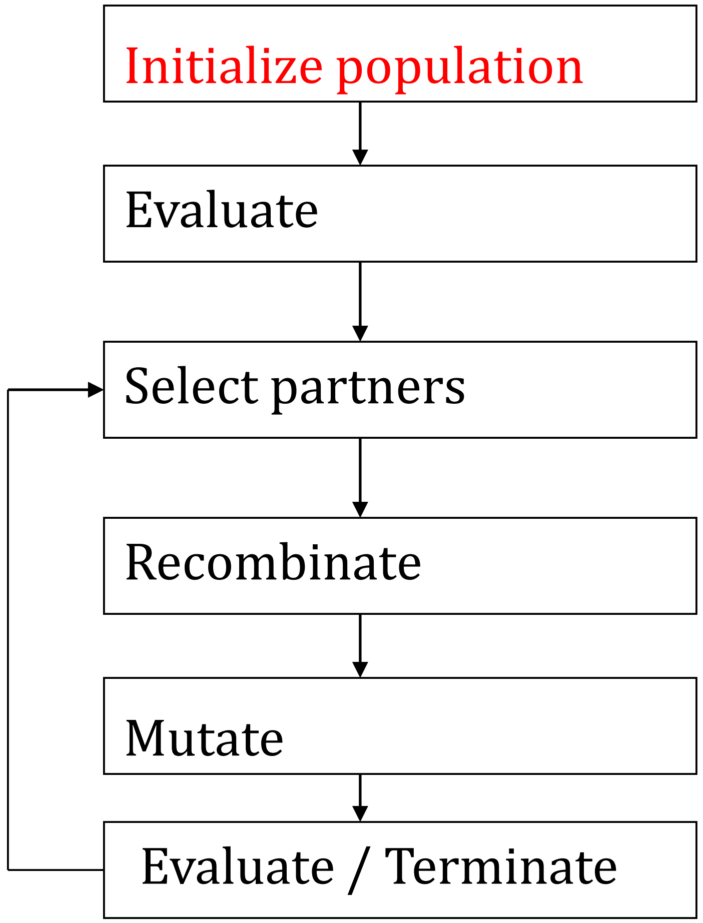
```
# Main Algorithm code

# Parameters settings
# Number of generation
ngen = 100
# Number of offspring
lamb = 100

# Number of parents mu/lambda (ratio 1/7)
mu = 15

# Expected rate of convergence
varcon = 1

# Mean step size
sig0 = 1
```



Evolutionary Strategies ES

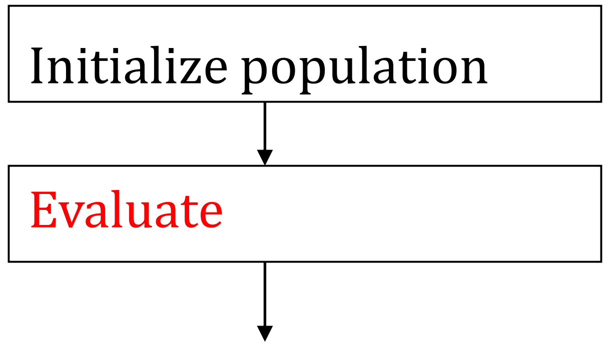
Evolutionary Strategies ES

```
# Initial setting
# Decision variables min and max values
Rvar = np.array([
    [-512, -512],
    [ 513,  513]])
# Number of parameters
Npar = Rvar.shape[1]
Rmin = Rvar[0][0:Npar]
Rmax = Rvar[1][0:Npar]

# Strategies parameters
sig = np.ones((lamb,Npar))
tau = varcon/math.sqrt(Npar)

# Random population
Rpop = np.ones((lamb,Npar))*Rmin+np.random.random((lamb,Npar))*np.array(Rmax-Rmin)
```



Evolutionary Strategies ES

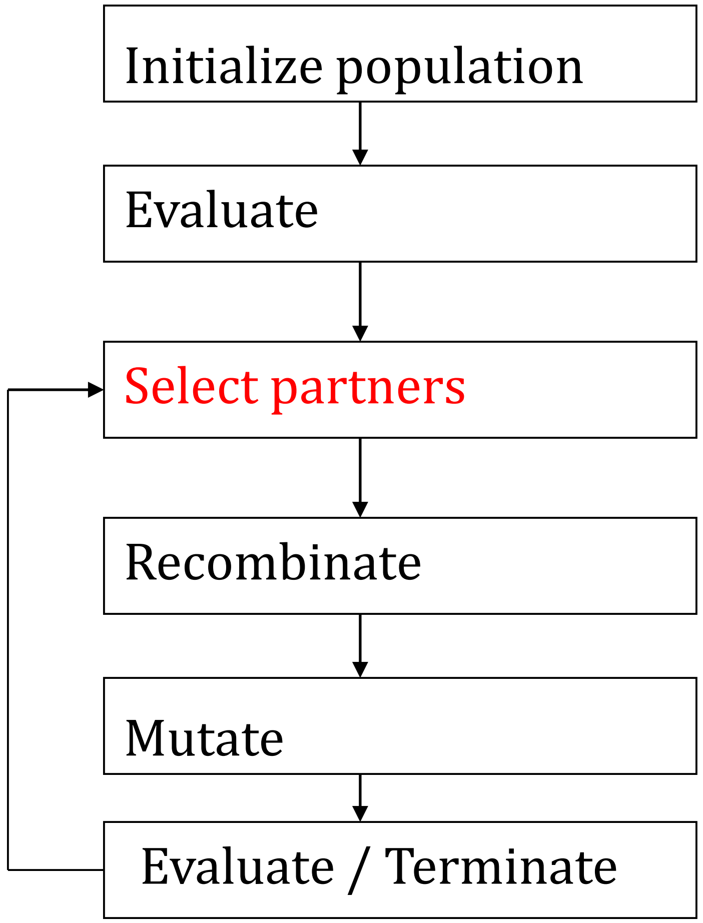
```
# Random population
Rpop = np.ones((lamb,Npar))*Rmin+np.random.random((lamb,Npar))*np.array(Rmax-Rmin)

PI_best_progress = [] # Tracks progress
Rbest = Rpop[0][:]

# Performance index
PI =  fitness_function(Rpop)
PI_best = np.min(PI)
ind = np.zeros(1)
ind = np.where(PI == PI_best)
Pbest = np.array(Rpop[ind[0]][:])

print ('Starting best score, % target: ',PI_best)
# Add starting best score to progress tracker
PI_best_progress.append(PI_best)
```



Evolutionary Strategies ES

Evolutionary Strategies ES

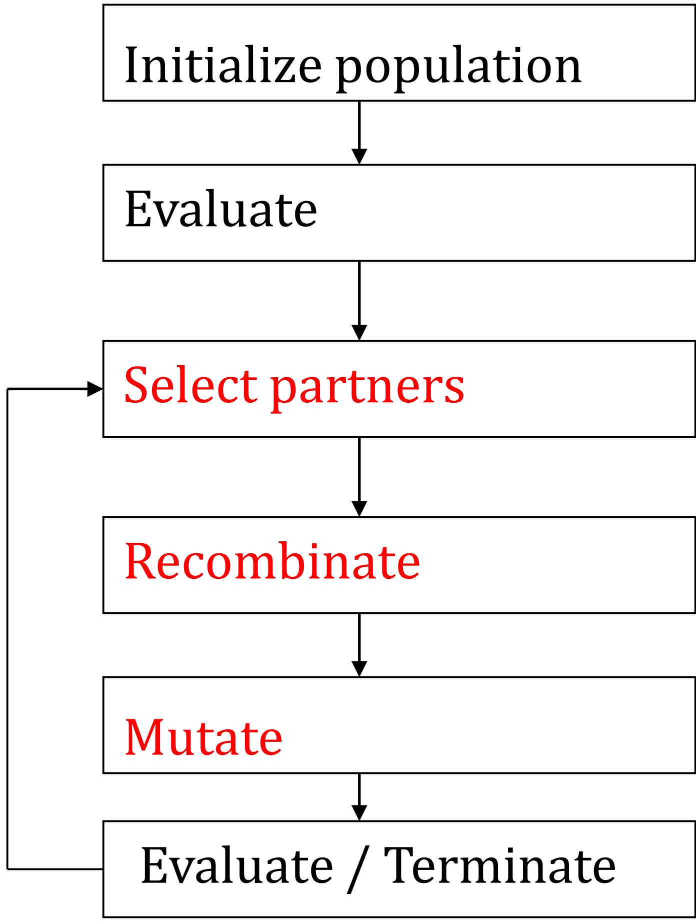
```
isort = np.argsort(PI)
PIopt = PI[isort]
Ropt  = Rpop[isort][:]

# Index selection vector
isel = np.zeros(lamb)
icou = 0
for i in range (lamb):
    isel[i] = icou
    icou = icou + 1
    if(icou == mu): icou = 0
isel = isel.astype(int)
```



Evolutionary Strategies ES

Evolutionary Strategies ES

```
for igen in range(ngen):
    # Ranking
    isort = np.argsort(PI)
    # Selection
    Rpop = Rpop[isort[isel[:]]][:]
    sig = sig[isort[isel[:]]][:]
    sig = sig*np.exp(np.random.random((lamb,Npar)))
 # Variation of variables
    Rpop = Rpop + np.random.random((lamb,Npar))*sig
    Rminm = np.ones((lamb,Npar))*Rmin
    Rmaxm = np.ones((lamb,Npar))*Rmax
    imin = np.where(Rpop<Rminm)
    imax = np.where(Rpop>Rmaxm)
    Rpop[imin[:][0],imin[:][1]]=Rminm[imin[:][0],imin[:][1]]
    Rpop[imax[:][0],imax[:][1]]=Rmaxm[imax[:][0],imax[:][1]]
```



Evolutionary Strategies ES Performance Index (Eggholder function)

```
import random
import math
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

def fitness_function(Rpop):
    npop = Rpop.shape[0]
    npvar = Rpop.shape[1]
    PI = np.zeros((npop))
    # Eggholder
    for ip in range(npop):
        x = Rpop[ip][:]
        PI[ip] =(-(x[1] + 47) * np.sin(np.sqrt(abs(x[0]/2 + (x[1]  + 47))))
            -x[0] * np.sin(np.sqrt(abs(x[0] - (x[1]  + 47)))))
    return PI
```

Simulated Annealing

Simulated annealing (SA) :
is a random-search technique which exploits an analogy between the way in which a metal cools and freezes into a minimum energy crystalline structure (the annealing process) and the search for a minimum in a more general system.

Numerical simulation of Annealing:

$$P(\delta E) = e^{(-\,\delta E/kT)}$$

$P(\delta E)$	: Probability of an increase in energy by δE
T	: Temperature
k	: Boltzmann's constant

Simulated Annealing

Simulated annealing (SA) :

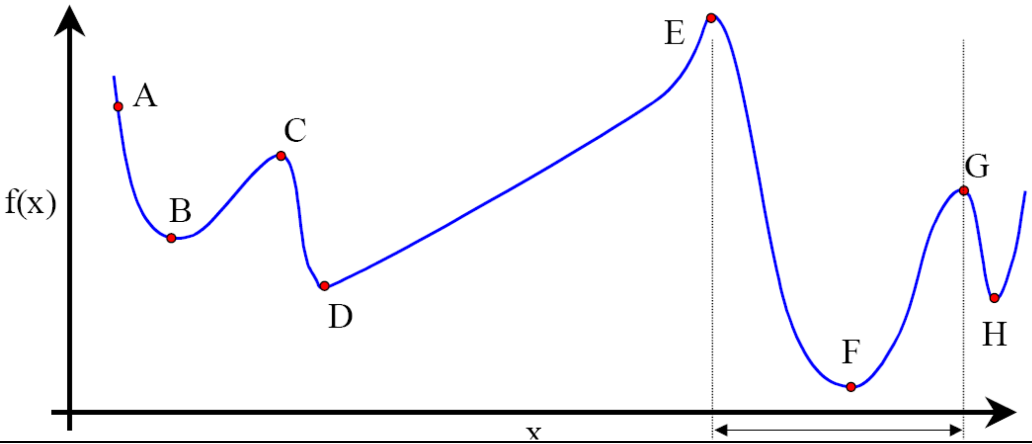
It forms the basis of an optimization technique for combinatorial and other problems.

Simulated annealing was developed in 1983 to deal with highly nonlinear problems.

Simulated Annealing

Simulated annealing (SA):

SA approaches the global maximization problem similarly to using a bouncing ball that can bounce over mountains from valley to valley. It begins at a high "*temperature" which enables the ball to make very high bounces,*
which enables it to bounce over any mountain to access any valley, given enough bounces.



Simulated Annealing

Simulated annealing (SA):

- SA's major advantage is an ability to avoid becoming trapped in local minima.
- The algorithm employs a random search which not only accepts changes that decrease the objective function f (assuming a minimization problem), but also some changes that increase it.

The latter are accepted with a probability

$$\delta f = f(x_{i+1}) - f(x_i)$$
$$p(\delta f) = \exp(-\delta f / T)$$

Simulated Annealing

Simulated annealing (SA):

Thermodynamic Simulation	Combinatorial Optimization
System States	Feasible Solutions
Energy	Objective
Change of State	Neighboring Solutions
Temperature	Control Parameter
Frozen State	Heuristic Solution

SA Algorithm: Part 1

Initial steps:

Solution Space $S(\mathbf{x})$

Objective Function $f(\mathbf{x})$

Select Initial Point $\mathbf{x}_o$ in S

Select Initial Temperature To

Select Temperature Reduction Function

Simulated Annealing

SA Algorithm: Part 2

Iteration steps:

For i_iteration = 1,N_iteration

Generate New Solution $x_{i+1}=x_i + \mathbf{D}\,\mathbf{u}$ D: max change **u**:R[-1 1]

Assess New Solution $\delta f = f(x_{i+1}) - f(x_i)$, if $\delta f < 0$ or
Random number r : [0 1] if $r < e^{-\delta f/Tk}$

Accept New Solution (No: Continue / Yes: Update)

Update $\mathbf{D}_{i+1} = (1-a)\,\mathbf{D}_i + a\,w\,\mathbf{R}$

Adjust Temperature Exp. cooling scheme $T_{k+1} = \alpha\, T_k$ ($\alpha = 0.95$)

End i_iteration (Terminate Search)

Simulated Annealing

SA Algorithm:

$$x_{i+1}=x_i + \mathbf{D}\,\mathbf{u}$$

where **u** is a vector of random numbers in the range (-1,1) and **D** is a diagonal matrix which defines the maximum change allowed in each variable.

After a successful trial, i.e. after an accepted change in solution, **D** is updated:

$$\mathbf{D}_{i+1} = (1\text{-}a)\,\mathbf{D}_i + \alpha\omega\,\mathbf{R}$$

where α is a damping constant and controls the rate at which information from **R** is folded into **D** with weighting ω. **R** is a diagonal matrix the elements of which consist of the magnitudes of the successful changes made to each control variable.

Simulated Annealing

SA Algorithm:

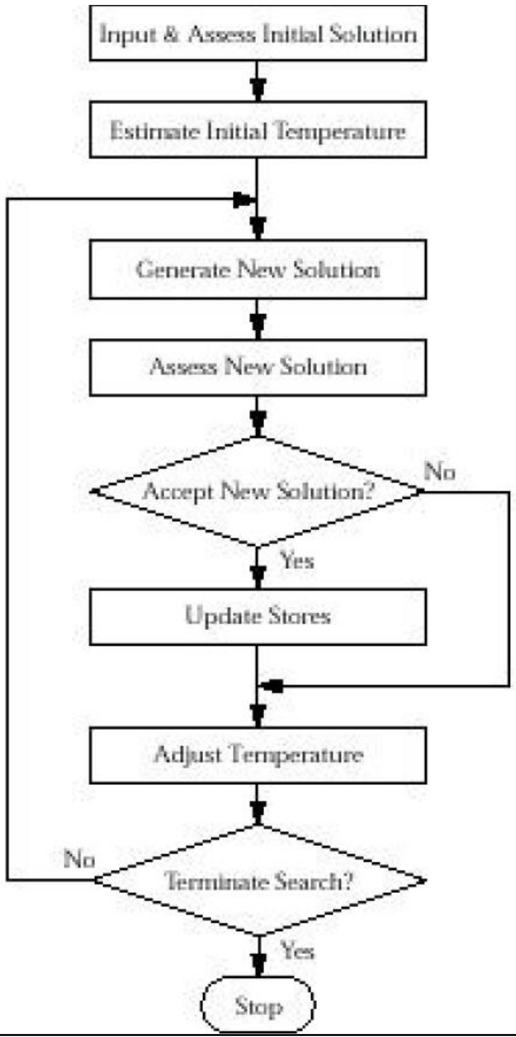
For problems with integer control variables, the simple strategy whereby new trial solutions are generated according to the formula:

$$x_{i+1}=x_i + \mathbf{u}$$

where **u** is a vector of random integers in the range (-1, 1) often suffices.

Simulated Annealing

SA Algorithm:



Particle Swarm Optimization

Introduction:

Particle swarm optimization (PSO) is a population based stochastic optimization technique developed by Dr. Eberhart and Dr. Kennedy in 1995, inspired by social behavior of bird flocking or fish schooling.

Particle Swarm Optimization

Introduction:

PSO shares many similarities with evolutionary computation techniques such as Genetic Algorithms (GA).

The system is initialized with a population of random solutions and searches for optima by updating generations.

However, unlike GA, PSO has no evolution operators such as crossover and mutation.

In PSO, the potential solutions, called particles, fly through the problem space by following the current optimum particles.

Particle Swarm Optimization

Introduction:

Each particle keeps track of its coordinates in the problem space which are associated with the best solution (fitness) it has achieved so far. (The fitness value is also stored.)
-This value is called *pbest*.

Another "best" value that is tracked by the particle swarm optimizer is the best value, obtained so far by any particle in the neighbors of the particle.
-This location is called *lbest*.

When a particle takes all the population as its topological neighbors,
-The best value is a global best and is called *gbest*.

Particle Swarm Optimization

Introduction:

The particle swarm optimization concept consists of,
- At each time step, changing the velocity of (accelerating) each particle toward its p*best* and *lbest* locations (local version of PSO).

- Acceleration is weighted by a random term, with separate random numbers being generated for acceleration toward *pbest* and *lbest* locations.

Particle Swarm Optimization

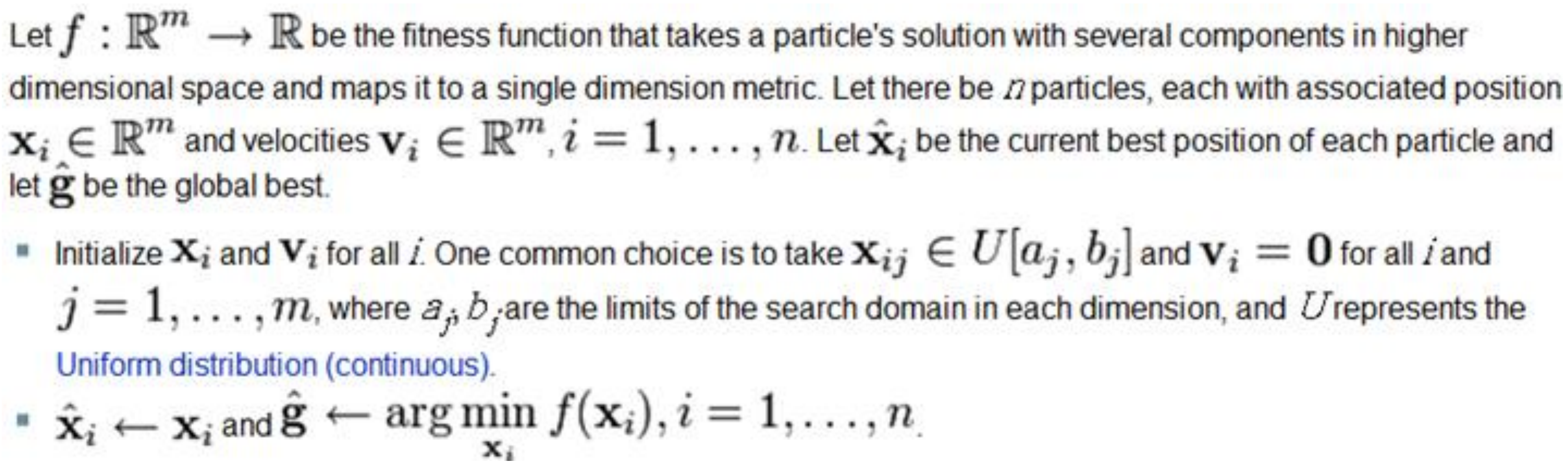
Method:

The particle position and velocity update equations in the simplest form that govern the PSO are given by

- $v_{i,j} \leftarrow c_0 v_{i,j} + c_1 r_1 (globalbest_j - x_{i,j}) + c_2 r_2 (localbest_{i,j} - x_{i,j}) + c_3 r_3 (neighborhoodbest_j - x_{i,j})$
- $x_{i,j} \leftarrow x_{i,j} + v_{i,j}.$

Particle Swarm Optimization

Algorithm:

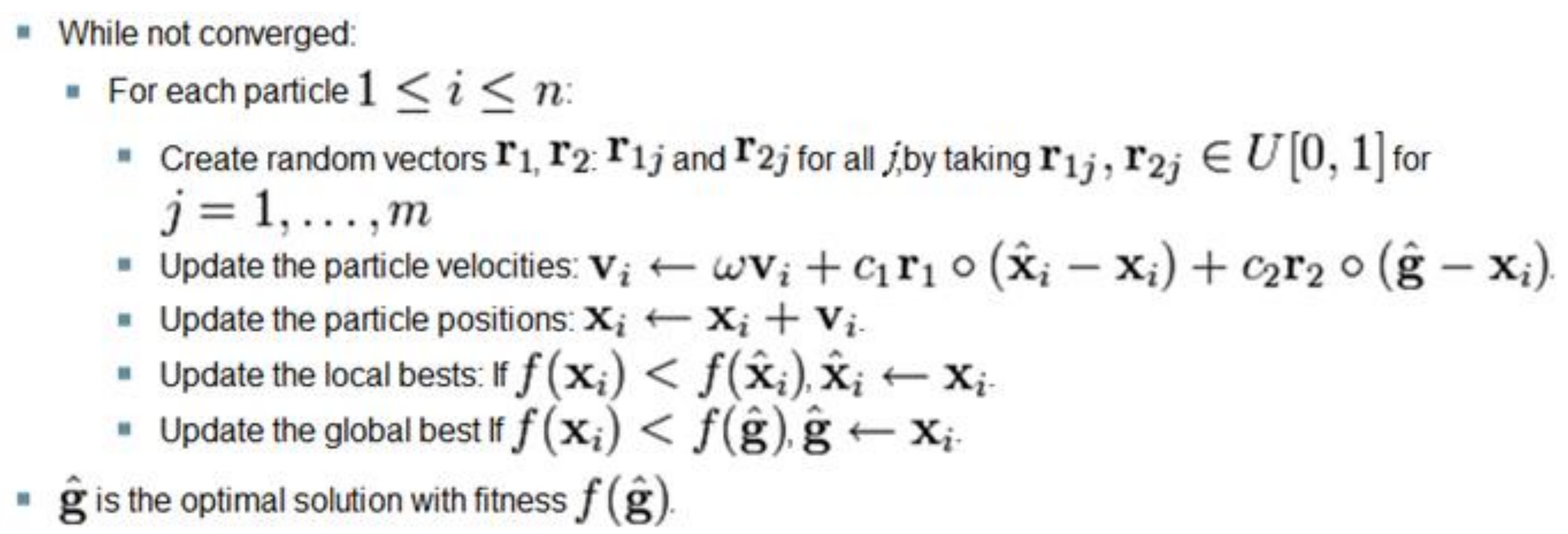


Let $f : \mathbb{R}^m \rightarrow \mathbb{R}$ be the fitness function that takes a particle's solution with several components in higher dimensional space and maps it to a single dimension metric. Let there be n particles, each with associated position $\mathbf{x}_i \in \mathbb{R}^m$ and velocities $\mathbf{v}_i \in \mathbb{R}^m, i = 1, \ldots, n$. Let $\hat{\mathbf{x}}_i$ be the current best position of each particle and let $\hat{\mathbf{g}}$ be the global best.

- Initialize $\mathbf{x}_i$ and $\mathbf{v}_i$ for all i. One common choice is to take $\mathbf{x}_{ij} \in U[a_j, b_j]$ and $\mathbf{v}_i = \mathbf{0}$ for all i and $j = 1, \ldots, m$, where a_j, b_j are the limits of the search domain in each dimension, and U represents the Uniform distribution (continuous).
- $\hat{\mathbf{x}}_i \leftarrow \mathbf{x}_i$ and $\hat{\mathbf{g}} \leftarrow \arg\min_{\mathbf{x}_i} f(\mathbf{x}_i), i = 1, \ldots, n$.

Particle Swarm Optimization

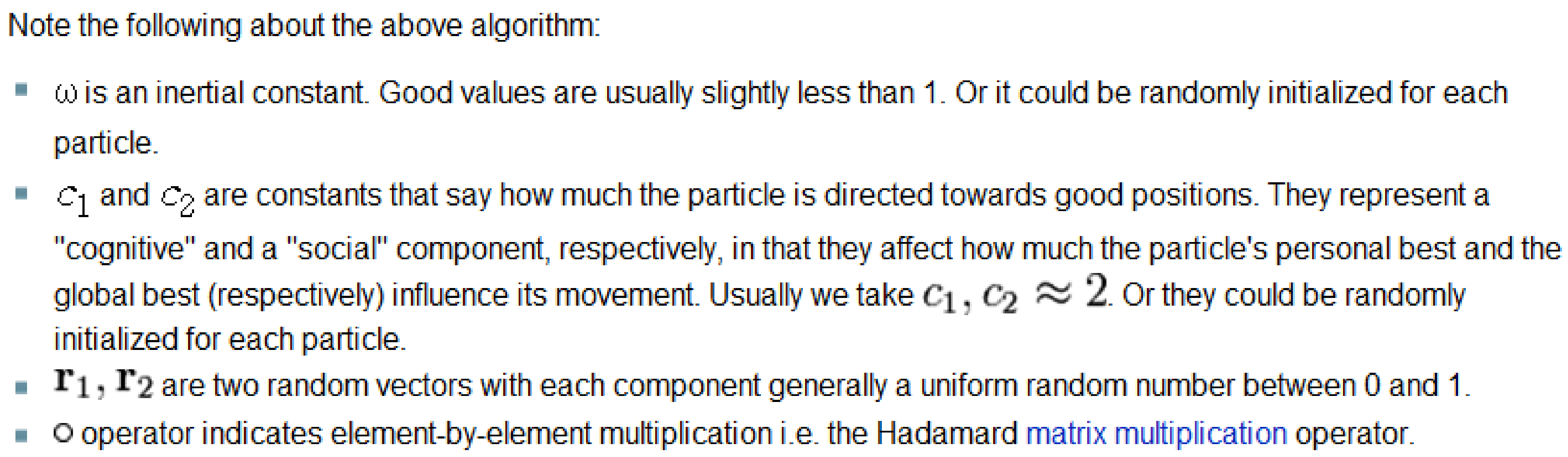
Algorithm:



- While not converged:
  - For each particle $1 \le i \le n$:
    - Create random vectors $\mathbf{r}_1$, $\mathbf{r}_2$: $\mathbf{r}_{1j}$ and $\mathbf{r}_{2j}$ for all j, by taking $\mathbf{r}_{1j}, \mathbf{r}_{2j} \in U[0,1]$ for $j = 1, \ldots, m$
    - Update the particle velocities: $\mathbf{v}_i \leftarrow \omega \mathbf{v}_i + c_1 \mathbf{r}_1 \circ (\hat{\mathbf{x}}_i - \mathbf{x}_i) + c_2 \mathbf{r}_2 \circ (\hat{\mathbf{g}} - \mathbf{x}_i)$.
    - Update the particle positions: $\mathbf{x}_i \leftarrow \mathbf{x}_i + \mathbf{v}_i$.
    - Update the local bests: If $f(\mathbf{x}_i) < f(\hat{\mathbf{x}}_i)$, $\hat{\mathbf{x}}_i \leftarrow \mathbf{x}_i$.
    - Update the global best If $f(\mathbf{x}_i) < f(\hat{\mathbf{g}})$, $\hat{\mathbf{g}} \leftarrow \mathbf{x}_i$.
- $\hat{\mathbf{g}}$ is the optimal solution with fitness $f(\hat{\mathbf{g}})$.

Particle Swarm Optimization

Algorithm:



Note the following about the above algorithm:

- ω is an inertial constant. Good values are usually slightly less than 1. Or it could be randomly initialized for each particle.
- c_1 and c_2 are constants that say how much the particle is directed towards good positions. They represent a "cognitive" and a "social" component, respectively, in that they affect how much the particle's personal best and the global best (respectively) influence its movement. Usually we take $c_1, c_2 \approx 2$. Or they could be randomly initialized for each particle.
- $\mathbf{r}_1, \mathbf{r}_2$ are two random vectors with each component generally a uniform random number between 0 and 1.
- $\circ$ operator indicates element-by-element multiplication i.e. the Hadamard matrix multiplication operator.

Particle Swarm Optimization

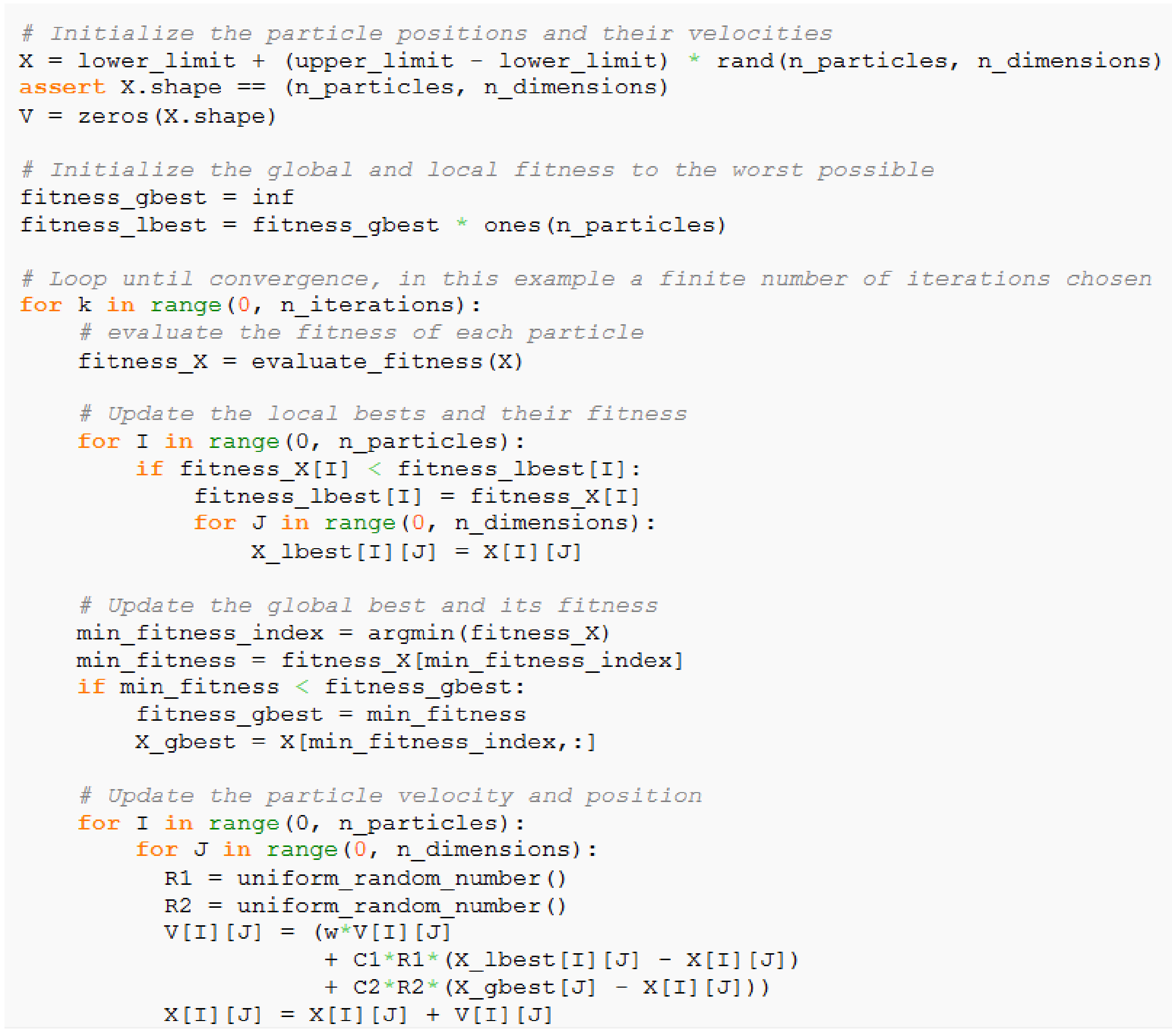
Algorithm:

- There is a misconception arising from the tendency to write the velocity formula in a "vector notation" (see for example D.N. Wilke's papers). The original intent (see M.C.'s "Particle Swarm Optimization, 2006") was to multiply a NEW random component per dimension, rather than multiplying the same component with each dimension per particle. Moreover, r_1 and r_2 are supposed to consist of a single number, defined as c_{max}, which normally has a relationship with ω (defined as c_1 in the literature) through a transcendental function, given the value Φ: $c_1 = 1.0 / (\Phi - 1.0 + (v_\Phi * v_\Phi) - (2.0 * v_\Phi))$ - and - $c_{max} = c_1 * \Phi$. Optimal "confidence coefficients" are therefore approximately in the ratio scale of $c_1 = 0.7$ and $c_{max} = 1.43$. The pseudo code shown below however, describes the intent correctly.

HÁSKÓLI ÍSLANDS
VERKFRÆÐI- OG NÁTTÚRUVÍSINDASVIÐ
IÐNAÐARVERKFRÆÐI-, VÉLAVERKFRÆÐI-
OG TÖLVUNARFRÆÐIDEILD

Particle Swarm Optimization

Algorithm:



```
# Initialize the particle positions and their velocities
X = lower_limit + (upper_limit - lower_limit) * rand(n_particles, n_dimensions)
assert X.shape == (n_particles, n_dimensions)
V = zeros(X.shape)

# Initialize the global and local fitness to the worst possible
fitness_gbest = inf
fitness_lbest = fitness_gbest * ones(n_particles)

# Loop until convergence, in this example a finite number of iterations chosen
for k in range(0, n_iterations):
    # evaluate the fitness of each particle
    fitness_X = evaluate_fitness(X)

    # Update the local bests and their fitness
    for I in range(0, n_particles):
        if fitness_X[I] < fitness_lbest[I]:
            fitness_lbest[I] = fitness_X[I]
            for J in range(0, n_dimensions):
                X_lbest[I][J] = X[I][J]

    # Update the global best and its fitness
    min_fitness_index = argmin(fitness_X)
    min_fitness = fitness_X[min_fitness_index]
    if min_fitness < fitness_gbest:
        fitness_gbest = min_fitness
        X_gbest = X[min_fitness_index,:]

    # Update the particle velocity and position
    for I in range(0, n_particles):
        for J in range(0, n_dimensions):
          R1 = uniform_random_number()
          R2 = uniform_random_number()
          V[I][J] = (w*V[I][J]
                     + C1*R1*(X_lbest[I][J] - X[I][J])
                     + C2*R2*(X_gbest[J] - X[I][J]))
          X[I][J] = X[I][J] + V[I][J]
```