Particle Swarm Optimization

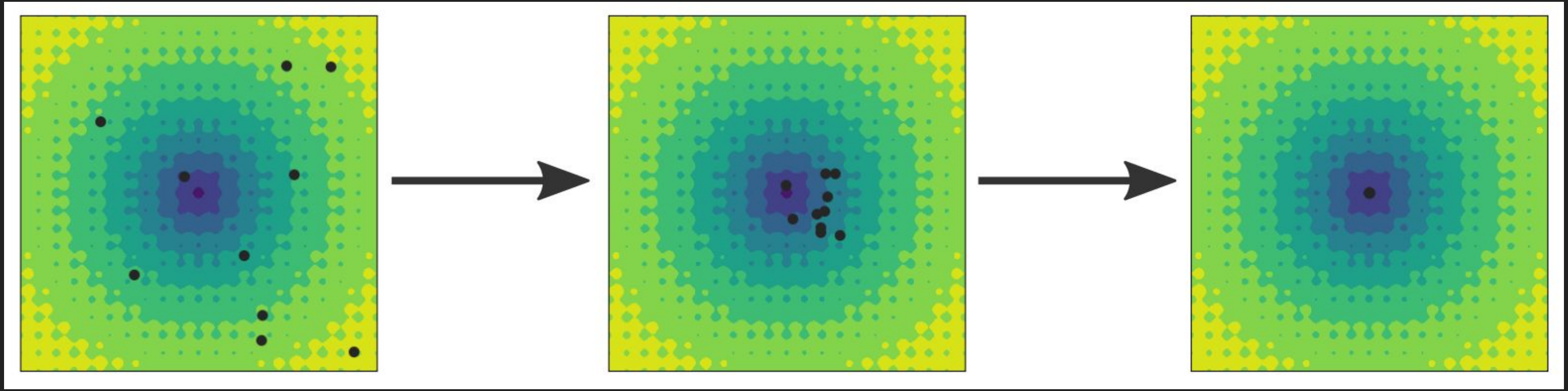
Leoni Mota Loris. CMP585

Swarm intelligence.

- Swarm intelligence is a type of artificial intelligence based on the collective behavior of self-organized systems.
- In general, such systems are made up of a population of simple agents interacting locally with one another and with the environment.
- Individual agents follow simple rules dictating how it should interact with the environment.
- Natural examples of swarm intelligence are bacterial growth, bird flocking, herds of animals, etc.

Particle Swarm Optimization

- Models the problem to be optimized as an *n-dimensional* surface where one wants to find its global minimum.
- Each particle represents a possible solution (minimum) to the problem.
- The entire set of particles searches for an optimal solution by upgrading its generations.
- But **it does not resample** its particles, unlike other population-based algorithms.

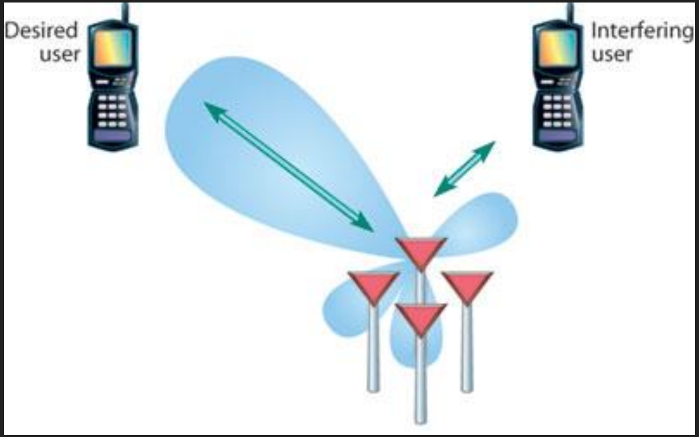


Direct Applications

- Adaptive Antenna Arrays
- Molecular Docking

Adaptive Antenna Arrays

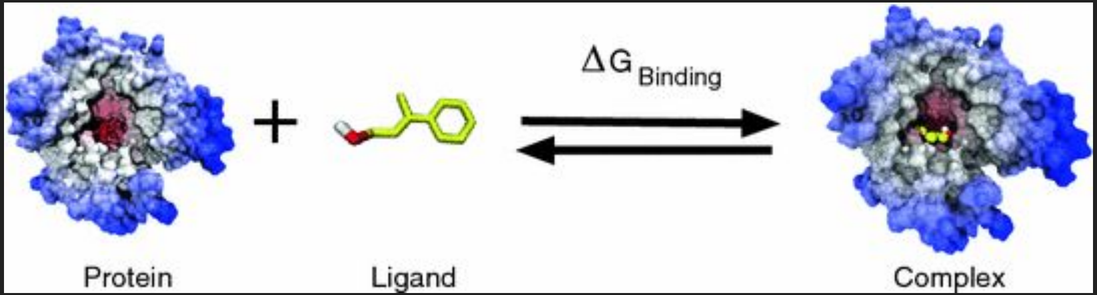
- Each antenna applies a gain and a phase-shift
- Thus it is possible to place `Lobes` and `nulls` on the irradiation pattern.
- In an adaptive array, the antennas need to be able to determine the optimal parameters in real time when a signal is detected.
- Thus, position, phase and amplitude of each antenna will be parameters to be optimized.



Yong-Chang Jiao, et al, "Synthesis of antenna array using particle swarm optimization", APMC2010

Molecular Docking.

- Given a geometric and chemical description of a protein and an arbitrary small organic molecule. We want to know if it will bind, and how strong will be this binding.
  - Example: *Benzamidine* (*Trypsin* inhibitor), docks into the active site of *Trypsin* (trypsin catalyzes the hydrolysis of peptide bonds, so it will eventually be absorbed down to the bloodstream), a protease involved in digestion.



Molecular Docking.

- The PSO is used to optimize the following parameters:
  - Translation: Ligand center with respect to the grid that encloses the binding site. (x, y, z).
  - Orientation: *[n_x, n_y, n_z]* and *alpha* representing the normal vector and its rotation.
  - Torsions: Torsion angle for each if *T* its rotating bonds.

$$E_{tot} = E_{vdW} + E_{Hbonds} + E_{elecpot} + E_{intern}$$

Liu BF, Chen HM, Huang HL, Hwang SF and Ho SY (2005) Flexible protein-ligand docking using particle swarm optimization, in Proc. of Congress on Evolutionary Computation (CEC 2005), IEEE Press, Washinton DC.

PSO Details.

- Each particle needs to know two values (besides its own velocity and position)
  - Global best position: The fittest position (optimizer parameter) yielded by the swarm.
  - Personal best position: The fittest position (optimizer parameter) yielded by the particle itself, so far.

```
class Particle:
    def __init__(self, inital_position):
        self._min_error = np.Inf
        self._n_dimensions = len(inital_position)
        self._error = np.Inf
        self._position = inital_position.copy()
        self._velocity = np.random.uniform(-1.0, 1.0, size=len(inital_position))
        self._best_position = np.zeros_like(self._position)


    def update_velocity(self, global_best_position):
        ...
```

PSO Details.

- At each step, a new speed must be calculated, the speed tells the particle which direction it should look for a fitter solution.
- Each particle must consider three points whenever it will compute a new speed
    - It should not vary its own speed too much, since it was heading towards a good solution (**momentum**).
    - It should try to go towards its own fittest position, so far, for it may be the globally fittest (**cognitive**).
    - It should try to go towards the fittest solution found by the entire swarm (**social**).

$$\begin{gathered} V_{t+1} = momentum * V_t + \\ r_1 * cognitive * (X_{particlebest} - X_{particle}) + \\ r_2 * social * (X_{swarmbest} - X_{particle}) \end{gathered}$$

PSO Details.

```
def update_velocity(self, global_best_position):
    inertia = 0.9
    cognitive_constant = 2
    social_constant = 1

    r1 = 0.5 * (np.random.uniform(size=self._n_dimensions) + 1)
    r2 = 0.5 * (np.random.uniform(size=self._n_dimensions) + 1)

    cognitive_velocity = cognitive_constant * r1 * (self._best_position - self._position)
    social_velocity = social_constant * r2 * (global_best_position - self._position)
    self._velocity = inertia * self._velocity + cognitive_velocity + social_velocity
```

PSO Details.

- Thus, each particle will update its position and evaluate how fit it was.
  - It will compute the new position as follows:

$$\frac{X_{t+1} - X_t}{t + 1 - t} = V_t$$

```
def update_fitness(self, cost_func):
    self._error = cost_func(self._position)
    if self._error < self._min_error:
        self._best_position, self._min_error = self._position, self._error
    return self._error

def update_fitness(self, min_position=-100, max_position=100):
    self._position = self._position + self._velocity
```

Code example, PSO variations and discussions:

- Why using a constant coefficients?
  - One particle could reason following the logics:
    - The better I am the more I follow my own way.
    - The better is my best neighbour the more I tend to go towards him

- Why using a constant population?
  - One particle could also reason following the logics:
    - The system improved as a whole, but I'm the worst. Gonna kill myself!
    - I'm the best particle, but I'm not improving that much, comparing to the swarm. Generate more particles then!