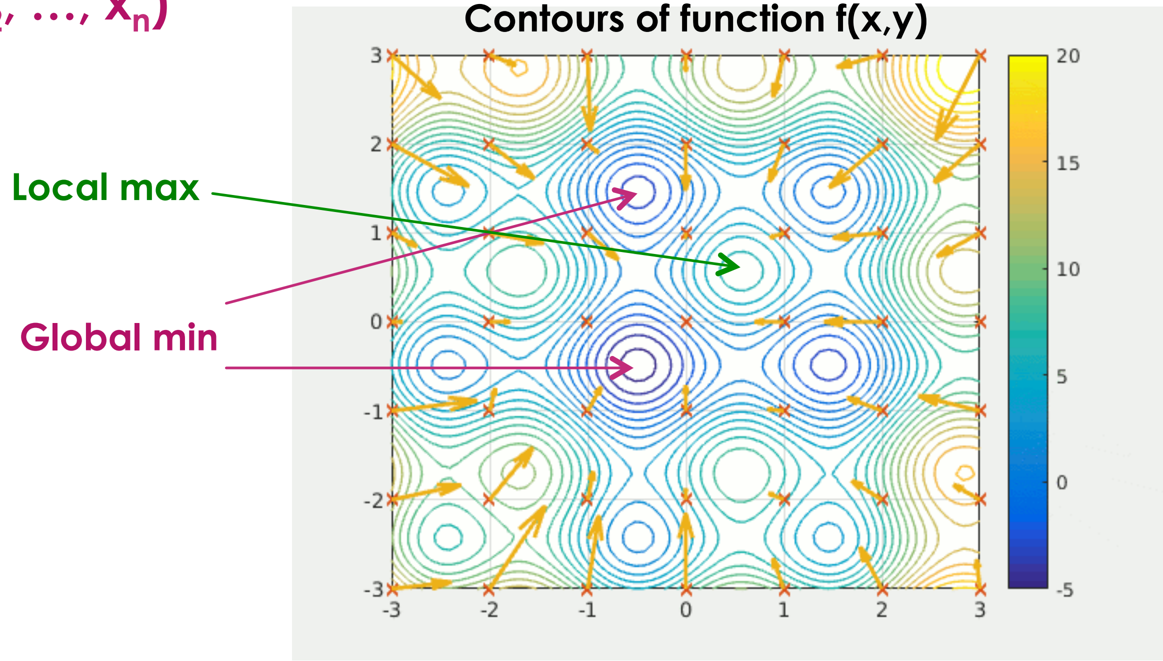
Particle Swarm Optimization (PSO)

Particle Swarm Optimization

- ✓ **Optimization of an *n*-dimensional function f(x_1, x_2, …, x_n)**
  - Gradient-based family (**faster**)
    - **Requires** partial derivatives ($\partial f / \partial x_i$)
  - Derivative-less family
    - Particle Swarm Optimization (PSO)
    - No need to compute derivatives
  - Swarm behavior (bees, ants) is random
    - BUT follows a **general objective**
    - An interesting idea for optimizing a function
      - Global or local extrema (maximum, minimum) → Depends on the strength of the social interaction of particles



- **References:**

[1] https://www.sciencedirect.com/topics/engineering/particle-swarm-optimization

[2] https://en.wikipedia.org/wiki/Particle_swarm_optimization

Particle Swarm Optimization

Particle swarm optimization **From Wikipedia**

From Wikipedia, the free encyclopedia

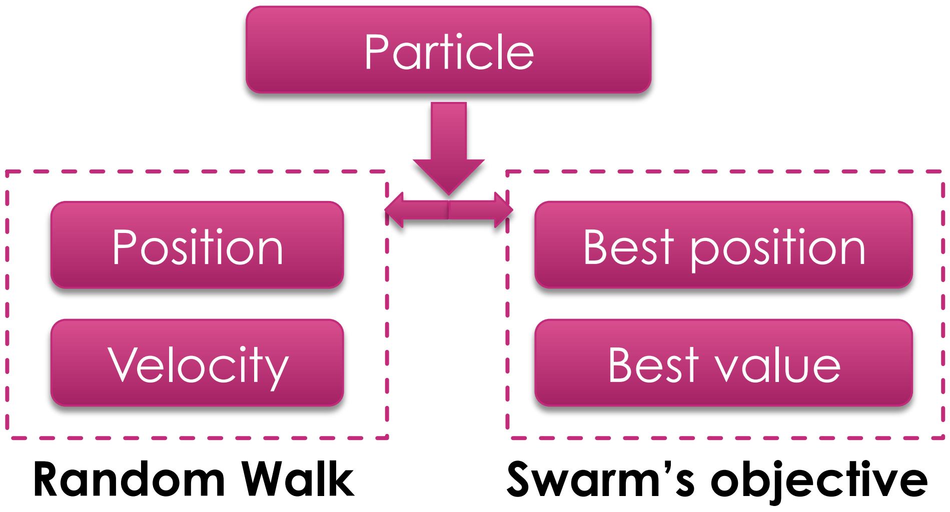
In computational science, particle swarm optimization (PSO) is a computational method that optimizes a problem by **iteratively** trying to improve a candidate solution with regard to a given measure of quality. It solves a problem by having a **population** of candidate solutions, here dubbed particles, and moving these particles around in the **search-space** according to **simple** mathematical formulae over the particle's **position** and **velocity**. Each particle's movement is influenced by its local best known position, but is also guided toward the best known positions in the search-space, which are updated as better positions are found by other particles. This is expected to move the swarm toward the best solutions.

PSO is originally attributed to Kennedy, Eberhart and Shi and was first intended for simulating social behaviour, as a stylized representation of the movement of organisms in a bird flock or fish school. The algorithm was simplified and it was observed to be performing optimization. The book by Kennedy and Eberhart describes many philosophical aspects of **PSO** and **swarm intelligence**. An extensive survey of PSO applications is made by Poli. Recently, a comprehensive review on theoretical and experimental works on PSO has been published by Bonyadi and Michalewicz.

PSO is **metaheuristic** as it makes **few or no assumptions about the problem being optimized** and can search very large spaces of candidate solutions. However, metaheuristics such as PSO **do not guarantee an optimal solution** is ever found. Also, **PSO does not use the gradient** of the problem being optimized, which means PSO does not require that the optimization problem be differentiable as is required by classic optimization methods such as gradient descent and quasi-newton methods.

Particle Swarm Optimization

- **Fitness Function $f(x_1, x_2, \ldots, x_n)$**
  - N-dimensional function (N variables)
  - Each variable, x_i, is initialized in a given interval
- **Search Space**
  - How many particles are used
    - Dimensionality of the **swarm**
- Dimension of PSO space = number of particles
- Dimension of each particle = number of variables
- **Optimization Objective (swarm's objective)**
  - Best known position for each particle
  - Best known value of the fitness function for each particle
  - **Minimizing**: best value > fitness(position) → update best value and best position
  - **Maximizing**: best value < fitness(position) → update best value and best position



Particle Swarm Optimization

✓ How to implement PSO

- **Step 1:** create a given number of particles
  - Each particle has an initial random position in a given interval
  - Each particle has a simple "move" method (random walk)

Iteration index

Velocity (*n*-dim)

Movement

$$v_i^{(m+1)} = \underline{w} \times v_i^{(m)} + \underline{c_1 r_1}(x_{best,i}^{(m)} - x_i^{(m)}) + \underline{c_2 r_2}(g_{best,i}^{(m)} - x_i^{(m)})$$

$$x_i^{(m+1)} = x_i^{(m)} + v_i^{(m+1)}$$

Position (*n*-dim)

Particle's behavior

Swarm's behavior

Particle index

- **Step 2:** create a space of particles
  - This is the search space in which the particles can move
  - Can move the swarm of particles based on the fitness function
- **Step 3:** implement PSO
  - "solve" method
    - Runs the iteration and updates the best solution

inertia weight

Social interaction

Parameters in each iteration

w: 0.5 → [0.5,0.9]
c_1: 0.8
c_2: 0.9
r_1: random [0,1]
r_2: random [0,1]

Particle Swarm Optimization

- **Pseudocode** (from Wikipedia)

S = search space dimension

Initial location randomly in the search interval

```
for each particle i = 1, ..., S do
   Initialize the particle's position with a uniformly distributed random vector: xi ~ U(blo, bup)
   Initialize the particle's best known position to its initial position: pi ← xi
   if f(pi) < f(g) then
       update the swarm's best known  position: g ← pi
   Initialize the particle's velocity: vi ~ U(-|bup-blo|, |bup-blo|)
while a termination criterion is not met do:
   for each particle i = 1, ..., S do
      for each dimension d = 1, ..., n do
         Pick random numbers: rp, rg ~ U(0,1)
         Update the particle's velocity: vi,d ← ω vi,d + φp rp (pi,d-xi,d) + φg rg (gd-xi,d)
      Update the particle's position: xi ← xi + vi
      if f(xi) < f(pi) then
         Update the particle's best known position: pi ← xi
         if f(pi) < f(g) then
            Update the swarm's best known position: g ← pi
```

f → Fitness function

Use interval object

Particle's best position

We will initialize to zero

We set the termination criterion to the **number of iterations**

w

c_1

c_2

Movement equation

Swarm's objective

This algorithm **minimizes** the function

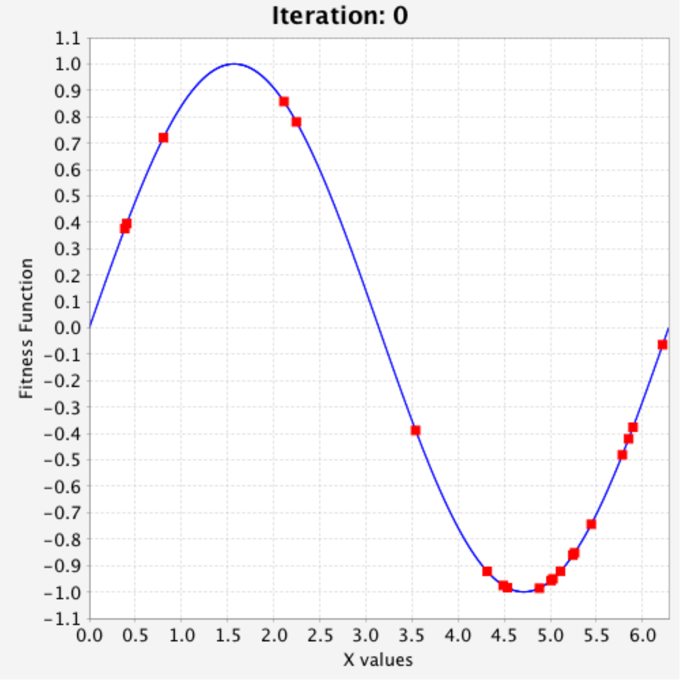
Tip: Swarm's objective can be set to <u>minimize</u> or <u>maximize</u> the function

Particle Swarm Optimization

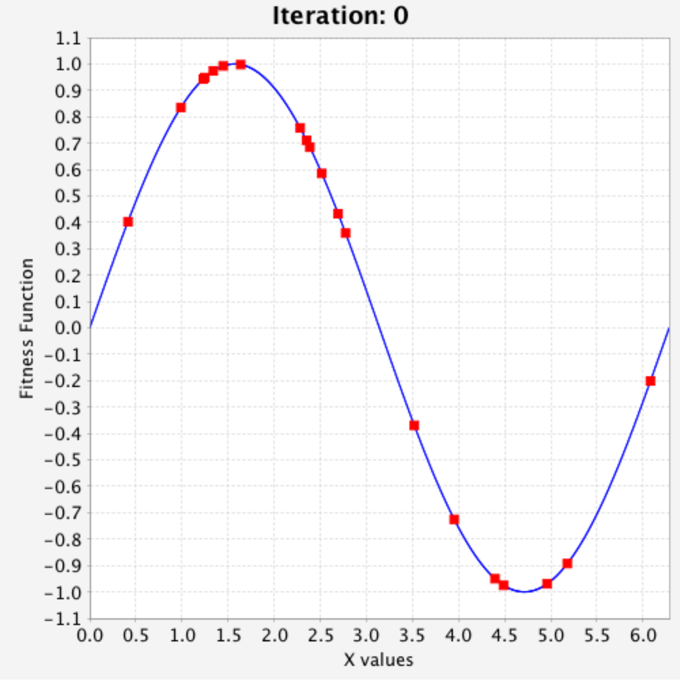
- Example of 1-D function
  - **f(x) = sin(x)**
  - 20 particles, 10 iterations
  - Search interval = [0, 2π]
  - Objective = Maximize
- How to create animations?
  - Use **ImageJ** library
  - Open source java library
  - Stand-alone application
  - Save particles at each iteration
  - Import a sequence of images
  - Create a gif animation image
  - Export as AVI movie

$$f(x) = \sin(x)$$

Maximization



Minimization

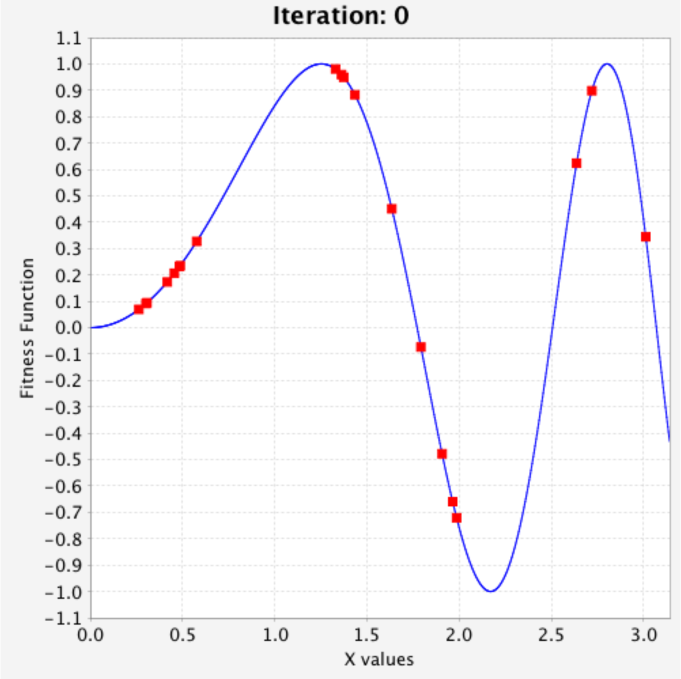


Particle Swarm Optimization

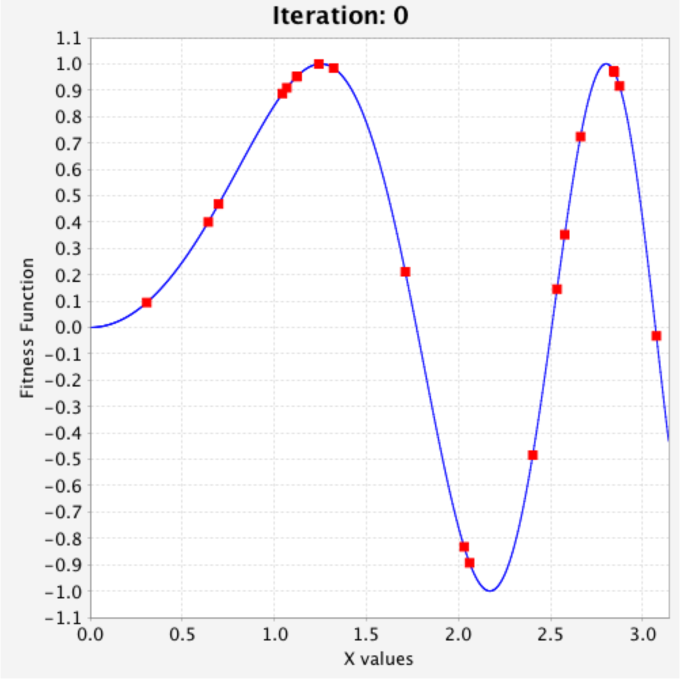
- Example of 1-D function
  - **f(x) = sin(x²)**
  - 20 particles
  - Search interval = [0, π]
  - Objective = Maximize
- How to create animations?
  - Use **ImageJ** library
  - Open source java library
  - Stand-alone application
  - Save particles at each iteration
  - Import a sequence of images
  - Create a gif animation image

$$f(x) = \sin(x^2)$$

Maximization



Minimization



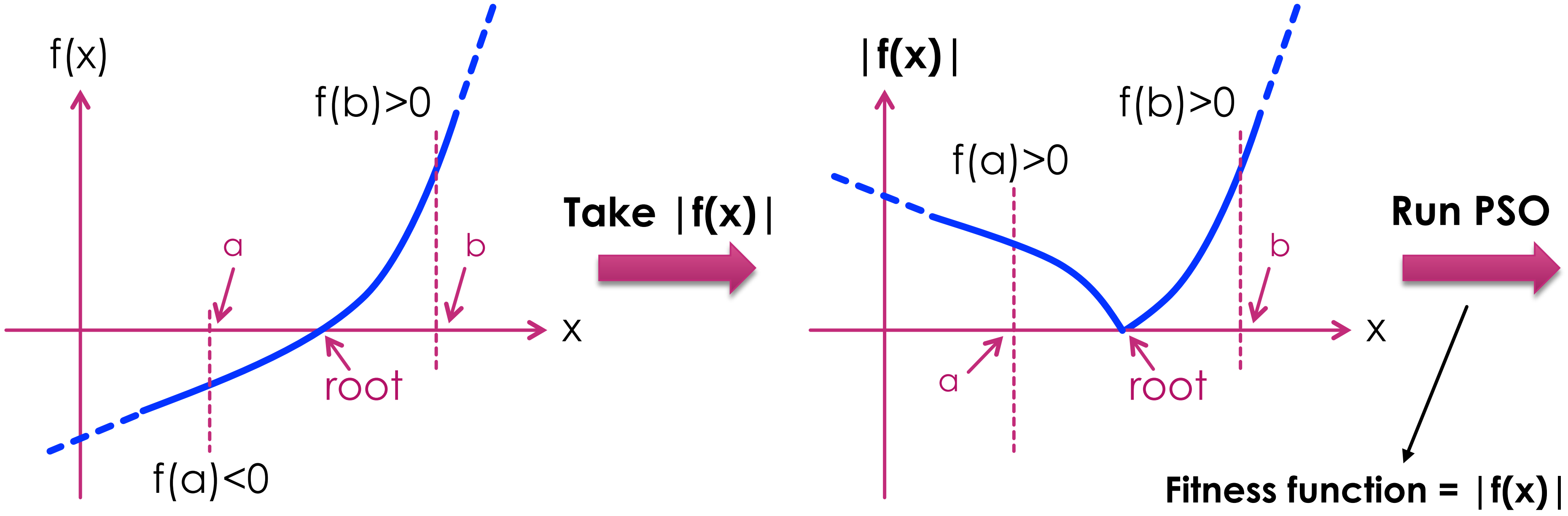
PSO for Solving Equations

- Using PSO to solve equations (1-D): **$f(x) = 0$**
  - Real function
  - Real roots
- PSO is a very lightweight optimization routine: function evaluation & multiplications & additions
- Problems with the common root finding algorithms → **Root bracketing**
  - General lemma $f(a).f(b) < 0$ ➡ f(x) MUST be continuous over (a,b)
  - Bisection, Trisection ➡ (a,b) MUST be a bracket. Failure at even tangent roots: $f(x) \sim (x-r)^{2n}$
  - Secant
  - False position (improved secant)
  - Brent

  ➡ f(x) MUST be continuous over (a,b)
  (a,b) MUST be a bracket for the root
  - Newton-Raphson ➡ f(x) MUST be smooth over (a,b). $f'(x_n) = 0$ was a problem.
- We want an algorithm that finds the roots for
  - Discontinuous functions, non-smooth functions, a non-bracket interval, …

PSO for Solving Equations

- Using PSO to solve equations: **f(x) = 0**
- A root occurs when f(x) touches (not crosses) x-axis
  - Minimum point of |**f(x)**| → Run PSO and check if the minimum is the root



- Initialize in (a, b)
- 1D fitness function
- Small No. of Particles
- Objective: **Minimize**
- Final test: **f(min) =?= 0**

Tip: 10 Particles should suffice. Use 50 to 100 iterations for PSO.

Bounded PSO

- Bounded PSO optimization
- Implemented PSO algorithm results in an **unbounded** optimization
- Sometimes we want particles to always remain in the initial search space
- Use a class that checks the position of each particle at each step with regard to the search space
  - If particle goes **outside**
    - move it back (or don't move it)
  - If particle is outside
    - don't update the velocity

```java
public class ParticleSpace {

    ArrayList<Interval> intervals ;

    // vararg constructor
    public ParticleSpace(Interval...intervals) {
        this.intervals = new ArrayList<>() ;
        for(Interval interval : intervals)
            this.intervals.add(interval) ;
    }

    public boolean isParticleInside(Particle particle) {
        double[] position = particle.position.x ;
        // component i of the position[i] --> interval[i]
        for(int i=0; i<intervals.size(); i++)
            if(!intervals.get(i).isInside(position[i]))
                return false ;
        return true ; // default state
    }
}
```

Solving Inequalities with PSO

- We can use PSO to find a desired solution for a set of constrained inequalities

$$\underbrace{\begin{array}{l} a_0 < x_0 < b_0 \\ a_1 < x_1 < b_1 \\ \dots \\ a_{n-1} < x_{n-1} < b_{n-1} \end{array}}_{\text{Constraints on variables}} \quad + \quad \underbrace{\begin{array}{l} c_0 < f_0(x_0, \dots, x_{n-1}) < d_0 \\ c_1 < f_1(x_0, \dots, x_{n-1}) < d_1 \\ \dots \\ c_{m-1} < f_{m-1}(x_0, \dots, x_{n-1}) < d_{m-1} \end{array}}_{\text{Desired inequalities (}\mathbf{m}\text{ inequalities)}}$$

- Define a fitness function that is minimized or maximized when all conditions of $f_n(x)$ hold true
  - Set the "bounded optimization" flag to **true** in the PSO

Fitness function

$$f(x) = \begin{cases} +1 & \text{if} \quad c_i < f_i(x_0, \dots, x_{n-1}) < d_i \\ -1 & \text{else} \end{cases}$$

This function will be maximized if all the inequalities are simultaneously satisfied

Solving Inequalities with PSO

- **Example:**
- Inequality functions
  - $f_1(x, y) = x^2 + y^2 <= 1$
  - $0 < f_2(x, y) = x + y < 1$
- Variables' domains [PSO bounds]
  - $-2 < x < 2$
  - $-5 < y < 5$
- Fitness function
  - Func = -1 if f_1 is true and f_2 is true
  - Func = +1 else
  - **Minimize** this function