Particle Swarm Optimization

Introduction

The problem: In this assignment, we will be using the Particle Swarm algorithm to find the global minimum of a non-convex function. Particle Swarm Optimization is a biologically inspired algorithm that mimics a flock of birds or a school of fish. Similar to the Genetic Algorithm, PSO starts with a randomly created initial population, a set of possible solutions (called particles), and updates these particles after every iteration. However, GA uses crossover and mutation to create new individuals, whereas PSO maintains the same population but changes each particle over time.

We have provided code stubs in this notebook to get you started, and give hints about the structure of the code.

You need to implement Particle Swarm Optimization, and analyze the results.

Approach:

Let's start with a few definitions.

- Particle (aka "individual"): a solution to the non-convex function $f(\overrightarrow{x})$, which we are trying to minimize. The particle's position in the swarm is defined to be this vector, which represents a possible solution to the problem. The particle also has a velocity vector, and variables that record the "best" (lowest) value found so far.
- Population: a collection of possible solutions to the function (i.e., a collection of Particles)
- Fitness: a function that tells us how good each Particle is (in our case, how low of a value the Particle yields. The lower the fitness the better. -Best Value: another way to talk about the fitness; this is defined as the Particle's lowest value found so far, from plugging in a position into $f(\overrightarrow{x})$

Our Particle Swarm will proceed in the following steps:

1. Create the population (initialized with random positions and velocities)
2. Initialize the global best value and all the Particles' individual best values. All best value initializations should be to $+\infty$
3. Update each Particle's individual best as well as the global best value found so far
4. Update each Particle's velocity subject to the equation learned in class, and then update the Particle's position
5. Update w
6. Repeat steps 3-4 for until the solution has converged
7. Plot your results (best values)

Algorithm:

At every iteration, the position of each Particle gets updated. Each Particle has a position vector, x, and a velocity vector, v. The velocity vector is changed, and then each element of the velocity vector gets added to each element of the position vector. The length of x and v is equal to dimension (n).

The velocity vector is updated in the following way:

v_i^{t+1} = inertia + cognitive velocity + social velocity

- inertia = w * v_i^t
- cognitive velocity = c_1 r_1 (personal best position - current position)
- social velocity = c_2 r_2 (global best position - current position)

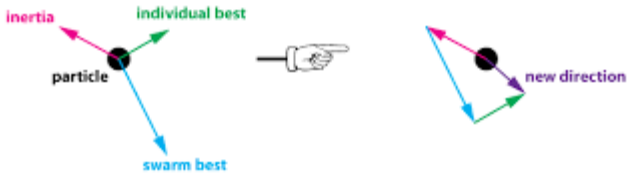
And the position vector is then updated:

x_i^{t+1} = x_i^t + v_i^{t+1}

r_1 and r_2 are random numbers between 0 and 1, and c_1 and c_2 are the cognitive and social parameters, respectively.

The direction that the particle will move in each iteration depends on these three vectors (inertia, cognitive velocity, social velocity). Cognitive velocity pushes the Particle towards its personal best, and social (global)

velocity pushes the Particle towards the global best.



In [72]:
```
import numpy as np, random, operator, pandas as pd, matplotlib.pyplot as pl
t
import operator
import math
import sys
from datetime import datetime
from util import plot_particles, record_to_pv
```

We will set global variable dimension as the number of variables in the function we are minimizing.

$$f(\overrightarrow{x}) = x_1^2 + (x_2 + 1)^2 - 5cos(1.5x_1 + 1.5) - 5cos(2x_2 - 1.5)$$

The function is written below.

In [73]:
```
dim = 2
def func(x):
    x_ = x[0]
    y_ = x[1]
    return x_**2 + (y_+1)**2 - 5*np.cos(1.5*x_ + 1.5) - 5 * np.cos(2* y_ -
1.50)
```

Create necessary classes and functions

Next, we create a class that will represent and handle an individual Particle in the population. A Particle only needs its position vector to be initialized. Velocity starts with random values, and we have variables that keep track of this Particle's best position so far and the value that this position yields. This means that this is the position, or solution, found by this Particle that gives the lowest value when plugged into the polynomial.

Fitness is simply the value of the polynomial when we plug in this Particle's position.

There are functions that are used to update the velocity and position of the Particle based on the equations above. When updating position, we must make sure each element of position stays within the desired bounds. This means that if the new position value is outside of the interval $[-100, 100]$, we set the value to the boundary value.

1.1 (5 points)

TO-DO

1. `update_velocity`
2. `update_position`

In [56]:
```
class Particle:
    def __init__(self, position):
        """
        position: the position of this Particle, a list of length 2
        velocity: the velocity of this Particle, a list of length 2
        best_position: the best position found so far by this Particle
        best_value: the best (lowest) value found so far by this Particle
        """
        self.position = position
        self.velocity = np.array([random.uniform(-2,2) for i in range(di
m)])
        self.best_position = np.copy(position)
        self.best_value = math.inf

    def fitness(self):
        """
        This fitness function is defined as the value of this Particle
        when plugged into the function polynomial.
        """
        return func(self.position)

    def update_velocity(self, w, c1, c2, best_position_global):
        """
        w, c1, c2: constants
        best_position_global: the best position found in the whole populati
on so far
        This function updates the Particle's velocity
        """
        new_velocity = (w * self.velocity) + (c1 * random.uniform(0, 1) *
(self.best_position - self.position)) + \
        (c2 * random.uniform(0, 1) * best_position_global)
        self.velocity = new_velocity
        #YOUR CODE HERE

    def update_position(self):
        """
        Updates the Particle's position
        """
        new_position = self.position + self.velocity
        self.position = new_position
        #YOUR CODE HERE
```

Next, we create the PSO_Simulation class. This will start and run the swarm for the algorithm. We create a PSO_Simulation object with initial population, which is the list of Particles that will be used, and the constants.

There is a function used to update the best values of a Particle, and update the global best if necessary.

The main driver of the program is PSO_Simulation.run(), which first updates the best values for each Particle. Then, we begin the iterations. In each iteration, we go through every Particle, update its velocity, update its position, and update its best value if necessary. At the end of each iteration, we update w (set $w = 0.98 * w$), and record the current global best value (lowest value). This function then returns a list of the global bests after each iteration.

1.2 (5 points)

TO-DO

1. `update_best`
2. Complete `run`.

In [57]:
```
class PSO_Simulation:
    def __init__(self, initialPop, w, c1, c2):
        """
        initialPop: list of pop_size Particles
        w, c1, c2: constants
        best_position_global: the best position found by any Particle so fa
r
        best_value_global: the best value found by any Particle so far
        """
        self.particles = initialPop
        self.pop_size = len(initialPop)
        self.w = w
        self.c1 = c1
        self.c2 = c2
        self.best_position_global = np.copy(initialPop[1].position)
        self.best_value_global = np.inf

    def update_best(self, particle):
        """
        Updates the global/personal best position and value if the current
Particle's position is better.
        """
        if particle.fitness() < particle.best_value:
            particle.best_position = particle.position
            particle.best_value = particle.fitness()
        if particle.fitness() < self.best_value_global:
            self.best_position_global = particle.position
            self.best_value_global = particle.fitness()
        #YOUR CODE HERE


    def run(self, iterations):
        best_global_values = [] # Stores the best global values of every it
eration
        particles_position = [] # Stores the particles' positions in each i
teration
        particles_velocity = [] # Stores the particles' velocities in every
iteration
        weights = []            # Stores the weights in each iteration

        #updates the best values for each particle
        for particle in self.particles:
            self.update_best(particle)

        for i in range(iterations):

            # YOUR CODE HERE

            #update particle velocity, position and if necessary, best valu
e
            for particle in self.particles:
                particle.update_velocity(w, c1, c2, self.best_position_glob
al)
                particle.update_position()
```

```
                self.update_best(particle)


            positions = []
            velocities = []

            for particle in self.particles:
                positions.append(particle.position)
                velocities.append(particle.velocity)

            particles_position.append(positions)
            particles_velocity.append(velocities)
            weights.append(self.w)

            self.w = self.w*0.98    #Weight update

            best_global_values.append(self.best_value_global)

        particles_record = [particles_position, particles_velocity]
        return weights, best_global_values, particles_record
```

Create our initial population

Particle generator. We now can make our initial population. To do so, we need a function that produces random Particles. To create an individual, we randomly generate a position vector, setting each element equal to a random number between -30 and 30. Even though we are starting out with a completely random initial population, there is still a chance for convergence.

The first function here produces one random Particle, and in the next function, we create the whole initial population by repeatedly calling randomParticle(). Note: we only have to use these functions to create the initial population.

1.3 (2 points)

TO-DO:

1. `randomParticles`

In [58]:
```
def randomParticle():
    particle = Particle(np.random.uniform(-30, 30, 2))
    return particle
    #YOUR CODE HERE


def initialPopulation(pop_size):
    """
    Create inital population of a given size.
    Returns a list of random Particles
    """
    pop = []
    for i in range(pop_size):
        temp = randomParticle()
        pop.append(temp)

    return pop
```

Final Step

The particleSwarm function puts everything together. We start off by creating the initial population, then create a PSO_Simulation object, which we use to run the program. You should capture the initial best value, final best value, and also the best values over time. We then plot the best value found at every iteration, using the list returned by PSO_Simulation.run(). For particle visualization, you will also need to return an array of particle positions and velocities in each iteration. (An example of the structure of the particle positions : positions.shape = (num. of iterations, pop_size, dim))

In [59]:
```
def particleSwarm(iterations, pop_size, w, c1, c2):
    start = datetime.now()

    pop=initialPopulation(pop_size)
    weights, best_global_values, particles_record = PSO_Simulation(pop,w,c
1,c2).run(iterations)

    print(f'initial best value:{best_global_values[0]}')
    print(f'final best value:{best_global_values[-1]}')

    plt.plot(best_global_values)
    plt.xlabel('iteration')
    plt.ylabel('fitness')

    end = datetime.now()
    print("Time Elapsed for Particle Swarm: " + str(end - start))
    return weights, particles_record , best_global_values
```

Finally, call the function with your desired parameters. Please choose what parameters to use which give you good results.

We recommend keeping iterations less than 100, pop_size less than 200, w under 1, and choosing small values (less than 1) for c_1 and c_2.

1.4 (4 points)

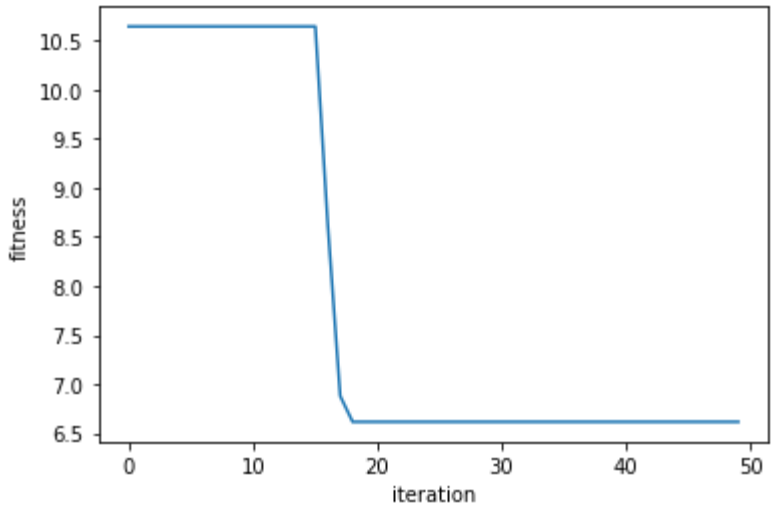
To-do

1. Run `particleSwarm` . (1 point)
2. `plot_particles` will generate two .gif visualizations and they will be saved in the folder where this notebook is located. Experiment with different combinations of parameters to see how the visulizations change.
3. How do c_1 and c_2 affect the particles convergence. (3 points)

In [65]:
```
# Change the parameters
w = 0.5
c1 = 0.05 #originally .05
c2 = 0.05
iterations = 50
pop_size = 100


weights, particles_record, best_global_values = particleSwarm(
    iterations=iterations, pop_size=pop_size, w=w, c1=c1, c2=c2)
positions, velocities = record_to_pv(particles_record)
plot_particles(positions=positions, velocities=velocities, normalize_velocity=True, alphas=weights,
               progresses=best_global_values, name='PSO visualization', additonal_info=f"c1={c1}, c2={c2}")
```

```
initial best value:10.646000727377396
final best value:6.615434471367951
Time Elapsed for Particle Swarm: 0:00:00.120468
```



Increasing c1 and c2 increases the cognitive and social velocity, and makes inertia have less of an effect on the particle's movement.

Firefly Algorithm

The firefly algorithm (FA) was first developed in late 2007. FA was based on the flashing patterns and behavior of fireflies. The flashing light is to attract mating partners and to attract potential prey, and possibly other functions scientists are still debating.

Now we can idealize some of the flashing characteristics of fireflies so as to develop firefly-inspired algorithms. For simplicity in describing the standard FA, we now use the following three idealized rules:

1. All fireflies are unisex, so one firefly will be attracted to other fireflies regardless of their sex.
2. Attractiveness is proportional to a firefly's brightness. Thus for any two flashing fireflies, the less bright one will move towards the brighter one. The attractiveness is proportional to the brightness, both of which decrease as their distance increases. If there is no brighter one than a particular firefly, it will move randomly.
3. The brightness of a firefly is affected or determined by the distance between the itself and the other firefly.

The pseudocode of FA is below: (In our case, since we are minimizing the function, the lower the fitness the better.)

```
Data: Objective function f(x)

Result: Best or optimal solution

Intialization of parameters (pop_size,alpha,beta,gamma)
Generate an initial population of n fireflies x_i (i =1,2,.. pop_size)

while t < iterations do:
    for i = 1: pop_size (all fireflies) do
        for j = 1 :pop_Size (all fireflies) do
            if (firefly i fitness > firefly j fitness) then
                Move firefly i towards j  using the position update equation
        end
    end

    rank the fireflies and find the current global best
    move the firefly with the lowest fitness randomly
    update alpha ( alpha = 0.98 * alpha)
```

The position update equation for the fireflies:

$$x_i^{t+1} = x_i^t + \beta_0 e^{-\gamma r_{ij}^2}(x_j^t - x_i^t) + \alpha \epsilon_i^t$$

, where ϵ is a vector of random numbers drawn from a Gaussian distribution centered around 0 with a standard deviation of 1.

r_{ij} is the distance between fly i and fly j. We can use any distance function for distance r. In this assignment, we will be using the Euclidean distance.

$$r_{ij} = \sqrt{\sum_{k=1}^{d} (x_{i,k} - x_{j,k})^2}$$

β_0 is the brightness at distance = 0. It is common to set it equal to 1.

Now, let's setup the FA. It is similar to the setup of the PSO.

2.1 (5 points)

To-do

In [81]:
```
class Firefly:
    def __init__(self, position):
        """
        position: the position of this firefly.

        """
        self.position = position

    def fitness(self):
        """
        This fitness function is defined as the value of this Particle
        when plugged into the function polynomial.
        """
        return func(self.position)

    def update_position(self, firefly, alpha,  beta, gamma):
        """
        Updates the fly's position
        """
        #YOUR CODE HERE
        distance = np.linalg.norm(self.position - firefly.position)
        new_position = self.position + (beta * math.exp(-gamma * (distance
** 2)) * (self.position - firefly.position)) \
        + (alpha * np.random.normal(0, 1))
        self.position = new_position
```

2.2 (5 points)

To-do:

Follow the pseudocode and complete the `run` method.

In [82]:
```
class FF_Simulation:
    def __init__(self, initialFlies, alpha, beta, gamma):
        """
        initialFlies: list of pop_size Particles
        alpha, beta, gamma: constants
        """
        self.flies = initialFlies
        self.pop_size = len(initialFlies)
        self.alpha = alpha
        self.beta = beta
        self.gamma = gamma


    def run(self, iterations):
        best_global_values = [] # Store the global fitness in every iterati
on
        flies_position = []     # Stores the position of each firefly in ea
ch iteration
        alphas = []             # Stores the alpha values of every iteratio
n

        for t in range(iterations):
            # YOUR CODE HERE
            for i in range(pop_size):
                for j in range(pop_size):
                    if self.flies[i].fitness() > self.flies[j].fitness():
                        self.flies[i].update_position(self.flies[j], alpha,
beta, gamma)


            # Update the positions for plotting.
            positions = []
            for fly in self.flies:
                positions.append(fly.position)

            flies_position.append(np.array(positions))

            alphas.append(self.alpha)

            self.alpha = 0.98*self.alpha # Update alpha

            self.flies.sort(key=lambda x: x.fitness(),reverse=True)
            best_global_values.append(self.flies[-1].fitness())

        return best_global_values, flies_position,alphas
```

Create our initial population

Firefly generator. We now can make our initial population. You can reuse the code you wrote for the PSO functions.

2.3 (2 points)

To-do

1. Complete `randomFlies` .

In [83]:
```
def randomFlies():
    firefly = Firefly(np.random.uniform(-30, 30, 2))
    return firefly
 # YOUR CODE HERE

def initialSwarm(pop_size):
    """
    Create inital population of a given size.
    Returns a list of random flies.
    """
    pop = []
    for i in range(pop_size):
        temp = randomFlies()
        pop.append(temp)

    return pop
```

In [84]:
```
def fireFly(iterations, pop_size, alpha, beta, gamma):

    start = datetime.now()

    pop=initialSwarm(pop_size)
    best_global_values, positions, alphas =  FF_Simulation(pop,alpha,beta,
gamma).run(iterations)

    print(f'initial best value:{best_global_values[0]}')
    print(f'final best value:{best_global_values[-1]}')

    plt.plot(best_global_values)
    plt.xlabel('iteration')
    plt.ylabel('fitness')

    end = datetime.now()
    print("Time Elapsed for Particle Swarm: " + str(end - start))
    return best_global_values, positions, alphas
```

2.4 (7 points)

To-do

1. Run `fireFly` . (1 point) We recommand the following setting:

`iterations` = 100

`pop_size` = 200

`alpha` > 0.4

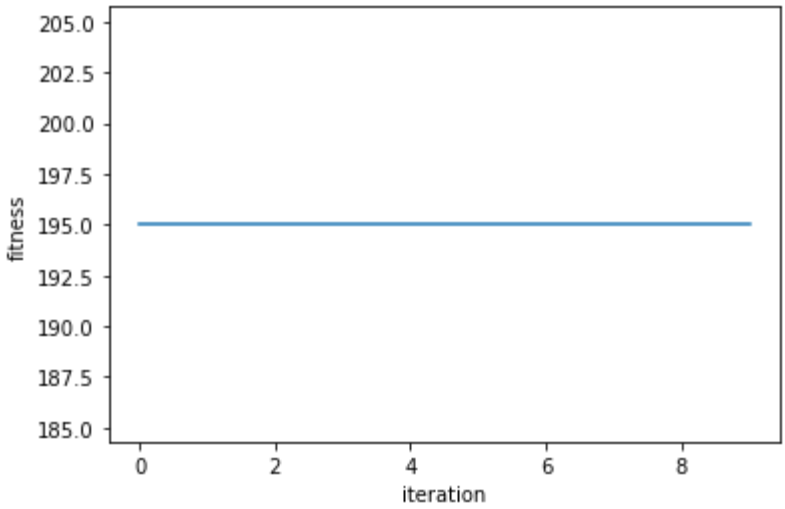
`beta` = 1

`gamma` > 2

1. `plot_particles` will generate two .gif visualizations and they will be saved in the folder where this notebook is located. Experiment with different combinations of parameters to see how the visulizations change.
2. How do the parameters ffect the particles convergence?(3 points)
3. Discuss the similarities and differences between PSO and Firefly algorithm. How is PSO better than the Firefly, or vice versa. (3 points)

In [87]:
```
# Change the parameters.
iterations = 10
pop_size = 20
alpha = 0.5
beta = 1
gamma = 3
best_global_values, positions, alphas = fireFly(iterations=iterations, pop_
size=pop_size, alpha = alpha, beta = beta, gamma =gamma)
plot_particles(positions=positions, velocities=None, normalize_velocity=Tru
e, progresses=best_global_values,
               name='firefly', alphas=alphas, additonal_info=f"beta: {bet
a}, gamma: {gamma}")
```

```
initial best value:195.0242575968058
final best value:195.0242575968058
Time Elapsed for Particle Swarm: 0:00:00.124144
```



Alpha affects randomness of movement, beta affects brightness and makes fireflies move towards each other more, gamma makes them move towards each other less.

PSO and firefly both use random movement combined with knowing the position of other members of the population. Firefly seems like it runs faster and performs better in most cases.

In [ ]: