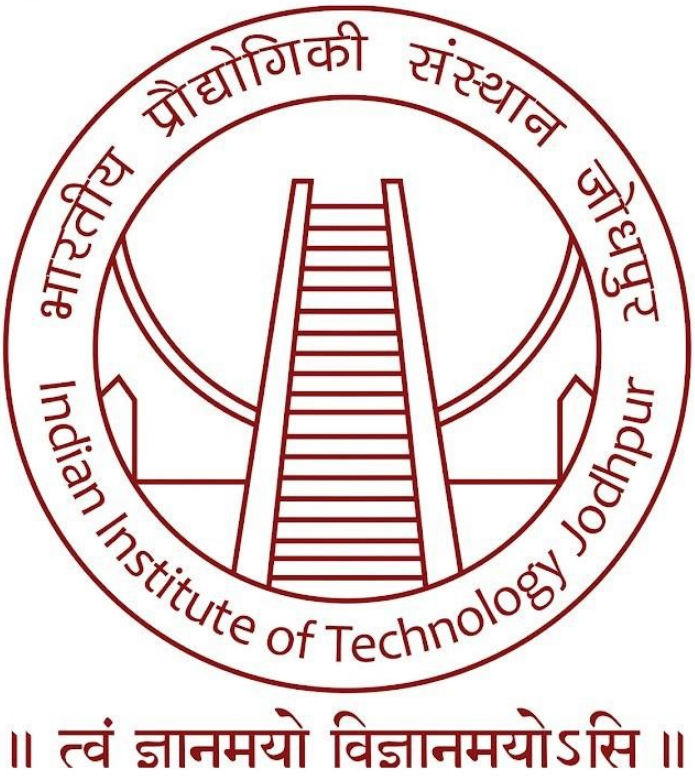
Genetic and PSO based ANC algorithms Report

Manish
B21CS044
IIT Jodhpur
manish.2@iitj.ac.in

Himanshu Gupta
B21CS034
IIT Jodhpur
gupta.103@iitj.ac.in



Abstract

This report explores the application of Genetic Algorithm (GA) in Active Noise Control (ANC), a technology aimed at reducing unwanted noise in various environments. One of the key challenges in ANC is to design an optimal filter that can cancel out noise while preserving the desired signal. GA, a powerful optimization technique inspired by natural selection, has been successfully applied in many fields including ANC. The principles of GA, its advantages, and limitations are discussed, along with a comparison of its performance with other optimization techniques. A case study is presented where GA is used to optimize a digital filter for reducing incoming noise, demonstrating how GA can improve noise cancellation performance. Additionally, Particle Swarm Optimization (PSO) based ANC is also discussed in this report, highlighting its potential benefits and limitations. The results show that both GA and PSO are promising optimization techniques for ANC, capable of achieving excellent noise cancellation performance.

1. Acknowledgement

We would like to express our special thanks to Prof. Amrita Puri for providing us with guidance with this project and the opportunity to work in such an exciting industry. We would especially also like to thank Prof. Amrita Puri's student Arvind for taking the time out of their busy schedules to answer the plethora of questions sent their way and providing us keen insight into completing this project.

2. Introduction

Active Noise Control (ANC) is a technology used to reduce unwanted noise in various environments such as aircraft cabins, car interiors, and industrial plants. The ANC systems produce a sound wave that cancels out the unwanted noise, resulting in a quieter environment. Designing an optimal filter is one of the main challenges in ANC, which cancels out the noise while preserving the desired signal. Digital filters can model the frequency response of the noise and generate a corresponding anti-noise signal. Optimization techniques such as Gradient Descent, Least Mean Square (LMS), Genetic Algorithm (GA), and Particle Swarm Optimization (PSO) can be used to design an optimal ANC filter.

2.1. Genetic Algorithm (GA) Introduction

Genetic Algorithm (GA) is a powerful optimization technique inspired by natural selection. It has been successfully applied in many fields, including engineering, finance, and computer science. GA works by modeling the evolution of a population of potential solutions to a given problem and selecting the fittest solutions for reproduction and mutation. GA has several advantages over other optimization techniques, including its ability to handle nonlinear and non-convex problems, its ability to escape from local optima, and its ability to handle constraints and multiple objectives.

2.2. Particle Swarm Optimization (PSO) Introduction

Particle Swarm Optimization (PSO) is a population-based stochastic optimization technique inspired by the collective behavior of bird flocks or fish schools. PSO uses a swarm of particles to explore the search space and find the optimal solution. Each particle moves in the search space based on its current position and the best position found by itself and the swarm. PSO has been successfully applied in many optimization problems, including ANC. PSO has several advantages over other optimization techniques, including its simplicity, its ability to handle constraints, and its fast convergence rate.

3. Genetic Algorithm Based ANC

Step 1: Setting the parameters

The first step is to set the parameters for the genetic algorithm. The population size is set to "p".

Step 2: Initialization

The next step is to initialize the weights. The weights are represented as Wj(n)=[wj(0,n),wj(1,n),...,wj(L-1,n)], where j=1,. . . p.

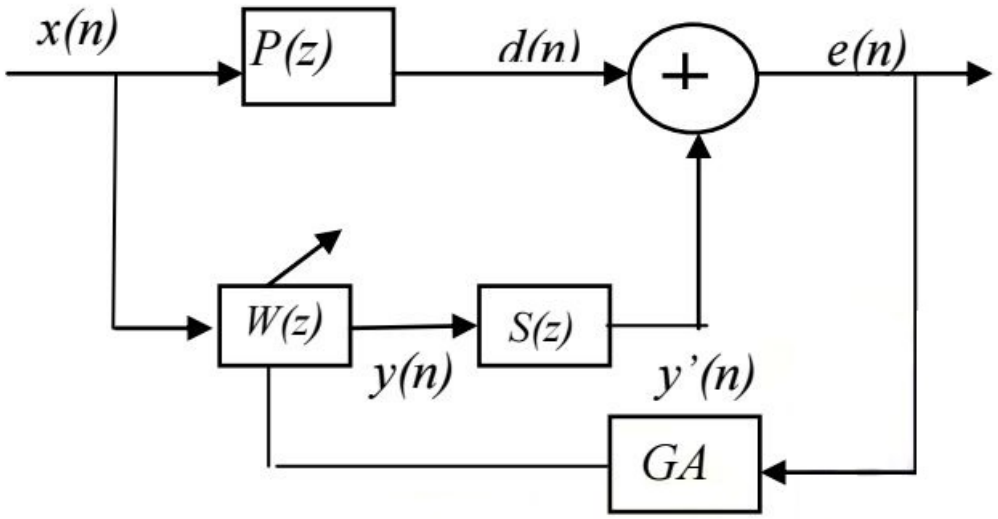


Figure 1. GA Based ANC

Step 3: Fitness function evaluation

For each member of the population, a fitness function is calculated. The fitness function is used to evaluate how well a particular set of weights cancels out the noise signal. The best fitness score is set to 1/e[j], where e[j] is the minimum error.

Step 4: Selection

The next step is to select the parents for the next generation. This can be done using different types of selection methods, such as tournament selection or roulette wheel selection. In this case, we will use tournament selection.

Step 5: Crossover

After selecting the parents, the next step is to perform crossover operations. This involves combining the genetic information from two parents to create a new offspring.

Step 6: Mutation

Mutation is an important step in genetic algorithms. It introduces diversity in the population by randomly changing some of the genes in an individual. After the crossover operation, mutation is performed on the selected offspring.

Step 7: Repeat

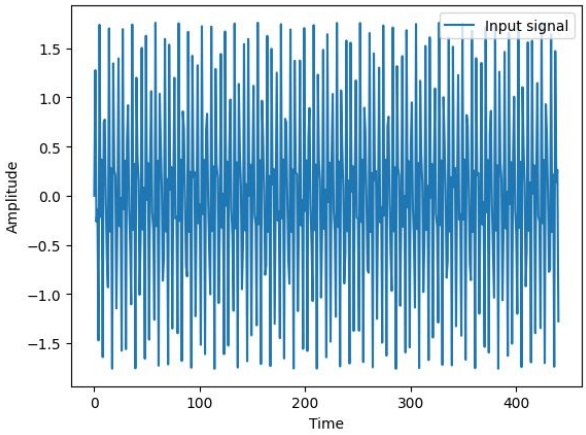
The process is repeated until the desired level of noise reduction is achieved or a certain number of generations have been evaluated.

3.1. Code for Genetic algorithm

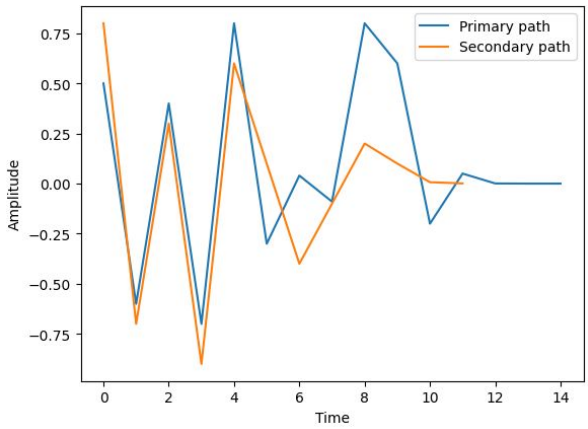
```
import numpy as np
import matplotlib.pyplot as plt
import random

#Input signal for the genetic algorithm
fs = 44100  # Sample rate (Hz)
duration = 0.1  # Duration of the noise (s)
t = np.arange(0, duration, 1/fs)  # Time vector
f1 = 1000  # Frequency of the first tone (Hz)
f2 = 2000  # Frequency of the second tone (Hz)
A1 = 1  # Amplitude of the first tone
A2 = 1  # Amplitude of the second tone
x = A1*np.sin(2*np.pi*f1*t) + A2*np.sin(2*np.pi*f2*t)
print(len(x))

Primary_path = [0.5, -0.6, 0.4, -0.7, 0.8, -0.3, 0.04, -0.09, 0.8, 0.6, -0.2, 0.05, 0.0007, 0, 0]  # primary path
Secondary_path = [0.8, -0.7, 0.3, -0.9, 0.6, 0.1, -0.4, -0.1, 0.2, 0.1, 0.007, 0.001]  # secondary path
N1 = len(Primary_path)
N2 = len(Secondary_path)
primary_path_signal = np.zeros(N1)
Secondary_path_signal = np.zeros(N2)
Reference_signal = np.zeros(len(x))
error = []
arr = np.zeros(len(x))

# Fitness function for the genetic algorithm according to primary path and secondary path
def fitness(weights, x,i):
    # Add zeros to x to match the length of Reference_signal
    arr[i] +=1
    e = 0
    if(arr[i] == 1):
        primary_path_signal[1:] = primary_path_signal[:-1]
        primary_path_signal[0] = x[i]
        d1 = np.sum(primary_path_signal * Primary_path)

        Reference_signal[1:] = Reference_signal[:-1]
        Reference_signal[0] = x[i]
        yout = np.sum(Reference_signal* weights)
        Secondary_path_signal[1:] = Secondary_path_signal[:-1]
        Secondary_path_signal[0] = yout
        d2 = np.sum(Secondary_path_signal * Secondary_path)
        e = d1+d2
    else:
        d1 = np.sum(primary_path_signal * Primary_path)
        yout = np.sum(Reference_signal* weights)
        Secondary_path_signal [0] = yout

d2 = np.sum(Secondary_path_signal * Secondary_path)
        e = d1+d2
    return abs(1/e)

# Genetic algorithm
def genetic_algorithm(x, pop_size, num_generations):
    # Initialize the population with random weights
    population = [np.random.randn(len(x)) for i in range(pop_size)]
    # Evaluate the fitness of the initial population
    fitness_scores = []
    for i in range(pop_size):
        fitness_scores.append(fitness(population[i], x,0))
    # Keep track of the best fitness score and best individual in each generation
    best_fitness_scores = []
    best_individuals = []

    # Iterate over the specified number of generations
```

```
    for i in range(num_generations):
        # Select the parents for the next generation using tournament selection
        parents = []
        for j in range(pop_size):
            # Choose two random individuals from the population
            a, b = random.sample(range(pop_size), 2)
            # Select the parent with lower fitness score
            if fitness_scores[a] < fitness_scores[b]:
                parents.append(population[a])
            else:
                parents.append(population[b])

        # Create the offspring for the next generation using crossover and mutation
        offspring = []
        for j in range(pop_size):
            # Choose two random parents from the selected parents
            a, b = random.sample(range(pop_size), 2)

            # perform crossover by randomly selecting coefficients from the parents
            child = np.zeros(len(x))
            for k in range(len(x)):
                if random.random() < 0.5:
                    child[k] = parents[a][k]
                else:
                    child[k] = parents[b][k]

            # Perform mutation by randomly changing some coefficients
            for k in range(len(x)):
                if random.random() < 0.1:
                    child[k] += np.random.randn()

            offspring.append(child)

        # Evaluate the fitness of the offspring
        offspring_fitness_scores = [fitness(p, x,i) for p in offspring]
        # Replace the population with the offspring
        population = offspring
        fitness_scores = offspring_fitness_scores

        # Keep track of the best fitness score and best individual in the population
        best_fitness_scores.append(max(fitness_scores))
        error.append(fitness_scores)
        # print(error)
        best_individuals.append(population[fitness_scores.index(max(fitness_scores))])

    return best_individuals, best_fitness_scores
# Run the genetic algorithm
best_individuals, best_fitness_scores = genetic_algorithm(x, 100, len(x))

# Plot the best fitness score in each generation
plt.plot(best_fitness_scores)
plt.xlabel('Generation')
plt.ylabel('Fitness score')
plt.show()
```

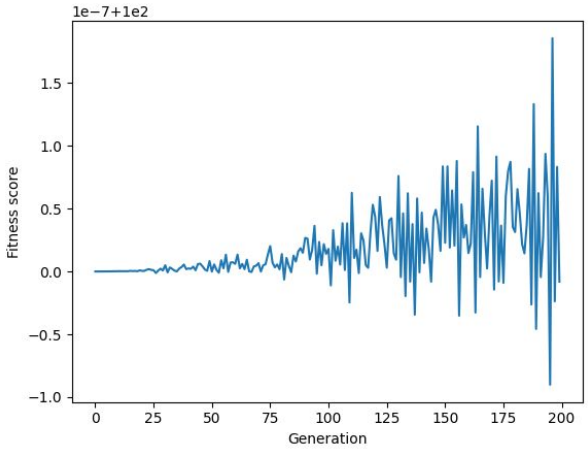
3.2. Simulation Results



(a) Input Signal



(b) Primary and Secondary Path



(c) Fitness Score

3.3. Explaination of Code

- Defined an input signal x as a sum of two sinusoids with different frequencies and amplitudes, and split the signal into two paths: Primary-path and Secondary-path.
- Defined a fitness function that takes a set of weights as input, processes the input signal through the system, and returns the inverse of the error between the processed output signal and the original input signal.
- Implement a genetic algorithm to optimize the set of weights that minimize the error between the processed output signal and the original input signal.
- Run the genetic algorithm for a specified number of generations, with a specified population size.
- Randomly generate an initial population with weights that are the same length as the input signal.
- Evaluate the fitness of each individual in the population using the fitness function.
- Select parents for the next generation through tournament selection who are having more fitness scores.
- Create offspring for the next generation through crossover and mutation.
- Evaluate the fitness of the offspring using the fitness function.

- Replace the population with the offspring, and repeat the process for the specified number of generations.
- Store the best fitness score and best individual in each generation, and return the best individual from the final generation as the optimal set of weights.
- Plot the best fitness score in each generation to show the progress of the genetic algorithm over time.

4. PSO based ANC

Step 1: Setting the parameters

The first step is to set the parameters for the PSO algorithm. These parameters include the population size, maximum number of iterations, acceleration coefficients, and inertia weight.

Step 2: Initialization

Next, initialize the swarm by randomly generating the positions and velocities of particles in the search space. The positions represent the filter coefficients, while the velocities represent the change in position at each iteration.

Step 3: Fitness function evaluation

For each particle, calculate its fitness function value based on its position. The fitness function represents the performance of the filter in cancelling out the noise signal.

Step 4: Updating personal best position

For each particle, update its personal best position based on its fitness value. This personal best position represents the best position the particle has achieved so far.

Step 5: Updating global best position

Update the global best position based on the personal best positions of all particles in the swarm. This global best position represents the best position achieved by any particle in the swarm.

Step 6: Updating particle velocity and position

Update the velocity and position of each particle using the acceleration coefficients and inertia weight. The new velocity is calculated based on the particle's current velocity, personal best position, and global best position. The new position is then calculated based on the new velocity.

Step 7: Repeat

Repeat steps 3 to 6 until the desired level of noise reduction is achieved or the maximum number of iterations is reached. At the end of the algorithm, the filter coefficients corresponding to the global best position represent the optimal filter for cancelling out the noise signal.

4.1. Code for PSO based ANC

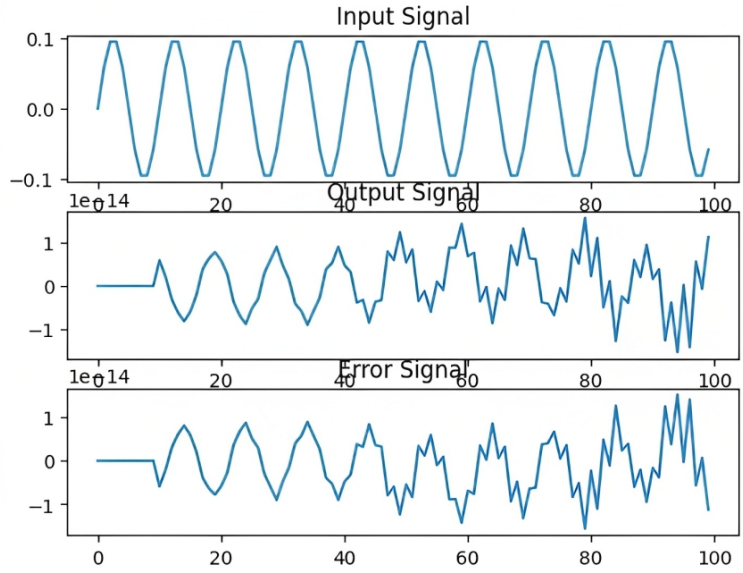
```
import numpy as np
import matplotlib.pyplot as plt

class PSO:
    def __init__(self, num_particles, num_features, w, c1, c2, max_iterations):
        self.num_particles = num_particles
        self.num_features = num_features
        self.w = w
        self.c1 = c1
        self.c2 = c2
        self.max_iterations = max_iterations
        self.particles = np.zeros((num_particles, num_features))
        self.particle_best_pos = np.zeros((num_particles, num_features))
        self.particle_best_error = np.ones(num_particles) * float('inf')
        self.global_best_pos = np.zeros(num_features)
        self.global_best_error = float('inf')
        self.velocity = np.zeros((num_particles, num_features))

    def optimize(self, X, d):
        for i in range(self.num_particles):
            self.particles[i, :] = np.random.rand(self.num_features)
            error = self.calculate_error(X, d, self.particles[i, :])
            self.particle_best_pos[i, :] = self.particles[i, :]
            self.particle_best_error[i] = error
            if error < self.global_best_error:
                self.global_best_pos = self.particles[i, :]
                self.global_best_error = error
        for i in range(self.max_iterations):
            for j in range(self.num_particles):
                self.velocity[j, :] = self.w * self.velocity[j, :] + \
                                      self.c1 * np.random.rand() * (self.particle_best_pos[j, :] - self
    .particles[j, :]) + \
                                      self.c2 * np.random.rand() * (self.global_best_pos - self.
    particles[j, :])
                self.particles[j, :] += self.velocity[j, :]
                error = self.calculate_error(X, d, self.particles[j, :])
                if error < self.particle_best_error[j]:
                    self.particle_best_pos[j, :] = self.particles[j, :]
                    self.particle_best_error[j] = error
                    if error < self.global_best_error:
                        self.global_best_pos = self.particles[j, :]
                        self.global_best_error = error
        return self.global_best_pos

    def calculate_error(self, X, d, w):
        y_hat = self.filter(X, w)
        error = np.sum((d - y_hat) ** 2)
        return error

    def filter(self, X, w):
        y_hat = np.zeros_like(X)
        for i in range(len(w), len(X)):
            y_hat[i] = np.dot(X[i-len(w):i], w)
        y_hat = np.concatenate((np.zeros(len(w)), y_hat[len(w):]))
        return y_hat


# runnning the class PSO with the following parameters
num_particles = 100
num_features = 10
w = 0.5
c1 = 0.5
c2 = 0.5
max_iterations = 100
pso = PSO(num_particles, num_features, w, c1, c2, max_iterations)
```

```
64
65 #initialize the noisy signal and we want output as 0
66 fs = 1000  # Sample rate (Hz)
67 duration = 0.1  # Duration of the noise (s)
68 t = np.arange(0, duration, 1/fs)  # Time vector
69 f1 = 100  # Frequency of the first tone (Hz)
70 f2 = 500  # Frequency of the second tone (Hz)
71 A1 = 0.1  # Amplitude of the first tone
72 A2 = 0.1  # Amplitude of the second tone
73 x = A1*np.sin(2*np.pi*f1*t) + A2*np.sin(2*np.pi*f2*t)
74 d = np.zeros_like(x)
75
76 #optimize the weights
77 w = pso.optimize(x, d)
78
79 #plot the input signal ,output signal and error on subplots
80 plt.figure()
81 plt.subplot(3, 1, 1)
82 plt.plot(x)
83 plt.title('Input Signal')
84 plt.subplot(3, 1, 2)
85 plt.plot(pso.filter(x, w))
86 plt.title('Output Signal')
87 plt.subplot(3, 1, 3)
88 plt.plot(d - pso.filter(x, w))
89 plt.title('Error Signal')
90 plt.show()
```

4.2. Working of PSO ANC code

- Define a class called PSO which performs Particle Swarm Optimization
- Takes in several parameters such as the number of particles, number of features, inertia weight, and cognitive and social learning parameters
- Initializes the particles and their positions randomly Calculates the error of each particle and updates its best position and best error accordingly
- Updates the global best position and global best error
- In each iteration, Updates the velocity of each particle based on its previous velocity, its best position, and the global best position
  Updates the position of each particle based on its velocity
- Calculates the error of each particle and if it's better than its previous best, updates its best position and best error
- If the error is better than the global best error, updates the global best position and global best error
- The optimize() method returns the global best position
- Generate a noisy signal
  Initialize the target output signal as all zeros
- Use the PSO class to optimize the weights of a filter that converts the noisy signal to the target output signal
- Plot the input signal, output signal, and error signal in separate subplots

4.3. Simulation Results for PSO



5. Conclusion

In conclusion, both Particle Swarm Optimization (PSO) and Genetic Algorithm (GA) based Active Noise Control (ANC) are effective optimization techniques for designing adaptive filters that can attenuate noise from a noisy signal.

PSO is a population-based optimization technique that searches for the optimal solution by simulating the behavior of a swarm of particles in a multidimensional search space. PSO has the advantage of being easy to implement and computationally efficient, making it a popular choice for solving optimization problems in various domains.

On the other hand, GA is a genetic-inspired optimization technique that mimics the process of natural selection and evolution. GA works by evolving a population of candidate solutions through a process of selection, crossover, and mutation. GA has the advantage of being able to handle complex and nonlinear optimization problems, but it can be computationally expensive.

Both PSO and GA-based ANC techniques have been shown to provide significant noise reduction in various applications. The choice between these techniques depends on the specific application requirements, the size of the search space, and the available computational resources.

In summary, PSO and GA-based ANC techniques are powerful optimization tools that can be used to design adaptive filters for noise reduction. These techniques can significantly improve the quality of audio signals in various applications, including speech recognition, audio recording, and communication systems.

6. References

1 S.M. kuo, D.R. Morgan, Active Noise Control Systems and DSP implementations, Newyork: Willey, 1996.

2 S. Haykin, Adaptive Filter Theory, Englewood Clifton, NJ: Prientice-Hall, 1986

3 Sven Johansoon, Lars Hankansson and Ingvar Claesson, Aircraft cabin noise and vibration prediction and active control, Handbook of noise and vibration control, John Willey, pp. 1207-1215, 2007.

4 David E. Goldberg, Genetic Algorithms in search, Optimization and Machine Learning, Addison Wesley Longmann Publishing Co., Boston, 1989