EE 553 Final Project

Filter Coefficients via Particle Swarm Optimization

Prepared for Dr. Aldo Morales

By: Congzhou Sha and Ryan Thompson

Abstract

In this project, we examined particle swarm optimization (PSO), which is a method of solving nonlinear problems such as finding minima of non-convex cost functions. The basic idea behind particle swarm optimization is that by searching a variety of randomly chosen starting configurations, minima can be found such that the search does not stagnate. All code was implemented in MATLAB.

Introduction

The "swarm" heuristic takes inspiration from experience with biological design in which a large number of simple interacting particles (e.g. schools of fish, colonies of bacteria, etc.) can generate complex emergent behavior [1]. For highly nonlinear functions, this emergent behavior can be tuned so that extrema are found with accuracy and speed; choosing parameters for the particles is important to the performance of the PSO. In fact, incorrectly tuned PSO may not find these extrema. The basic algorithm has few restrictions on its input, but as a result also can guarantee few conditions on the output, particularly in convergence.

Theory

The following general algorithm is taken from [2]. We require only a cost function f, which is a mapping from n-dimensional space to the real line. The cost function contains n variables, and generally maps to a single location on the real line.

$$f: \mathbb{R}^n \to \mathbb{R} \tag{1}$$

The goal is to find a position $\boldsymbol{a}$ such that $\forall \boldsymbol{b} \in U \subseteq \mathbb{R}^n, f(\boldsymbol{a}) < f(\boldsymbol{b})$. If this condition is satisfied, then we will have found the global (or local) minimum, depending on how we restrict our configuration space. Next, we take n particles, with the i^{th} particle having position $\boldsymbol{x}_i$ and velocity $\boldsymbol{v}_i$ with $\boldsymbol{x}_i, \boldsymbol{v}_i \in \mathbb{R}^n$. For the entire swarm, let $\boldsymbol{g}$ be the position of the particle with least cost

which has been found thus far, and $\boldsymbol{p_i}$ be the best known position for that particle. Using pseudocode, the basic structure of the PSO algorithm is shown below.

for each particle i = 1, 2, …, n
    initialize the particle with random position, $\boldsymbol{x}_i$ (usually chosen uniformly in a range)
    initialize the particle's best known position to that initial position, $\boldsymbol{p}_i \leftarrow \boldsymbol{x}_i$
    if $f(\boldsymbol{p}_i) < f(\boldsymbol{g})$
        $\boldsymbol{g} \leftarrow \boldsymbol{p}_i$
    initialize the velocity with magnitude less than the bounds of the initial position, $\boldsymbol{v}_i$
while we have not reached a tolerance for termination
    for each particle i = 1, 2, …, n
        for each dimension d = 1, 2, …, k
            pick two random numbers uniformly from [0, 1]: a, b
            update the particles velocity, weighting the displacement vector from the particle to its best known position relative to the vector from the particle to the global best known position, $\boldsymbol{v}_{i,d} \leftarrow \omega \boldsymbol{v}_{i,d} + \phi_p a(\boldsymbol{p}_{i,d} - \boldsymbol{x}_{i,d}) + \phi_g b(\boldsymbol{g}_{i,d} - \boldsymbol{x}_{i,d})$
        update the position: $\boldsymbol{x}_i \leftarrow \boldsymbol{x}_i + \boldsymbol{v}_i$
        if $f(\boldsymbol{x}_i) < f(\boldsymbol{p}_i)$
            $\boldsymbol{p}_i \leftarrow \boldsymbol{x}_i$
            if $f(\boldsymbol{p}_i) < f(\boldsymbol{g})$
                $\boldsymbol{g} \leftarrow \boldsymbol{p}_i$

The intuition is thus for each particle, we want to head toward the local minimum. There is information about this in two directions, we can either head toward the current global minimum, or to the particle's minimum from its past. We constantly update the particle's own minimum and the global minimum as well as soon as improvements are found, which allows sharing of information between all particles in the swarm. The parameters ω, ϕ_p, ϕ_g are chosen so that this weighting of the particle's history and the global history guides the search properly. As seen, we assume velocities are roughly normalized to the size of the search area. These can also be learned parameters in the backpropagation gradient of the procedure, as there are few trainable parameters. That improvement is known as adaptive PSO. In this project, we tuned these weights by hand.

Implementation

The implementation of the PSO algorithm involved many variables. Because some tuning was anticipated, the code was written with a parameterized approach. A 2 Hz sin wave with an amplitude of 1, as shown in Fig. A1, was chosen as an input signal and discretized to 100 points over a 1 second period. This yielded a sampling period of roughly 10 ms and ensured that the sampling frequency was well over the minimum Nyquist rate. The high resolution was expected to lead to a more reliable filter. A training set of random noise was created by sampling a random variable which is distributed uniformly on the interval [-0.1 0.1]. Three additional test sets were created using a similar approach for later use. The training signal resulting from adding the noise to the sine wave is shown in Fig. A2.

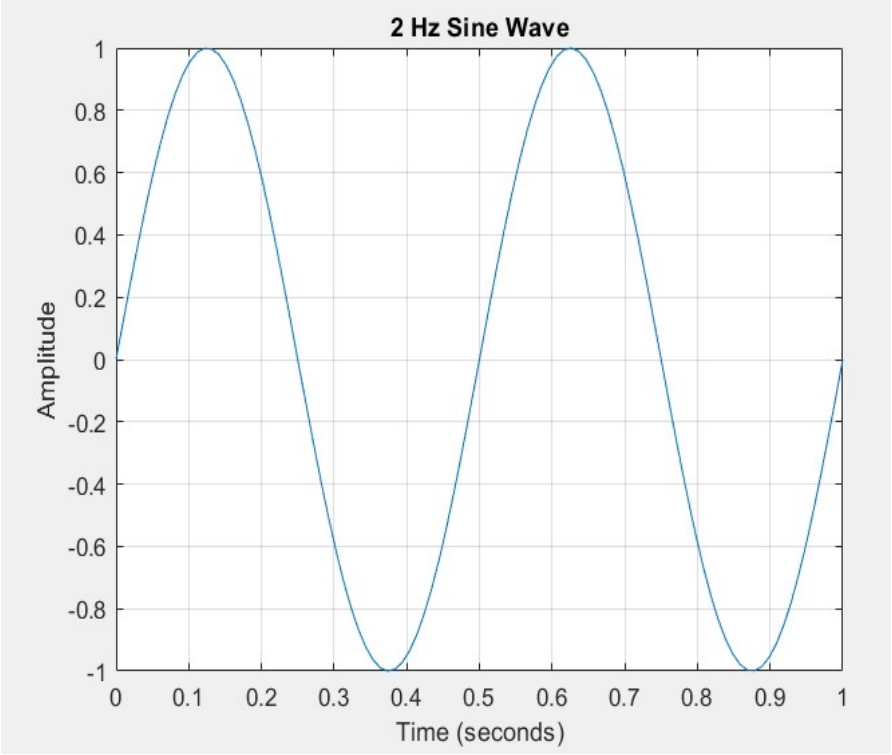

Fig. A1: Desired signal, 2 Hz sinusoid with amplitude of 1

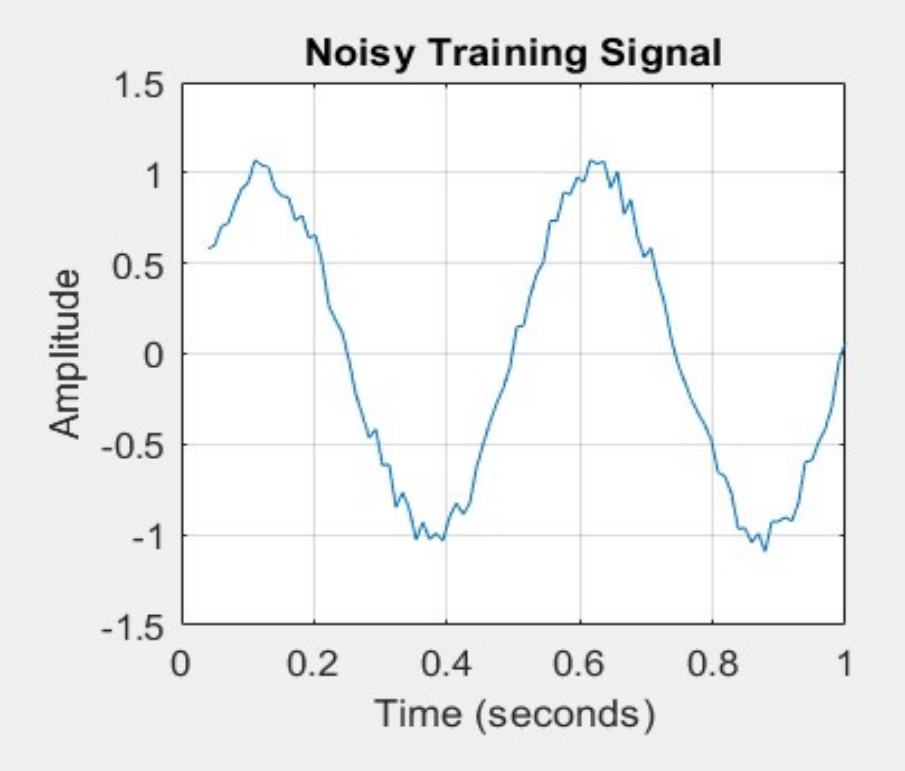

Fig. A2: Noisy training signal

In order to find effective parameters for this problem, a set of baseline parameters were chosen, and adjustments were made to individual parameters to see the effects. The baseline parameters are shown in Table 1. The initial baseline numbers were chosen as the result of some preliminary tinkering.

Table 1: Baseline parameters

Number of Particles	Filter Order	Inertial Scalar	Personal Scalar	Global Scalar	Number of Iterations
100	9	0.008	0.008	0.010	500

Table 2 shows results of parameter adjustments compared to the baseline results. Quality of the resulting filter and consistency of the algorithm are demonstrated by the mean and sample variance of the ASE from five trials. As a general rule, a smaller mean shows a higher quality filter with lower overall error. A smaller variance indicates that the five solutions converge more closely on an optimal solution. Each set of trials was run with only the single specified parameter changed. All other parameters should be assumed to be those listed in Table 1.

Table 2: Results of parameter adjustments

Parameter Adjusted	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Mean	Sample Variance
Baseline	0.235	0.2977	0.1318	0.2336	0.0713	0.19388	0.008231727
Particles ↑ to 300	0.0125	0.0111	0.0908	0.0465	0.064	0.04498	0.001166587
Particles ↑ to 1000	0.0202	0.0187	0.0262	0.0034	0.041	0.0219	0.00018467
Filter Order ↑ to 19	0.2854	0.6022	0.8384	0.589	0.4181	0.54662	0.043693512
Filter Order ↓ to 4	0.0071	0.0165	0.0067	0.004	0.0325	0.01336	0.000136838
Iterations ↑ to 1500	0.1578	0.1282	0.1749	0.0635	0.2303	0.15094	0.003770503

A new baseline set of parameters was created based on these results. These parameters are shown in Table 3. An increase in the number of particles was shown to decrease variance, thereby increasing the algorithm's consistency. With additional particles, the number of iterations was increased slightly. The original 500 iterations did not prove to be enough to settle all particles in a single solution, though 750 was found to be sufficient. Interestingly, decreasing the filter order drastically improved both quality and reliability. This is likely due to the large filter length relative to the sample signal length.

Table 3: New baseline PSO parameters

Number of Particles	Filter Order	Inertial Scalar	Personal Scalar	Global Scalar	Number of Iterations
1000	4	0.008	0.008	0.010	750

Another round of testing was done with this new set of parameters. The results are shown in Table 4. The intention was to further adjust the scaling coefficients after this round of trials, but the parameter set was shown to be incredibly consistent with the parameters in Table 3. If computing time were a major concern, more adjustments would have been helpful, but they were not found necessary for the purposes of this paper.

Table 4: Error results from parameters in Table 3

Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Mean	Sample Variance
0.0019	0.0019	0.0019	0.0019	0.0019	0.0019	0

The five filter solutions from these trials were saved and are presented in Table 5. Although the tap coefficients are not precisely the same, they proved to be quite similar. This result signals that these coefficient sets are clustered very tightly around the optimal solution in R5 (five-dimensional space).

Table 5: Five sets of coefficients presumed to be nearly optimal

	h_0	h_1	h_2	h_3	h_4
Trial 1	0.5625	0.3446	0.2543	0.0016	-0.1813
Trial 2	0.5109	0.5	0.1262	-0.0084	-0.1464
Trial 3	0.5805	0.3775	0.1568	0.0275	-0.1578
Trial 4	0.5776	0.3796	0.1615	0.022	-0.1567
Trial 5	0.5738	0.3798	0.1644	0.0231	-0.1573

Results

The effectiveness of the solutions was tested on three other noise sets. The ASE for each test is shown in Table 6. The results are similar in magnitude to the training set results, although slightly higher. This is to be expected, as the filters were specifically optimized to the training set. For comparison, the ASE from the original baseline parameters in Table 1 was around 100 times this amount (0.19 mean ASE). These test set results signal a filter which is quite effective at removing low-amplitude random noise from a pure sine wave.

Table 6: ASE of each filter solution on noise test sets

	Training Set	Test Set 1	Test Set 2	Test Set 3
Filter 1	0.0019	0.002576	0.002177	0.002114
Filter 2	0.0019	0.0025814	0.002208	0.002201
Filter 3	0.0019	0.0025358	0.002148	0.002107
Filter 4	0.0019	0.0025343	0.002149	0.002108
Filter 5	0.0019	0.0025506	0.002163	0.002116

Fig. B1-B4 give visual confirmation of these results. The plots were all obtaining using the “Filter 1” coefficients. Fig. B1 shows the effectiveness when applied to the training set. Fig. B2-B4 show the results when applied to the test sets.

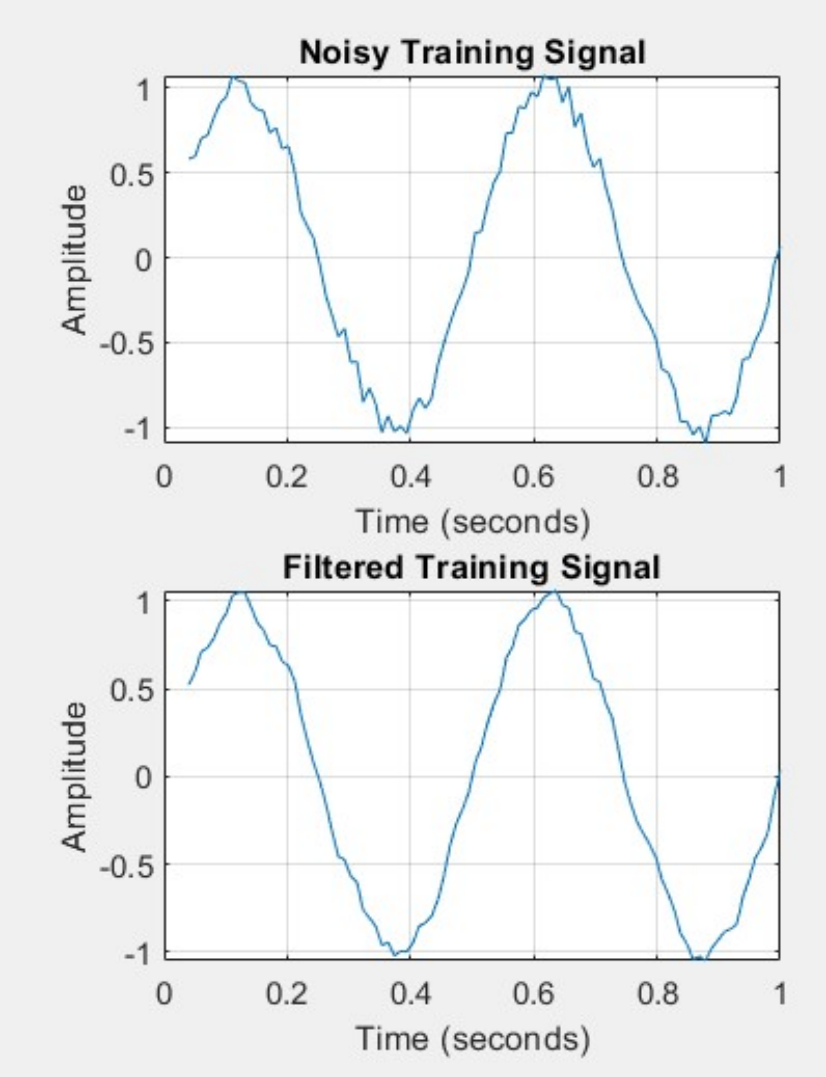

Fig. B1: Training signal, unfiltered and filtered

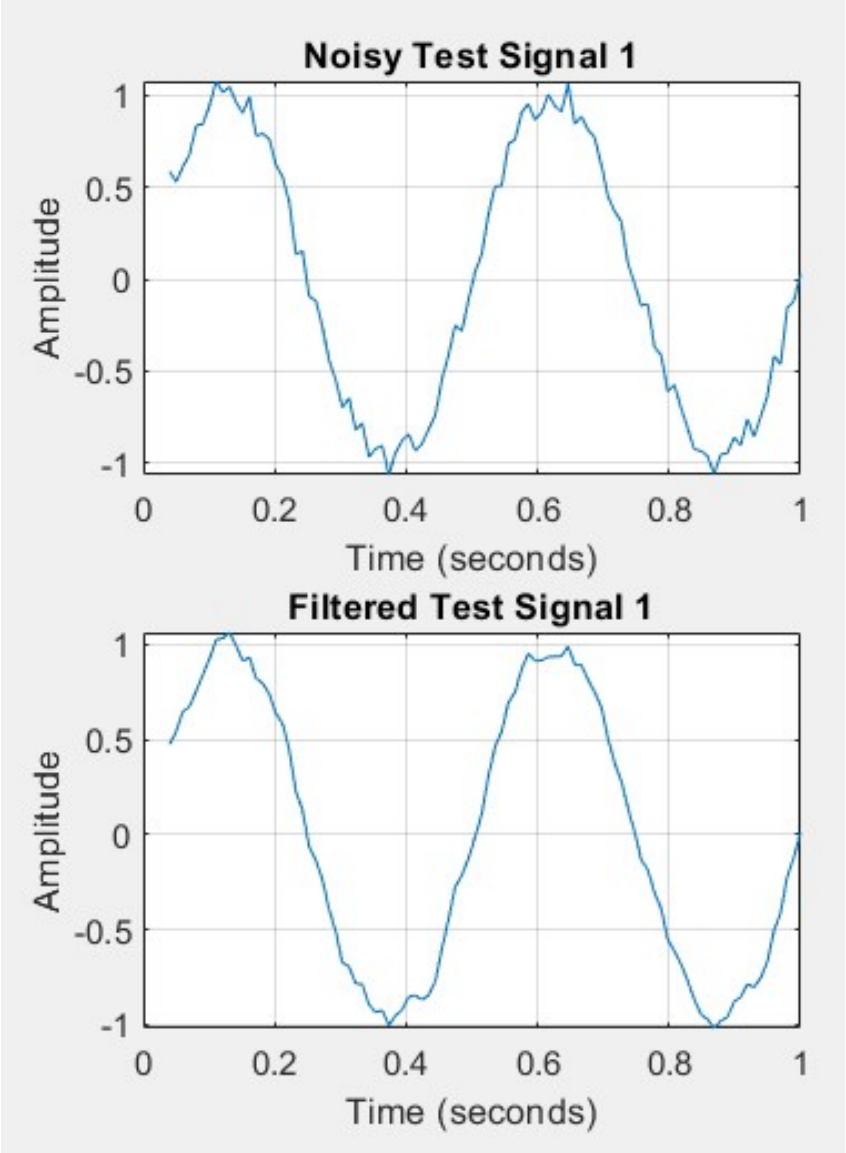

Fig. B2: First test signal, unfiltered and filtered

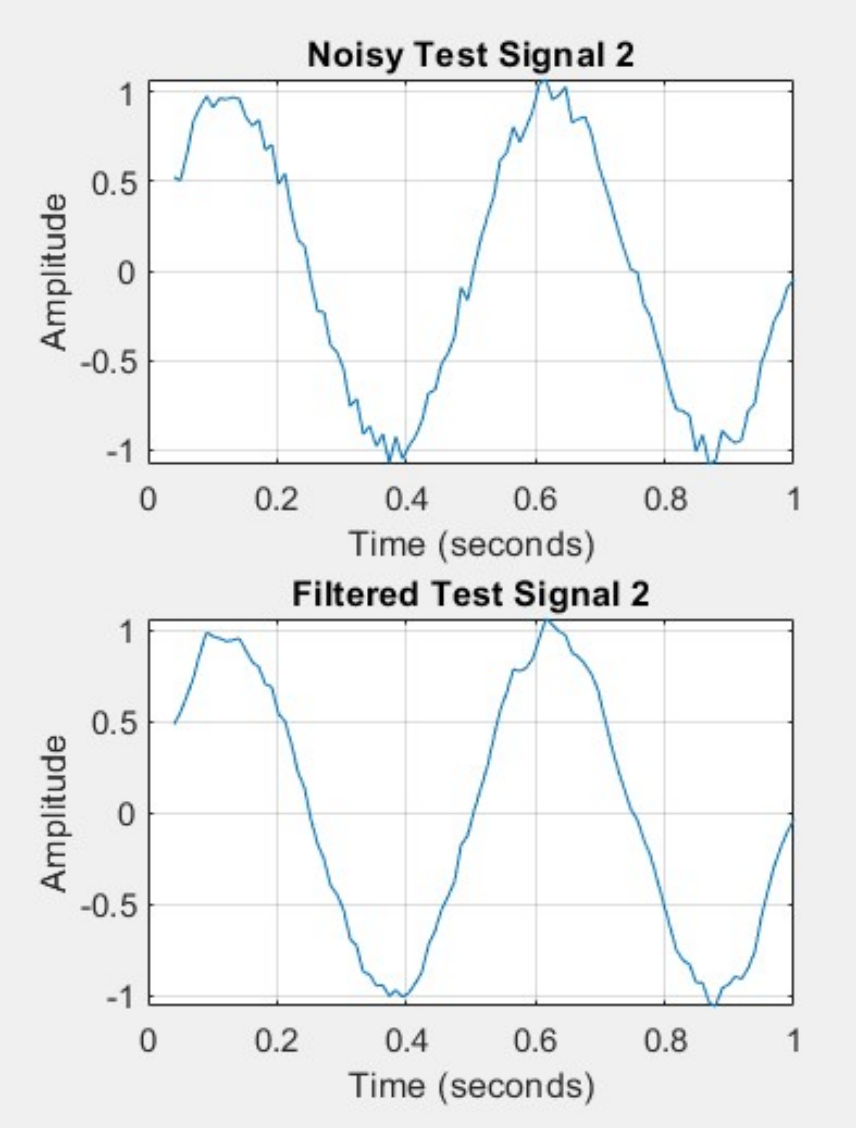

Fig. B3: Second test signal, unfiltered and filtered

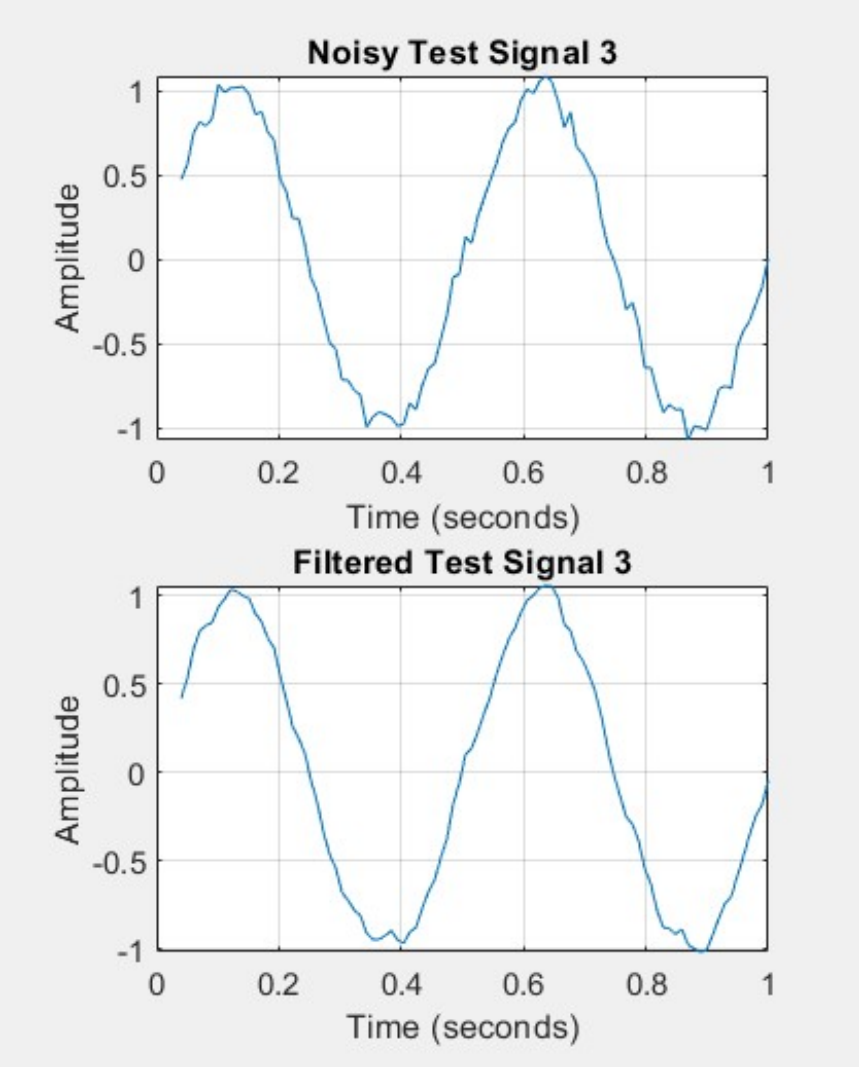

Fig. B4: Third test signal, unfiltered and filtered

Conclusions

The PSO algorithm is a reliable approach to finding optimal solutions to problems with many variables, such as digital filters. The main downside of this algorithm is that its effectiveness is predicated on the user selecting appropriate parameters. Very large amounts of particles with very small movements and a sufficient number of iterations will be very effective, but can be computationally expensive and very slow for complex problems. One potential solution to this would be to remove redundant particles which have converged to within a specified distance of another particle and have slowed to below a reasonable speed. There is no need to continue calculating new positions for many redundant particles with similar positions, velocities, and personal bests.

For the specific application of low-order digital filters to simple low-noise signals, the PSO algorithm proved to be quite effective. The ASE was in the range of 25/10,000 of the amplitude of the desired signal when the filter was applied to test signals with similar noise levels. Besides the removal of redundant particles, future alterations could potentially include longer and more complex signals, as well as non-uniformly distributed noise, such as Gaussian noise.

Code Section 1: Parameterized PSO Program Designed to Optimize Digital Filters

```
clear all;

%~~~~~~~~~~~~~~~~~~~~~~~~~~
% Parameters for adjustment
%~~~~~~~~~~~~~~~~~~~~~~~~~~

np = 1000;      % Number of particles
order = 4;     % Filter order
UB = 10;      % Filter coefficient upper bound
LB = -10;      % Filter coefficient lower bound
iterations = 750;    % Number of iterations
inertialCoefficient = 0.008;    % Inertial scale factor
pBestInfluence = 0.008;    % Personal best scale factor
gBestInfluence = 0.01;    % Global best scale factor

% Time vector, default set for 100 points on the interval [0 1]
t = linspace(0,1,100);

% Create discrete ideal input signal vector
% For simplicity, this is a simple 2 Hz sine wave
% with an amplitude of 1
cleanSignal = sin(2*pi*2*t);
%plot(t,cleanSignal)

% Create noisy version of input signal vector
% This can be customized. Currently loading a vector
% of noise which is uniformly distributed on [-1 1]
% then scaling so noise amplitude interval is 10%
% of signal amplitude
noiseVector = csvread('noiseVector.csv');
noisySignal = cleanSignal + 0.1 * noiseVector;
%plot(t,noisySignal)


%~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
% After this line, no code should need to be altered. All
% remaining code is parameterized based on the above variables
%~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~


% Initialize random starting points for particles
% Each column will contain the coefficients for each particle
CurrentParticlePositions = rand(order + 1, np) * (UB - LB) +…
 (LB * ones(order + 1, np));
NewParticlePositions = CurrentParticlePositions;
PreviousParticlePositions = CurrentParticlePositions;

% Run filter on initial values, then initialize personal best and global
% best vectors
currentASE = zeros(1,np);
newASE = currentASE;
for i = 1:np
```

```
    currentASE(i) = ComputeASE(cleanSignal, noisySignal,
NewParticlePositions(:,i));
end
pBest = currentASE;
pBestPositions = NewParticlePositions;
[gBest, minIndex] = min(pBest);
gBestPositions = NewParticlePositions(:,minIndex);



% Main algorithm loop
for i = 1:(iterations - 1)

    % Find contribution from personal best
    pBestContribution = pBestPositions - CurrentParticlePositions;

    % Find contribution from global best
    for j = 1:np
        gBestContribution(:,j) = gBestPositions -
CurrentParticlePositions(:,j);
    end

    % Find contribution from intertia
    inertialContribution = CurrentParticlePositions -
PreviousParticlePositions;

    % Find next particle positions, but don't move particles yet
    NewParticlePositions = CurrentParticlePositions + (pBestContribution *
pBestInfluence) ...
                + (gBestContribution * gBestInfluence) + ...
                (inertialContribution * inertialCoefficient);

    % Remember current positions for future inertia calculation
    PreviousParticlePositions = CurrentParticlePositions;

    % Move particles to new positions
    CurrentParticlePositions = NewParticlePositions;

    % Find ASE for each current particle position and check it against global
    % and personal bests, then update those if appropriate
    for j = 1:np
        % Calculate ASE for particle j
        ASEj = ComputeASE(cleanSignal, noisySignal,
CurrentParticlePositions(:,j));
        % Update personal best if appropriate
        if (ASEj < pBest(j))
            pBest(j) = ASEj;
            pBestPositions(:,j) = CurrentParticlePositions(:,j);
        end
        % Update global best if appropriate
        if (ASEj < gBest)
            gBest = ASEj;
            gBestPositions = CurrentParticlePositions(:,j);
        end
    end
end
```

```
% display best solution found by algorithm
gBest
transpose(gBestPositions)
%pBestPositions


% This function passes the noisy signal through the desired filter,
% compares the output to the original clean signal, and computes the ASE
function ASE = ComputeASE(cleanSignal, noisySignal, coefficients)

    iterations = length(noisySignal) - length(coefficients) + 1;
    outputSignal = zeros(1, iterations);

    % Calculate and build output signal
    for i=1:iterations
        outputSignal(i) = flip(noisySignal(i:(i + length(coefficients) - 1)))
...
                          * coefficients;
    end

    % Compute error
    errorVector = cleanSignal(length(coefficients):end) - outputSignal;
    ASE = sum(errorVector .^ 2) / length(errorVector);

end
```

Code Section 2: Program for Testing Filter on Additional Signals and Producing Plots

```
clear all;

t = linspace(0,1,100);    % time vector

% Read and populate noise and tap coefficient vectors
coeffs = csvread('NewFilters.csv');
Noise2 = csvread('noiseVector2.csv');
Noise3 = csvread('noiseVector3.csv');
Noise4 = csvread('noiseVector4.csv');
OriginalNoise = csvread('noiseVector.csv');

% Create discrete ideal input signal vector
cleanSignal = sin(2*pi*2*t);
%plot(t,cleanSignal)

% Create noisy version of input signal vector
noisySignal = cleanSignal + 0.1 * OriginalNoise;
noisySignal2 = cleanSignal + 0.1 * Noise2;
noisySignal3 = cleanSignal + 0.1 * Noise3;
noisySignal4 = cleanSignal + 0.1 * Noise4;
%plot(t,noisySignal)

% Create and fill matrix with ASE for each filter/signal pair
ASE_matrix = zeros(5,3)
for k = 1:5
    ASE_matrix(k,1) = ComputeASE(cleanSignal, noisySignal2, coeffs(k,:));
    ASE_matrix(k,2) = ComputeASE(cleanSignal, noisySignal3, coeffs(k,:));
    ASE_matrix(k,3) = ComputeASE(cleanSignal, noisySignal4, coeffs(k,:));
end

% display the matrix
ASE_matrix


% Plot original and filtered signals for each noise set
% using set 1 of filter coefficients
[A, OS] = ComputeASE(cleanSignal, noisySignal, coeffs(1,:))
subplot(2,4,1)
plot(t(5:end), noisySignal(5:end))
title('Noisy Training Signal')
xlabel('Time (seconds)')
ylabel('Amplitude')
grid on
subplot(2,4,5)
plot(t(5:end), OS)
title('Filtered Training Signal')
xlabel('Time (seconds)')
ylabel('Amplitude')
grid on

[A, OS] = ComputeASE(cleanSignal, noisySignal2, coeffs(1,:))
subplot(2,4,2)
```

```
plot(t(5:end), noisySignal2(5:end))
title('Noisy Test Signal 1')
xlabel('Time (seconds)')
ylabel('Amplitude')
grid on
subplot(2,4,6)
plot(t(5:end), OS)
title('Filtered Test Signal 1')
xlabel('Time (seconds)')
ylabel('Amplitude')
grid on

[A, OS] = ComputeASE(cleanSignal, noisySignal3, coeffs(1,:))
subplot(2,4,3)
plot(t(5:end), noisySignal3(5:end))
title('Noisy Test Signal 2')
xlabel('Time (seconds)')
ylabel('Amplitude')
grid on
subplot(2,4,7)
plot(t(5:end), OS)
title('Filtered Test Signal 2')
xlabel('Time (seconds)')
ylabel('Amplitude')
grid on

[A, OS] = ComputeASE(cleanSignal, noisySignal4, coeffs(1,:))
subplot(2,4,4)
plot(t(5:end), noisySignal4(5:end))
title('Noisy Test Signal 3')
xlabel('Time (seconds)')
ylabel('Amplitude')
grid on
subplot(2,4,8)
plot(t(5:end), OS)
title('Filtered Test Signal 3')
xlabel('Time (seconds)')
ylabel('Amplitude')
grid on

% This function passes the noisy signal through the desired filter,
% compares the output to the original clean signal, and computes the ASE
% This function is the same as the function from the original
% program except for the required transposition of the tap
% coefficient matrix
function [ASE, outputSignal] = ComputeASE(cleanSignal, noisySignal,
coefficients)

    iterations = length(noisySignal) - length(coefficients) + 1;
    outputSignal = zeros(1, iterations);

    for i=1:iterations
        outputSignal(i) = flip(noisySignal(i:(i + length(coefficients) - 1)))
...
                            * transpose(coefficients);
    end
```

```
    errorVector = cleanSignal(length(coefficients):end) - outputSignal;
    ASE = sum(errorVector .^ 2) / length(errorVector);

end
```

Works Cited

[1] J. Kennedy and R. Eberhart, Particle Swarm Optimization, Perth, WA, Australia: Proceedings of ICNN'95 - International Conference on Neural Networks, 1995.

[2] M. Clerc, Standard Particle Swarm Optimization, HAL Open Access Archive, 2012.