

# ANNEX

## MATLAB Script for Damage Detection in Bridge Structures

Leonardo Zunino

June 25, 2024

```
clc;
clear;
close all;

%% INPUT: Acceleration signals
num_sensors = 14; % Number of sensors used
num_matrices = 32; % Number of matrices (cases) for
    each configuration
acc_signals = cell(num_matrices, num_sensors);

% Load the signals (example file names; replace with
    actual file names)
damage_acc_mat_files = {
    'd_08_8_11_2.mat', 'd_08_8_12_2.mat', 'd_08_8_11_8
        .mat', 'd_08_8_12_8.mat', 'd_08_8_11_12.mat', '
        d_08_8_12_12.mat', 'd_08_8_11_19.mat', '
        d_08_8_12_19.mat', ...
    'd_08_8_8_2.mat', 'd_08_8_9_2.mat', 'd_08_8_8_10.
        mat', 'd_08_8_9_10.mat', 'd_08_8_8_12.mat', '
        d_08_8_9_12.mat', 'd_08_8_8_19.mat', '
        d_08_8_9_19.mat',
};

undamage_acc_mat_files = {
    'd_08_1_2_2.mat', 'd_08_1_2_10.mat', '
        d_08_1_2_12.mat', 'd_08_1_2_19.mat', '
        d_08_1_3_2.mat', 'd_08_1_3_10.mat', '
        d_08_1_3_12.mat', 'd_08_1_3_19.mat', ...
    'd_08_1_4_2.mat', 'd_08_1_4_10.mat', '
        d_08_1_4_12.mat', 'd_08_1_4_19.mat', '
        d_08_1_8_2.mat', 'd_08_1_8_10.mat', '
        d_08_1_8_12.mat', 'd_08_1_8_19.mat',
};
```

```

acc_mat_files = [undamage_acc_mat_files,
                 damage_acc_mat_files];

% Load acceleration data (replace column indices with
    actual sensor indices)
sensor_columns = 2:15; % Use all available sensors

for i = 1:num_matrices
    acc_mat = load(acc_mat_files{i});
    for j = 1:num_sensors
        acc_signals{i, j} = acc_mat.Data(3001:9000,
            sensor_columns(j));
    end
end

% Sampling frequency
fs = 100; %[Hz]
t_tot = 60; %[s] % First 60 seconds
t = (0:(t_tot*fs-1))/fs;
t = t.';

%% Chebyshev Filtering (II type)
fc = 39; % Filter cutoff frequency [Hz]
rp = 1; % Maximum allowable ripple in passband [dB]
rs = 100; % Minimum required attenuation in stopband [
    dB]
Wp = fc / (fs/2); % Passband cutoff frequency
Ws = (fc + 10) / (fs/2); % Stopband cutoff frequency

[n, Wn] = cheb2ord(Wp, Ws, rp, rs);
[b, a] = cheby2(n, rs, Wn);

filtered_signals = cell(num_matrices, num_sensors);
for i = 1:num_matrices
    for j = 1:num_sensors
        filtered_signals{i, j} = filter(b, a,
            acc_signals{i, j});
    end
end

% Visualize the filter frequency response
figure;
freqz(b, a, 512, fs);

for j = 1:num_sensors
    % Plot undamaged cases in groups of 8
    for k = 0:1
        figure('units','normalized','outerposition',[0
            0 1 1]); % Create full-screen figure
        for i = 1:8

```

```

case_index = k * 8 + i;
signal = acc_signals{case_index, j};

% Time domain signal
subplot('Position', [0.1, 0.92 - (i *
    0.11), 0.5, 0.08]);
plot(t, signal);
if i == 1
    title(['Sensor ', num2str(j), ' - Time
        Domain']);
end
ylabel('Acc.(m/s^2)');
text(-0.1, 0.5, ['Case ', num2str(
    case_index)], 'Units', 'normalized', '
    HorizontalAlignment', 'right');
if i == 8
    xlabel('Time (s)');
end

% Fourier transform
N = length(signal);
Y = fft(signal);
f = (0:N-1)*(fs/N); % Frequency vector
P = abs(Y)/N; % Two-sided spectrum

% Limit the frequency range to 0-50 Hz
f_limit = f(1:floor(N/2));
P_limit = P(1:floor(N/2));
idx_limit = f_limit <= 50;

subplot('Position', [0.65, 0.92 - (i *
    0.11), 0.3, 0.08]);
plot(f_limit(idx_limit), P_limit(idx_limit
));
if i == 1
    title(['Sensor ', num2str(j), ' -
        Fourier Transform']);
end
ylabel('Magnitude');
xlim([0 50]); % Limit x-axis to 0-50 Hz
if i == 8
    xlabel('Frequency (Hz)');
end
end
sgtitle(['Undamaged Cases ', num2str(k*8+1), '
    to ', num2str((k+1)*8), ' - Sensor ',
    num2str(j)], 'FontSize', 14, 'FontWeight',
    'bold');

% Save the figure as PNG

```

```

        saveas(gcf, ['Undamaged_Cases_Sensor_',
                     num2str(j), '_Group_', num2str(k+1), '.png'
                     ]);
        close(gcf); % Close the figure after saving
    end

    % Plot damaged cases in groups of 8
    for k = 0:1
        figure('units','normalized','outerposition',[0
            0 1 1]); % Create full-screen figure
        for i = 1:8
            case_index = k * 8 + i;
            signal = acc_signals{16 + case_index, j};

            % Time domain signal
            subplot('Position', [0.1, 0.92 - (i *
                0.11), 0.5, 0.08]);
            plot(t, signal);
            if i == 1
                title(['Sensor ', num2str(j), ' - Time
                    Domain']);
            end
            ylabel('Acc.(m/s^2)');
            text(-0.1, 0.5, ['Case ', num2str(
                case_index)], 'Units', 'normalized', '
                HorizontalAlignment', 'right');
            if i == 8
                xlabel('Time (s)');
            end

            % Fourier transform
            N = length(signal);
            Y = fft(signal);
            f = (0:N-1)*(fs/N); % Frequency vector
            P = abs(Y)/N; % Two-sided spectrum

            % Limit the frequency range to 0-50 Hz
            f_limit = f(1:floor(N/2));
            P_limit = P(1:floor(N/2));
            idx_limit = f_limit <= 50;

            subplot('Position', [0.65, 0.92 - (i *
                0.11), 0.3, 0.08]);
            plot(f_limit(idx_limit), P_limit(idx_limit
                ));
            if i == 1
                title(['Sensor ', num2str(j), ' -
                    Fourier Transform']);
            end
            ylabel('Magnitude');

```

```

        xlim([0 50]); % Limit x-axis to 0-50 Hz
        if i == 8
            xlabel('Frequency (Hz)');
        end
    end
    sgtitle(['Damaged Cases ', num2str(k*8+1), '
        to ', num2str((k+1)*8), ' - Sensor ',
        num2str(j)], 'FontSize', 14, 'FontWeight',
        'bold');

    % Save the figure as PNG
    saveas(gcf, ['Damaged_Cases_Sensor_', num2str(
        j), '_Group_', num2str(k+1), '.png']);
    close(gcf); % Close the figure after saving
end
end

%% VMD and HT on each filtered signal
% Parameters for VMD and HT
MaxIterations = 10000;
LMUpdateRate = 0.1;
RelativeTolerance = 1e-5;
AbsoluteTolerance = 0.1;

optPenaltyFactors = zeros(num_matrices, num_sensors);
optNumIMFs = zeros(num_matrices, num_sensors);

colors = lines(NumIMFs); % To have distinct colors for
    each IMF

combined_features = cell(num_sensors, 1);

parfor j = 1:num_sensors
    combined_data = [];
    for i = 1:num_matrices
        signal = filtered_signals{i, j};
        [optPenaltyFactors(i, j), optNumIMFs(i, j)] =
            optimizeVMDParameters(signal, fs);

        [IMFs, residual] = vmd(signal, 'MaxIterations'
            , MaxIterations, 'NumIMFs', optNumIMFs(i, j)
            ), ...
            'PenaltyFactor', optPenaltyFactors(i, j),
            'LMUpdateRate', LMUpdateRate, ...
            'RelativeTolerance', RelativeTolerance, '
            AbsoluteTolerance', AbsoluteTolerance);

        filtered_IMFs = zeros(size(IMFs));
        for k = 1:NumIMFs
            filtered_IMFs(:, k) = filter(b, a, IMFs(:,

```

```

        k));
end

% Hilbert-Huang (HHT) to obtain the
% instantaneous frequencies
[HS, ~, ~, inst_freq, inst_energy] = hht(
    filtered_IMFs, fs);

% Apply a median filter to smooth
% instantaneous frequencies
inst_freq_smooth = medfilt1(inst_freq, 11); %
    Apply median filter with window size 11

% Collect combined data for PCA
combined_data = cat(3, combined_data,
    inst_freq_smooth);

% Plotting for all signals
% Plot IMFs and their Fourier transforms
fig1 = figure('units','normalized','
    outerposition',[0 0 1 1]); % Create full-
    screen figure
for k = 1:NumIMFs
    % Plot IMF
    subplot('Position', [0.1, 1 - (k / 8) -
        0.02, 0.35, 0.12]);
    plot(IMFs(:, k));
    if k == 1
        title(['IMFs for Signal ', num2str(i),
            ' - Sensor ', num2str(j)]);
    end
    ylabel(['IMF ', num2str(k)]);
    if k == NumIMFs
        xlabel('Time (s)');
    end

    % Fourier transform of IMF
    N = length(IMFs(:, k));
    Y = fft(IMFs(:, k));
    f = (0:N-1)*(fs/N); % Frequency vector
    P = abs(Y)/N; % Two-sided spectrum

    % Limit the frequency range to 0-50 Hz
    f_limit = f(1:floor(N/2));
    P_limit = P(1:floor(N/2));
    idx_limit = f_limit <= 50;

    subplot('Position', [0.55, 1 - (k / 8) -
        0.02, 0.35, 0.12]);
    plot(f_limit(idx_limit), P_limit(idx_limit)

```

```

    ));
    if k == 1
        title('Fourier Transform');
    end
    if k == NumIMFs
        xlabel('Frequency (Hz)');
    end
end
sgtitle(['IMFs and Fourier Transforms for
Signal ', num2str(i), ' - Sensor ', num2str(
j)]);
% Save the figure as PNG
saveas(fig1, ['IMFs_Fourier_Signal_', num2str(
i), '_Sensor_', num2str(j), '.png']);
close(fig1); % Close the figure after saving

% Plot instantaneous frequencies before and
after median filtering
fig2 = figure('units','normalized','
outerposition',[0 0 1 1]); % Create full-
screen figure
% Plot original instantaneous frequencies
subplot(2, 1, 1);
hold on;
for k = 1:NumIMFs
    plot(inst_freq(:, k), 'Color', colors(k,
:));
end
hold off;
title(['Original Instantaneous Frequencies for
Signal ', num2str(i), ' - Sensor ',
num2str(j)]);
ylabel('Frequency (Hz)');
legend(arrayfun(@(x) ['IMF ', num2str(x)], 1:
NumIMFs, 'UniformOutput', false));

% Plot smoothed instantaneous frequencies
subplot(2, 1, 2);
hold on;
for k = 1:NumIMFs
    plot(inst_freq_smooth(:, k), 'Color',
colors(k, :));
end
hold off;
title(['Smoothed Instantaneous Frequencies for
Signal ', num2str(i), ' - Sensor ',
num2str(j)]);
xlabel('Time (s)');
ylabel('Frequency (Hz)');
legend(arrayfun(@(x) ['IMF ', num2str(x)], 1:

```

```

        NumIMFs, 'UniformOutput', false));
    % Save the figure as PNG
    saveas(fig2, ['InstFreq_Signal_', num2str(i),
        '_Sensor_', num2str(j), '.png']);
    close(fig2); % Close the figure after saving
end
combined_features{j} = combined_data;
end

%% Combine all features from all sensors for PCA
combined_all_features = [];
for j = 1:num_sensors
    combined_all_features = cat(3,
        combined_all_features, combined_features{j});
end

% Reshape the 3D matrix into a 2D matrix
% First, permute the matrix to make dimensions 6 x
    6000 x (num_matrices * num_sensors)
combined_all_features = permute(combined_all_features,
    [2, 1, 3]);

% Now combine the second and third dimensions into a
    single dimension
% This will result in a matrix of dimensions 6 x
    (6000*num_matrices*num_sensors)
data_resaped = reshape(combined_all_features, 6, [])
';

% Apply PCA on all cases
Z_normalized_all = zscore(data_resaped);
[coeff_all, score_all, latent_all, ~, explained_all] =
    pca(Z_normalized_all);

% Plot explained variance for all cases
figure;
pareto(explained_all(1:6), 1); % up to 100%
title('Explained Variance of Principal Components for
    All Cases (All Sensors)');
xlabel('Principal Components');
ylabel('Explained Variance (%)');

% Select components to retain (discarding only the
    first component)
components_to_discard = [1, 2];
components_to_keep = setdiff(1:size(coeff_all, 2),
    components_to_discard);

% Scores of the retained components
filtered_score_all = score_all(:, components_to_keep);

```



```

% Coefficients (loadings) of the retained components
filtered_coeff_all = coeff_all(:, components_to_keep);

% Reconstruct the data using the selected components
Z_reconstructed_all = filtered_score_all *
    filtered_coeff_all';

% Rescale the reconstructed data to the original scale
mean_data = mean(data_reshaped);
std_data = std(data_reshaped);
Z_backscaled_all = Z_reconstructed_all .* std_data +
    mean_data;

% Transform back into a 3D matrix with original
    dimensions (6 * 6000 * (num_matrices * num_sensors)
    )
num_samples = size(combined_all_features, 2);
num_cases = num_matrices * num_sensors;
Z_backscaled_all = reshape(Z_backscaled_all', [6,
    num_samples, num_cases]);

% Now Z_backscaled_all has the original dimensions
    with rescaled data

%% Separate the rescaled data back to individual
    sensors
reconstructed_features = cell(num_sensors, 1);
for j = 1:num_sensors
    reconstructed_features{j} = Z_backscaled_all(:, :,
        (j-1)*num_matrices + (1:num_matrices));
end

% Ensure lengths match before plotting
min_len = min(length(t), size(reconstructed_features
    {1}, 2));
t_inst = t(1:min_len);

% Plot the rescaled instantaneous frequencies and the
    differences with the original frequencies
for j = 1:num_sensors
    Z_backscaled_sensor = reconstructed_features{j}(:,
        1:min_len, :);

    for i = 1:num_matrices
        % Plot rescaled instantaneous frequencies and
            their differences in the same figure
        fig = figure('units','normalized','
            outerposition',[0 0 1 1]); % Create full-
            screen figure
    end
end

```

```

% Plot rescaled instantaneous frequencies
subplot(2, 1, 1);
hold on;
for k = 1:NumIMFs
    plot(t_inst, squeeze(Z_backscaled_sensor(k
        , :, i)), 'DisplayName', ['IMF ',
            num2str(k)]);
end
hold off;
title(['Rescaled Instantaneous Frequencies (
    Sensor ', num2str(j), ', Case ', num2str(i)
        , ')']);
xlabel('Time [s]');
ylabel('Frequency [Hz]');
legend show;
grid on;

% Plot the differences between the original
    and rescaled instantaneous frequencies
original_freqs = squeeze(combined_features{j
    }(:, :, i)); % Correctly index original
    frequencies
original_freqs = original_freqs(1:min_len, :)
    '; % Transpose to match dimensions with
    reconstructed_freqs
reconstructed_freqs = squeeze(
    Z_backscaled_sensor(:, :, i));
diff_freqs = original_freqs -
    reconstructed_freqs;

subplot(2, 1, 2);
hold on;
for k = 1:NumIMFs
    plot(t_inst, diff_freqs(k, :), '
        DisplayName', ['IMF ', num2str(k)]);
end
hold off;
title(['Difference Between Original and
    Rescaled Instantaneous Frequencies (Sensor
        ', num2str(j), ', Case ', num2str(i), ')'])
    ;
xlabel('Time [s]');
ylabel('Frequency Difference [Hz]');
legend show;
grid on;

% Save the figure as PNG
saveas(fig, ['
    Rescaled_And_Diff_InstFreq_Sensor_',

```

```

        num2str(j), '_Case_', num2str(i), '.png']);
    close(fig); % Close the figure after saving
end
end

%% K-means clustering for damage detection using third
    instantaneous frequency
L = 1344; % Symbolic data length
window_size = 5; % Size of moving window
num_matrices = 32; % Number of matrices (cases)
num_sensors = 14; % Number of sensors
damage_labels = [zeros(1, 16), ones(1, 16)]; % 0 for
    undamaged, 1 for damaged (16 undamaged, 16 damaged
    cases)
fs = 100; % Sampling frequency in Hz
num_samples = 6000; % Number of samples for the first
    30 seconds assuming 100 Hz

silhouette_scores = zeros(num_sensors, 1); % Array to
    store silhouette scores

for j = 1:num_sensors
    % Collect and concatenate the first 30 seconds of
        the third instantaneous frequency for each case
    third_inst_freqs = [];
    case_labels = [];

    % Include all undamaged cases
    for i = 1:16
        inst_freq_smooth = Z_backscaled_all(3, 1:
            num_samples, (i-1)*num_sensors + j); % Use
            the third instantaneous frequency
        third_inst_freqs = [third_inst_freqs;
            inst_freq_smooth(:)'];
        case_labels = [case_labels; repmat({'
            Undamaged Case ', num2str(i)}], num_samples
            , 1)];
    end

    % Include all damaged cases
    for i = 17:num_matrices
        inst_freq_smooth = Z_backscaled_all(3, 1:
            num_samples, (i-1)*num_sensors + j); % Use
            the third instantaneous frequency
        third_inst_freqs = [third_inst_freqs;
            inst_freq_smooth(:)'];
        case_labels = [case_labels; repmat({'Damaged
            Case ', num2str(i-16)}], num_samples, 1)];
    end
end

```

```

% Determine the point to separate undamaged and
    damaged data
length_U = 16 * num_samples; % 16 undamaged cases
    times number of samples

% K-means clustering and DI calculation for all
    cases
concatenated_data = third_inst_freqs(:);
num_points = length(concatenated_data);
num_boxes = floor(num_points / L);
box_data = zeros(num_boxes, L);

for k = 1:num_boxes
    start_idx = (k-1) * L + 1;
    end_idx = k * L;
    if end_idx > length(concatenated_data)
        break;
    end
    box_data(k, :) = concatenated_data(start_idx:
        end_idx);
end

% Normalize box data
box_data = zscore(box_data, 0, 2); % Normalize
    each box to have zero mean and unit variance

% Calculate symbolic data
symbolic_data = calculate_symbolic_data(box_data);

% Calculate Detection Index for all cases
DI = calculate_DI(symbolic_data, window_size);

% Calculate the boundary box index correctly
boundary_box = floor(length(DI) / 2);

% Plot DI for all cases and save as PNG
plot_and_save_DI(DI, j, boundary_box); % Plot DI
    for all cases

% Calculate silhouette score for the clustering
[idx, C] = kmeans(symbolic_data, 2, 'Replicates',
    50); % Perform K-means clustering with 2
    clusters
silhouette_scores(j) = mean(silhouette(
    symbolic_data, idx)); % Calculate the mean
    silhouette score for the sensor
end

% Display silhouette scores for all sensors
disp('Silhouette Scores for each sensor:');

```

```

disp(silhouette_scores);

% Helper function to calculate symbolic data
function symbolic_data = calculate_symbolic_data(
    box_data)
    symbolic_data = zeros(size(box_data));
    for i = 1:size(box_data, 1)
        q75 = prctile(box_data(i, :), 75);
        q25 = prctile(box_data(i, :), 25);
        symbolic_data(i, :) = q75 - q25; % Calculate
            interquartile range
    end
end

% Helper function to calculate Detection Index (DI)
function DI = calculate_DI(symbolic_data, window_size)
    num_windows = floor(size(symbolic_data, 1) /
        window_size);
    DC_values = zeros(num_windows, 1);
    for w = 1:num_windows
        start_idx = (w-1) * window_size + 1;
        end_idx = w * window_size;
        window_data = symbolic_data(start_idx:end_idx,
            :);
        if size(window_data, 1) > 2
            sil_scores = zeros(1, min(10, size(
                window_data, 1) - 1));
            for k = 2:min(10, size(window_data, 1) -
                1)
                [idx, ~] = kmeans(window_data, k, '
                    Replicates', 10, 'Display', 'off');
                sil_scores(k) = mean(silhouette(
                    window_data, idx));
            end
            [~, optimal_k] = max(sil_scores);
            [idx, C] = kmeans(window_data, optimal_k,
                'Replicates', 50);
            DC = 0;
            for c1 = 1:optimal_k
                for c2 = 1:optimal_k
                    if c1 ~= c2
                        DC = DC + norm(C(c1, :) - C(c2
                            , :));
                    end
                end
            end
            DC = DC / (optimal_k * (optimal_k - 1));
            DC_values(w) = DC;
        else
            DC_values(w) = 0;
        end
    end
end

```

```

        end
    end
    mean_DC = mean(DC_values);
    std_DC = std(DC_values);
    CB = mean_DC + tinv(0.999, num_windows-1) * std_DC
        / sqrt(num_windows);
    DI = DC_values - CB;

    % Scale down DI values
    DI = DI / 10; % Adjust scaling factor as needed
end

% Helper function to plot and save Detection Index (DI)
function plot_and_save_DI(DI, sensor_num, boundary_box)
    figure('units','normalized','outerposition',[0 0 1
        1]); % Create full-screen figure
    % Ensure DI is a column vector
    DI = DI(:);

    b = bar(1:length(DI), DI, 'FaceColor', 'flat');
    for k = 1:length(DI)
        if DI(k) >= 0
            b.CData(k,:) = [1 0 0]; % Red for positive
                values
        else
            b.CData(k,:) = [0 1 0]; % Green for
                negative values
        end
    end
    hold on;
    xline(boundary_box, 'k--', 'LineWidth', 2); % Add
        a vertical dashed line to separate undamaged
        and damaged
    title(['Detection Index (DI) for Sensor ', num2str
        (sensor_num)]);
    xlabel('Window Number');
    ylabel('DI Value');
    grid on;

    % Calculate y-axis limits based on the DI values
    min_DI = min(DI);
    max_DI = max(DI);
    y_margin = (max_DI - min_DI) * 0.1; % Add 10%
        margin to the y-axis limits
    ylim([min_DI - y_margin, max_DI + y_margin]);

    hold off;

```

```

        % Save the figure as PNG
        saveas(gcf, ['DI_Sensor_', num2str(sensor_num), '.
        png']);
        close(gcf); % Close the figure after saving
    end

%% Function to Optimize VMD Parameters
function [optPenaltyFactor, optNumIMFs] =
    optimizeVMDParameters(signal, fs)
    % Reduced range of PenaltyFactor values to test
    % for debugging
    penaltyValues = linspace(400, 1000, 100); %
    % Example range, adjust as necessary
    % Reduced range of numbers of IMFs for simpler
    % debugging
    numIMFsValues = 4:6;
    minEntropy = inf;

    optPenaltyFactor = NaN;
    optNumIMFs = NaN;

    for penaltyFactor = penaltyValues
        for numIMFs = numIMFsValues
            [IMFs, ~] = vmd(signal, 'PenaltyFactor',
                penaltyFactor, 'NumIMFs', numIMFs, '
                MaxIterations', 500);
            % Calculate the entropy for this
            % configuration
            entropy = spectralEntropy(IMFs, fs);
            % Select the minimum
            if entropy < minEntropy
                minEntropy = entropy;
                optPenaltyFactor = penaltyFactor;
                optNumIMFs = numIMFs;
            end
        end
    end
    if isnan(optPenaltyFactor)
        error('Failed to optimize PenaltyFactor: No
        valid values found.');
```

```

    end
end

function entropy = spectralEntropy(IMFs, fs)
    numModes = size(IMFs, 2);
    entropy = 0;
    for i = 1:numModes
        powerSpectrum = abs(fft(IMFs(:, i)))^2;
        powerSpectrum = powerSpectrum / sum(
            powerSpectrum); % Normalize
    end
end

```

```
        entropy = entropy - sum(powerSpectrum .* log(
            powerSpectrum + eps)); % Add eps for
            numerical stability
    end
end
```