# Investigation Into Pitch-Tracking Systems in Auto-Tune

Edward Silva Jonas Krider Tappen Douglas

***Abstract*—This study evaluates and compares the performance of the Fast Fourier Transform (FFT) and Autocorrelation (ACF) algorithms for real-time pitch tracking in auto-tune applications. System effectiveness is compared and contrasted using specific performance metrics, specifically latency and residual pitch tracking error.**

## I. Introduction

Real-time automated pitch correction systems must carefully balance computational latency with pitch-tracking accuracy. To explore these tradeoffs, this project implements and evaluates a system that extracts the fundamental frequency ($f_0$) of an incoming audio signal and retune it to a desired frequency ($f_\mathrm{d}$). The system specifically compares the efficiency and accuracy of Fast Fourier Transform (FFT) and Autocorrelation (ACF) pitch detection techniques using a custom MATLAB signal chain.

The performance of the system is quantified through algorithmic processing time and end-to-end latency. By analyzing the system from both $f_0$ to the tuned output, and the tuned output back to the tracked fundamental, the efficiency tradeoffs within in digital pitch correction are characterized to understand and analyze.

To accomplish this, individual scope of work was as follows: Edward Silva managed latency and error reporting, data visualization, and results verification; Tappen Douglas developed the core MATLAB pitch-tracking and auto-tune algorithms, alongside the live input/output; and Jonas Krider directed the overall system architecture, filter tuning, and the development of an additional MaxMSP live demonstration.

The remainder of this paper is organized as follows: Section II details the mathematical background of the selected algorithms. Section III describes the system architecture and filtering methodology. Section IV presents the experimental latency and processing metrics, and Section V concludes the findings.

## II. Background

Real-time digital pitch tracking requires extracting the fundamental frequency ($f_0$) of an incoming acoustic source and manipulating it. To process these audio sources, continuous analog sound waves are first converted into discrete electrical signals via an analog-to-digital converter (ADC). To optimize the efficiency and prevent aliasing, these discrete-time signals are typically band-limited using a cascade of high-pass and low-pass filters to isolate the target human vocal range, followed by decimation to reduce the sample rate.

To gather frequency-domain information from the discrete-time signal, the discrete Fourier transform is computed using the Fast Fourier Transform (FFT) algorithm, defined as

$$X[k] = \sum_{n=0}^{N-1} x[n] \cdot e^{-j2\pi \frac{k}{N} n} \tag{1}$$

where the incoming signal $x[n]$, the window size $N$, the sample index $n$, and the frequency bin $k$, are used to determine the frequency response. The primary purpose of this FFT operation is to determine the fundamental frequency ($f_0$) of the incoming signal by identifying the frequency bin with the maximum magnitude [1].

Alternatively, time-domain Autocorrelation (ACF) can be performed to achieve a similar goal. ACF computes the similarity of a signal with a delayed copy of itself, given by

$$R_{\mathrm{xx}}[k] = \sum_{n=0}^{N-1-k} x[n] \cdot x[n+k] \tag{2}$$

where $R_{xx}[k]$ is the ACF function evaluated at lag $k$. This process extracts $f_0$ by identifying the lag where the ACF reaches a periodic maximum, corresponding to the fundamental period [1].

Once a stable $f_0$ is extracted, the signal must be mapped to a desired target frequency ($f_\mathrm{d}$), typically a discrete value within the 12-tone equal temperament scale. The fundamental signal processing challenge of retuning lies in shifting this frequency without altering the signal's time duration or distorting its formants. A naive resampling approach would compress or expand the audio in time, changing both pitch and playback speed simultaneously.

The implementation of these pitch tracking and retuning algorithms in a live environment introduces design tradeoffs between computational complexity, latency, and resolution. Frequency-domain methods like the FFT require large window sizes ($N$, Equation (1)) to achieve high frequency resolution, which directly increases system latency. This is often detrimental to real-time applications where target delays must be kept generally in real-time (typically $\leq 100$ ms). Conversely, time-domain methods like ACF offer high precision for low frequencies with comparatively shorter windows, but they require intensive computation across many lags.

A real-time digital signal processing system must carefully balance these parameters, using techniques such as input filtering, down-sampling, and efficient buffer management to ensure rapid execution without sacrificing accurate pitch detection.

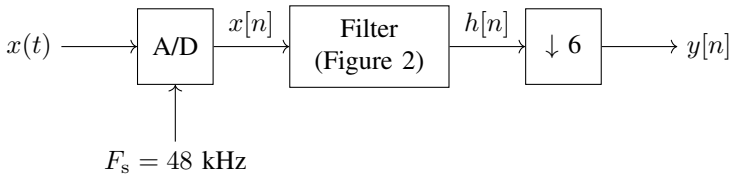


Fig. 1: System block diagram

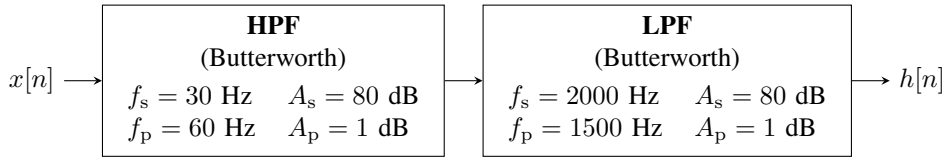


Fig. 2: Filter Architecture: $h[n]$

## III. Architecture & Methodology

The FFT method computes signal frequencies in $O(n \log n)$ time while the ACF method compute signal frequencies in $O(n^2)$ time [2], [3]. However, ACF determines the fundamental frequency while offering a lower space complexity. Since the FFT is faster, it can naively be chosen for this use-case, but a direct comparison of their implementation highlights the specific advantages of one over the other [4], [5].

The beginning of the implementation required defining parameters for FFT, ACF, and retuning algorithms as discussed in Table I.

| **Parameter** | **Value** |
|---|---|
| Sampling Frequency ($F_{\mathrm{s}}$) | 48 kHz |
| Frame Size | 1024 samples |
| FFT Size ($N_{\mathrm{FFT}}$) | 2048 |
| Gain | 10 dB |
| Window Function | Hamming |

Table I: General chosen system parameters

| **Parameter** | **Value** |
|---|---|
| Filter Type | Butterworth |
| stop-band Frequency ($F_{\mathrm{stop}}$) | 30 Hz |
| pass-band Frequency ($F_{\mathrm{pass}}$) | 60 Hz |
| stop-band Attenuation ($A_{\mathrm{stop}}$) | 80 dB |
| pass-band Ripple ($A_{\mathrm{pass}}$) | 1 dB |
| Match | stop-band |

Table II: high-pass filter parameters for band-pass

The system implemented uses a simple structure, visually depicted in Figure 2, with the code in Section V, where the incoming audio signal is converted to a digital signal through an analog-to-digital converter (ADC), sampled at a rate of $48$ kHz. From there the signal is filtered through a cascaded band-pass filter, comprised of low-pass and high-pass filters, which attenuate the signal outside of the bounds of a regular human voice. These parameters are detailed in Table II and Table III, respectively.

| **Parameter** | **Value** |
|---|---|
| Filter Type | Butterworth |
| pass-band Frequency ($F_{\mathrm{pass}}$) | 1500 Hz |
| stop-band Frequency ($F_{\mathrm{stop}}$) | 2000 Hz |
| pass-band Ripple ($A_{\mathrm{pass}}$) | 1 dB |
| stop-band Attenuation ($A_{\mathrm{stop}}$) | 80 dB |
| Match | stop-band |

Table III: low-pass filter parameters for band-pass

Following the cascaded filtering, the system decimates the signal to reduce the sampling rate to $8000$ Hz. This decimation occurs in two sequential stages utilizing factors of 3 and 2 to prevent aliasing and reduce the computational load for the tracking algorithms. A $2048$ sample circular buffer maintains the recent pitch history. The current offset is removed from this buffered signal prior to analysis.

For the FFT method, the signal is multiplied by a Hamming window and evaluated using a high resolution $8192$ bin FFT. The bin with the maximum amplitude is identified in the magnitude spectrum. To mitigate any quantization error within the discrete frequency bins, a parabolic interpolation is applied across the peak bin and its two adjacent neighbors to calculate a precise fundamental frequency estimate, optimizing resolution without increasing the FFT size.

The ACF method uses the same buffered history by computing an unbiased cross correlation, as described in Section II. To optimize execution speed and prevent false tracking on higher harmonics, the search bounds are restricted to lags corresponding to a frequency range of $55$ Hz to $2000$ Hz. The algorithm identifies the first local maximum that exceeds a $0.2$ amplitude threshold. Parabolic interpolation is then utilized on the correlation peak to find the fractional sample lag, which is inverted to determine the fundamental frequency.

Once the fundamental frequency is extracted, the retuning stage maps the pitch to a target frequency. The system references a stored map of $12$ tone equal temperament notes and

selects the nearest frequency midpoint. An exponential moving average smoothing filter with an alpha coefficient of 0.2 prevents erratic switching between adjacent target notes. The ratio between the target frequency and the tracked fundamental frequency determines the scaling factor. The current audio frame is then resampled by this rational factor, windowed using a Hann function, and amplitude normalized before being passed to the audio output device for playback.

## IV. Results & Analysis

To evaluate the system frequency response boundaries, and isolate the capabilities between the FFT/ACF algorithms, a continuous frequency sweep was processed through the cascaded filter. The primary test signal moves upward, and both the FFT and ACF algorithms accurately track the fundamental frequency, within the pass-band. However, once the signal crosses 1500 Hz threshold into the low-pass band, the ACF method fails. The FFT on the other hand continues tracking into the transition band by isolating the frequency bin, but also collapses once the signal reaches the 2000 Hz stop-band.

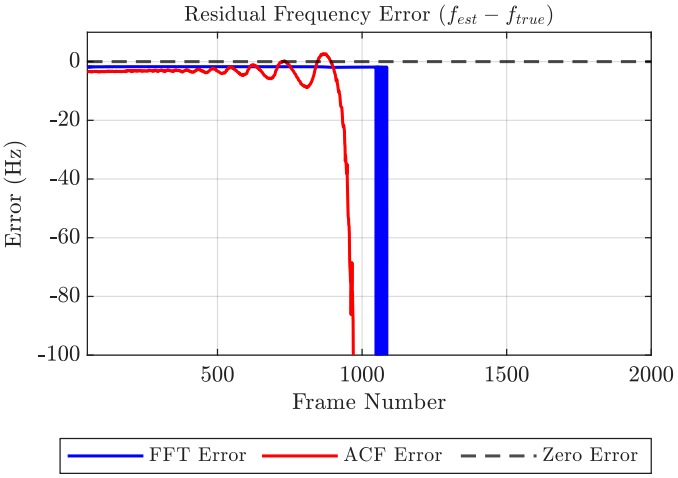


Fig. 3: Residual tracking error ($f_{est} - f_{true}$) mapped across the simulation frame span

This behavior, observed in Figure 3, demonstrates the out-of-band rejection, such that the primary signal is fully attenuated, and both algorithms abandon the noise from the signal and move onto a secondary valid signal, which enters the system shortly after, as described in Section III. Notably, the first 50 Hz of the frequency spectrum are omitted due to them not being viably observed in this system. This lapse is due to the calculation method used which cannot go lower than 55 Hz.

During the MATLAB execution, the computational latency for fundamental frequency ($f_0$) estimation was measured for both Fast Fourier Transform (FFT) and ACF (ACF) methods. As illustrated in Figure 4 and detailed in Table IV, the FFT method demonstrates higher efficiency. Specifically, the mean processing time ($\overline{x}$) for the FFT (0.22 ms) is approximately 2.68 times faster than that of ACF (0.59 ms). Additionally, the FFT displays stability improvements, with a variance ($\sigma^2$) of

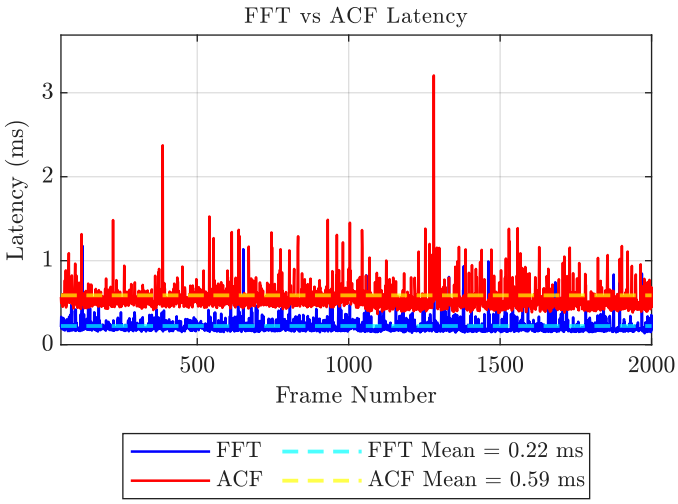


Fig. 4: Computational latency time for FFT vs ACF

0.26 compared to the 0.74 observed in the ACF method. These metrics indicate that the FFT is computationally more efficient and has a higher consistency across varying frequency bins.

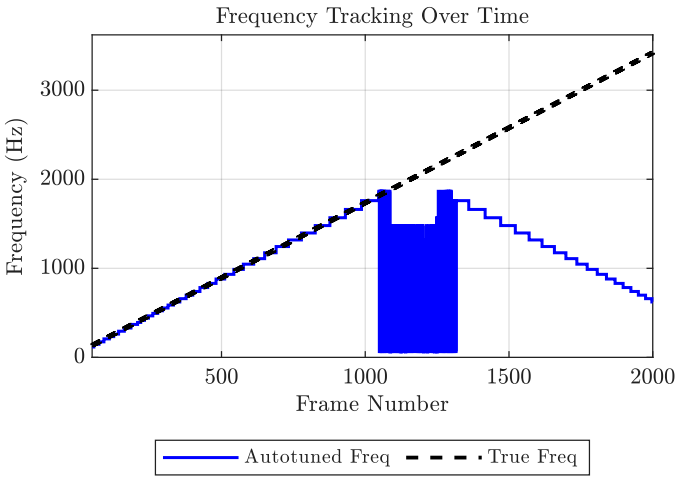


Fig. 5: Frequency error tracking for retuning based against $f_0$

As seen in Figure 5, the retuning algorithm is able to map accurately the current signal to the $f_0$ desired. At 2 kHz, the retuning fails due to the pitch tracking algorithms implemented and their inability to track in the attenuation zone of the filter, described in Section III. Around frame 1300, the retuning begins to work again as aliased (wrapped) frequencies re-enter the detectable spectrum. Due to the cyclic nature of the frequency mapping, the signal would periodically re-enter the attenuation zone if the observation window were extended

| **Metric** | **FFT** (ms) | **ACF** (ms) | $\Delta$ |
|---|---|---|---|
| $\overline{x}$ | 0.22 | 0.59 | 0.37 |
| $\sigma^2$ | 0.26 | 0.74 | 0.48 |
| max | 1.18 | 3.21 | 2.03 |

Table IV: FFT vs ACF: Performance Comparison

## V. Conclusion

This study evaluated the computational tradeoffs between FFT and ACF pitch-tracking algorithms using a real-time MATLAB implementation. In this implementation there were two functional calls, live demo, and test demo. In live demo the microphone would take in audio, while in test demo a chirp signal would move across the usable frequency spectrum and perform metric calculations for both functions.

The FFT showed promise in having a lesser computational latency than the ACF, while the ACF showed better tracking against error, and space complexity management. Results concluded that for an auto-tune system with a simple filter, both methods were accurate within relatively the same margins and could both be used in simple applications.

Future work would include retuning efficiency. This step would reduce artifacts, such as octave errors, and high frequencies. Additionally, overlapping frames which would cause a tremolo sound could also be a point to extend this project into.

## References


[1] A. V. Oppenheim, A. S. Willsky, and S. H. Nawab, *Signals and Systems*, 2nd. Upper Saddle River, NJ: Prentice Hall, 1997, ISBN: 0-13-814757-4.

[2] CP-Algorithms Contributors, *Fast fourier transform*, https://cp-algorithms.com/algebra/fft.html, Accessed: 2026-05-11, 2025.

[3] Jake, *Practical guide to autocorrelation*, https://scicoding.com/practical-guide-to-autocorrelation/, Accessed: 2026-05-11, May 2023.

[4] Wikipedia contributors, *Autocorrelation*, [Online; accessed 1-May-2026], 2026. [Online]. Available: https://en.wikipedia.org/wiki/Autocorrelation.

[5] Wikipedia contributors, *Fast fourier transform*, [Online; accessed 1-May-2026], 2026. [Online]. Available: https://en.wikipedia.org/wiki/Fast_Fourier_transform.

# APPENDIX

*Contents*



## A. *Github Repository*

edwardasilva/DSP-Project

## B. *MaxMSP Implementation*

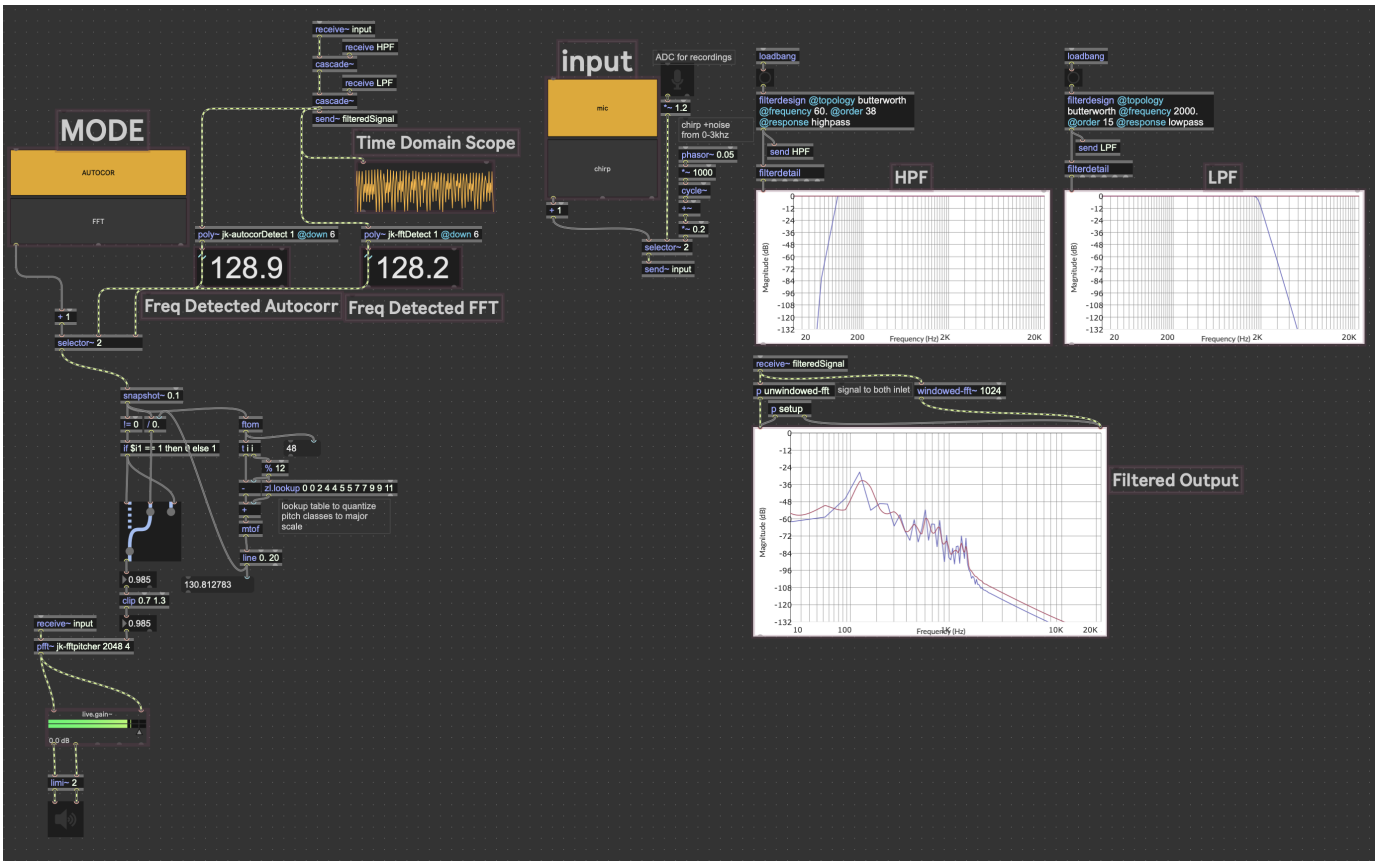


Fig. 6: MaxMSP: Main Patch

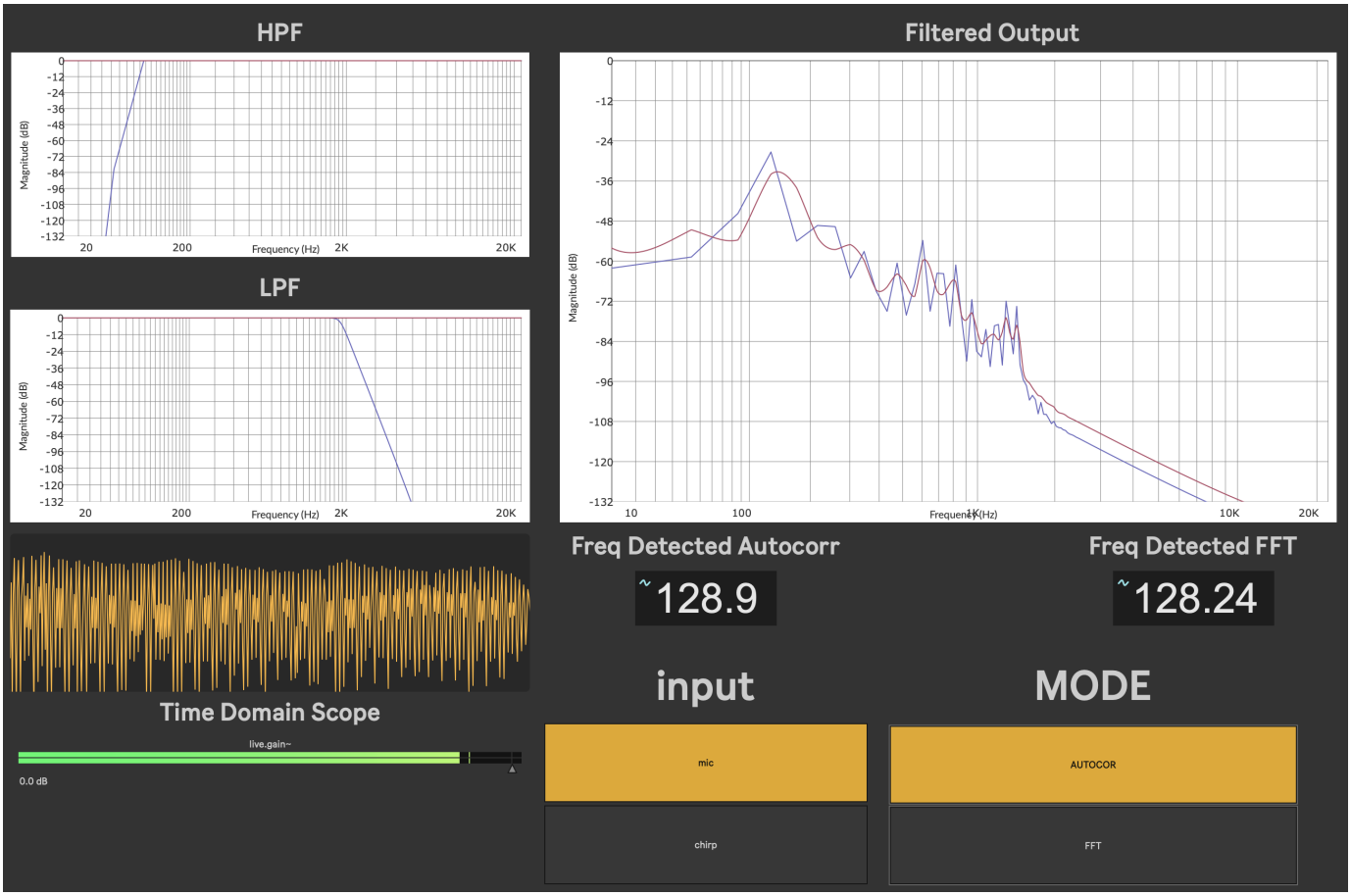


Fig. 7: MaxMSP Presentation: User Interface

```
Data history(8192);
Data acBuf(1024);
History ptr(0), counter(0), filled(0);
History lastFreq(0);
History lastOutput(0);
History x1(0), y1(0);

Fs     = samplerate;
W      = 2048;
stride = 2;
minLag = int(Fs / 2000);
maxLag = int(Fs / 55);
maxLag = min(maxLag, 1023);

// DC removal
alpha = 0.995;
dc    = in1 - x1 + alpha * y1;
x1    = in1;
y1    = dc;
poke(history, dc, ptr);
ptr = (ptr + 1) % 8192;

// Buffer fill
if (filled == 0) {
    if (ptr >= (W + maxLag + 10)) {
        filled = 1;
    };
};

// Analysis block - runs every 256 samples
counter = counter + 1;
if (counter >= 256) {
    counter = 0;
    if (filled == 1) {
        // RMS gate
        rmsSum = 0;
        for (i = 0; i < W; i = i + stride) {
            idx1   = (ptr - i + 8192) % 8192;
            v1     = peek(history, idx1);
            rmsSum = rmsSum + v1 * v1;
        };
        rms = sqrt(rmsSum / (W / stride));
        if (rms < 0.005) {
            lastOutput = 0;
        } else {
            // Normalized autocorrelation
            for (tau = minLag; tau <= maxLag; tau = tau + 1) {
                sum_acc = 0;
                energy1 = 0;
                energy2 = 0;
                for (i = 0; i < W; i = i + stride) {
                    idx1    = (ptr - i + 8192) % 8192;
                    idx2    = (ptr - i - tau + 16384) % 8192;
                    v1      = peek(history, idx1);
                    v2      = peek(history, idx2);
                    sum_acc = sum_acc + v1 * v2;
                    energy1 = energy1 + v1 * v1;
                    energy2 = energy2 + v2 * v2;
                };
                norm = sqrt(energy1 * energy2 + 0.000000000001);
                poke(acBuf, sum_acc / norm, tau);
            };
            // Peak search
            foundPeak = 0;
            prevCorr  = peek(acBuf, minLag);
            currCorr  = peek(acBuf, minLag + 1);
            for (tau = minLag + 2; tau <= maxLag; tau = tau + 1) {
                nextCorr = peek(acBuf, tau);
                if (foundPeak == 0) {
                    if (currCorr > prevCorr) {
                        if (currCorr > nextCorr) {
                            if (currCorr > 0.30) {
                                foundPeak = 1;
                                peakTau   = tau - 1;
```

```
                                yL  = peek(acBuf, peakTau - 1);
                                yM  = peek(acBuf, peakTau);
                                yR  = peek(acBuf, peakTau + 1);
                                den = yL - 2 * yM + yR;
                                p   = 0;
                                if (abs(den) > 0.00000001) {
                                    p = 0.5 * (yL - yR) / den;
                                };
                                p          = min(p, 1);
                                p          = max(p, -1);
                                refinedTau = peakTau + p;
                                newFreq    = Fs / refinedTau;
                                if (newFreq >= 55) {
                                    if (newFreq <= 2000) {
                                        lastFreq   = lastFreq + 0.2 * (newFreq - lastFreq);
                                        lastOutput = lastFreq;
                                    };
                                };
                            } else {
                                lastOutput = 0;
                            };
                        };
                    };
                };
                prevCorr = currCorr;
                currCorr = nextCorr;
            };
            if (foundPeak == 0) {
                lastOutput = 0;
            };
        };
    };
};
out1 = lastOutput;
```

Listing 1: MaxMSP Gen: Autocorrelation implementation

```
Data mags(4096);
History maxMag(0);
History peakBin(0);
History lastValidFreq(0);
History stableOut(0);
History silentFrames(0);
History prevBin(0);

magnitude = in1;
binIdx    = in2;
N         = vectorsize * 2;
halfN     = vectorsize;
Fs        = samplerate;
minFreq   = 55;
maxFreq   = 2000;

minBin = int((minFreq * N) / Fs);
maxBin = int((maxFreq * N) / Fs);
maxBin = min(maxBin, halfN - 2);

poke(mags, magnitude, binIdx);
if (binIdx >= minBin && binIdx <= maxBin) {
    if (magnitude > maxMag) {
        maxMag  = magnitude;
        peakBin = binIdx;
    }
}

frameDone = 0;
if (prevBin > binIdx) {
    frameDone = 1;
}
prevBin = binIdx;

if (frameDone) {
    if (maxMag > 0.0001) {
        silentFrames = 0;
```

```
        yL = peek(mags, peakBin - 1);
        yM = peek(mags, peakBin);
        yR = peek(mags, peakBin + 1);
        denom = yL - 2 * yM + yR;
        p = 0;
        if (abs(denom) > 0.000001) {
            p = 0.5 * (yL - yR) / denom;
        }
        p = clip(p, -1, 1);
        refinedBin = peakBin + p;
        binWidth   = Fs / N;
        newFreq    = refinedBin * binWidth;
        if (newFreq >= minFreq && newFreq <= maxFreq) {
            lastValidFreq = lastValidFreq + 0.4 * (newFreq - lastValidFreq);
            stableOut = lastValidFreq;
        }
    } else {
        silentFrames = silentFrames + 1;
        if (silentFrames > 8) {
            lastValidFreq = 0;
            stableOut = 0;
        }
    }
    maxMag  = 0;
    peakBin = 0;
}
out1 = stableOut / 2.0;
```

Listing 2: MaxMSP Gen: FFT implementation

### *C. Main.m*

```
function Main(runMode)
    % MAIN runs the autotuner program.
    %   Main() runs in 'test' mode by default.
    %   Main('live') runs in 'live' mode with audio processing.

    if nargin < 1
        runMode = 'test';
    end

    clc; close all;
    addpath('src'); % Add src functions/classes to path

    %% Start

    Fs                  = 48000;
    frameSize           = 4096;
    gainDB              = 0;
    frameCounter        = 0;
    totalSample         = 0;

    % Filter setup
    Fstop = 30;          % Stopband Frequency
    Fpass = 60;          % Passband Frequency
    Astop = 80;          % Stopband Attenuation (dB)
    Apass = 1;           % Passband Ripple (dB)
    match = 'stopband';  % Band to match exactly
    h  = fdesign.highpass(Fstop, Fpass, Astop, Apass, Fs);
    hpFilt = design(h, 'butter', 'MatchExactly', match);

    Fpass = 1500;        % Passband Frequency
    Fstop = 2000;        % Stopband Frequency
    Apass = 1;           % Passband Ripple (dB)
    Astop = 80;          % Stopband Attenuation (dB)
    match = 'stopband';  % Band to match exactly
    h  = fdesign.lowpass(Fpass, Fstop, Apass, Astop, Fs);
    lpFilt = design(h, 'butter', 'MatchExactly', match);

    % Synthetic source parameters (for test mode)
    sweepDuration   = 200.0;    % Seconds per chirp sweep
    f0              = 50;        % Start frequency (Hz)
    f1              = 4000;     % End frequency (Hz)
```

```
noiseAmplitude  = 0.01;     % White noise level
rng(0);                     % Repeatable test input

disp('Processing in terminal. Use Ctrl+C to stop.');

% Live audio reader
deviceReader = audioDeviceReader('SampleRate', Fs, 'SamplesPerFrame', frameSize); % , 'Device','M4'

runSimulation = true;
maxIterations = 2000; % Set a finite limit: 2000 basically covers a full cycle

hist_fft              = zeros(maxIterations, 1);
hist_autocorr         = zeros(maxIterations, 1);
hist_true             = zeros(maxIterations, 1);
hist_autotune         = zeros(maxIterations, 1);
hist_fft_latency      = zeros(maxIterations, 1);
hist_autocorr_latency = zeros(maxIterations, 1);

% Main Loop
while runSimulation && frameCounter < maxIterations
    frameCounter = frameCounter + 1;

    switch lower(runMode)
        case 'live'
            % Capture a frame of live audio
            audioChunk = deviceReader();
            f_true = NaN; % Unknown in live mode

        case 'test'
            % Generate a frame of synthetic test audio (chirps + noise)
            idx         = (0:frameSize-1).';
            tAbs        = (totalSample + idx) / Fs;
            tSweep      = mod(tAbs, sweepDuration);
            upChirp     = chirp(tSweep, f0, sweepDuration, f1, 'linear');
            downChirp   = 0.6 * chirp(tSweep, f1, sweepDuration, f0, 'linear');
            noise       = noiseAmplitude * randn(frameSize, 1);
            audioChunk  = 0.5 * upChirp + 0.5 * downChirp + 0.25 * noise;
            totalSample = totalSample + frameSize;

            % Calculate the true instantaneous frequency of the up-chirp
            % at the midpoint of the current frame
            tMid = (totalSample + frameSize/2) / Fs;
            tSweepMid = mod(tMid, sweepDuration);
            f_true = f0 + (f1 - f0) * (tSweepMid / sweepDuration);

            fprintf("Freq: %.2f\n", f_true);
        otherwise
            error('Invalid runMode: %s. Use ''live'' or ''test''.', runMode);
    end

    % Call Run function -> calculations
    res = Run(audioChunk, hpFilt, lpFilt, Fs, gainDB);
    hist_fft(frameCounter)              = res.f0_fft;
    hist_autocorr(frameCounter)          = res.f0_autocorr;
    hist_true(frameCounter)              = f_true;
    hist_fft_latency(frameCounter)       = res.fftTime_ms;
    hist_autocorr_latency(frameCounter)  = res.autocorrTime_ms;

    % Retuning: fft
    [audioOutFrame, noteFreqsOut] = Decide_op_freq(res.f0_fft, audioChunk);
    hist_autotune(frameCounter)   = noteFreqsOut;
end

release(deviceReader);

% Truncate arrays to the exact number of frames processed
hist_fft              = hist_fft(1:frameCounter);
hist_autocorr         = hist_autocorr(1:frameCounter);
hist_true             = hist_true(1:frameCounter);
hist_autotune         = hist_autotune(1:frameCounter);
hist_fft_latency      = hist_fft_latency(1:frameCounter);
hist_autocorr_latency = hist_autocorr_latency(1:frameCounter);
frames                = (1:frameCounter)';
```

```
    % Generate final plot
    e = errorplots(hist_fft, hist_autocorr, hist_true, hist_autotune, frames);
    l = latencyplots(hist_fft_latency, hist_autocorr_latency, frames);

    %% Export as SVG
    export_dir = 'Figures';
    if ~exist(export_dir, 'dir')
       mkdir(export_dir);
    end

    all_figs = [e.hFig1, e.hFig2, l.hFig1];

    for i = 1:length(all_figs)
       fig = all_figs(i);
       filename = sprintf('%s.svg', fig.Name);
       full_path = fullfile(export_dir, filename);
       exportgraphics(fig, full_path, 'ContentType', 'vector', 'BackgroundColor', 'none');
    end

end
```

Listing 3: Main.m MATLAB implementation

### *D. Run.m*

```
function res = Run(audioChunk, hpFilt, lpFilt, Fs, gainDB)
% Script which will run the audiochunk given to it. Real cases for project
% are as follows:
%   Case 1: Live
%           This is live audio input from a microphone
%   Case 2: Test
%           This is sliding chirp inputs to show perfect scenario

    persistent pitchHistory
    if isempty(pitchHistory)
        pitchHistory = zeros(2048, 1);
    end

    % Apply gain
    processedChunk = max(min(audioChunk * 10^(gainDB/20), 1), -1);

    % FILTER input (Cascaded HP -> LP)
    tempSignal     = filter(hpFilt, processedChunk);    % highpass first
    filteredOutput = filter(lpFilt, tempSignal);        % pass highpass signal through lowpass

    % DECIMATE TO fs = 8kHz
    stage1      = decimate(filteredOutput, 3);
    pitchSignal = decimate(stage1, 2);
    Fs_pitch    = Fs / 6;
    % decimation causing sample rate to drop by 1/6

    % Update Circular Buffer
    numNewSamples = length(pitchSignal);
    pitchHistory(1:end-numNewSamples) = pitchHistory(numNewSamples+1:end);
    pitchHistory(end-numNewSamples+1:end) = pitchSignal(:);

    pitchSignal_NoDc = pitchSignal(:) - mean(pitchSignal(:)); % Remove DC offset

    %% FFT Implementation: Get F0
    tFFT = tic;
    L_pitch = length(pitchSignal_NoDc);
    nfft_hires = 8192; % Increase bin resolution

    spectraPitch = fft(pitchSignal_NoDc .* hamming(L_pitch), nfft_hires);
    [~, I] = max(abs(spectraPitch(1:floor(end/2)))); % Max amplitude bin

    % Frequencies corresponding to the higher-resolution bins
    if I > 1 && I < floor(nfft_hires/2)
        left_bin  = abs(spectraPitch(I-1));
        max_bin   = abs(spectraPitch(I));
        right_bin = abs(spectraPitch(I+1));

        p = 0.5 * (left_bin - right_bin) / (left_bin - 2*max_bin + right_bin);
```

```
    if isnan(p) || abs(p) > 1
        p = 0;
    end

    f0_fft = (I - 1 + p) * (Fs_pitch / nfft_hires);
else
    f0_fft = (I - 1) * (Fs_pitch / nfft_hires);
end

fftTime_ms = toc(tFFT) * 1000;

%% AUTOCORRELATION IMPLEMENTATION
tAutocorr = tic;
% Use only the valid (non-zero or filled) history segment
validHistory = pitchHistory(:);
validHistory = validHistory - mean(validHistory);

[acor, lags] = xcorr(validHistory, 'coeff'); % Use unbiased or raw cross-correlation
centerIdx = ceil(length(acor) / 2);

% Limit search range corresponding to 55 Hz - 2000 Hz
minLag = floor(Fs_pitch / 2000);
maxLag = floor(Fs_pitch / 55);
searchIndices = (centerIdx + minLag) : min(centerIdx + maxLag, length(acor));

if ~isempty(searchIndices)
    % Autocorrelate search indexes sample segments
    segment = acor(searchIndices);

    % local peaks
    [peaks, locs] = findpeaks(segment);

    if ~isempty(peaks)
        % first valid peak (not global largest)
        peakIdx = locs(1);
        maxVal = peaks(1);
        actualIdx = searchIndices(peakIdx);
    else
        f0_autocorr = 0;
        maxVal = 0; % prevent fall-through error
    end

    % gain threshold checks whether the peak is prominent enough
    if maxVal > 0.2
       left_lag  = acor(actualIdx - 1);
       peak_lag  = acor(actualIdx);
       right_lag = acor(actualIdx + 1);

        % Parabolic interpolation
        den = (left_lag - 2*peak_lag + right_lag);
        if abs(den) < 1e-6
           p_corr = 0;
        else
           p_corr = 0.5 * (left_lag - right_lag) / den;
        end

        lag_samples = lags(actualIdx) + p_corr; % get sample lag

        if lag_samples <= 0 % prevent divide by 0
           f0_autocorr = 0;
        else
           f0_autocorr = Fs_pitch / lag_samples;
        end

        if f0_autocorr < 55 || f0_autocorr > 2000 % limit range to realistic fundamental range
           f0_autocorr = 0;
        end
    else
       if ~exist('f0_autocorr', 'var')
           f0_autocorr = 0;
       end
    end
else
   f0_autocorr = 0;
```

```
    end

    autocorrTime_ms = toc(tAutocorr) * 1000;

    % Store base signals
    res.processedChunk    = processedChunk;
    res.filteredOutput    = filteredOutput;
    res.pitchSignal       = pitchSignal;
    res.Fs_pitch          = Fs_pitch;
    res.f0_fft            = f0_fft;
    res.f0_autocorr       = f0_autocorr;
    res.fftTime_ms        = fftTime_ms;
    res.autocorrTime_ms   = autocorrTime_ms;
end
```

Listing 4: Run.m MATLAB Implementation

### *E. Decide_op_freq.m*

```
function [audioOutFrame, noteFreqsOut] = Decide_op_freq(f_naught, call_deviceReader)
    if nargin < 2
        error('Usage: Decide_op_freq(f_naught, call_deviceReader)');
    end

    persistent fs noteFreqs player lastFreq lockedFreq lastScale notes

    if isempty(noteFreqs)
        fs = 48000;
        fid = fopen('NOTES.txt');
        if fid == -1
            error('cannot open file NOTES.txt')
        end
        noteFreqs = [];
        notes = {};
        note = '';
        while ~feof(fid)
            line = strtrim(fgetl(fid));
            if isempty(line), continue; end
            if startsWith(line, '#')
                note = strtrim(line(2:end));
                continue;
            end
            vals = sscanf(line, '%f,%f');
            if numel(vals) < 1
                continue;
            end
            noteFreqs(end+1,1) = vals(1);
            notes{end+1,1} = note;
        end
        fclose(fid);

        player = audioDeviceWriter('SampleRate', ...
            fs, 'Device', 'Default', ...
            'SupportVariableSizeInput', true);

        lastFreq = 0;
        lockedFreq = 0;
        lastScale = 1;
    end

    %% FIND NOTE
    closestIndex = numel(noteFreqs);
    for i = 1:numel(noteFreqs)-1
        midpoint = (noteFreqs(i) + noteFreqs(i+1)) / 2;
        semitone_spacing = (noteFreqs(i+1)-noteFreqs(i));
        if f_naught < midpoint
            closestIndex = i;
            n_semitones_up = ((noteFreqs(i)-f_naught)/semitone_spacing);

            break;
        end
    end
    f_out = noteFreqs(closestIndex);
```

```
    matchedNote = notes{closestIndex};

    %% SMOOTH + LOCK
    alpha = 0.2;
    f_smooth = alpha*f_out + (1-alpha)*lastFreq;
    if abs(f_smooth - lockedFreq) > 20
        lockedFreq = f_out;
    end
    lastFreq = f_smooth;
    f_out = lockedFreq;

    %% SCALE
    scale = f_out / max(f_naught, 1);
    beta = 0.2;
    scale = beta*scale + (1-beta)*lastScale;
    lastScale = scale;

    %% PITCH SHIFT FRAME

    targetN = length(call_deviceReader);

    y = shiftPitch(call_deviceReader, n_semitones_up);

    % force valid vector
    y = y(:);

    % ensure constant size
    if length(y) > targetN
        y = y(1:targetN);
    elseif length(y) < targetN
        y(end+1:targetN) = 0;
    end

    % windowing
    y = y .* hann(targetN);

    % normalize
    y = y / (max(abs(y)) + 1e-6) * 0.5;

    %% AUDIO OUTPUT

    step(player, y);
    audioOutFrame = y;
    noteFreqsOut = f_out;

    fprintf('Selected frequency: %g\n', f_out);
    fprintf('Associated note: %s\n', matchedNote);
end
```

Listing 5: Decide_op_freq.m MATLAB Implementation: Logic for calculating correct frequency operation

### *F. errorplots.m*

```
function e = errorplots(f_fft_arr, f_autocorr_arr, f_true_arr, f_autotuned_arr, frames)
    set(groot, 'defaultTextInterpreter', 'latex');
    set(groot, 'defaultAxesTickLabelInterpreter', 'latex');
    set(groot, 'defaultLegendInterpreter', 'latex');
    set(groot, 'defaultAxesYLimMode', 'auto');
    set(groot, 'defaultAxesXLimMode', 'auto');
    set(groot, 'defaultAxesTickDir', 'out');

    xleftend = 50;

    % Standardize aesthetics for IEEE Print (Single Column Width ~ 3.5in)
    fs_title    = 12;
    fs_labels   = 10;
    fs_axes     = 9;
    fs_legend   = 9;
    lw          = 1.2;

    err_fft      = f_fft_arr - f_true_arr;
    err_autocorr = f_autocorr_arr - f_true_arr;
    start_idx    = 50;
```

```
    if length(frames) > start_idx
        frames          = frames(start_idx:end);
        f_true_arr      = f_true_arr(start_idx:end);
        f_fft_arr       = f_fft_arr(start_idx:end);
        f_autocorr_arr  = f_autocorr_arr(start_idx:end);
        f_autotuned_arr = f_autotuned_arr(start_idx:end);
        err_fft         = err_fft(start_idx:end);
        err_autocorr    = err_autocorr(start_idx:end);
    end

    % FIGURE 1: Frequency Tracking
    hFig1 = figure('Name', 'FreqTracking', 'NumberTitle', 'off', ...
                   'Units', 'inches', 'Position', [1, 1, 4, 3]);
    hFig1.PaperPositionMode = 'auto';

    stairs(frames, f_autotuned_arr, 'b-', 'LineWidth', lw); hold on;
    stairs(frames, f_true_arr, 'k--', 'LineWidth', lw + 0.3);
    hold off;

    title('Frequency Tracking Over Time', 'FontSize', fs_title);
    xlabel('Frame Number', 'FontSize', fs_labels);
    ylabel('Frequency (Hz)', 'FontSize', fs_labels);

    % Split legend into 2 columns so it fits in the 3.5in width
    legend('Autotuned Freq', 'True Freq', 'Location', 'southoutside', 'NumColumns', 2, 'FontSize',
    fs_legend);
    set(gca, 'FontSize', fs_axes);
    grid on;

    xlim([50, max(frames)]);
    ylim([0, max(f_true_arr) + 200]);

    % FIGURE 2: Residual Error
    hFig2 = figure('Name', 'ErrorTracking', 'NumberTitle', 'off', ...
                   'Units', 'inches', 'Position', [5, 1, 4, 3]);
    hFig2.PaperPositionMode = 'auto';

    plot(frames, err_fft, 'b-', 'LineWidth', lw); hold on;
    plot(frames, err_autocorr, 'r-', 'LineWidth', lw);
    yline(0, 'k--', 'LineWidth', lw);
    hold off;

    title('Residual Frequency Error ($f_{est} - f_{true}$)', 'FontSize', fs_title);
    xlabel('Frame Number', 'FontSize', fs_labels);
    ylabel('Error (Hz)', 'FontSize', fs_labels);

    legend('FFT Error', 'ACF Error', 'Zero Error', 'Location', 'southoutside', 'NumColumns', 3, 'FontSize',
     fs_legend);
    set(gca, 'FontSize', fs_axes);
    grid on;

    xlim([xleftend, max(frames)]);
    ylim([-100, 10]);

    % Return both figure handles
    e.hFig1 = hFig1;
    e.hFig2 = hFig2;
end
```

Listing 6: errorplots.m MATLAB implementation: Helper for plotting out the error bounds for FFT / Autocorrelation

### G. *latencyplots.m*

```
function l = latencyplots(fft_latency, autocorr_latency, frames)
    set(groot, 'defaultTextInterpreter', 'latex');
    set(groot, 'defaultAxesTickLabelInterpreter', 'latex');
    set(groot, 'defaultLegendInterpreter', 'latex');
    set(groot, 'defaultAxesYLimMode', 'auto');
    set(groot, 'defaultAxesXLimMode', 'auto');
    set(groot, 'defaultAxesTickDir', 'out');
    xleftend = 50;
```

```
    % Standardize aesthetics for IEEE Print (Single Column Width ~ 3.5in)
    fs_title    = 12;
    fs_labels   = 10;
    fs_axes     = 9;
    fs_legend   = 9;
    lw          = 1.2;

    % Statistics
    meanFFT  = mean(fft_latency);
    meanAC   = mean(autocorr_latency);

    if numel(fft_latency) > xleftend
        maxFFT = max(fft_latency(xleftend+1:end));
    else
        maxFFT = max(fft_latency);
    end
    if numel(autocorr_latency) > xleftend
        maxAC = max(autocorr_latency(xleftend+1:end));
    else
        maxAC = max(autocorr_latency);
    end

    stdFFT   = std(fft_latency);
    stdAC    = std(autocorr_latency);

    % FFT vs Autocorrelation Latency
    hFig1 = figure('Name', 'LatencyComparison', 'NumberTitle', 'off', ...
                'Units', 'inches', 'Position', [9, 1, 4, 3]);
    hFig1.PaperPositionMode = 'auto';

    plot(frames, fft_latency, 'b-', 'LineWidth', lw * .8); hold on;
    plot(frames, autocorr_latency, 'r-', 'LineWidth', lw * .8);

    % Plot mean lines on top by bringing them to front after plotting data
    y1 = yline(meanFFT, 'c--', 'LineWidth', lw * 1.2);
    y2 = yline(meanAC,  'y--', 'LineWidth', lw * 1.2);
    uistack(y1, 'top');
    uistack(y2, 'top');
    hold off;

    title('FFT vs ACF Latency', 'FontSize', fs_title);
    xlabel('Frame Number', 'FontSize', fs_labels);
    ylabel('Latency (ms)', 'FontSize', fs_labels);

    legend('FFT', 'ACF', ...
           sprintf('FFT Mean = %.2f ms', meanFFT), ...
           sprintf('ACF Mean = %.2f ms', meanAC), ...
           'Location', 'southoutside', 'NumColumns', 2, 'FontSize', fs_legend);
    set(gca, 'FontSize', fs_axes);
    grid on;

    xlim([xleftend, max(frames)]);
    ylim([0, max(maxFFT, maxAC) * 1.15]);

    % Statistics printed as a markdown table in the command window
    % Prepare table lines
    header           = '|  Metric   |  FFT (ms) | Autocorr (ms)|';
    meanLine = sprintf('|   Mean    |   %.2f    |      %.2f    |', meanFFT, meanAC);
    stdLine  = sprintf('|   Std     |   %.2f    |      %.2f    |', stdFFT,  stdAC);
    maxLine  = sprintf('|   Max     |   %.2f    |      %.2f    |', maxFFT,  maxAC);

    % Display
    fprintf('%s\n%s\n%s\n%s\n', header, meanLine, stdLine);
    fprintf('%s\n', maxLine);

    % Return figure handle
    l.hFig1 = hFig1;
end
```

Listing 7: latencyplots.m MATLAB implementation: Helper for plotting FFT vs Autocorrelation computation latency