Development and Evaluation of Pitch Detection Algorithms for a Karaoke Training Application

David Rostcheck
Independent Researcher
david@rostcheck.com

Amazon Q Developer CLI w/ Anthropic Claude Sonnet 3.7
AI Assistant

***Abstract*—This paper presents the research, development, and evaluation of pitch detection algorithms and vocal line extraction methods for a karaoke training application called PitchTrack. We compare several established pitch detection algorithms including YIN, pYIN (probabilistic YIN), CREPE (Convolutional Representation for Pitch Estimation), and McLeod Pitch Method (MPM), evaluating their accuracy, computational efficiency, and suitability for real-time vocal pitch tracking. We implement and test these algorithms using Python libraries including librosa and aubio, and develop post-processing techniques to improve pitch tracking for vocal applications. Additionally, we evaluate vocal line extraction methods including Spleeter and Demucs to isolate vocals from mixed audio recordings, enabling users to practice with just the vocal line. Our findings indicate that pYIN offers the best balance of accuracy and computational efficiency for vocal pitch tracking, while Demucs provides superior vocal isolation quality despite longer processing times. We also discuss the development of a visualization system that provides intuitive feedback to users through a piano roll interface.**

***Index Terms*—pitch detection, audio source separation, vocal extraction, music information retrieval, karaoke, vocal training, pYIN, Demucs, real-time audio processing**

I. Introduction

Accurate pitch detection is a fundamental challenge in music information retrieval and audio signal processing. For applications like vocal training and karaoke systems, the ability to precisely track the fundamental frequency of a singing voice in real-time is essential for providing meaningful feedback to users. However, vocal pitch detection presents unique challenges due to the complex harmonic structure of the human voice, presence of vibrato, rapid pitch transitions, and varying timbres across different singers [2].

In this paper, we describe the research and development process for PitchTrack, a karaoke training application that provides real-time visual feedback on vocal pitch. We focus on the evaluation of different pitch detection algorithms, their implementation using available libraries, and the development of post-processing techniques to improve pitch tracking specifically for vocal applications.

The main contributions of this paper are:

- A comparative analysis of pitch detection algorithms for vocal applications
- Implementation and evaluation of post-processing techniques to improve pitch tracking quality
- Development of a real-time visualization system for intuitive pitch feedback
- Practical recommendations for implementing pitch detection in vocal training applications

II. Background and Related Work

A. Pitch Detection Algorithms

Pitch detection algorithms can be broadly categorized into time-domain and frequency-domain approaches [5]. Time-domain methods typically analyze the periodicity of the waveform, while frequency-domain methods examine the spectral content of the signal.

1) YIN Algorithm: The YIN algorithm [1] is a widely used time-domain pitch detection method based on autocorrelation with additional processing steps to reduce errors. It computes a modified autocorrelation function called the cumulative mean normalized difference function, which helps reduce octave errors common in basic autocorrelation methods.

The YIN algorithm can be summarized in the following steps:

1) Compute the difference function:

$$d_t(\tau) = \sum_{j=1}^{W} (x_j - x_{j+\tau})^2 \tag{1}$$

where x is the input signal, τ is the lag, and W is the window size.

2) Compute the cumulative mean normalized difference function:

$$d'_t(\tau) = \begin{cases} 1, & \text{if } \tau = 0 \\ \frac{d_t(\tau)}{\frac{1}{\tau}\sum_{j=1}^{\tau} d_t(j)}, & \text{otherwise} \end{cases} \tag{2}$$

3) Find the minimum of $d'_t(\tau)$ that is below a threshold.
4) Refine the estimate using parabolic interpolation.

2) pYIN (Probabilistic YIN): pYIN [2] extends the YIN algorithm by incorporating a probabilistic model to improve pitch tracking accuracy. It uses multiple pitch candidates from the YIN algorithm and applies a hidden Markov model (HMM) to find the most likely pitch trajectory over time. This approach is particularly effective for vocal pitch tracking as it better handles note transitions and vibrato.

3) CREPE (Convolutional Representation for Pitch Estimation): CREPE [3] represents a more recent approach using deep learning. It employs a convolutional neural network trained on a large dataset of labeled audio to predict pitch. CREPE has demonstrated state-of-the-art accuracy, particularly for vocal pitch tracking, but comes with higher computational requirements.

4) McLeod Pitch Method (MPM): The McLeod Pitch Method [4] is based on the normalized square difference function (NSDF) and is designed to be computationally efficient while maintaining good accuracy. It is particularly effective at handling noisy signals.

B. Libraries for Pitch Detection

Several libraries implement these algorithms and provide accessible interfaces for developers:

- **Aubio**: A C library with Python bindings that implements YIN, YinFFT, and other algorithms.
- **Librosa**: A Python package for music and audio analysis that includes implementations of pYIN.
- **TarsosDSP**: A Java library that implements MPM and other algorithms.
- **CREPE**: A TensorFlow implementation of the CREPE algorithm.

III. Methodology

A. Evaluation Criteria

We evaluated pitch detection algorithms based on the following criteria:

- **Accuracy**: How precisely the algorithm tracks pitch variations, particularly for vocal audio with vibrato and transitions.
- **Latency**: The delay between audio input and pitch detection output, critical for real-time applications.
- **Robustness**: How well the algorithm handles different voices, microphones, and acoustic environments.
- **Computational Efficiency**: The computational resources required, affecting the feasibility for real-time applications.
- **Implementation Complexity**: The effort required to implement and integrate the algorithm.

B. Implementation

We implemented pitch detection using both the aubio and librosa libraries in Python. The following code snippet shows our implementation of the YIN algorithm using aubio:

```
def detect_pitch_aubio(file_path, method="yin"
    ,
                       buffer_size=2048,
                           hop_size=512,
                       sample_rate=44100):
    # Create pitch object
    pitch_o = aubio.pitch(method, buffer_size,
         hop_size,
                           sample_rate)
    pitch_o.set_unit("Hz")
    pitch_o.set_silence(-40)
    pitch_o.set_tolerance(0.8)

    # Load audio file
    source = aubio.source(file_path,
        sample_rate, hop_size)
    sample_rate = source.samplerate

    # Lists to store results
    pitches = []
    confidences = []

    # Process audio file
    while True:
        samples, read = source()
        pitch = pitch_o(samples)[0]
        confidence = pitch_o.get_confidence()

        pitches.append(float(pitch))
        confidences.append(float(confidence))

        if read < hop_size:
            break

    # Convert frame indices to time
    times = [t * hop_size / float(sample_rate)
            for t in range(len(pitches))]

    return times, pitches, confidences
```

For pYIN, we used the implementation provided by librosa:

```
def detect_vocal_pitch(file_path, hop_length
    =512,
                       fmin=80.0, fmax=800.0):
    # Load audio file
    y, sr = librosa.load(file_path, sr=None)

    # Use pYIN algorithm
    f0, voiced_flag, voiced_probs = librosa.
        pyin(
        y,
        fmin=fmin,
        fmax=fmax,
        sr=sr,
        hop_length=hop_length,
        fill_na=None  # Don't fill unvoiced
            sections
    )

    # Convert frame indices to time
    times = librosa.times_like(f0, sr=sr,
                               hop_length=
                                   hop_length)

    return times, f0, voiced_probs
```

C. Post-Processing Techniques

To improve the quality of pitch tracking specifically for vocal applications, we implemented several post-processing techniques:

1) Energy Thresholding: We used the root mean square (RMS) energy of the signal to distinguish between voiced and unvoiced segments:

```
# Calculate energy for voice activity
    detection
```

```
energy = librosa.feature.rms(
    y=y, frame_length=hop_length*2,
    hop_length=hop_length)[0]
energy = energy / np.max(energy) if np.max(
    energy) > 0 else energy

# Apply threshold to remove low-confidence
    segments
for i in range(len(f0)):
    if confidence[i] > energy_threshold and f0
        [i] > 0:
        processed_pitch[i] = f0[i]
    else:
        processed_pitch[i] = 0
```

2) Continuity Constraints: To avoid octave jumps and other pitch tracking errors, we implemented continuity constraints that penalize large pitch changes between consecutive frames:

```
# Apply continuity constraints to avoid octave
    jumps
for i in range(1, len(processed_pitch)):
    if processed_pitch[i] > 0 and
        processed_pitch[i-1] > 0:
        # Calculate octave difference
        octave_diff = np.abs(np.log2(
            processed_pitch[i] /
                processed_pitch[i-1]))

        # If jump is too large, try to correct
            it
        if octave_diff > continuity_tolerance:
            # Check if it's likely an octave
                error
            if abs(octave_diff - 1.0) < 0.1:
                # Close to an octave jump
                # Adjust to previous octave if
                    confidence allows
                if confidence[i] < confidence[
                    i-1] * (1 + octave_cost):
                    if processed_pitch[i] >
                        processed_pitch[i-1]:
                        processed_pitch[i] =
                            processed_pitch[i]
                            / 2.0
                    else:
                        processed_pitch[i] =
                            processed_pitch[i]
                            * 2.0
```

3) Median Filtering: To smooth the pitch contour and reduce jitter, we applied median filtering to segments of detected pitch:

```
# Apply median filtering to smooth the pitch
    contour
valid_indices = processed_pitch > 0
if np.any(valid_indices):
    # Create a copy for filtering
    smoothed_pitch = np.copy(processed_pitch)

    # Only apply filtering to segments with
        valid pitch
    segments = []
    segment_start = None

    # Find continuous segments
    for i in range(len(valid_indices)):
        if valid_indices[i] and segment_start
            is None:
            segment_start = i
        elif not valid_indices[i] and
            segment_start is not None:
            segments.append((segment_start, i)
                )
            segment_start = None

    # Add the last segment if it exists
    if segment_start is not None:
        segments.append((segment_start, len(
            valid_indices)))

    # Apply median filtering to each segment
    for start, end in segments:
        if end - start > median_filter_size:
            segment = processed_pitch[start:
                end]
            smoothed_segment = medfilt(segment
                , median_filter_size)
            smoothed_pitch[start:end] =
                smoothed_segment

    processed_pitch = smoothed_pitch
```

D. Visualization System

We developed a piano roll visualization system using PyQt6 to provide intuitive feedback to users. The visualization includes:

- A piano roll display with evenly spaced pitch lines
- Colored key indicators showing black and white piano keys
- Connected pitch lines showing the continuous pitch trajectory
- Key highlighting to indicate the currently sung note

The visualization maps detected pitch to musical notes using the following frequency-to-MIDI conversion:

$$MIDI = 69 + 12 \times \log_2\left(\frac{f}{440}\right) \tag{3}$$

where f is the frequency in Hz, and 69 corresponds to A4 (440 Hz).

IV. Results and Discussion

A. Algorithm Comparison

We evaluated the performance of different pitch detection algorithms on vocal recordings. Table I summarizes our findings.

TABLE I
COMPARISON OF PITCH DETECTION ALGORITHMS FOR VOCAL APPLICATIONS

Algorithm	Accuracy	Latency	CPU Usage	Robustness
YIN	Medium	Low	Low	Medium
pYIN	High	Medium	Medium	High
CREPE	Very High	High	Very High	Very High
MPM	Medium	Low	Low	Medium

Our experiments showed that pYIN consistently outperformed the basic YIN algorithm for vocal pitch tracking, particularly in handling vibrato and note transitions. CREPE achieved the highest accuracy but at the cost of significantly higher computational requirements, making it less suitable for real-time applications on devices with limited resources.

B. Post-Processing Effectiveness

The post-processing techniques we implemented significantly improved the quality of pitch tracking. Table II shows the impact of different techniques.

TABLE II
IMPACT OF POST-PROCESSING TECHNIQUES ON PITCH TRACKING QUALITY

Technique	Improvement	CPU Cost	Latency Impact
Energy Thresholding	High	Low	Negligible
Continuity Constraints	Medium	Low	Negligible
Median Filtering	High	Low	Medium
All Combined	Very High	Medium	Medium

Energy thresholding was particularly effective at removing spurious pitch detections during unvoiced segments, while median filtering significantly reduced jitter in the pitch contour. The combination of all techniques provided the best results, with only a moderate increase in computational cost.

C. Visualization Effectiveness

User testing of the piano roll visualization indicated that it provided intuitive feedback on pitch accuracy. The evenly spaced pitch lines were found to be more readable than a traditional piano keyboard visualization, while still maintaining the visual connection to musical notes through the colored key indicators.

V. VOCAL LINE EXTRACTION

In addition to pitch detection, our karaoke training application requires the ability to isolate vocal lines from mixed audio recordings. This capability allows users to practice with just the vocal line, helping them learn melodies more effectively before attempting to sing along with the full accompaniment.

A. Motivation

Vocal line extraction serves several important purposes in a karaoke training application:

- Enables users to hear the target melody in isolation
- Provides a reference for pitch and timing without instrumental distractions
- Allows for direct comparison between the user's voice and the original vocal
- Facilitates progressive learning approaches (vocal only → vocal with accompaniment → accompaniment only)

B. Algorithms Evaluated

We evaluated three approaches to vocal line extraction:

1) Traditional DSP Methods (Librosa): While Librosa provides excellent audio processing capabilities, its harmonic-percussive source separation (HPSS) [6] is not specifically designed for vocal isolation:

```
import librosa
import numpy as np

# Load the audio file
y, sr = librosa.load('song.mp3', sr=None)

# Perform harmonic-percussive source
    separation
y_harmonic, y_percussive = librosa.effects.
    hpss(y)

# The harmonic component contains vocals but
    also other harmonic instruments
```

This approach proved insufficient for clean vocal isolation, as it doesn't specifically target vocals and produces significant bleed-through from other harmonic instruments.

2) Spleeter (Deezer): Spleeter [7] is an open-source music source separation library developed by Deezer Research that uses deep learning to separate audio into different stems:

```
from spleeter.separator import Separator
import numpy as np

# Initialize the separator with the desired
    model
separator = Separator('spleeter:4stems')

# Load and separate audio
waveform, sr = librosa.load('song.mp3', sr
    =44100, mono=False)
# Ensure correct shape (samples, channels)
if waveform.shape[0] == 2:
    waveform = waveform.T

# Apply high-pass filter to improve vocal
    clarity
from scipy import signal
cutoff = 150  # Hz
nyquist = 0.5 * sr
normal_cutoff = cutoff / nyquist
b, a = signal.butter(4, normal_cutoff, btype='
    high', analog=False)
filtered = np.zeros_like(waveform)
for i in range(waveform.shape[1]):
    filtered[:, i] = signal.filtfilt(b, a,
        waveform[:, i])

# Separate the audio
prediction = separator.separate(filtered)
vocals = prediction['vocals']
```

We tested Spleeter with multiple model configurations:

- 2 stems (vocals + accompaniment)
- 4 stems (vocals, drums, bass, other)
- 5 stems (vocals, drums, bass, piano, other)

3) Demucs (Facebook Research): Demucs (Deep Extractor for Music Sources) [8] is a state-of-the-art music source separation model developed by Facebook Research:

```
import torch
from demucs.pretrained import get_model
from demucs.apply import apply_model
import numpy as np
import librosa

# Load the model
model = get_model('htdemucs')
model.cpu()
model.eval()

# Load and process audio
wav, sr = librosa.load('song.mp3', sr=model.
    samplerate, mono=False)
if wav.ndim == 1:
    wav = np.stack([wav, wav])

# Apply the model
wav = torch.tensor(wav)
with torch.no_grad():
    sources = apply_model(model, wav[None])[0]

# Extract vocals
sources = sources.cpu().numpy()
vocals = sources[model.sources.index('vocals')
    ]
```

C. Evaluation Methodology

We evaluated each algorithm based on:

- **Vocal Isolation Quality**: How cleanly the vocals were separated from instruments
- **Processing Speed**: Time required to process a 3:09 minute song
- **Implementation Complexity**: Ease of integration into our application
- **Architecture Compatibility**: Performance on Apple Silicon (arm64) hardware

Testing was performed on an Apple Silicon Mac using a Python 3.11 virtual environment with arm64 architecture. We used the song "Code Monkey" by Jonathan Coulton (3:09 in length) as our primary test case.

D. Results

1) Vocal Isolation Quality: Our extensive listening tests revealed significant differences in vocal isolation quality across the evaluated algorithms. Demucs consistently demonstrated superior vocal isolation with minimal instrumental bleed-through, producing the cleanest separation between vocals and accompaniment. The Spleeter 5-stem model performed reasonably well but exhibited noticeable instrumental artifacts in the vocal track. The Spleeter 4-stem model provided moderate separation quality, while the 2-stem model showed significant instrumental bleed-through. As expected, Librosa's HPSS, which was not specifically designed for vocal isolation, performed poorly in this task. Table III summarizes these findings.

2) Processing Speed and Implementation Challenges: The processing times for a 3:09 minute song revealed an inverse relationship between quality and speed. Librosa's HPSS was the fastest at approximately 3 seconds, followed by Spleeter's 2-stem model at 8.88 seconds, 4-stem model at 11.32 seconds, and 5-stem model at 12.99 seconds. Demucs was significantly slower, requiring exactly 56.66 seconds to process the same audio file. These performance differences reflect the complexity of the underlying models and their computational requirements. Table IV provides a comparison of processing times and implementation considerations.

TABLE III
COMPARISON OF VOCAL ISOLATION QUALITY

Algorithm	Isolation Quality	Artifacts	Bleed-through
Demucs	Excellent	Minimal	Very Low
Spleeter (5-stem)	Good	Moderate	Low
Spleeter (4-stem)	Moderate	Moderate	Medium
Spleeter (2-stem)	Fair	High	High
Librosa HPSS	Poor	Very High	Very High

TABLE IV
PROCESSING SPEED AND IMPLEMENTATION CONSIDERATIONS

Algorithm	Processing Time (3:09 song)	Implementation Complexity	Architecture Compatibility
Librosa HPSS	∼3.00 seconds	Low	High
Spleeter (2-stem)	8.88 seconds	Medium	Medium
Spleeter (4-stem)	11.32 seconds	Medium	Medium
Spleeter (5-stem)	12.99 seconds	Medium	Medium
Demucs	56.66 seconds	Medium	Medium

Both Spleeter and Demucs presented implementation challenges on Apple Silicon architecture. Spleeter required specific versions of TensorFlow (2.12.0) and NumPy (1.23.5) to work properly, and it uses deprecated TensorFlow APIs (Estimator) that may cause future compatibility issues. Demucs implementation was more straightforward but required PyTorch with MPS acceleration for reasonable performance on Apple Silicon. Both algorithms required careful handling of audio format (samples vs. channels orientation) and sample rate consistency to function correctly.

3) Optimizations: For Spleeter, we implemented several optimizations based on best practices:

- Applied high-pass filtering (150 Hz) to improve vocal clarity
- Ensured proper audio format handling (samples, channels)
- Used local model caching to avoid repeated downloads

E. Discussion

Our evaluation revealed a clear trade-off between vocal isolation quality and processing speed. Demucs consistently produced superior vocal isolation but at the cost of significantly longer processing times. Spleeter offered faster processing but with more instrumental bleed-through, even when using the more complex 5-stem model.

For our karaoke training application, where accurate pitch tracking depends on clean vocal isolation, we determined that

the superior quality of Demucs outweighs the performance penalty. The cleaner vocal isolation enables more accurate pitch detection and provides a better reference for users learning to sing a melody.

VI. Conclusion and Future Work

Our research and development of the PitchTrack application has demonstrated that pYIN, combined with appropriate post-processing techniques, offers the best balance of accuracy and computational efficiency for vocal pitch tracking in a karaoke training application. The piano roll visualization system provides intuitive feedback to users, helping them improve their singing pitch accuracy.

For vocal line extraction, Demucs emerged as the superior solution despite its higher computational cost. The clean vocal isolation it provides is essential for accurate pitch detection and for providing users with high-quality reference audio for practice. While Spleeter offers faster processing, the quality difference justifies the performance trade-off for our application's needs.

Future work could explore:

- Real-time microphone input for live singing practice
- Integration of CREPE for offline analysis with higher accuracy
- Adaptive thresholding based on voice characteristics
- Performance metrics and scoring for user feedback
- Multi-voice pitch tracking for harmony training
- Optimizations to improve Demucs processing speed
- Hybrid approaches combining the speed of Spleeter with the quality of Demucs
- Progressive learning modes that gradually reduce the volume of the reference vocal

AI Disclosure

This research and paper were developed with the assistance of Amazon Q Developer CLI with Anthropic Claude Sonnet 3.7, an AI assistant. The AI was used both in conducting the research, analyzing code, and in preparing this article. The human author (David Rostcheck) provided direction, domain expertise, and final editorial oversight.

References

[1] A. de Cheveigné and H. Kawahara, "YIN, a fundamental frequency estimator for speech and music," The Journal of the Acoustical Society of America, vol. 111, no. 4, pp. 1917-1930, 2002.

[2] M. Mauch and S. Dixon, "pYIN: A fundamental frequency estimator using probabilistic threshold distributions," in Proc. IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), pp. 659-663, 2014.

[3] J. W. Kim, J. Salamon, P. Li, and J. P. Bello, "CREPE: A convolutional representation for pitch estimation," in Proc. IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), pp. 161-165, 2018.

[4] P. McLeod and G. Wyvill, "A smarter way to find pitch," in Proc. International Computer Music Conference (ICMC), 2005.

[5] D. Gerhard, "Pitch extraction and fundamental frequency: History and current techniques," Technical Report TR-CS 2003-06, Department of Computer Science, University of Regina, 2003.

[6] D. Fitzgerald, "Harmonic/percussive separation using median filtering," in Proc. Int. Conf. Digital Audio Effects (DAFx), Graz, Austria, Sep. 2010, pp. 246–253.

[7] F. Stöter, S. Uhlich, A. Liutkus, and Y. Mitsufuji, "Spleeter: A Fast and State-of-the-Art Music Source Separation Tool with Pre-trained Models," Journal of Open Source Software, vol. 5, no. 50, pp. 2154, 2020.

[8] A. Defossez, S. Watanabe, E. Vincent, and N. Usunier, "Demucs: Deep Extractor for Music Sources with Improved Source Separation," arXiv preprint arXiv:1909.01174, 2019.