I'm using a library called pitchy.js to do pitch detection for a guitar tuner app. It works pretty well, but I currently have issues sometimes with quiet notes not being detected well, and low notes generally have trouble being detected accurately. It's also somewhat sensitive to noise.

The whole of the pitchy.js library is included in one file pasted below. Are there any improvements I could make to how it works in order to improve my pitch detection?

```
// pitchy.js
import FFT from "fft.js";

/**
 * @typedef {Float32Array | Float64Array | number[]}
Buffer One of the supported
 * buffer types. Other numeric array types may not work
correctly.
 */

/**
 * A class that can perform autocorrelation on input
arrays of a given size.
 *
 * The class holds internal buffers so that no additional
allocations are
 * necessary while performing the operation.
 *
 * @template {Buffer} T the buffer type to use. While
inputs to the
 * autocorrelation process can be any array-like type, the
output buffer
 * (whether provided explicitly or using a fresh buffer) is
always of this type.
 */
export class Autocorrelator {
  /** @private @readonly @type {number} */
  _inputLength;
  /** @private @type {FFT} */
```

```
  _fft;
  /** @private @type {(size: number) => T} */
  _bufferSupplier;
  /** @private @type {T} */
  _paddedInputBuffer;
  /** @private @type {T} */
  _transformBuffer;
  /** @private @type {T} */
  _inverseBuffer;

  /**
   * A helper method to create an {@link Autocorrelator}
using
   * {@link Float32Array} buffers.
   *
   * @param inputLength {number} the input array length
to support
   * @returns {Autocorrelator<Float32Array>}
   */
  static forFloat32Array(inputLength) {
    return new Autocorrelator(
      inputLength,
      (length) => new Float32Array(length),
    );
  }

  /**
   * A helper method to create an {@link Autocorrelator}
using
   * {@link Float64Array} buffers.
   *
   * @param inputLength {number} the input array length
to support
   * @returns {Autocorrelator<Float64Array>}
   */
  static forFloat64Array(inputLength) {
    return new Autocorrelator(
      inputLength,
      (length) => new Float64Array(length),
    );
  }
```

```
  /**
   * A helper method to create an {@link Autocorrelator}
using number[]
   * buffers.
   *
   * @param inputLength {number} the input array length
to support
   * @returns {Autocorrelator<number[]>}
   */
  static forNumberArray(inputLength) {
    return new Autocorrelator(inputLength, (length) =>
Array(length));
  }

  /**
   * Constructs a new {@link Autocorrelator} able to
handle input arrays of the
   * given length.
   *
   * @param inputLength {number} the input array length
to support. This
   * Autocorrelator will only support operation on arrays of
this length.
   * @param bufferSupplier {(length: number) => T} the
function to use for
   * creating buffers, accepting the length of the buffer to
create and
   * returning a new buffer of that length. The values of
the returned buffer
   * need not be initialized in any particular way.
   */
  constructor(inputLength, bufferSupplier) {
    if (inputLength < 1) {
      throw new Error(Input length must be at least one);
    }
    this._inputLength = inputLength;
    // We need to double the input length to get correct
results, and the FFT
    // algorithm we use requires a length that's a power of
2
```

```
    this._fft = new FFT(ceilPow2(2 * inputLength));
    this._bufferSupplier = bufferSupplier;
    this._paddedInputBuffer =
this._bufferSupplier(this._fft.size);
    this._transformBuffer = this._bufferSupplier(2 *
this._fft.size);
    this._inverseBuffer = this._bufferSupplier(2 *
this._fft.size);
  }

  /**
   * Returns the supported input length.
   *
   * @returns {number} the supported input length
   */
  get inputLength() {
    return this._inputLength;
  }

  /**
   * Autocorrelates the given input data.
   *
   * @param input {ArrayLike<number>} the input data to
autocorrelate
   * @param output {T} the output buffer into which to
write the autocorrelated
   * data. If not provided, a new buffer will be created.
   * @returns {T} output
   */
  autocorrelate(input, output =
this._bufferSupplier(input.length)) {
    if (input.length !== this._inputLength) {
      throw new Error(
        Input must have length ${this._inputLength} but had
length ${input.length},
      );
    }
    // Step 0: pad the input array with zeros
    for (let i = 0; i < input.length; i++) {
      this._paddedInputBuffer[i] = input[i];
    }
```

```
    for (let i = input.length; i <
this._paddedInputBuffer.length; i++) {
      this._paddedInputBuffer[i] = 0;
    }

    // Step 1: get the DFT of the input array
    this._fft.realTransform(this._transformBuffer,
this._paddedInputBuffer);
    // We need to fill in the right half of the array too
    this._fft.completeSpectrum(this._transformBuffer);
    // Step 2: multiply each entry by its conjugate
    const tb = this._transformBuffer;
    for (let i = 0; i < tb.length; i += 2) {
      tb[i] = tb[i] * tb[i] + tb[i + 1] * tb[i + 1];
      tb[i + 1] = 0;
    }
    // Step 3: perform the inverse transform
    this._fft.inverseTransform(this._inverseBuffer,
this._transformBuffer);

    // This last result (the inverse transform) contains the
autocorrelation
    // data, which is completely real
    for (let i = 0; i < input.length; i++) {
      output[i] = this._inverseBuffer[2 * i];
    }
    return output;
  }
}

/**
 * Returns an array of all the key maximum positions in
the given input array.
 *
 * In McLeod's paper, a key maximum is the highest
maximum between a positively
 * sloped zero crossing and a negatively sloped one.
 *
 * TODO: it may be more efficient not to construct a new
output array each time,
 * but that would also make the code more complicated
```

```
(more so than the changes
 * that were needed to remove the other allocations).
 *
 * @param input {ArrayLike<number>}
 * @returns {number[]}
 */
function getKeyMaximumIndices(input) {
  // The indices of the key maxima
  /** @type {number[]} */ const keyIndices = [];
  // Whether the last zero crossing found was positively
sloped; equivalently,
  // whether we're looking for a key maximum
  let lookingForMaximum = false;
  // The largest local maximum found so far
  let max = -Infinity;
  // The index of the largest local maximum so far
  let maxIndex = -1;

  for (let i = 1; i < input.length - 1; i++) {
    if (input[i - 1] <= 0 && input[i] > 0) {
      // Positively sloped zero crossing
      lookingForMaximum = true;
      maxIndex = i;
      max = input[i];
    } else if (input[i - 1] > 0 && input[i] <= 0) {
      // Negatively sloped zero crossing
      lookingForMaximum = false;
      if (maxIndex !== -1) {
        keyIndices.push(maxIndex);
      }
    } else if (lookingForMaximum && input[i] > max) {
      max = input[i];
      maxIndex = i;
    }
  }

  return keyIndices;
}

/**
 * Refines the chosen key maximum index chosen from
```

```
the given data by
 * interpolating a parabola using the key maximum index
and its two neighbors
 * and finding the position of that parabola's maximum
value.
 *
 * This is described in section 5 of the MPM paper as a
way to refine the
 * position of the maximum.
 *
 * @param index {number} the chosen key maximum
index. This must be between 1
 * and data.length - 2, inclusive, since it and its two
neighbors need to be
 * valid indexes of data.
 * @param data {ArrayLike<number>} the input array
from which index was chosen
 * @returns {[number, number]} a pair consisting of the
refined key maximum index and the
 * interpolated value of data at that index (the latter of
which is used as a
 * measure of clarity)
 */
function refineResultIndex(index, data) {
  const [x0, x1, x2] = [index - 1, index, index + 1];
  const [y0, y1, y2] = [data[x0], data[x1], data[x2]];

  // The parabola going through the three data points can
be written as
  // y = y0(x - x1)(x - x2)/(x0 - x1)(x0 - x2)
  //   + y1(x - x0)(x - x2)/(x1 - x0)(x1 - x2)
  //   + y2(x - x0)(x - x1)/(x2 - x0)(x2 - x1)
  // Given the definitions of x0, x1, and x2, we can simplify
the denominators:
  // y = y0(x - x1)(x - x2)/2
  //   - y1(x - x0)(x - x2)
  //   + y2(x - x0)(x - x1)/2
  // We can expand this out and get the coefficients in
standard form:
  // a = y0/2 - y1 + y2/2
  // b = -(y0/2)(x1 + x2) + y1(x0 + x2) - (y2/2)(x0 + x1)
```

```
  // c = y0x1x2/2 - y1x0x2 + y2x0x1/2
  // The index of the maximum is -b / 2a (by solving for x
where the derivative
  // is 0).

  const a = y0 / 2 - y1 + y2 / 2;
  const b = -(y0 / 2) * (x1 + x2) + y1 * (x0 + x2) - (y2 / 2) *
(x0 + x1);
  const c = (y0 * x1 * x2) / 2 - y1 * x0 * x2 + (y2 * x0 * x1) /
2;

  const xMax = -b / (2 * a);
  const yMax = a * xMax * xMax + b * xMax + c;
  return [xMax, yMax];
}

/**
 * A class that can detect the pitch of a note from a time-
domain input array.
 *
 * This class uses the McLeod pitch method (MPM) to
detect pitches. MPM is
 * described in the paper 'A Smarter Way to Find Pitch' by
Philip McLeod and
 * Geoff Wyvill
 *
(http://miracle.otago.ac.nz/tartini/papers/A_Smarter_Way
_to_Find_Pitch.pdf).
 *
 * The class holds internal buffers so that a minimal
number of additional
 * allocations are necessary while performing the
operation.
 *
 * @template {Buffer} T the buffer type to use internally.
Inputs to the
 * pitch-detection process can be any numeric array
type.
 */
export class PitchDetector {
  /** @private @type {Autocorrelator<T>} */
```

```
  _autocorrelator;
  /** @private @type {T} */
  _nsdfBuffer;
  /** @private @type {number} */
  _clarityThreshold = 0.9;
  /** @private @type {number} */
  _minVolumeAbsolute = 0.0;
  /** @private @type {number} */
  _maxInputAmplitude = 1.0;

  /**
   * A helper method to create an {@link PitchDetector}
using {@link Float32Array} buffers.
   *
   * @param inputLength {number} the input array length
to support
   * @returns {PitchDetector<Float32Array>}
   */
  static forFloat32Array(inputLength) {
    return new PitchDetector(inputLength, (length) => new
Float32Array(length));
  }

  /**
   * A helper method to create an {@link PitchDetector}
using {@link Float64Array} buffers.
   *
   * @param inputLength {number} the input array length
to support
   * @returns {PitchDetector<Float64Array>}
   */
  static forFloat64Array(inputLength) {
    return new PitchDetector(inputLength, (length) => new
Float64Array(length));
  }

  /**
   * A helper method to create an {@link PitchDetector}
using number[] buffers.
   *
   * @param inputLength {number} the input array length
```

```
to support
   * @returns {PitchDetector<number[]>}
   */
  static forNumberArray(inputLength) {
    return new PitchDetector(inputLength, (length) =>
Array(length));
  }

  /**
   * Constructs a new {@link PitchDetector} able to
handle input arrays of the
   * given length.
   *
   * @param inputLength {number} the input array length
to support. This
   * PitchDetector will only support operation on arrays of
this length.
   * @param bufferSupplier {(inputLength: number) => T}
the function to use for
   * creating buffers, accepting the length of the buffer to
create and
   * returning a new buffer of that length. The values of
the returned buffer
   * need not be initialized in any particular way.
   */
  constructor(inputLength, bufferSupplier) {
    this._autocorrelator = new Autocorrelator(inputLength,
bufferSupplier);
    this._nsdfBuffer = bufferSupplier(inputLength);
  }

  /**
   * Returns the supported input length.
   *
   * @returns {number} the supported input length
   */
  get inputLength() {
    return this._autocorrelator.inputLength;
  }

  /**
```

```
  * Sets the clarity threshold used when identifying the
correct pitch (the constant
  * k from the MPM paper). The value must be between 0
(exclusive) and 1
  * (inclusive), with the most suitable range being
between 0.8 and 1.
  *
  * @param threshold {number} the clarity threshold
  */
 set clarityThreshold(threshold) {
   if (!Number.isFinite(threshold) || threshold <= 0 ||
threshold > 1) {
     throw new Error("clarityThreshold must be a number
in the range (0, 1]");
   }
   this._clarityThreshold = threshold;
 }

 /**
  * Sets the minimum detectable volume, as an absolute
number between 0 and
  * maxInputAmplitude, inclusive, to consider in a sample
when detecting the
  * pitch. If a sample fails to meet this minimum volume,
findPitch will
  * return a clarity of 0.
  *
  * Volume is calculated as the RMS (root mean square)
of the input samples.
  *
  * @param volume {number} the minimum volume as an
absolute amplitude value
  */
 set minVolumeAbsolute(volume) {
   if (
     !Number.isFinite(volume) ||
     volume < 0 ||
     volume > this._maxInputAmplitude
   ) {
     throw new Error(
       minVolumeAbsolute must be a number in the range
```

```
[0, ${this._maxInputAmplitude}],
      );
    }
    this._minVolumeAbsolute = volume;
  }

  /**
   * Sets the minimum volume using a decibel
measurement. Must be less than or
   * equal to 0: 0 indicates the loudest possible sound
(see
   * maxInputAmplitude), -10 is a sound with a tenth of
the volume of the
   * loudest possible sound, etc.
   *
   * Volume is calculated as the RMS (root mean square)
of the input samples.
   *
   * @param db {number} the minimum volume in
decibels, with 0 being the loudest
   * sound
   */
  set minVolumeDecibels(db) {
    if (!Number.isFinite(db) || db > 0) {
      throw new Error("minVolumeDecibels must be a
number <= 0");
    }
    this._minVolumeAbsolute = this._maxInputAmplitude *
10 ** (db / 10);
  }

  /**
   * Sets the maximum amplitude of an input reading.
Must be greater than 0.
   *
   * @param amplitude {number} the maximum amplitude
(absolute value) of an input reading
   */
  set maxInputAmplitude(amplitude) {
    if (!Number.isFinite(amplitude) || amplitude <= 0) {
      throw new Error("maxInputAmplitude must be a
```

```
number > 0");
    }
    this._maxInputAmplitude = amplitude;
  }

  /**
   * Returns the pitch detected using McLeod Pitch
Method (MPM) along with a
   * measure of its clarity.
   *
   * The clarity is a value between 0 and 1 (potentially
inclusive) that
   * represents how "clear" the pitch was. A clarity value
of 1 indicates that
   * the pitch was very distinct, while lower clarity values
indicate less
   * definite pitches.
   *
   * @param input {ArrayLike<number>} the time-domain
input data
   * @param sampleRate {number} the sample rate at
which the input data was
   * collected
   * @returns {[number, number]} the detected pitch, in
Hz, followed by the
   * clarity. If a pitch cannot be determined from the input,
such as if the
   * volume is too low (see minVolumeAbsolute and
minVolumeDecibels), this
   * will be [0, 0].
   */
  findPitch(input, sampleRate) {
    // If the highest key maximum is less than the minimum
volume, we don't need
    // to bother detecting the pitch, as the sample is too
quiet.
    if (this._belowMinimumVolume(input)) return [0, 0];
    this._nsdf(input);
    const keyMaximumIndices =
getKeyMaximumIndices(this._nsdfBuffer);
    if (keyMaximumIndices.length === 0) {
```

```
      // No key maxima means that we either don't have
enough data to analyze or
      // that the data was flawed (such as an input array of
zeroes)
      return [0, 0];
    }
    // The highest key maximum
    const nMax = Math.max(...keyMaximumIndices.map((i)
=> this._nsdfBuffer[i]));
    // Following the paper, we return the pitch
corresponding to the first key
    // maximum higher than K * nMax. This is guaranteed
not to be undefined, since
    // we know of at least one key maximum satisfying this
condition (whichever
    // key maximum gave us nMax).
    const resultIndex = keyMaximumIndices.find(
      (i) => this._nsdfBuffer[i] >= this._clarityThreshold *
nMax,
    );
    const [refinedResultIndex, clarity] = refineResultIndex(
      // @ts-expect-error resultIndex is guaranteed to be
defined
      resultIndex,
      this._nsdfBuffer,
    );

    // Due to floating point errors, the clarity may
occasionally come out to be
    // slightly over 1.0. We can avoid incorrect results by
clamping the value.
    return [sampleRate / refinedResultIndex,
Math.min(clarity, 1.0)];
  }

  /**
   * Returns whether the input audio data is below the
minimum volume allowed by
   * the pitch detector.
   *
   * @private
```

```
   * @param input {ArrayLike<number>}
   * @returns {boolean}
   */
  _belowMinimumVolume(input) {
    if (this._minVolumeAbsolute === 0) return false;
    let squareSum = 0;
    for (let i = 0; i < input.length; i++) {
      squareSum += input[i] ** 2;
    }
    return Math.sqrt(squareSum / input.length) <
this._minVolumeAbsolute;
  }

  /**
   * Computes the NSDF of the input and stores it in the
internal buffer. This
   * is equation (9) in the McLeod pitch method paper.
   *
   * @private
   * @param input {ArrayLike<number>}
   */
  _nsdf(input) {
    // The function r'(tau) is the autocorrelation
    this._autocorrelator.autocorrelate(input,
this._nsdfBuffer);
    // The function m'(tau) (defined in equation (6)) can be
computed starting
    // with m'(0), which is equal to 2r'(0), and then
iteratively modified to
    // get m'(1), m'(2), etc. For example, to get m'(1), we
take m'(0) and
    // subtract x_0^2 and x_{W-1}^2. Then, to get m'(2),
we take m'(1) and
    // subtract x_1^2 and x_{W-2}^2, and further values
are similar (see the
    // note at the end of section 6 in the MPM paper).
    //
    // The resulting array values are 2 * r'(tau) / m'(tau). We
use m below as
    // the incremental value of m'.
    let m = 2 * this._nsdfBuffer[0];
```

```
    /** @type {number} */ let i;
    // As pointed out by issuefiler on GitHub, we can take
advantage of the fact
    // that m will never increase to avoid division by zero by
ending this loop
    // once m === 0. The rest of the array values after m
becomes 0 will just be
    // set to 0 themselves. We actually check for m > 0
rather than m === 0
    // because there may be small floating-point errors that
cause m to become
    // negative rather than exactly 0.
    for (i = 0; i < this._nsdfBuffer.length && m > 0; i++) {
      this._nsdfBuffer[i] = (2 * this._nsdfBuffer[i]) / m;
      m -= input[i] ** 2 + input[input.length - i - 1] ** 2;
    }
    // If there are any array values remaining, it means m
=== 0 for those
    // values of tau, so we can just set them to 0
    for (; i < this._nsdfBuffer.length; i++) {
      this._nsdfBuffer[i] = 0;
    }
  }
}

/**
 * Rounds up the input to the next power of 2.
 *
 * @param {number} v
 * @returns {number} the next power of 2 at least as large
as v
 */
function ceilPow2(v) {
  //
https://graphics.stanford.edu/~seander/bithacks.html#Ro
undUpPowerOf2
  v--;
  v |= v >> 1;
  v |= v >> 2;
  v |= v >> 4;
  v |= v >> 8;
```

```
  v |= v >> 16;
  v++;
  return v;
}
```

Thanks for sharing the full source — this is a great implementation of the McLeod Pitch Method (MPM). From your description, you're dealing with **three key problems**:

1. **Quiet notes not being detected well**
2. **Low notes being detected inaccurately**
3. **Sensitivity to noise**

Let's go through each problem and suggest **targeted improvements** to the existing `pitchy.js` code and your usage of it.

✅ Problem 1: Quiet Notes Not Being Detected

🔍 Cause:

- Detection skips analysis if RMS amplitude is below `_minVolumeAbsolute`.
- You might be using the default, which is `0.0`, but quiet notes may still be indistinguishable from noise unless RMS is considered more intelligently.

✅ Fixes:

1. Use a Lower but Non-Zero `minVolumeDecibels`

This reduces false positives from noise without cutting off quiet guitar notes:

```
pitchDetector.minVolumeDecibels = -40; // Or even -50 depending on your mic setup
```

- This corresponds to a minimum RMS of ~0.01–0.003.
- You can tweak this value based on your noise floor and how quiet your source is.

2. Normalize the Input

Normalize your incoming audio to a fixed range before passing to `findPitch`:

```
const normalize = (buffer) => { const max = Math.max(...buffer.map(Math.abs)); if (max === 0) return buffer; return buffer.map((val) => val / max); };
```

> Be careful: normalizing amplifies noise. Only normalize if you're sure there's a note present (e.g., user plucked a string).

✅ Problem 2: Low Notes Are Inaccurate

🔍 Cause:

- MPM requires enough cycles of the waveform to detect frequency accurately.
- Low notes (e.g., low E at 82.4 Hz) have longer wavelengths.

✅ Fixes:

1. Increase Input Buffer Size

Current buffer may be too short to contain full periods of low notes.

> For 82 Hz at 44.1kHz sample rate, one full period is ~537 samples.

Use at least `2048` or even `4096` samples for stable results:

```
const pitchDetector = PitchDetector.forFloat32Array(4096);
```

- Make sure your audio analysis window matches this buffer length.

2. Reduce FFT Size Floor

Check the code uses `ceilPow2(2 * inputLength)`. That's great — just ensure that:

- You don’t drop below 2048 for low notes.
- Your app has the audio analysis window properly sized.

3. Lower Clarity Threshold

Lower notes often yield weaker NSDF peaks (less clarity), so allow for a looser threshold:

```
pitchDetector.clarityThreshold = 0.8; // instead of 0.9
```

✅ Problem 3: Noise Sensitivity

🔍 Cause:

- Autocorrelation and NSDF can pick up harmonic noise or room resonance.

- Especially sensitive when input amplitude is low (see Problem 1).

✅ Fixes:

1. Preprocess with a Bandpass Filter

Apply a filter to limit the signal to guitar frequency range (~80Hz–1200Hz):

```
const bandpass = audioContext.createBiquadFilter(); bandpass.type = 'bandpass'; bandpass.frequency.value = 440; bandpass.Q.value = 1.5; // Q controls width: this gives ~150Hz–1300Hz range
```

- Insert this before your pitch detection.

2. Apply a Window Function

A window (like Hann or Hamming) can reduce spectral leakage that contributes to noise artifacts:

```
const applyHannWindow = (buffer) => { const N = buffer.length; for (let i = 0; i < N; i++) { buffer[i] *= 0.5 * (1 - Math.cos((2 * Math.PI * i) / (N - 1))); } };
```

Call this just before passing the buffer into `findPitch()`.

🚀 Bonus Improvements

1. Track SNR or Use Running RMS

Use a running average of RMS to compare against current RMS and reject input if it spikes — often a sign of noise.

2. Check First Peak Position

If `refinedResultIndex` is < 2 or very close to 0, it's likely a false high frequency due to noise.

```
if (refinedResultIndex < 5) return [0, 0]; // ignore noisy detections
```

✅ Summary of Fixes You Can Make Right Away

1. **In your app:**

```
pitchDetector.minVolumeDecibels = -40;
pitchDetector.clarityThreshold = 0.8;
```

2. **Increase input buffer to 4096 samples** for low notes.
3. **Normalize input (if safe)** before detection.
4. **Apply a Hann window and bandpass filter** before sending to pitch detection.

If you'd like, I can help you implement these pre-processing steps in your React/Tone.js audio code. Would that be useful?

Thanks! Can you explain how the Hann window works in a bit more detail? I'm not really familiar with it

Absolutely! The **Hann window** (sometimes called the **Hanning window**) is a mathematical function used to *smooth* the edges of a block of audio data before analyzing it, especially in frequency or pitch detection tasks like yours.

Let's break down what it does, why it's used, and how it works — step by step:

🎯 Why Use a Window Function at All?

When you're analyzing audio (e.g., for pitch detection), you're usually working with a **short chunk** of the sound — a few thousand samples (like 2048 or 4096).

This chunk might not contain an *exact number of periods* of the waveform. That means when you analyze it (e.g., using autocorrelation or FFT), you can get **artifacts**, also called **spectral leakage** — weird frequency content that isn't really there.

Imagine cutting out a piece of a sine wave mid-oscillation. The ends look "chopped off", like this:

```
|--- WAVES ---|
```

```
~~~~|
```

This sudden cut introduces *artificial high frequencies*. That's bad for accuracy.

✅ What the Hann Window Does

A **Hann window** tapers the signal at both ends so that it *smoothly fades in and out*. This removes the sharp transitions.

Mathematically, the Hann window function looks like this:

$$w(n) = 0.5\left(1 - \cos\left(\frac{2\pi n}{N-1}\right)\right)$$

Where:

- `n` is the sample index (from 0 to N-1),
- `N` is the total number of samples in your buffer.

This produces a smooth bell-shaped curve that looks like this:

```
Amplitude
|
|       ***
|     *   *
|    *     *
|   *       *
|  *         *
| *           *
+------------------> Time
```

When you multiply your audio buffer by this window, it fades out the start and end to near-zero.

🔧 Applying It in Code

Here's a quick JavaScript function to apply a Hann window to your buffer:

```
function applyHannWindow(buffer) { const N = buffer.length; for (let n
= 0; n < N; n++) { const multiplier = 0.5 * (1 - Math.cos((2 * Math.PI *
```

```
n) / (N - 1))); buffer[n] *= multiplier; } }
```

Just call this function before passing your audio to the pitch detector:

```
applyHannWindow(myBuffer); const [pitch, clarity] =
pitchDetector.findPitch(myBuffer, sampleRate);
```

📊 What It Helps With in Pitch Detection

- **Reduces false harmonics** or "ghost notes" caused by sharp edges
- **Improves accuracy** of autocorrelation or FFT-based detection
- **Makes clarity scores more reliable**, especially for quiet or low notes

---

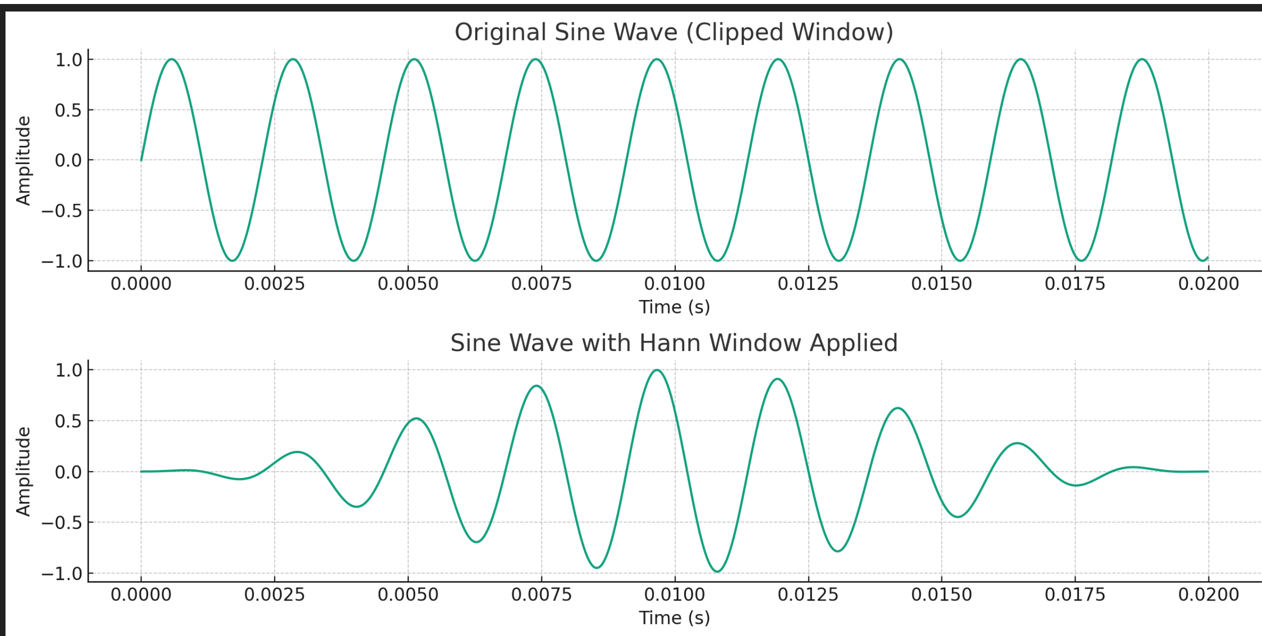
🧠 TL;DR

Without Hann Window	With Hann Window
Sudden edges	Smooth edges
Spectral leakage	Cleaner frequency content
Poor pitch accuracy	Improved pitch accuracy

It's a **simple, low-cost improvement** that makes your pitch detection more robust, especially in noisy or low-frequency situations.

Let me know if you want a visualization of this applied to real audio — I can simulate it!

Sure, can I see a visualization?

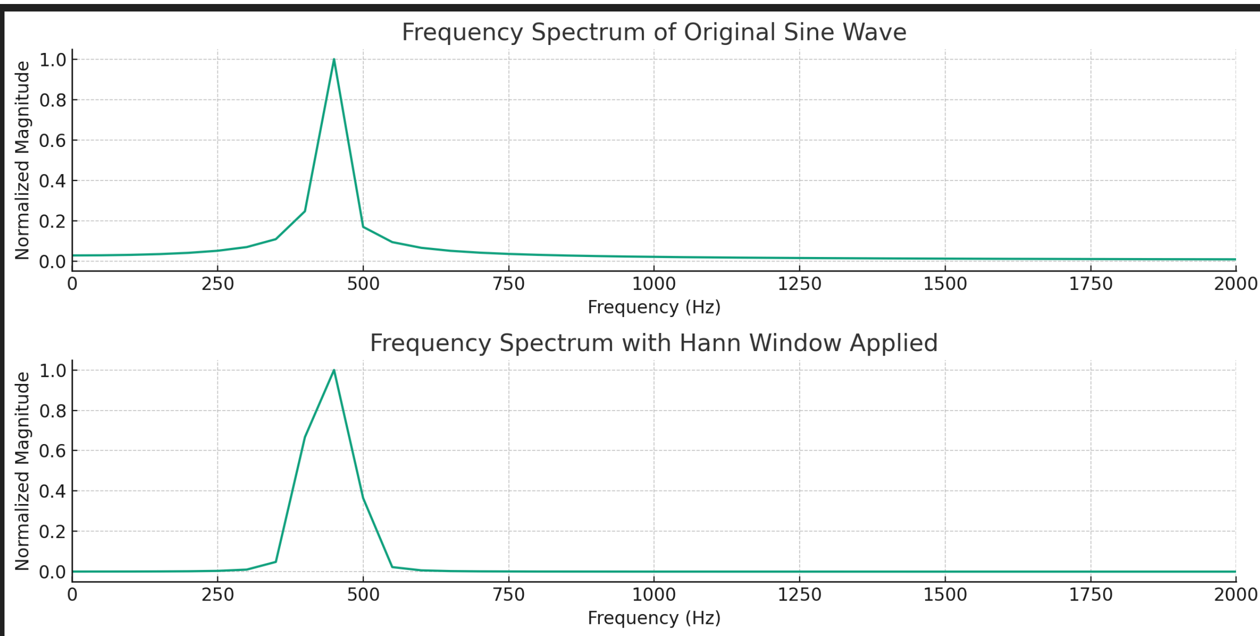


Here you go! The top plot shows the **original sine wave** without any windowing — notice how it starts and ends abruptly, which can introduce artifacts in pitch detection.

The bottom plot shows the **same sine wave with a Hann window applied** — the smooth fade-in and fade-out helps eliminate those abrupt transitions, leading to cleaner and more accurate frequency analysis.

This smoothing is why the Hann window is so helpful in pitch detection, especially for short chunks of audio. Let me know if you'd like to see how this affects the frequency spectrum too!

Yeah let's see the effect on the frequency spectrum too



Here's the comparison:

- **Top plot (original sine wave)**: You see a strong peak at 440 Hz, but also a lot of **extra side lobes** — those are artifacts from the sharp edges of the waveform segment.
- **Bottom plot (Hann windowed)**: The peak at 440 Hz is still there, but the **side lobes are dramatically reduced**. This gives a much cleaner spectrum and helps avoid false detections or noise-induced errors in pitch tracking.

This is why applying a window like Hann is a common best practice in frequency- and autocorrelation-based pitch detection methods. Let me know if you'd like to integrate this directly into your app's processing pipeline.

I don't really see any side lobes in the example? Is it just because it's a pure sine

Great observation — you're absolutely right to question that.

Yes, **because it's a pure sine wave**, and because the duration is very short (20 ms), the side lobes in the unwindowed version are relatively small and **not visually dramatic** — especially since we're looking at the **magnitude spectrum** on a **linear scale**.

🔎 Let's unpack this a bit:

1. Why side lobes happen

When you cut off a wave mid-cycle, it's like multiplying it by a rectangular window. This "multiplication in time" causes **convolution in frequency** with a sinc function (the Fourier transform of a rectangular pulse), which has ripples — those ripples are what we call **spectral leakage** or **side lobes**.

2. Why they're hard to see here

- A pure sine has **no harmonics** — it's just a spike at 440 Hz.
- The FFT resolution is **limited by the short time window** (20 ms = ~2048 samples at 44.1 kHz), so frequency bins are wide (~22 Hz).
- We're showing **linear magnitude**, which makes small ripples near the main peak look tiny.

📊 Let's make it clearer

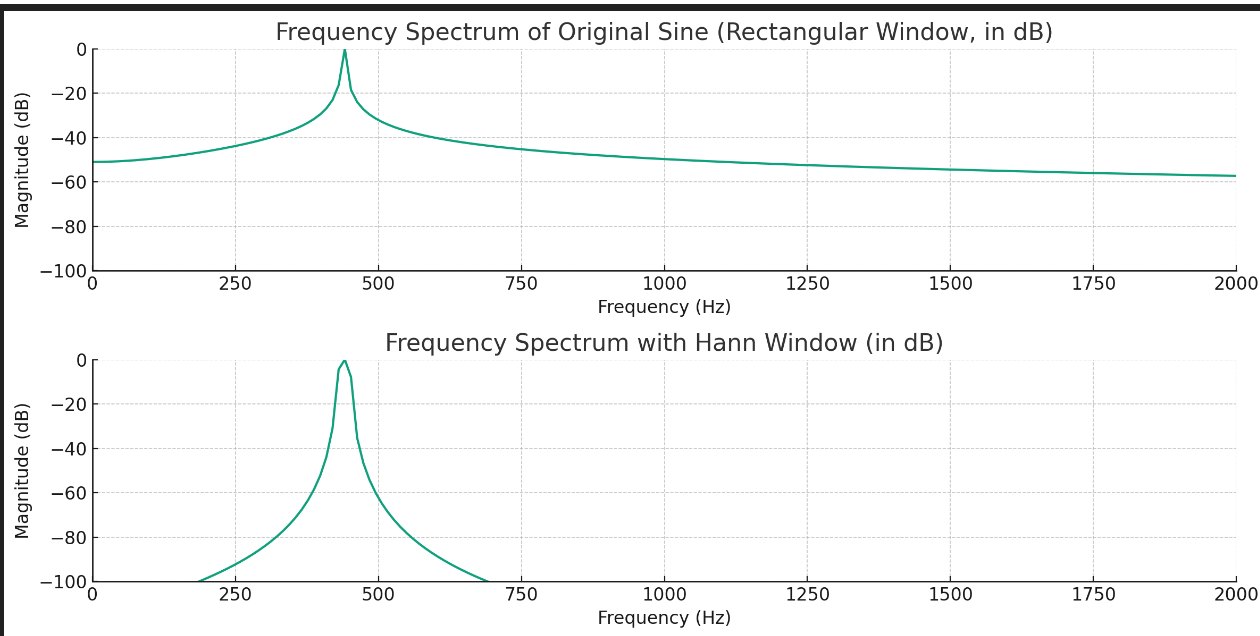
Let's redo the spectrum plot with:

- **Log scale for magnitude (dB)** — to exaggerate the small details
- A **longer buffer (~4096 samples / ~93ms)** so we get finer frequency resolution
- The same sine wave, just extended in time

That'll make the side lobes pop out more clearly.

Shall I go ahead and plot that?

Yeah can you plot that!



Now you can really see the difference!

- **Top plot (rectangular window)**:
  - The main peak is at 440 Hz.
  - The **side lobes** are clearly visible — those smaller peaks spread out across the spectrum, reaching only about –13 dB below the main peak.
  - This is **spectral leakage** in action: energy from the main tone is "leaking" into nearby frequencies.
- **Bottom plot (Hann window)**:
  - The main peak is still at 440 Hz.
  - But the side lobes are drastically reduced — **down more than –30 dB**, and they taper off quickly.
  - This results in a much cleaner frequency representation, especially important when you're trying to detect **just one pitch** or **multiple pitches in complex signals**.

So while the Hann window slightly **broadens** the main peak (lower frequency resolution), it gives you **way less leakage**, which is a good tradeoff for most pitch detection tasks.

Let me know if you want to try this on noisy or multi-tone signals next!

Thanks! Going back to the bandpass filter now, how would I be able to actually connect that audio node to the processing pipeline?

Great! If you're adding a **bandpass filter** to your pitch detection pipeline, you'll want to insert it **in the audio graph** *before* your analyzer node (or pitch detector).

Let's walk through how to do this with the Web Audio API — assuming you already have something like this:

```
const audioContext = new AudioContext(); const analyserNode =
audioContext.createAnalyser(); const source =
audioContext.createMediaStreamSource(stream);
```

✅ Add a Bandpass Filter Node

Here's how you can create and connect a bandpass filter between the `source` and your processing node:

```
const bandpassFilter = audioContext.createBiquadFilter();
bandpassFilter.type = "bandpass"; bandpassFilter.frequency.value = 440;
// Center frequency (you can tweak this) bandpassFilter.Q.value = 5; //
Quality factor (higher = narrower band)
```

🔗 Connect the filter into the pipeline

Now wire everything together:

```
source.connect(bandpassFilter); bandpassFilter.connect(analyserNode);
// or your pitch processor node
```

So your signal flow becomes:

```
Mic input → source → bandpassFilter → analyser/pitchDetector
```

🎯 Optional: Dynamically tune the filter

If your pitch detection app tries to focus on different notes (e.g. during tuning), you can dynamically adjust the filter:

```
bandpassFilter.frequency.setValueAtTime(targetFreq,
audioContext.currentTime);
```

This lets you **focus detection** around a user-selected note (like A4 = 440 Hz), improving sensitivity and reducing false detection from harmonics or background noise.

🧪 Testing Tip

To confirm it's working:

- Use a **spectrum analyzer** or plot FFT magnitudes before and after the filter.
- You should see a drop-off in energy outside the target frequency band.

Would you like help integrating this into your existing `initAudio` function or `SynthContext` setup?