Digital Signal Processing Lab

Take Home Exam - Q1

Saad Zubairi
shz2020

November 8, 2025

Solution

Premise

The basic premise of this question was to create a program that lets us shift the frequency of an input signal by a scaling factor while also preserving the harmonic structure of the signal, and letting the user shift the frequency through a GUI based slider that repeats the audio in real time. The suggested way of approaching this problem is by taking FFT of the input, shifting the frequency spectrim by the scaling factor, and then inversing the FFT.

a) Approach

The approach was simple. Taking hints from `Demo 13`, `Demo 14`, and `Demo 56`, I implemented frequency scaling using FFT processing with overlapping block processing. To ensure smooth transitions between block boundaries and avoid artifacts, the signal is broken down into overlapping blocks with a 50% overlap factor, obtained through trial and error. Each block is processed independently using FFT, frequency scaling via interpolation, and IFFT. The processed blocks are then combined using the overlap-add method.

Detailed steps to solution

- **Blocks Forming:** Before processing, the signal is divided into overlapping blocks using the function `frames_to_process_with_hops()`. The overlap factor is set to 0.5.

```python
def frames_to_process_with_hops(all_frames, block_size, overlap_factor):
    frames = []
    hop_count = math.floor(block_size * (1 - overlap_factor))
    idx_i = 0
    idx_l = block_size
    while idx_l < len(all_frames) - 1:
        frames.append(all_frames[idx_i:idx_l])
        idx_i += hop_count
        idx_l = idx_i + block_size
    return frames

```

Snippet 1: Block formation with overlap

- **FFT and Frequency Scaling:** Each block is processed using the `process_block_fft_scaling()` function. For each block, this function:

  1. Computes the real FFT of the input block.
  2. Scales the frequency spectrum by the factor α using linear interpolation
  3. Computes the inverse real FFT to obtain the time-domain output

  The frequency scaling is achieved by mapping each bin k to a destination bin k/α. Of course since k/α may rarely be an integer, linear interpolation is used between bins:

$$\mathbf{y}[k] = (1-t)\cdot \mathbf{x}[\lfloor k/\alpha \rfloor] + t \cdot \mathbf{x}[\lfloor k/\alpha \rfloor + 1]$$

  where $t = k/\alpha - \lfloor k/\alpha \rfloor$ is of course the fractional part.

```python
def process_block_fft_scaling(input_block, alpha):
    X = np.fft.rfft(input_block)
    Y = np.zeros_like(X)
    for src_ind in range(X.size):
        dst_ind = src_ind / alpha
        if dst_ind < X.size - 1:
            i0 = int(np.floor(dst_ind))
            i1 = i0 + 1
            t = dst_ind - i0
            Y[src_ind] = (1 - t) * X[i0] + t * X[i1]
    y = np.fft.irfft(Y)
    return y

```

Snippet 2: FFT-based frequency scaling with interpolation

- **Overlap-Add Processing:** For smooth combinig of blocks, the tail of the previous block (which essentially is the overlapping portion) is stored and added to the beginning of the current block with a 0.5 weighting (a sort of averaging) to prevent abrupt artifacts:

```python
# overlap and add
output_block[:len(prev_tail)] = 0.5 * (output_block[:len(prev_tail)] +
    prev_tail)

```

Snippet 3: Overlap-add processing

- **Real-time Processing Loop:** The main processing loop continuously:
  1. Reads the current scaling factor α from the slider value
  2. Retrieves the next input block from the frames array
  3. Processes the block with the current scaling factor
  4. Applies overlap-add with the previous block's tail
  5. Clips and writes the output chunk to the audio stream
  6. Stores the tail portion for the next iteration
  7. Cycles through frames in a circular manner for continuous playback using the modulo operator

```python
prev_tail = np.zeros(BLOCKLEN - Hop)
frames = frames_to_process_with_hops(all_samples, BLOCKLEN, OVERLAP_FACTOR)
frame_idx = 0
while CONTINUE:
    root.update()

    # get slider value in real time
    alpha_from_slider = alpha.get()

    # get next input frame
    input_block = frames[frame_idx]

    # process the scaled
    output_block = process_block_fft_scaling(input_block, alpha_from_slider)

    # overlap and add
    output_block[:len(prev_tail)] = 0.5 * (output_block[:len(prev_tail)] +
    prev_tail)

    # clipping
    output_chunk = np.clip(output_block[:Hop], -MAXVALUE, MAXVALUE)
    output_chunk = np.around(output_chunk).astype(np.int16)
    stream.write(output_chunk.tobytes())

    prev_tail = output_block[Hop:]
    # increment to the next frame (circular)
    frame_idx = (frame_idx + 1) % len(frames)

```

Snippet 4: Main real-time processing loop

- **Parameters:** For the solution, I implmented the following main parameters after a bit of trial and error:
  - Block size: 1024 samples
  - Overlap factor: 0.5
  - Scaling factor range: 0.5 to 2.0

Addendum

The implementation successfully achieves real-time frequency scaling while preserving the harmonic structure of the input signal. The key design choices include:

- **50% Overlap:** The 50% overlap factor was chosen through experimentation to balance between smooth transitions and computational efficiency. This overlap ensures that block boundary artifacts are minimized.
- **Linear Interpolation:** Linear interpolation in the frequency domain provides a good balance between quality and computational cost. More sophisticated interpolation methods (e.g., cubic) could be used for potentially better quality at the cost of increased computation.
- **Circular Frame Buffer:** The frames are processed in a circular manner, allowing for continuous playback of the audio file while maintaining real-time responsiveness to slider changes.
- **Clipping Protection:** The output is clipped to prevent overflow, which is essential for maintaining audio quality and preventing distortion when the scaling factor causes amplitude changes.

The implementation demonstrates effective use of overlapping block processing with FFT-based frequency domain manipulation for real-time audio processing applications.