Optimizing the Constant-Q Transform in Octave

Hans Fugal
New Mexico State University
P.O. Box 30001, MSC CS
Las Cruces, New Mexico 88003
United States of America
hfugal@cs.nmsu.edu

Abstract

The constant-Q transform calculates the log-frequency spectrum and is therefore better suited to many computer music tasks than the discrete Fourier transform, which computes the linear-frequency spectrum. The direct calculation of the constant-Q transform is computationally expensive, and so the Brown-Puckette efficient constant-Q transform algorithm is often used. The original constant-Q transform and the Brown-Puckette constant-Q transform algorithm were implemented and benchmarked in Octave. The Brown-Puckette algorithm was faster than the direct algorithm with a high minimum frequency, but it was not faster for a low minimum frequency. Additional vectorization of the algorithm is slower under most circumstances, and using sparse matrices gives marginal speedup.

Keywords

Octave, spectrum, spectrogram, constant-Q, optimization

1 Introduction

Due to the log-frequency nature of the musical scale and human hearing, and the linear-frequency nature of the discrete Fourier transform (DFT), the DFT is sometimes inconvenient to work with. For example, one semitone in a low register spans a few Hz, but in a high register one semitone spans over 100 Hz. When working with pitches, the log-frequency scale is desirable. Harmonic partials are integer multiples of the fundamental frequency. The spacing between harmonics is therefore not uniform in the linear-frequency scale, which is inconvenient for algorithms which try to recognize the harmonic signature of instruments. The constant-Q transform uses a varying time window to calculate the log-frequency spectrum, and is better suited for these and many other problems in computer music. Figures 1 and 2 visually demonstrate the advantages of the constant-Q transform.

The constant-Q filter is so named because the ratio of frequency to bandwidth (Q) is held constant for each frequency bin. The Q of a filter is defined as $f/\Delta f$, where f is the center frequency and Δf is the bandwidth. The bandwidth is inversely proportional to the number of samples

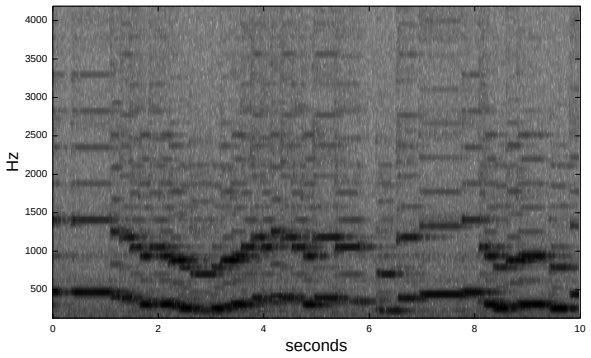

Figure 1: A portion of a normal (linear-frequency) spectrogram of a monophonic melody recorded on a pipe organ. Observe the harmonic is not equidistant from the fundamental frequency because of the log-frequency nature of harmonics.

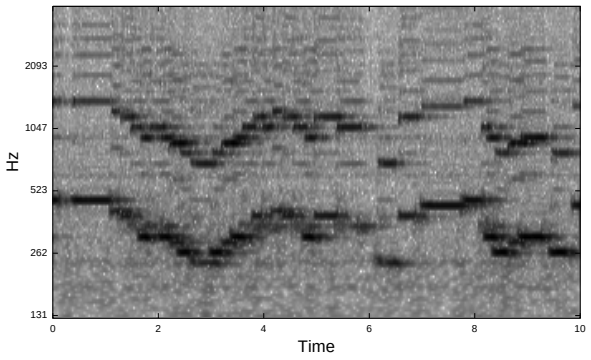

Figure 2: Constant-Q (log-frequency) spectrogram of the same recording portrayed in figure 1. Observe the harmonics move with the fundamental frequency, and the distance between semitones is constant. The vertical range is more effectively used (the same limits are used on both plots).

evaluated, and the constant-Q transform varies the number of samples according to the center frequency in order to keep Q constant.

Brown [1] defines the constant-Q transform as:

$$X_{cq}(k_{cq}) = \frac{1}{N_{k_{cq}}} \sum_{n=0}^{N_{k_{cq}}-1} w_{k_{cq}}(n) x(n) e^{-j2\pi Q n / N_{k_{cq}}} \tag{1}$$

$Q = 1/(2^{1/b} - 1)$ and b is the number of frequency bins per octave (often 12, for semitone spacing). $N_{k_{cq}} = Q f_s / f_{k_{cq}}$ and $f_{k_{cq}} = 2^{k_{cq}/b} f_{\min}$. $w_{k_{cq}}$ is a window function of length $N_{k_{cq}}$. I will use B to refer to the total number of constant-Q bins calculated.

Brown and Puckette [2] introduced an efficient algorithm for calculating the constant-Q transform, which leverages the Fast Fourier Transform and a precomputed sparse spectral kernel to compute the transform. The kernel calculation is expensive but depends only on the sample rate, number of bins per octave, and minimum and maximum frequencies. It is defined as:

$$K(k_{cq}, k) = \sum_{n=0}^{N_0-1} w_{k_{cq}}(n) e^{j\omega_{k_{cq}}} e^{-j2\pi k n / N_0} \tag{2}$$

The precomputed kernel and FFT of the signal are used to calculate the constant-Q transform:

$$X_{cq}(k_{cq}) = \frac{1}{N_0} \sum_{k=0}^{N_0-1} X(k) K(k_{cq}, k) \tag{3}$$

As an optimization, the product may only be evaluated where the absolute value of an element of the spectral kernel is greater than MINVAL = 0.15. Only 1–2% of the spectral kernel elements are larger than MINVAL (see figure 3).

This paper explores the performance and optimization of Octave[1] code to calculate the constant-Q transform using the original and Brown-Puckette algorithms described above. The Brown-Puckette algorithm is not always the fastest, and some would-be optimizations actually harm performance.

[1] Octave is an open-source high-level language for numerical computation, and is mostly compatible with MATLAB.

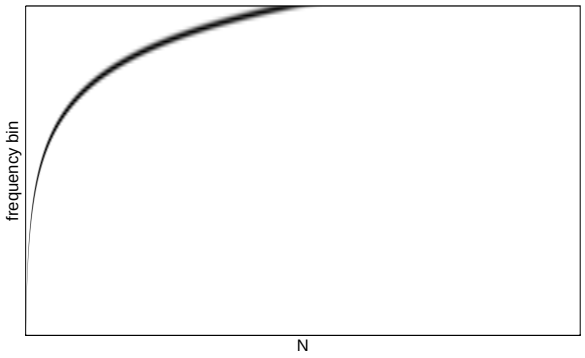

Figure 3: The spectral kernel. Most of the kernel is very close to 0, and can be converted into a sparse matrix.

2 Methods

There were five implementations: the original constant-Q algorithm (`cq`), the Brown-Puckette constant-Q algorithm (`ecq`), the vectorized Brown-Puckette constant-Q algorithm (`vecq`), the Brown-Puckette constant-Q algorithm with a sparse-matrix kernel (`secq`), and the vectorized Brown-Puckette constant-Q algorithm with a sparse-matrix kernel (`svecq`). The code for each implementation along with the benchmark code can be found online at http://hans.fugal.net/research/cq-octave/. Relevant snippets (edited for line length and brevity) are included below.

`cq` is a straightforward implementation of (1), using one for loop with a vectorized dot product.

```
for kcq = 0:B-1
  f = f_min * r^kcq;
  Nkcq = round(Q*sr/f);
  Xcq(kcq+1) = \
    sum(hamming(Nkcq) .* (x(1:Nkcq) .* \
      exp(j2piQn(1:Nkcq)/Nkcq))) / Nkcq;
end
```

`ecq` is a straightforward implementation of (3), using one for loop with a vectorized dot product. The kernel used for `ecq` is the full kernel without truncation below MINVAL. It was expected that because of the precomputation of the spectral kernel, `ecq` would outperform `cq`.

```
X = fft(x, N0);
for kcq = 1:B
  Xcq(kcq) = sum(X .* K(kcq, :)) / N0;
end
```

This code is already somewhat vectorized using the `.*` and `sum` operations, in place of another

for loop, and is natural for an Octave or MATLAB programmer. The cardinal rule of optimization in Octave and MATLAB is to avoid for loops in favor of vectorization wherever possible. By employing an additional vectorization technique using the `repmat` function, the outer for loop can also be eliminated, and this is done in `vecq`. B rows of X are replicated to give a $B \times N$ matrix, which is the size of the kernel K. These two matrices are then element-wise multiplied, and the sum is taken across the rows. It was expected that this optimization would improve performance over `ecq`.

```
X = fft(x, N0);
Xcq = sum((repmat(X,B,1) .* K), 2)/N0;
```

The motivation behind the Brown-Puckette constant-Q algorithm is that the spectral kernel is sparse and few multiplications per bin are needed. The efficient way to implement this in Octave is to use a sparse matrix, which allows Octave to avoid doing expensive operations on zeros without expensive for loops and conditionals in the Octave code. `secq` is `ecq` executed with a sparse kernel. Likewise, `svecq` is `vecq` with a sparse kernel. `secq` and `svecq` were expected to perform better than `ecq` and `vecq` respectively because of the sparse kernel.

`cqkern` is the code to calculate the spectral kernel and `sparse` is the code to set values of K less than MINVAL to 0 and then convert K into a sparse matrix. Their execution time is the one-time overhead needed to use the Brown-Puckette algorithm family.

The parameters used for the first experiment are a sample rate of 44100 Hz, 12 bins per octave, $f_{\min} = 16.35$ Hz (C0, MIDI note 12), and $f_{\max}$ at the Nyquist frequency.

Each algorithm ran 100 times in succession, timed using `cputime`. The experiment was done on two systems. The first is a MacBook with a 2GHz Intel Core Duo processor and 2G of RAM, running OS X 10.5 with Octave 3.0.3 from MacPorts. The second is an Athlon 64 2800+ (1.8GHz) desktop with 1G of RAM, running 32-bit Ubuntu 8.10 with Octave 3.0.1 stock. The `measure` FFTW planner was used.

The experiment was then repeated with the same parameters except $f_{\min} = 130.81$ Hz (C3, MIDI note 48).

3 Results

Figures 4 and 5 show the mean execution times with standard deviation for each experiment.

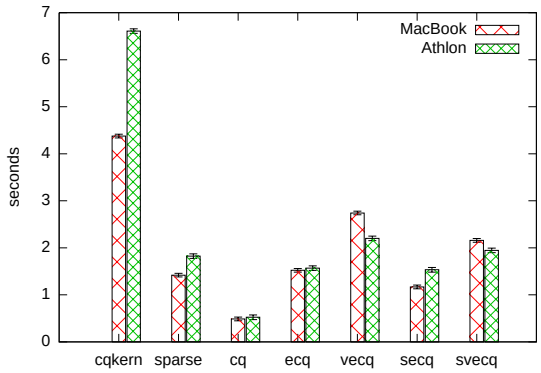

Figure 4: Mean execution time and standard deviation over 100 runs with $f_{\min} = 16.35$ Hz (C0, MIDI note 12), on two computers.

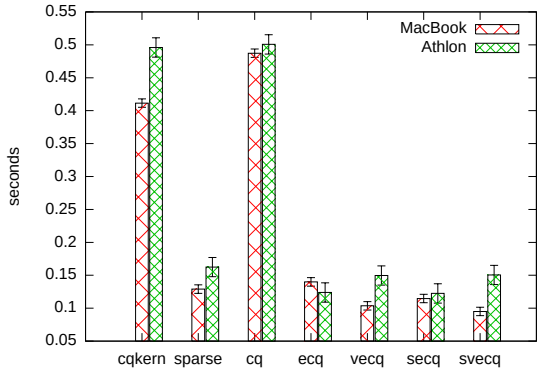

Figure 5: Mean execution time and standard deviation over 100 runs with $f_{\min} = 130.81$ Hz (C3, MIDI note 48), on two computers.

For the first experiment ($f_{\min} = 16.35$ Hz), `cq` is the fastest by far. `vecq` and `svecq` are slower than `ecq` and `secq` respectively. `secq` and `svecq` are faster then `ecq` and `vecq` respectively.

For the second experiment ($f_{\min} = 130.81$ Hz), `cq` is the slowest by far. On the Athlon `vecq` and `svecq` are still slower than `ecq` and `secq` respectively, but on the MacBook `vecq` and `svecq` are faster than `ecq` and `secq` respectively. In this experiment the effect of the sparse kernel is negligible on the Athlon.

4 Discussion

The Brown-Puckette algorithm as implemented was not always more efficient. For lower $f_{\min}$ the direct algorithm is considerably faster. But for higher $f_{\min}$ the Brown-Puckette algorithm is considerably faster, and the overhead for calculating the spectral kernel is paid for almost immediately. An analysis of the algorithms reveals that they are all of the same order:

$O(N_0 \log N_0)$. (N_0 depends primarily on $f_{\min}$ in practice, when the sample rate and Q are held constat.) It may be that when N_0 is large the speed of memory access and memory management issues dominate, but when N_0 is small the inefficiencies of for loops in Octave dominate. Each algorithm is of the same order but the implementations depend on the size of the input in complex ways.

Conventional wisdom says that the more vectorized the code is, the faster Octave can execute it. However, the `repmat` vectorization of the Brown-Puckette algorithm was slower than the straightforward implementation, except on the MacBook for higher $f_{\min}$. The working-copy space complexity of the straightforward implementation is $O(N_0)$, and that of the repmat implementation is $O(N_0 \log N_0)$. Care was taken to avoid swapping to disk during the experiments, but we may be seeing a slowdown due to more memory accesses and memory management. The speedup of `vecq` on the MacBook for higher $f_{\min}$ may be due to parallelization on the dual cores.

Sparse matrices improve speed (or at least do not slow it down), and the overhead of making the kernel sparse is not great, so when using the Brown-Puckette algorithm sparse matrices should be used. But the sparseness does not make an order-of-magnitude difference, and if higher accuracy is desired it can be omitted.

5 Future Work

It would be instructive to run the same experiments on MATLAB and compare the performance to that of Octave, both in relative and absolute terms. Would the `repmat` optimization be slower on MATLAB as well?

There could be further investigation of the inherent differences in the two algorithms, perhaps comparing implementations in C, where memory management is more transparent and loops are not inefficient, making a direct implementation of the Brown-Puckette constant-Q algorithm more true in spirit to the original paper.

References

[1] Judith C. Brown. Calculation of a constant Q spectral transform. *J. Acoust. Soc. Am*, 89(1):425–434, January 1991.

[2] Judith. C. Brown and Miller S. Puckette. An efficient algorithm for the calculation of a constant Q transform. *J. Acoust. Soc. Am*, 92(5):2698–2701, November 1992.