Sliding DFT for Fun and Musical Profit

John FITCH
Codemist Ltd
Horsecombe Vale
BATH BA2 7AY,
United Kingdon
jpff@codemist.co.uk

Richard DOBSON
Composers Desktop Project
c/o University of Bath
BATH BA2 5QR,
United Kingdom,
richarddobson@blueyonder.co.uk

Russell BRADFORD
Dept. of Computer Science
University of Bath
BATH BA2 5QR,
United Kingdom,
rjb@cs.bath.ac.uk

Abstract

Many Csound users may have seen reference to the SDFT or Sliding Discrete Fourier Transform in recent months. Here we review the basic ideas, and describe the Csound implementation. The main musical message is that it is integrated with the f-signals for the streaming phase vocoder written by one of us earlier, and so is easy to use. The down side is that it is slow, and definitely not yet for real-time performance. We briefly discuss approaches to improved performance.

Keywords

DFT, Csound, HiPAC, TransformationalFM

1 The Sliding Discrete Fourier Transform

The use of the spectral domain is well known and there is no reason to reiterate the underlying concepts. Most people use an algorithm called the FFT (Fast Fourier Transform) to calculate the Discrete Fourier transform of a window of audio signal, and then perform whatever operation on it they wish. The reconstruction of the time-domain signal is then performed by the IFFT (Inverse Fast Fourier Transform) algorithm, together with some averaging process.

The SDFT is an alternative algorithm, whereby the analysis window is always moved on by one sample ("hopsize = 1"), and the value of the DFT for the new window is calculated from the previous one. That such a process should exist is not a surprise, as the FFT calculates the spectral components in a window, and if that window is moved (slid) by one sample, it is obvious that most of the information will remain the same. An early music-related description of this form of the phase vocoder appears in [1], which highlighted its value for frequency analysis. More recently the SDFT was highlighted by James Moorer, *Audio in the New Millennium*[2], which in turn formed the starting point for our own work [3].

The bottom line is that for each frequency bin the process of moving one involves adding the new sample, subtracting the sample that has fallen off the start of the window, and a multiplication (rotation) by a complex value. If we have a function $f(n)$, represented as discrete samples f_0, f_1, ..., and we assume a window size of N, which is to say that we assume that the signal repeats over that period, then the DFT, starting at time t, is

$$F_t(n) = \sum_{j=0}^{N-1} f_{j+t} e^{-2\pi i j n/N},$$

where $F_t(n)$ is the value in the nth bucket in the frequency domain.

As we are moving by 1 sample only:

$$\begin{aligned} & F_{t+1}(n) \\ &= \sum_{j=0}^{N-1} f_{j+t+1} e^{-2\pi i j \frac{n}{N}} \\ &= \sum_{j=1}^{N} f_{j+t} e^{-2\pi i (j-1) \frac{n}{N}} \\ &= \left(\sum_{j=0}^{N-1} f_{j+t} e^{-2\pi i j \frac{n}{N}} - f_t + f_{t+N} \right) e^{2\pi i \frac{n}{N}} \\ &= \left(F_t(n) - f_t + f_{t+N}\right) e^{2\pi i \frac{n}{N}} \end{aligned}$$

We start from all zeros so there is no starting problem for the first DFT frame.

There are a number of implications of this formulation, most obviously there is no requirement for the window size to be a power of two. Some of the more surprising implications can be found in [4]. Here we just draw attention to the band-width of each bin, which is up to Nyquist, and so each bin can contribute to any frequency.

But the point of this paper is not to discuss the mathematics which can be found elsewhere, but rather the Csound[5; 6; 7] implementation, and the opcodes that can use it. Before that

we do need to consider an important issue, windowing.

2 Windowing and Sliding

The normal process is to apply an envelope window to the sample period to improve frequency resolution and minimise smear. In the SDFT this cannot be done in the time domain. But it is well known that convolution in the time domain is multiplication in the frequency domain, and so we apply the window after calculating the transform. This does impose a slight restriction, that for practical reasons the window type must be built from cosines. This is acceptable for rectangular, Hamming, von Hann, Blackman (and variants), Blackman-Harris, and Nutall. It does not include Kaiser windows which is pity; we think we have a solution but it is a compromise, and is currently not implemented.

We are also forced to lose the ability to provide a window as a ftable, a CUSTOM window. Technically it would be possible to provide a user-window facility presented as coefficients of the cosine, but this does seem rather detailed for common use!

The windows that are available in the current implementation are shown in Table 1.

3 Integration into Csound

Due to earlier work by Richard Dobson, and extended by Victor Lazzarini[8], Csound has a useful and well constructed streaming phase vocoder. The process here is to construct a new DFT frame when sufficient samples have been obtained, with a restriction that this cannot be more than once per k-rate frame. Internally the f-variables have a structure to maintain the bin values, window size and type and various housekeeping data. The SDFT implementation in Csound reuses this structure, so from an elementary user point of view the introduction of the sliding option has no syntactic change. All existing use of the pvsXXX opcodes and f-variables should continue to work after the introduction of sliding.

In essence the SDFT is a transform with an overlap parameter of 1 sample. That plus the restriction on the existing **pvsanal** give the clue to the way the integration has been done. If the overlap is less than the ksmps or is small (currently set to less than 10 but maybe this should be 2) the sliding format is used. This requires a transform frame for each of the ksmps samples, and that happens underneath the user's view. The construction of the frames then proceeds using the formula above rather than the FFT.

The reconstruction of the signal is rather different. In the original Moorer paper in AES the inverse algorithm is very memory intensive while looking deceptively similar to the forward transform. Fortunately there is a better way, as described in [3]. As we have a transform for each sample we can treat each frame as a representation of a single sample, and then the fundamental Discrete Fourier Transform formula

$$f_t = \frac{1}{N}\sum_{j=0}^{N-1} F_t(j)e^{-2\pi ijt/N}$$

applies. There is a choice we can make; if each frame represents one sample, which one? If we choose the oldest sample then the t in this formula is zero, and the equation collapses to a straight addition of the bins! This does introduce a latency of the window size of course. By considering the middle sample to be represented by the frame this becomes an alternating add/subtract formula. A zero latency version is more expensive, and while we have had it implemented in our experimental code it seems rather too much.

It should not come as a surprise that such a simple formula exists, as it is just treating the reconstruction as an oscillator bank. In the Csound SDFT implementation of pvsynth we are using the midpoint version, which is a good compromise between latency and speed. We could make an option for alternatives but that seems a little over-complex.

3.1 Csound Opcodes

The opcode to create a sliding DFT frame is **pvsanal** as before. The arguments are not changed, but if the overlap parameter is 1 or less than **ksmps** then sliding is used. In the manual the example given is

```
ain     in
ffin    pvsanal  ain,1024,256,2048,0
```

If this is replaced by

```
ain     in
ffin    pvsanal  ain,1024,1,2048,0
```

the sliding mechanism will be used, and the internal structures will be different, but presented to the user in the same way.

To reconstruct the signal the opcode **pvsynth** is used; if the f-signal is a sliding

Name	Index	Formula
HAMMING	0	$0.53836 - 0.4614\cos(2\pi n/(N-1))$
HANN	1	$0.5 - 0.5\cos(2\pi n/(N-1))$
KAISER	2	*unavailable*
CUSTOM	3	*unavailable*
BLACKMAN	4	$0.42 - 0.5\cos(2\pi n/(N-1)) + 0.08\cos(4\pi n/(N-1))$
BLACKMAN_EXACT	5	$0.42659071367153912296 - 0.49656061908856405846\cos(2\pi n/(N-1))$ $+0.076848667239896818572\cos(4\pi n/(N-1))$
NUTTALLC3	6	$0.375 - 0.5\cos(2\pi n/(N-1)) + 0.125\cos(4\pi n/(N-1))$
BHARRIS_3	7	$0.44959 - 0.49364\cos(2\pi n/(N-1)) + 0.05676\cos(4\pi n/(N-1))$
BHARRIS_MIN	8	$0.42323 - 0.4973406\cos(2\pi n/(N-1)) + 0.0782792\cos(4\pi n/(N-1))$
RECT	9	1

Table 1: Window Types available in Csound SDFT

one then the appropriate mechanism is used. We found it convenient to have a Boolean field in the f-structure to say that this particular f-variable is a sliding one.

4 Transforming in the Spectral Domain

Just translating to and from the spectral domain is not very interesting. The value lies in the transformations. Csound has a rich (and growing) collection of these, and they all need to be made aware of the sliding format. In addition this opens the possibility for a-rate changes in some cases. In this section we indicate the progress so far in this. There are three classes; those known to work, those believed to work but not completely tested, and those not addressed.

4.1 Verified Transformations

A very common use of the spectral domain is to change the pitch of a signal. Csound provides two opcodes for this, **pvscale** which scales the frequency components in a harmonic way, and **pvshift** which shifts the frequency components, stretching/compressing its spectrum. Both these are implemented with sliding, but with one important additional feature, the shift amount (the second argument in both cases) can optionally be an a-rate variable. This gives much finer-grained control over the transformation, and allows a **kr** value as required elsewhere, possibly quite large.

```
instr 3
      al   line     400, p3, 500
      asig in
      fsig pvsanal  asig,128,1,128,1
      fs   pvshift  fsig, al, 10
      acln pvsynth  fs
           out      acln
endin

instr 4
      asig oscili   16000, 440, 1
      fsig pvsanal  asig,512,1,512,1
      fs   pvscale  fsig, 1.1
      acln pvsynth  fs
           out      acln
endin
```

With the increased bandwidth in each analysis bin it is possible to use a simpler and more accurate pitch-shifting algorithm, and the sliding form actually is better at preservation of the amplitude, but we lose the formant-preserving variant.

Another common use is the spectral freeze. Csound provides **pvsfreeze** which can also be used in a sliding fashion, but is otherwise unchanged

```
instr  1
       kl   line       100, p3, 1000
       asig oscili     16000, kl, 1
       fsig pvsanal    asig,128,1,128,1
       ktrig oscil     1.5, 2, 1
       ktrig = abs(ktrig)
       fou  pvsfreeze  fsig,ktrig,ktrig
       aa   pvsynth    fou
            out        aa
endin
```

There are different ways of filtering in the spectral domain. The **pvstencil** provides one, using a masking function table. For example

```
instr 2
       kl   line      100, p3, 1000
       asig oscili    16000, kl, 1
```

```
      fsig pvsanal   asig,128,1,128,1
      fcln pvstencil fsig, 0, 1, 1
      acln pvsynth   fcln
           out       acln
endin
```

with `f1 0 4096 10 1` in the score.

The last of the verified opcodes is **pvsmix** which does a spectral mix of two signals, taking the largest amplitude from each matching pair of bins

```
instr 5
      a2   diskin2    "latedemo.wav", 1
      f2   pvsanal    a2,1000,1,1000,1
      asig in
      fsig pvsanal    asig,1000,1,1000,1
      fs   pvsmix     fsig, f2
      acln pvsynth    fs
           out        acln
endin
```

The previous version of **pvscent** to calculate the spectral centroid of a signal has been changed to return either a k-rate answer or an a-rate one. In practice the a-rate values may need to be smoothed by another lowpass filter before use.

```
instr 8
   a1    in
   fsig  pvsanal   a1,1024,1,1024,1
   kcen  pvscent   fsig
   adm   oscil     32000, kcen, 1
         out       adm
endin
```

The opcode **pvsmaska** modifies the amplitudes of the bins using a table of modification values. This works essentially unchanged

```
  asig  buzz     20000,199,50,1
  fsig  pvsanal  asig,1024,1,1024,1
  kmod  linseg   0,p3/2,1,p3/2,0
  fsig2 pvsmaska fsig,2,kmod
  aout  pvsynth  fsig2
        out      aout
```

The values in the bins can be smoothed using a 1st order lowpass IIR filters with time-varying cutoff frequency; this is the same as in the non-sliding case.

```
  asig  in
  fim   pvsanal  asig,1024,1,1024,1
  fou   pvsmooth fim, 0.01, 0.01
  aout  pvsynth  fou
        out      aout
```

4.2 Converted but Untested or Different Transformations

It is only time that has not permitted the full checking of these opcodes.

Another filtering opcode **pvsfilter** filters one stream according to a second. This opcode runs but at present produces rather different (but interesting!) output. The same is true for **pvsbin**; this could return the a-rate values, but so far it is limited to returning the values at that start of each k-cycle.

The opcode **pvscross** has unverified code that may work and **pvsinfo**, the information opcode has not been changed, but should work the same — it does not give an explicit statement that it is doing sliding, but the overlap value includes that.

4.3 Non-functional Transformations

The modified opcodes fall into two major categories; those which have just not yet been looked at and those that have major design questions.

The first category includes **pvsdemix**, **pvsmorph**, **pvspitch**, **pvsifd**, **pvsosc**, **pvsblur** and **pvsarp**.

The problematic opcodes are those that read and write, as the data from the sliding variant is much larger. Reading externally generated `PVOC-EX` data will not work either, unless that is generated at 1 sample overlap, and the utility to do that needs to be written. The opcodes here are **pvsdiskin**, **pvsftr**, **pvsftw**, **pvsfread**, **pvsin**, **pvsout**, **pvsdisp**, **pvsfwrite**, **pvsvoc**, **pvsbuffer**, and **pvsbufread**.

As we are already using an oscillator-bank to reconstruct the signal there is no need for the **pvsadsyn** opcode.

5 Musical Uses

The addition of the SDFT to the established streaming phase vocoder creates what we call a Sliding Phase Vocoder (SPV). The two most significant aspects of the SPV are the synthesis by oscillator bank (with each bin able to contain any frequency below the Nyquist limit), and the single-sample update. Thus, any modification that can be applied to an additive oscillator bank can be employed in the SPV. With the conventional hopping phase vocoder, pitch modulation is a familiar process, but limited to relatively low modulation rates by the low analysis rate. At the lowest rates, vibrato and other sweeping effects can be applied to a sound; at

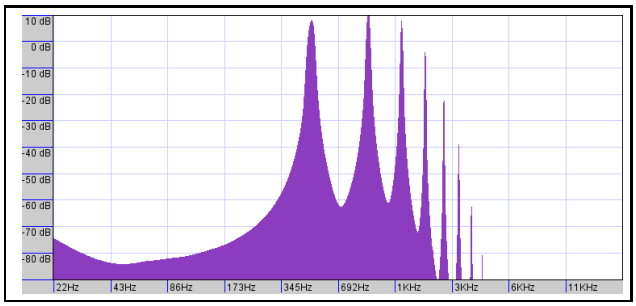

Figure 1: Spectrum of simple Transformational FM

higher rates the effect is more granular in nature, as the pitch shift is constant within each frame, frames being summed with overlaps to generate the output. With the SPV, the possibility arises to perform genuine per-sample modulation at the full range of audio rates. Indeed, we can mimic classic FM this way. Considering a plain FM patch such as:

```
instr 10
    amod oscili  1.5,500,1
    acar oscili  16000,1000*(1+amod),1
    aout dcblock acar
         out     aout
endin
```

We can implement a corresponding frequency-domain FM patch:

```
instr 1
    amod oscili 500,500,1    ; modulator
    acar oscili 16000,1000,1 ; carrier
    fcar pvsanal acar,1000,1,1000,1
 ; frequency-domain FM
    fs   pvshift fcar,amod,0
    asig pvsynth fs
    aout dcblock asig
         out  aout
endin
```

The spectrum of this signal is given in Figure 1. However, there is the equal possibility to derive input from an arbitrary external source, which we have accordingly termed *Transformational FM* (TFM):

```
instr 2
         ; carrier is input audio
    acar in
         ; 1KHz modulator
    amod oscili  0.2,1000,1
    fcar pvsanal acar,1000,1,1000,1
    fs   pvscale fcar,1 + amod,0
    asig pvsynth fs
    aout dcblock asig
         out     aout
endin
```

Here we use pvscale to preserve the harmonicity of the source, while noting that as pvscale takes a ratio argument, the resulting FM sidebands will not correspond to those of classic FM (while still being of musical value[4]). An alternative formulation we have used successfully (but not so far using Csound) employs a naïve peak-finding algorithm, and applies a common linear shift to the group of bins associated with each peak. The required shift factor for each group is calculated from the frequency at the detected peak. While this could certainly be implemented as an opcode, it is our hope that the recently introduced streaming partial-tracking opcodes[9] can themselves be adapted to use the SDFT. Of course in applying TFM in this way some care must be taken with the choice of input sound, as it is all too easy to transform a harmonically dense input into sheer noise; but we have obtained a range of satisfying transformations of a number of harmonically rich monophonic instrumental sounds, from horn and trumpet to a glockenspiel.

As in the example of Adaptive FM [10] it is also possible to incorporate pitch tracking into the TFM process (not least since the analysis information is already present), so that the modulation ratios can be maintained as the source instrument or voice changes pitch. The two techniques differ in that AFM is a time-domain technique based on a variable delay line (and requiring a separate analysis stream), while TFM (which can be used adaptively but is more general) employs audio-rate frequency-domain modulation. The possibilities are indeed limited largely by our imagination, especially when we realise that with TFM we do not have to confine ourselves to an identical modulation to all bins.

In using TFM, the reduced latency afforded by the SPV will make pvoc that much more attractive for interactive real-time performance, once suitably powerful hardware is available.

6 Some Thoughts and HiPAC

Clearly this is work in progress. The untested opcodes will be tested soon, and the opcodes that were overlooked will get attention. What to do about read/write and external data needs some reflection and advice.

There is also one important caveat. This process is not fast. We will repeat that: **THIS PROCESS IS NOT FAST**, and so do not expect to use it in real-time performance, at least until hardware improves and our next research project is funded. For example the transformational FM example would need about a 45GHz Intel processor to be real-time.

More technical information about the process can be found in the slides presented at ICMC2007 which are available[11] with audio examples.

We are addressing the speed issue by investigating one important feature of the SDFT, that it is totally parallel. The calculation for each bin is independent of the others, and so the process is a candidate for SIMD-style parallelism. With generous access to hardware and software products provided by ClearSpeed Plc[12], we are starting a number of projects in HiPAC (High Performance Audio Computing). In addition to continuing development of the systems described in this paper, in particular we are investigating the applications of SIMD to physical models.

7 Conclusions

We have presented the Sliding Discrete Fourier Transform as a practical tool within Csound, and hence via the Csound plugin class `csoundapi~`, in Pure Data[13]. Other systems can access the code now via OSC, but it would not be too hard to incorporate in other systems. At present the computational cost is high, and this limits the application to out-of-real-time calculations. However we are already progressing with parallel SIMD-style implementations, and it can be seen that it may produce a good use for multiple-core computers.

Our experience with Transformational FM leads us to believe that there are many such musical uses in this area waiting for faster hardware.

8 Acknowledgements

This work was funded largely by the **Arts and Humanities Research Council** under their *Speculative Research* theme, to whom we express our thanks.

References

[1] Miller S. Puckette and Judith C. Brown. Accuracy of frequency estimates using the phase vocoder. *IEEE Trans. Speech and Audio Processing*, 6(2):1660–176, March 1998.

[2] James A. Moorer. Audio in the New Millennium. *J. Audio Eng. Soc.*, 48(5):490–498, May 2000.

[3] Russell Bradford, Richard Dobson, and John ffitch. Sliding is Smoother than Jumping. In *ICMC 2005 free sound*, pages 287–290. Escola Superior de Música de Catalunya, 2005.

[4] Russell Bradford, Richard Dobson, and John ffitch. The sliding phase vocoder. In *Proceedings of the 2007 International Computer Music Conference*, volume II, pages 449–452. ICMA and Re:New, August 2007. ISBN 0-9713192-5-1.

[5] Barry Vercoe. *Csound — A Manual for the Audio Processing System and Supporting Programs with Tutorials*. Media Lab, M.I.T., 1993.

[6] Richard Boulanger, editor. *The Csound Book: Tutorials in Software Synthesis and Sound Design*. MIT Press, 2000.

[7] John ffitch. The Design of Csound5. In *Linux Audio Conference*, April 2005.

[8] Victor Lazzarini. Extensions to the Csound Language: from User-Defined to Plugin Opcodes and Beyond. In *Proceedings of LAC2005*, pages 14–19, Karlsruhe, April 2005. Linux Audio Consortium.

[9] Victor Lazzarini, Joe Timoney, and Tom Lysaght. Streaming Frequency-Domain DAFx in Csound 5. In *Proc. of the Int. Conf. on Digital Audio Effects (DAFx-06)*, Montreal, Quebec, Canada, Sept. 18–20, 2006.

[10] Victor Lazzarini, Joseph Timoney, and Thomas Lysaght. Adaptive FM synthesis. In *Proc. of DAFx07*, pages 21–26, Bordeaux, France, September 2007.

[11] Russell Bradford, Richard Dobson, and John ffitch. Slides from Presentation at ICMC2007. `http://dream.cs.bath.ac.uk/SDFT`.

[12] ClearSpeed Technology plc. ClearSpeed Whitepaper: CSX Processor Architecture. `http://www.clearspeed.com`, Feb 2007.

[13] `http://puredata.info/`, May 2007.