Video Fourier Transforms using PureData

Mathieu Bouchard

ABSTRACT

Fourier Transforms have been available for PureData for a long time. However they have never been applied to pixels, not counting the possibility that someone converted an image to sound in order to use [fft~] on it, which is fastidious and is still just a one-dimensional transform. This paper exposes the possibilities of using Fourier Transforms on whole images and on sequences of them that enables a new palette of effects that comes mostly from the many variations on the theme of applying the Convolution Theorem. These effects can be applied on realtime video, provided that the video stream uses sufficiently little bandwidth. This paper covers implementation issues both at generic and pd-specific levels and covers various effects that can be made using the [#fft] object class.

1. IMPLEMENTATION ISSUES

1.1 Number Field

To use Fourier methods, one needs a vector space, and to have a vector space, one needs a number field[1], such as the Real numbers, the Algebraic numbers, or the Rational numbers. For use with computers, true number fields are unwieldy, so approximations are in use, such as floating-point and fixed-point[2]. Fixed-point doesn't go well with Fourier methods, because Fourier components usually have values at much varying scales. However, video is usually all done in fixed-point, even in PureData and even though PureData is that much floating-point-oriented and that little fixed-point-oriented. Both GEM and PDP are devoid of any floating-point image formats, and even though GridFlow has supported floating-point images for over 4 years, little has been done using that feature.

[1]French *corps*, German *körper*

[2]a number system made of fractions with fixed denominator; this is completely unrelated to iterative numerical methods, unlike the other meaning of the same word.

Naturally, there's an incentive towards finding a way to adapt FFT to fixed-point. One way is to use a high-precision fixed point, which allows for high-amplitude low-frequency components without clipping, and low-amplitude high-frequency components without much quantization. Another way to compensate is to scale FFT components according to their frequency, just like how the energy of a sound has to do with the speed of the speaker membrane rather than its position, so in a typical blend of bass and treble meant to be heard, treble will operate on a much smaller scale of speaker positions. This can be done either by multiplying each component by its frequency, or by applying FFT on the derivative of the signal instead of the signal itself, but a lot of care has to be put into balancing quantization and the risk of overflow. No matter how it's done, a special case has to be made for frequency 0 (DC).

Fortunately, it is now possible to consider using floating-point for all video computations, given how fast floating point is in modern processors, if one is willing to accept 96 bits-per-pixel. GridFlow already defaults to that data rate (using integers instead), so from that perspective, making the jump towards floating-point is not a matter of RAM bandwidth. The FFTW3 library is used by recent versions of GridFlow to provide [#fft], PureData's only image-based FFT.

Note that all integer-based number fields that are not approximative wouldn't work with this kind of problem because they rely on modulo arithmetic, which introduces number relationships that have nothing to do with what we are trying to model.

1.2 Algebraic Extension

Fourier Theory is more understandable using Complex numbers than non-Complex numbers, because introducing the concept of square root of negative numbers reveals hidden relationships between concepts that seem very different in non-Complex numbers, making Complex mathematics less complex than non-Complex mathematics in some respects (but saying this doesn't make people any less scared of Complex numbers). However, there are some new complexities introduced by them.

A Complex number needs to be represented by a pair of non-Complex numbers, usually called "real" and "imaginary", which in the context of Fourier components, may be called "cosine" and "sine" instead (respectively). This duality has to be implemented in some way in PureData. For the built-in [fft~], this is done using two signals. If [fft~] gets a single signal, the positive and negative frequencies will mirror each other in a redundant way. Because of this, there are [rfft~] and [rifft~] which skip over the redundant processing at the level of FFT itself, but à priori the intervening frequency-domain objects don't know about the situation and so can't get any faster. To really eliminate all work done because of negative frequencies, [rfft~] and [rifft~] would have to support two different sample rates at the same time, or else find a way to merge both signals together (but those merged signals would only be realistically processable by new special objects designed to work with them). So maybe we have to accept this as a fact of life and just get used to always process signals in pairs or waste half of operations on zeroes getting discarded.

For grids, however, the equivalent of sample rate can get as variable as desired, and so can the logical organisation of data. A "real" FFT could be performed on a grid of size 256 by 256 by 3, outputting a pair of grids of size 256 by 128 by 3, where the duplicate frequencies have been removed (In 2-D, still only half of the values are wasted, which is why I'm not suggesting 128 by 128). In practice this is further complicated by the fact that DC and Nyquist don't really belong in the same bin (in 2-D, a column of bins), apart from being the two components of a complex FFT that don't have a sine part. If they were put in the same bin, one of them would have to pretend to be a sine. Due to this exclusion principle, 256 by 129 would be more appropriate, and if we wanted much alignment, 256 by 136 would be appropriate (and so would be 129 by 256).

For [#fft], I have decided to use a single grid instead of two. This turns out to be useful for avoiding procedures of interleaving and deinterleaving required by algorithms that want data in an interleaved fashion. Because of the order of dimensions, this grid's size can't be 2 by 256 by 256 by 3, it has to be 256 by 256 by 3 by 2 because the interleaving happens in the last dimension. Here, the 3 by 2 may be thought of as each channel of a bin having two subchannels (e.g. cosine green vs sine green); it may also be thought of as having 6 channels, and indeed it's sometimes useful to cast this to 256 by 256 by 6. I have avoided writing [#rfft] and [#rifft] on the grounds that this optimisation would be premature, just like eliminating duplicate frequencies would also be premature. So far, the best use of resources with minimum complexity is to [#redim] so that a 4-channel system becomes a 2-by-2-channel system, wherever the frequency-domain operations don't need the channels to be absolutely separate (e.g. uses of the Convolution Theorem).

1.3 Vector Space

After Complex numbers have been accepted, a vector space needs to be made using them. There is a need to deal with the fact that we have several pixels and that they are going to be converted into frequency bins. Complex numbers are taken to be the scalar field which is raised (by cartesian product) to a power equal to the number of dimensions, which in the case of sound is the block size. Here a dimension is synonymous with a degree of freedom. In the case of a single image, the number of dimensions is the number of pixels, but that is not all that is determining the structure of the vector space: the way that the FFT applies itself on the dimensions depends of relationships between the dimensions. In this case, the dimensions are organised in rows, columns and channels; that organisation could be called the dimensions of dimensions. The two-dimensional FFT sought after is actually a pair of FFT, one of which is along rows and the other is along columns. Each of those can be seen as many smaller FFT that operate on scalar components, or as one huge FFT that operates on large vector components (whole rows and whole columns of pixels).[3]

In a 2-D FFT, the concept of frequency is somewhat different, because frequency itself becomes "vectorial"[4]. A frequency is then a pair of a row frequency with a column frequency, the latter two being integers. For instance, there is one particular bin for a picture of size (240 320) represents the harmonic that has frequency (40 64), meaning that its row frequency is 40 cycles per 240 pixels (one sixth of a cycle/pixel) and its column frequency is 64 cycles per 320 pixels (one fifth of a cycle/pixel). It is possible to talk about diagonal frequencies and the angles of frequencies, but the precision and relevance of this concept is limited when the number of possible frequencies is finite: the pixel grid, aka "square lattice", is obviously not rotation independent, except at right angles, else no-one would ever need anti-aliasing of diagonal lines.

FFT tends to be fastest when the number of bins factorizes well. The classic algorithm works in an amount of time roughly proportional to the sum of the prime factors of the number of bins (repeated factors included). This encourages the number of bins to be a power of two. PureData's FFT works only on powers of two, but when it's time to do FFT on images, without any spatial block size (unlike JPEG, which has a 8-by-8 block size), then there's a serious need for padding before starting to do anything with power-of-two sizes, and if wanting to work in wrap-around coordinates, it's just impossible to pad. This means that non-power-of-two FFT has to be supported. Fortunately, FFTW does, for small prime factors (nowadays there exist variants of FFT for larger prime factors but they are not necessary here, as already enough possible widths are possible with small primes, even considering the additional restrictions that FFTW puts on this and that I'm not explaining).

There is never any FFT done along the channels dimension, because it's almost always useless to do that. For it to make sense, there has to be many channels and they have to form a finite ring in some intuitive way (but let's not get into that...).

There is rarely any FFT done along the time dimension. That dimension is usually not apparent in GridFlow because each image is a separate grid, but grid transmissions may be reframed using [#import] as a very large grid with a time dimension. FFT along the video time dimension faces the same problems as sound does, but in slow motion (and thus much more noticeably) because (among other reasons) the sample rate is over a thousand times lower. MPEG doesn't do any FFT along the time dimension. Currently, [#fft] doesn't support the time dimension.

2. APPLICATIONS

[3]This is said considering that a structure like a matrix or a tensor or a grid is in the end a vector, when seen by plain vector operations. Other things, like matrix product and the Einstein notation, use the organisation of dimensions where plain vector operations don't.

[4]Not in the exact sense of Linear Algebra, because indices don't usually form a field, and we don't want them to, and that's perfect because we don't need a true vector space as long as we don't need a well-defined exact division operator.

2.1 Fast Convolutions

There exist fast convolution algorithms for special convolution kernels. For example, a 15-by-15 square blur may be decomposed into the equivalent combination of a 15-by-1 rectangle blur and a 1-by-15 rectangle blur. Another example: a blur with kernel (1 2 3 4 5 4 3 2 1) can be decomposed into two identical blurs with kernel (1 1 1 1 1). Decomposition doesn't have to be by composition: if a kernel is (1 2 3 4 99 4 3 2 1) it can be decomposed as a sum of (94) and the previous kernel. However, it's not so easy to figure out an efficient decomposition. They're normally devised by human intervention and only for special cases.

There are several ways to apply large rectangle blurs in the same time that small ones can be applied. Kernels that have lots of zeroes can be partially optimised out (which GridFlow does but the others don't). There are plenty of tricks and they still work only for particular kernel patterns.

FFT can be used to compute convolutions with *any* kernel in the same amount of time. The kernel has to be zero-padded so that it is as big as the picture itself. Typically, if you are convolving with a 15x15 kernel the ordinary way, it's possible that three FFTs and a Complex product together are faster than that. It's also possible with something as small as a 11x11 kernel. The threshold at which FFT becomes more efficient than dumb convolution, is something that depends on the speed of both, but such a threshold always exist.

Fast convolution means the ability to make impressive large-scale blurs. Generally speaking, a blur is a low-pass filter. It can also be used for making edge-detection, which is high-pass filter, but it's less useful at that, because edge-detection tends to involve only very small areas so they only need very small kernels anyway, and then they usually need conversion back to space-domain because of non-linear operations that have to be done on them afterwards. For example, in computing a polar Sobel, one has to compute the magnitude from the local horizontal and vertical ripples around every pixel position: this is a space-domain operation. If we multiply the horizontal ripples with themselves from a space-domain point of view, but the data is in spatial-frequency-domain, then a mirror image of the usual Convolution Theorem applies, and so this operation is equivalent to convolving that spectrum with itself, which so abominably slow that it can only mean that the spectrum should be converted back to space-domain.

2.2 Notch Filter Design

Cameras are subject to electromagnetic interference due to unsufficient (or completely absent) shielding in their sensor. Images scanned from magazines may show moiré effects due to interference between the pixel grid and the grid of the dithering. In both cases the image can be smoothed using a notch filter, supposing that the picture does not feature patterns that coïncide with the interference in terms of frequencies (including the directions of the ripples).

In 2-D, notches may be more complex than just intervals. There are many moral equivalents of intervals:

- rectangle-shaped holes (cartesian products of intervals) and rhombus-shaped holes (a sheared rectangle hole)
- a disk-shaped hole (all points within a certain radius of a point) or an elliptic hole (a sheared disk made by slightly changing the definition of distance). This is what the Q-factor corresponds to in 2-D. In the case of the circle, the Q-factor is still a single number, for for the ellipse the Q-factor is a matrix.
- a circular band centered at the origin, in which case it is an anisotropic notch filter, which is a 1-D filter radially converted to 2-D.
- some infinite band with straight parallel borders, in which case it's a 1-D filter linearly aligned in a specific direction.
- a more special 1-D filter is the one that filters out some angles. This is an infinite-size pacman. A finite-size pacman is a low-pass filter and an angle-notch-filter at once.

To each of those filters, as usual, corresponds some a kind of noise that this filter is good at getting rid of.

In each case, the notch should be duplicated rotated 180 degrees around the origin. This is the same as making a double reflection, one along the x and one along the y.

2.3 Deconvolution

Just like convolution is a multiplication of spectra, deconvolution is division of spectra. Proper deconvolution is hard to do. First the kernel has to be invertible. This can be checked by looking at the spectrum of the kernel: it must not contain any zeroes, but even then, it must not contain values close to zero, else quantization effects will be extremely apparent. Quantization effects in general very quickly kill hopes of doing deconvolution the simple way.

2.4 Semiconvolution

It is possible to find which are the kernels that when applied twice have the same effect a given kernel. This can be done by Complex square root, which can be done as division by 2 in the log domain or the cepstrum. Other fractional convolutions may be performed too, by using other constants.

2.5 Crosscorrelation

A kind of similarity comparison may be done by superimposing the images in all possible ways, each time measuring the brightness of the two multiplied images. This is done by convolving an image with a mirror of the other image. That mirror is actually a double mirroring (both x and y) so it's again the same as a 180-degree rotation. The impact on the spectrum is to change the sign of the sine components (conjugation). Cross-correlation of an image with itself is called autocorrelation; it detects self-similarity in an image. Because this is a similarity of translation only (and not rotation nor scaling) it will detect only the periods and directions of periodic patterns, and not of other patterns. I use crosscorrelation to detect and measure motion of the camera (or of the whole scenery) in terms of distances and direction (which is much different from other things that are called "motion detection" in GridFlow).

2.6 Fourier Interpolation

A spectrum may be extended using zeroes so that it becomes the spectrum of a bigger image. Then an inverse FFT will yield that bigger image, which will be exactly band-limited, unlike all other interpolation methods. This isn't necessarily the most desirable, as diffraction patterns appear around any sharp edges in the image (this is called the Gibbs phenomenon in math, but it's equivalent to the diffraction patterns that are observed in physics).

2.7 Spectrum of common convolution kernels

Using a single Fourier transform, one can find the spectrum of that kernel in the context of a certain size of image (each size will yield a different spectrum). This can help understanding common FIR filters in FFT terms. Here are some examples. A rectangle blur is a low-pass filter that has a diffraction pattern in its spectrum (for rectangle sizes above 2). Binomial blurs have coefficients (1 1) or are made by composing those coefficients with themselves, effectively yielding rows from Pascal's triangle; those blurs have no diffraction patterns in their spectra, hence they are "smoother" in some way. The (-1 2 -1) kernel is an edge detector, that is, a high-pass filter. Kernels that are symmetric (palindromic) do not introduce any "phase delay" (that is, the phase delay is either 0 or pi). Phase delay has to be understood in image terms, which is that the contents of an image appear to have moved in a specific direction and for a specific distance when the filter is applied.