The General Fourier Family Transform (GFT) Library

Introduction

The General Fourier Family Transform (GFT) describes all the transforms that use a Fourier-style complex sinusoidal kernel. The most common of these are the Fourier transform itself, and the short-time Fourier transform. As efficient algorithms are widely available to calculate these transforms, this library focuses on a more recent addition, the S-transform (ST). An ST produces local frequency spectra like a short-time Fourier transform, but offers progressive resolution, like a wavelet transform. That is, as the frequency increases, the ST trades off frequency resolution for temporal (or spatial) resolution, so higher frequencies can be better localized. This smooth tradeoff is responsible for many of the desirable properties of wavelets, and is considered by many to offer a better time-frequency representation of a signal.

This package implements the fast frequency domain GFT algorithm published by Brown *et. al.* (IEEE Transactions on Signal Processing, 58, 281-290, 2010), in both 1 dimension (signals) and two dimensions (images). If you publish work using the GFT you should cite this paper. If you feel like writing some code you can easily use the 1D transform to perform the GFT of any number of dimensions.

The fast GFT is an O(N log N) algorithm, which is the same order as the fast Fourier transform. In general, the GFT implemented in this library will take roughly twice as long to calculate as the FFT of the same signal.

Copyright and License

Requirements

The GFT C library requires a recent version of the FFTW library (http://www.fftw.org/ - tested with version 3.1.2).

Building

The GFT C library consists of the files gft.h and gft.c. You can simply include these in your project just as you would any other C files. Additionally, when building your project,

you must add the FFTW library and headers. See the FFTW documentation for details on how to do this for your particular compiler and platform.

For example, on a UNIX-y system you would copy gft.c and gft.h to your project directory, put the following line in your source file:

```
#include "gft.h"
```

and build with a command like:

```
cc myprogram.c -I/usr/local/include -lfftw3 -o myprogram
```

Interface

One Dimensional GFT

```
void gft_1dComplex64(double *signal, unsigned int N, double *win, int
*pars, int stride);
```

This function calculates the 1D GFT.

***signal** is the signal to transform, stored in 128-bit interleaved complex floating point. For example, the amplitude of the first sample in the signal is stored in ***signal[0]** (the real part) and ***signal[1]** (the imaginary part), each being a double precision floating point value.

N is the *number of samples* in ***signal**. Note, the total length of the ***signal** array is equal to 2***N**.

***win** is a pointer to the set of Fourier-space window functions, stored in 128-bit interleaved complex floating point format. ***win** must be the same length as ***signal**. ***win** can be generated by a window set generating function, documented below, or as a custom set by the user. Note: this should probably be changed to take a pointer to a window-generating function, just as the 2D GFT does.

***pars** is a partition set defining how the spectrum will be divided. ***pars** can be generated by a partition generation function, documented below, or can be user generated. However, ***pars** should be constructed carefully, to match ***win** (or *vice versa*). Note: this should probably be changed to take a partition generating function, analogous to the window generating function.

stride describes the distance from one element in ***signal** to the adjacent element. Like **N** this value is in units of samples, not indices. For a 1D GFT this

value should be 1. The NxM standard form (from wavelet terminology) 2D GFT would require M 1D GFTs with stride 1 followed by N 1D GFTs with stride N.

Two Dimensional GFT

```
void gft_2dComplex64(double *signal, unsigned int N, unsigned int M,
windowFunction *window);
```

This function calculates the 2D GFT, in wavelet standard form (all rows transformed, then all columns, yielding a full decomposition).

***signal** is the image to transform, stored in 128-bit interleaved complex floating point. For example, the amplitude of the first sample in the signal is stored in ***signal[0]** (the real part) and ***signal[1]** (the imaginary part), each being a double precision floating point value. For an NxM image, the first sample of the second row is located at ***signal[N*2]** and ***signal[N*2+1]**.

N is the *number of samples* in one row of ***signal**.

M is the *number of samples* in one column of ***signal**.

***window** is a pointer to a window-generating function. This function may be one of the ones provided (documented below) or a custom function of type **windowFunction**.

Types

```
typedef void (windowFunction)(double*,int,int);
```

This is the required type for window generating functions used by the GFT functions. The first argument is an array of double pointers which will hold the window. This array must be allocated by the user (I recommend using one of the provided utility functions), freed by the user, and must be of the right length (the same length as the signal). The second argument is the length, in samples, of the window array, and the third argument is the frequency to use when constructing the window. The output is in the Fourier domain.

Windows

The particular window that is used in a GFT calculation determines which of the GFT subtype transforms will be calculated.

```
void gaussian(double *win, int N, int freq);
```

A window generating function that produces Gaussian windows. GFTs performed with this window are discrete S-transforms. Arguments are as described in the documentation for the **windowFunction** type.

```
void box(double *win, int N, int freq);
```

A window generating function that produces boxcar windows. GFTs performed with this window are discrete orthonormal S-transforms with critical sampling in both frequency and time/space dimensions. Arguments are as described in the documentation for the **windowFunction** type.

Window Set Generators

These are utility functions to generate a full set of windows for a non-overlapping GFT.

```
double *windows(int N, windowFunction *window);
```

A basic window set generator. **N** is the length of the signal that will be transformed, **window** is a window generating function. This function assumes that a standard power-of-two GFT partition layout will be used. The return value is a pointer to a window set array, which should be freed by the user when no longer required.

```
double *windowsFromPars(int N, windowFunction *window, int *pars);
```

A general purpose non-overlapping window set generator. **N** is the length of the signal that will be transformed, **window** is a window generating function and **pars** is a partition set (see next section). The return is a pointer to a window set array, which should be freed by the user when no longer required.

Partition Generators

The partition used in the GFT calculation determines how the time-frequency spectrum is sampled. If only a single partition is used, the result is an FFT. Multiple, constant width partitions produce a STFT. Dyadic partitions produce an ST. The partition map is also required for determining how the output of the GFT calculation is formatted.

```
int *gft_1dPartitions(unsigned int N);
```

This function produces a standard double dyadic arrangement where each successive higher frequency partition has twice the width of the one before

(except for the special cases in the very low frequencies). This arrangement has a consistent time-frequency resolution tradeoff and results in all power-of-two calculations if the original signal has a size that is a power of two. **N** is the length of the signal that will be transformed.

```
int *gft_1dMusicPartitions(unsigned int N, float samplerate, int
cents);
```

This function produces a partition arrangement that is tailored to match the musical scale. The frequency scale is sampled on an exponential scale with **cents** divisions per doubling, with all the musical notes on the 440 Hz reference 12-tone even tempered scale guaranteed to fall in the centre of a partition. If **cents** is set to 1 then each partition represents a semitone. **samplerate** is the sampling rate, in Hz, of the signal and is required to align notes correctly. **N** is the length (in samples) of the signal. This partition generator may serve as a template for others that correspond to other exponential scales of interest. Since this generator does not produce partitions that are powers of two, non-power-of-two FFTs are required to calculate the GFT. The FFTW library can do this quite efficiently, but performance will still be somewhat degraded.

Interpolators

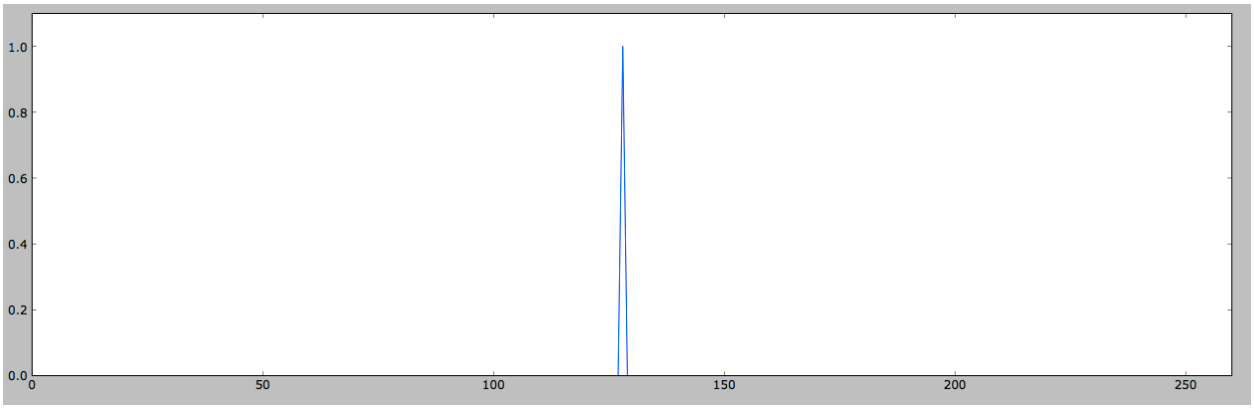
Since the discrete GFT frequently does not produce a uniformly sampled result, interpolation of some form is required for display as a spectrogram-style image. These interpolation functions transform a 1D GFT spectrum into a regularly sampled grid for display.

```
double *gft_1d_interpolateNN(double *signal, unsigned int N, unsigned
int M);
```

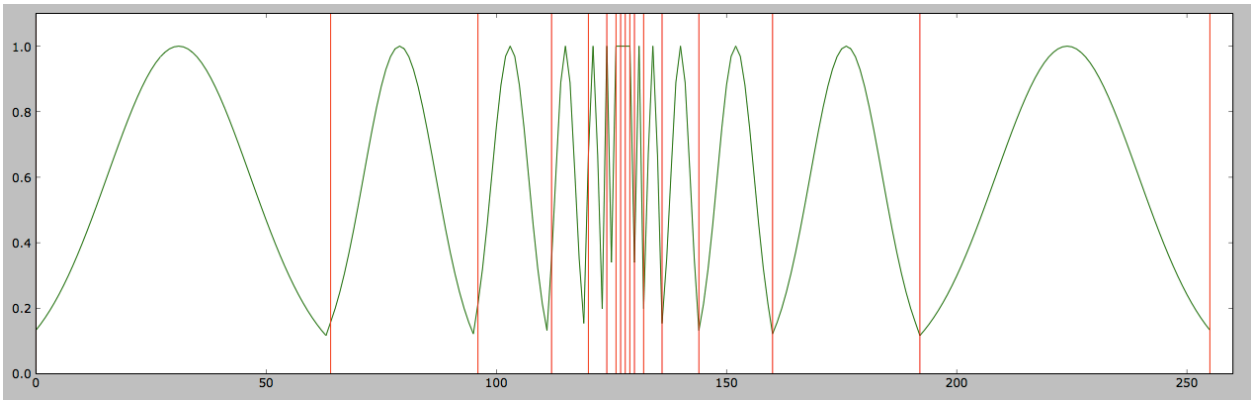
Implements basic nearest-neighbor interpolation of a discrete GFT spectrum. **signal** is the GFT spectrum (should change this), **N** is the length, in samples, of the GFT spectrum (equal to the length of the original signal), and **M** is the desired output size. The output will be a uniformly sampled **M**x**M** grid that can be easily displayed. If **M** is equal to **N** pure interpolation is performed. If **M** is less than **N**, downsampling will occur. Although some detail is lost, this option is useful to fit the spectrum to a particular size (to fill the screen, for example) and, since the uniformly sampled spectrum is frequently very large, may be required due to memory restrictions.

A GFT Example

Consider a simple signal, consisting of a single spike:

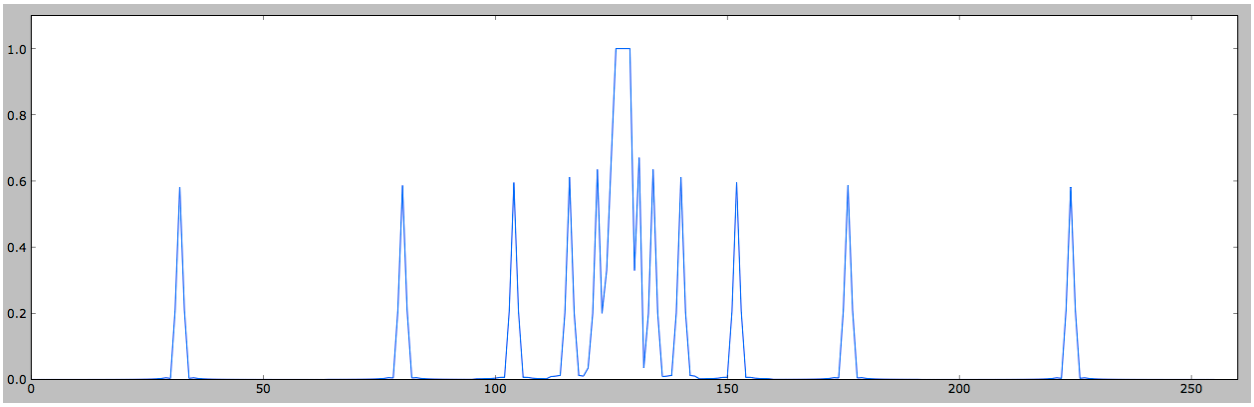


Since the frequency representation of a spike is a constant, the GFT spectrum should have a representation of the spike in each frequency band. To perform the GFT we must first choose windows and partitions. Gaussian windows and standard GFT partition layout are shown, windows in green and partitions in red, shifted so the 0 frequency is at the centre (zero frequency is normally the first element):

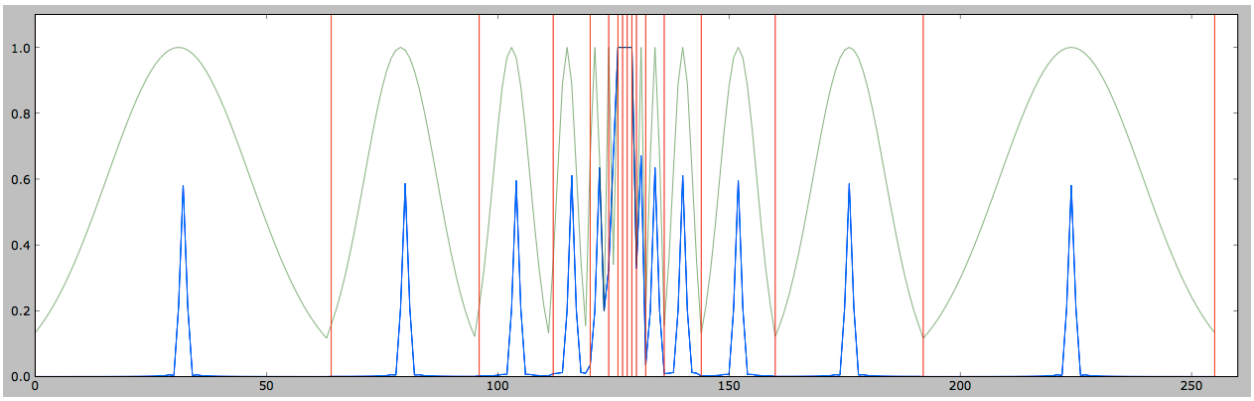


Note how, in the standard GFT partition scheme, the partition bandwidth doubles with each successively higher frequency band.

The GFT of the signal using the standard partition scheme and Gaussian windows looks like this:



Finally, with superimposed windows (transparent green) and partitions (red):



Note that the layout of the GFT spectrum is the same as a discrete wavelet spectrum except that negative frequencies are included. A more traditional spectrogram-like representation can be generated by interpolating the non-uniform sampling of the GFT partitions into uniformly sampled time and frequency axes: